UNIT-I

C++ Class Overview

1. Introduction:-

C in C++

To a large extent, C++ is a superset of C, and most carefully written ANSI C will compile as C++. There are a few major caveats though:

- All functions must be declared before they are used, rather than defaulting to type int.
- All function declarations and definition headers must use new-style declarations, e.g.,

  extern int foo(int a, char* b);

- The form **extern int foo();** means that **foo** takes *no* arguments, rather than arguments of an unspecified type and number.
- In fact, some advise using a C++ compiler even on normal C code, because it will catch errors like misused functions that a normal C compiler will let slide.
- If you need to link C object files together with C++, when you declare the C functions for the C++ files, they must be done like this:

  extern "C" int foo(int a, char* b);

- Otherwise the C++ compiler will alter the name in a strange manner.
- There are a number of new keywords, which you may not use as identifiers --- some common ones **are new, delete, const, and class**.

Basic Concepts:-

Before giving examples of C++ features, I will first go over some of the basic concepts of object-oriented languages. If this discussion at first seems a bit obscure, it will become clearer when we get to some examples.

1. **Classes and objects**. A class is similar to a C *structure*, except that the definition of the data structure, *and* all of the functions that operate on the data structure are grouped together in one place. An *object* is an instance of a class (an instance of the data

structure); objects share the same functions with other objects of the same class, but each object (each instance) has its own copy of the data structure. A class thus defines two aspects of the objects: the *data* they contain, and the *behavior* they have.

2. **Member functions**. These are functions which are considered part of the object and are declared in the class definition. They are often referred to as *methods* of the class. In addition to member functions, a class's behavior is also defined by:
3. What to do when you create a new object (the **constructor** for that object) -- in other words, initialize the object's data.
4. What to do when you delete an object (the **destructor** for that object).
5. **Private vs. public members**. A public member of a class is one that can be read or written by anybody, in the case of a data member, or called by anybody, in the case of a member function. A private member can only be read, written, or called by a member function of that class.

2. Classes:-

Classes are used for two main reasons:

(1) It makes it much easier to organize your programs if you can group together data with the functions that manipulate that data, and
(2) The use of private members makes it possible to do *information hiding*, so that you can be more confident about the way information flows in your programs.

C++ classes are similar to C structures in many ways. In fact, a C++ struct is really a class that has only public data members. In the following explanation of how classes work, we will use a stack class as an example.

1. **Member functions.** Here is a (partial) example of a class with a member function and some data members:

```
2.  class Stack {
3.    public:
4.      void Push(int value); // Push an integer, checking for overflow.
5.      int top;        // Index of the top of the stack.
6.      int stack[10];   // The elements of the stack.
7.  };
8.
9.  void
10. Stack::Push(int value) {
11.     ASSERT(top < 10);                    // stack should never overflow
12.     stack[top++] = value;
13. }
```

This class has two data members, top and stack, and one member function, Push. The notation *class*:: *function* denotes the *function* member of the class *class*. (In the style we use, most function names are capitalized.) The function is defined beneath it.

As an aside, note that we use a call to ASSERT to check that the stack hasn't overflowed; ASSERT drops into the debugger if the condition is false. It is an extremely good idea for you to use ASSERT statements liberally throughout your code to document assumptions made by your implementation. Better to catch errors automatically via ASSERTs than to let them go by and have your program overwrite random locations.

In actual usage, the definition of class Stack would typically go in the file stack.h and the definitions of the member functions, like Stack::Push, would go in the file stack.cc.

If we have a pointer to a Stack object called s, we can access the top element as s->top, just as in C. However, in C++ we can also call the member function using the following syntax:

```
s->Push(17);
```

Of course, as in C, s must point to a valid Stack object.

Inside a member function, one may refer to the members of the class by their names alone. In other words, the class definition creates a scope that includes the member (function and data) definitions.

Note that if you are inside a member function, you can get a pointer to the object you were called on by using the variable this. If you want to call another member function on the same object, you do not need to use the this pointer, however. Let's extend the Stack example to illustrate this by adding a Full() function.

```
class Stack {
  public:
    void Push(int value); // Push an integer, checking for overflow.
    bool Full();      // Returns TRUE if the stack is full, FALSE otherwise.
    int top;          // Index of the lowest unused position.
    int stack[10];    // A pointer to an array that holds the contents.
};
bool
Stack::Full() {
    return (top == 10);
}
```

Now we can rewrite Push this way:

```
void
Stack::Push(int value) {
    ASSERT(!Full());
    stack[top++] = value;
}
```

We could have also written the ASSERT:

```
ASSERT(!(this->Full());
```

but in a member function, the this-> is implicit.

The purpose of member functions is to encapsulate the functionality of a type of object along with the data that the object contains. A member function does not take up space in an object of the class.

14. **Private members.** One can declare some members of a class to be *private*, which are hidden to all but the member functions of that class, and some to be *public*, which are visible and accessible to everybody. Both data and function members can be either public or private.

In our stack example, note that once we have the Full() function, we really don't need to look at the top or stack members outside of the class -- in fact, we'd rather that users of the Stack abstraction *not* know about its internal implementation, in case we change it. Thus we can rewrite the class as follows:

```
class Stack {
  public:
    void Push(int value); // Push an integer, checking for overflow.
    bool Full();      // Returns TRUE if the stack is full, FALSE otherwise.
  private:
    int top;          // Index of the top of the stack.
    int stack[10];    // The elements of the stack.
};
```

Before, given a pointer to a Stack object, say s, any part of the program could access s->top, in potentially bad ways. Now, since the top member is private, only a member function, such as Full(), can access it. If any other part of the program attempts to use s->top the compiler will report an error.

You can have alternating public: and private: sections in a class. Before you specify either of these, class members are private, thus the above example could have been written:

```
class Stack {
    int top;          // Index of the top of the stack.
    int stack[10];    // The elements of the stack.
  public:
    void Push(int value); // Push an integer, checking for overflow.
    bool Full();      // Returns TRUE if the stack is full, FALSE otherwise.
};
```

Which form you prefer is a matter of style, but it's usually best to be explicit, so that it is obvious what is intended. In Nachos, we make everything explicit.

What is not a matter of style: **all data members of a class should be private.** All operations on data should be via that class' member functions. Keeping data private adds to the modularity of the system, since you can redefine how the data members are stored without changing how you access them.

15. **Constructors and the operator new.** In C, in order to create a new object of type Stack, one might write:

```
16.         struct Stack *s = (struct Stack *) malloc(sizeof (struct Stack));
17.         InitStack(s, 17);
```

The InitStack() function might take the second argument as the size of the stack to create, and use malloc() again to get an array of 17 integers.

The way this is done in C++ is as follows:

```
   Stack *s = new Stack(17);
```

The new function takes the place of malloc(). To specify how the object should be initialized, one declares a *constructor* function as a member of the class, with the name of the function being the same as the class name:

```
class Stack {
  public:
    Stack(int sz);    // Constructor:  initialize variables, allocate space.
    void Push(int value); // Push an integer, checking for overflow.
    bool Full();      // Returns TRUE if the stack is full, FALSE otherwise.
  private:
    int size;         // The maximum capacity of the stack.
    int top;          // Index of the lowest unused position.
    int* stack;       // A pointer to an array that holds the contents.
};

Stack::Stack(int sz) {
    size = sz;
    top = 0;
    stack = new int[size];   // Let's get an array of integers.
}
```

There are a few things going on here, so we will describe them one at a time.

The new operator automatically creates (i.e. allocates) the object and then calls the constructor function for the new object. This same sequence happens even if, for instance, you declare an object as an automatic variable inside a function or block -- the compiler allocates space for the object on the stack, and calls the constructor function on it.

In this example, we create two stacks of different sizes, one by declaring it as an automatic variable, and one by using new.

```
void
test() {
   Stack s1(17);
   Stack* s2 = new Stack(23);
}
```

Note there are two ways of providing arguments to constructors: with new, you put the argument list after the class name, and with automatic or global variables, you put them after the variable name.

It is crucial that you **always** define a constructor for every class you define, and that the constructor initialize **every** data member of the class. If you don't define your own constructor, the compiler will automatically define one for you, and believe me, it won't do what you want (``the unhelpful compiler''). The data members will be initialized to random, unrepeatable values, and while your program may work anyway, it might not the next time you recompile (or vice versa!).

As with normal C variables, variables declared inside a function are deallocated automatically when the function returns; for example, the s1 object is deallocated when test returns. Data allocated with new (such as s2) is stored on the heap, however, and remains after the function returns; heap data must be explicitly disposed of using delete, described below.

The new operator can also be used to allocate arrays, illustrated above in allocating an array of ints, of dimension size:

```
   stack = new int[size];
```

Note that you can use new and delete (described below) with built-in types like int and char as well as with class objects like Stack.

18. **Destructors and the operator delete.** Just as new is the replacement for malloc(), the replacement for free() is delete. To get rid of the Stack object we allocated above with new, one can do:

19. delete s2;

This will deallocate the object, but first it will call the *destructor* for the Stack class, if there is one. This destructor is a member function of Stack called ~Stack():

```
class Stack {
  public:
    Stack(int sz);    // Constructor:  initialize variables, allocate space.
    ~Stack();         // Destructor:   deallocate space allocated above.
    void Push(int value); // Push an integer, checking for overflow.
```

```
  bool Full();      // Returns TRUE if the stack is full, FALSE otherwise.
 private:
  int size;         // The maximum capacity of the stack.
  int top;          // Index of the lowest unused position.
  int* stack;       // A pointer to an array that holds the contents.
};

Stack::~Stack() {
  delete [] stack;  // delete an array of integers
}
```

The destructor has the job of deallocating the data the constructor allocated. Many classes won't need destructors, and some will

Example:-

```
class Point {
  int _x, _y;       // point coordinates

 public:            // begin interface section
  void setX(const int val);
  void setY(const int val);
  int getX() { return _x; }
  int getY() { return _y; }
 };

 Point apoint;


 void Point::setX(const int val) {
  _x = val;
 }

 void Point::setY(const int val) {
  _y = val;
 }

 apoint.setX(1);     // Initialization
 apoint.setY(1);



 void Point::setX(const int val) {
```

```
  this->_x = val;   // Use this to reference invoking
                    // object
 }

 void Point::setY(const int val) {
  this->_y = val;
 }
Constructors
class Point {
  int _x, _y;

 public:
  Point() {
    _x = _y = 0;
  }

  void setX(const int val);
  void setY(const int val);
  int getX() { return _x; }
  int getY() { return _y; }
 };
 class Point {
  int _x, _y;

 public:
  Point() {
    _x = _y = 0;
  }
  Point(const int x, const int y) {
    _x = x;
    _y = y;
  }

  void setX(const int val);
  void setY(const int val);
  int getX() { return _x; }
  int getY() { return _y; }
 };
Point apoint;          // Point::Point()
 Point bpoint(12, 34);   // Point::Point(const int, const int)



class Point {
  int _x, _y;
```

```
public:
  Point() {
    _x = _y = 0;
  }
  Point(const int x, const int y) {
    _x = x;
    _y = y;
  }
  Point(const Point &from) {
    _x = from._x;
    _y = from._y;
  }

  void setX(const int val);
  void setY(const int val);
  int getX() { return _x; }
  int getY() { return _y; }
};
Point bpoint(apoint);    // Point::Point(const Point &)
Point cpoint = apoint;   // Point::Point(const Point &)
  ~Point() { /* Nothing to do! */ }
```

<u>Other Basic C++ Features:-</u>

Here are a few other C++ features that are useful to know.

(1) When you define a class Stack, the name Stack becomes usable as a type name as if created with **typedef.** The same is true for **enums.**

(2) You can define functions inside of a class definition, whereupon they become ***inline functions***, which are expanded in the body of the function where they are used. The rule of thumb to follow is to only consider inlining one-line functions, and even then do so rarely.

As an example, we could make the Full routine an inline.

```
class Stack {
  ...
  bool Full() { return (top == size); };
  ...
            };
```

There are two motivations for **inlines**: convenience and performance. If overused, inlines can make your code more confusing, because the implementation for an object is no longer in one place, but spread between the .h and .c files. Inlines can sometimes speed up your code (by avoiding the overhead of a procedure call), but that shouldn't be your principal concern as a student (rather, at least to begin with, you should be most concerned with writing code that is simple and bug free). Not to mention that inlining sometimes slows down a program, since the object code for the function is duplicated wherever the function is called, potentially hurting cache performance.

1. Inside a function body, you can declare some variables, execute some statements, and then declare more variables. This can make code a lot more readable. In fact, you can even write things like:
2. for (int i = 0; i < 10; i++) ;

Depending on your compiler, however, the variable i may still visible after the end of the for loop, however, which is not what one might expect or desire.

3. Comments can begin with the characters // and extend to the end of the line. These are usually more handy than the /* */ style of comments.
4. C++ provides some new opportunities to use the const keyword from ANSI C. The basic idea of const is to provide extra information to the compiler about how a variable or function is used, to allow it to flag an error if it is being used improperly. You should always look for ways to get the compiler to catch bugs for you. After all, which takes less time? Fixing a compiler-flagged error, or chasing down the same bug using gdb?

For example, you can declare that a member function only reads the member data, and never modifies the object:

```
class Stack {
  ...
  bool Full() const;  // Full() never modifies member data
  ...
};
```

As in C, you can use const to declare that a variable is never modified:

```
   const int InitialHashTableSize = 8;
```

This is *much* better than using #define for constants, since the above is type-checked.

5. Input/output in C++ can be done with the >> and << operators and the objects cin and cout. For example, to write to stdout:
6. cout << "Hello world! This is section " << 3 << "!";

This is equivalent to the normal C code

```
fprintf(stdout, "Hello world!  This is section %d!\n", 3);
```

except that the C++ version is type-safe; with printf, the compiler won't complain if you try to print a floating point number as an integer. In fact, you can use traditional printf in a C++ program, but you will get bizarre behavior if you try to use both printf and << on the same stream. Reading from stdin works the same way as writing to stdout, except using the shift right operator instead of shift left. In order to read two integers from stdin:

```
int field1, field2;
cin >> field1 >> field2;
   // equivalent to fscanf(stdin, "%d %d", &field1, &field2);
   // note that field1 and field2 are implicitly modified
```

In fact, cin and cout are implemented as normal C++ objects, using operator overloading and reference parameters, but (fortunately!) you don't need to understand either of those to be able to do I/O in C++.

Function members in classes:-

Functions declared inside a class can be any of the following four types. This C++ Tutorial explains each one of them as below.

Ordinary member functions :

These are ordinary functions defined with a return type and parameters. The return type can also be void. The special trait about **member** functions is they can access the private/protected data members of their class and manipulate them. No external functions can access the private/protected data members of a class. The sample below this C++ Tutorial uses an ordinary member function Add(), returning an integer value.

Constructors:

Constructors in C++ are special member functions of a class. They have the same name as the Class Name. There can be any number of overloaded constructors inside a class, provided they have a different set of parameters. There are some important qualities for a constructor to be noted.

- Constructors have the same name as the class.

- Constructors do not return any values
- Constructors are invoked first when a class is initialized. Any initializations for the class members, memory allocations are done at the constructor.

In the example class given below in this C++ tutorial has the constructor Example_Class(), with the same name as the class.

Destructors:

Destructors in C++ also have the same name, except for the fact that they are preceded by a '~' operator. The destructors are called when the object of a class goes out of scope. It is not necessary to declare a constructor or a destructor inside a class. If not declared, the compiler will automatically create a default one for each. If the constructor/destructor is declared as private, then the class cannot be instantiated. Check below for the sample class of the C++ tutorial for an example of destructor.

3. Access Control:-

The classes in C++ have 3 important access levels. They are **Private, Public** and **Protected**. The explanations are as follows.

Private:

The members are accessible only by the member functions or friend functions.

Protected:

These members are accessible by the member functions of the class and the classes which are derived from this class.

Public:

Accessible by any external member. Look at the sample class below.

Example of a class:

```
class Example_class //Sample Class for the C++

{
   private:
     int x; //Data member
     int y; // Data member
   public:
     Example_Class() //Constructor for the C++

     {
        x = 0;
        y = 0;
```

```
    }
  ~Example_Class() //destructor for the C++

  { }
  int Add()
  {
    return x+y;
  }
};
```

4. Constructors:-

- Are used to initialize the member variables of the class when the objects of the class are created
- Must have the same name as that of class name
- Cannot return any value, not even a void type
- Class can have more than one constructors defined in it (known as overloaded constructors)
- Default constructors accept no parameters and are automatically invoked by the compiler

Need for Constructors

- To initialize a member variable at the time of declaration

Declaration of Constructors

Example:

```
class Calculator
  {
private:
        int number1, number2, tot;
public:
...
Calculator()
{
number1 = number2 = tot = 0;
cout << "Constructor invoked"<< endl;
}
};
```

Destructors:-

- Are used to de-initialize the objects when they are destroyed
- Are used to clear memory space occupied by a data member when an object goes out of scope
- Must have the same name as that of the class, preceded by a ~ (For example: ~Calculator())
- Are automatically invoked
- Can also be explicitly invoked when required
- Cannot be overloaded

Need for Destructors

- To de-initialize the objects when they are destroyed
- To clear memory space occupied by a data member when an object goes out of scope

Declaration of Destructors

Example:

```
/* This Code Shows The Use Of Destructor In The Calculator Class */
class Calculator
{
private:
        int number1, number2, tot;
public:
        ...
    ~Calculator()  //Body Of The Destructor
    {
    number1 = number2 = tot = 0;
    }
};
```

Just a Minute…

Given is a code snippet for the main() function of a program. How would you ensure that the following tasks are accomplished when the program is executed:

1. The member variables are initialized with zero

2. On exit or termination of the program the following message is displayed : “Bye Folks!!! Have a Nice Time”

```
#include <iostream>

int main()

{

        Number num1;num1.disp();

}
```

Hint: The Number class has only one member variable myNum.

Scope Resolution Operator (::)

- Is used to define member functions outside the class definition therefore making the class definition more readable
- Example:

```
class calculator
{
public:
     void input();
};
   void calculator::input ()
  {…}
```

Constructors with Parameters

- Allow member variables of the class to be initialized with user supplied values from main() function also called parameters

Example

```
class calculate
{
private:
int num1,num2,total;
public:
calculate(int, int);
};
calculate::calculate(int x, int y)
{
num1=x;num2=y;
total=0;
}
int main()
{//Accept values in two variable var1 & //var2
calculate c1(var1,var2);
}
```

Constructors with Default Arguments

```
calculate::calculate(int x, int y=10);

int main()
{
calculate c1(5);
}
```

Invoking Member Functions

- By using call by value
- By using call by reference

Call by Value

- Is useful when the function does not need to modify the values of the original variables
- Does not affect the values of the variables in caller functions

Problem Statement 5.P.1

Predict the Output:

```
#include <iostream>
void square(int);
class functionCall
```

```
{
        int number;
        public:
        functionCall();
};
functionCall::functionCall()
{
        number=10;square(number);
}
void square(int num)
{
        cout<<num<<endl;
        num *= num; //This Expression Is //Resolved As num = num * num
        cout << num << endl;
}
int main()
{
        functionCall f1;
        return 0;
                }
```

Call by Reference

Is implemented by using

(1) An alias
(2) Pointers

Call by Reference (Contd.)

- Using an alias

(1) The same variable can be referenced by more than one name by using the & or the alias operator
(2) The change in the value of the variable by the called or the calling program is reflected in all the affected functions

Call by Reference using Pointers

To define and declare a pointer variable:

(1) Using pointers

- Involves a pointer variable storing the memory address of any variable

- Is advantageous since it allows direct access to individual bytes in the memory and output devices

5. Parameter Passing Methods :-

- Overview
- Value Parameters
- Reference Parameters
- Const Reference Parameters
- Array Parameters

In Java, all parameters are passed by value. In C++, a parameter can be passed by:

1. value,
2. reference, or
3. const-reference

Each parameter's mode is determined by the way it is specified in the function's header (the mode is the same for all calls to the function). For example:

void f(int a, int &b, const int &c);
Parameter *a* is a value parameter, *b* is a reference parameter, and *c* is a const-reference parameter.

Value Parameters

When a parameter is passed by value, a *copy* of the parameter is made. Therefore, changes made to the formal parameter by the called function have no effect on the corresponding actual parameter. For example:

```
void f(int n) {
   n++;
}

int main() {
   int x = 2;
   f(x);
   cout << x;
}
```

In this example, *f's* parameter is passed by value. Therefore, although *f* increments its formal parameter *n*, that has no effect on the actual parameter *x*. The value output by the program is 2 (not 3).

Note that if a *pointer* is passed by value, then although the pointer itself is not affected by changes made to the corresponding formal parameter, the object pointed by the pointed *can* be changed. For example:

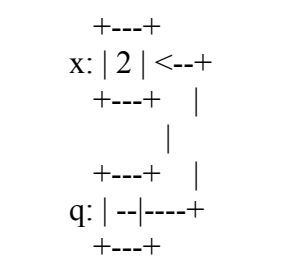
```
void f(int *p) {
   *p = 5;
   p = NULL;
}

int main() {
   int x=2;
   int *q = &x;

   f(q);

   // here, x == 5, but q != NULL
}
```

In this example, *f's* parameter is passed by value. Therefore, the assignment *p = NULL;* in *f* has no effect on variable *q* in main (since *f* was passed a copy of *q*, not *q* itself). However, the assignment **p = 5;* in *f*, **does** change the value pointed to by *q*. To understand why, consider what happens when the example program runs:

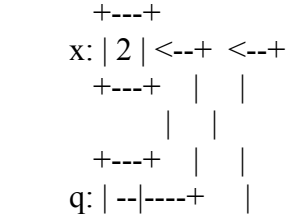
After executing the two statements:

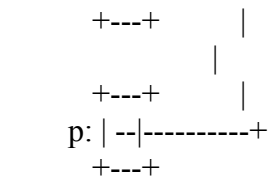
```
   int x=2;
   int *q = &x;
```

memory looks like this:



Now function f is called; the value of q (which is the address of x)
is copied into a new location named p:

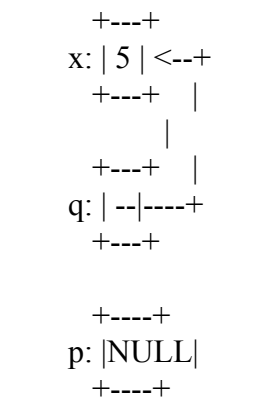




Executing the two statements in f:

```
*p = 5;
p = NULL;
```

causes the values of x (the thing pointed to by p) and p to be changed:



However, note that q is NOT affected.

Reference Parameters

When a parameter is passed by reference, conceptually, the actual parameter itself is passed (and just given a new name -- the name of the corresponding formal parameter). Therefore, any changes made to the formal parameter *do* affect the actual parameter. For example:

```
void f(int &n) {
   n++;
}

int main() {
   int x = 2;
   f(x);
   cout << x;
}
```

In this example, *f's* parameter is passed by reference. Therefore, the assignment to *n* in *f* is actually changing variable *x*, so the output of this program is 3.

When you write a function whose purpose is to compute two or more values, it makes sense to use reference parameters (since a function can *return* only one result). For example, if you want

to read a list of integers from a file, and you want to know both how many integers were read, as well as the average value that was read, you might use a function like the following:

```
void f(istream &input, int &numRead, double &average) {
   int k, sum = 0;
   numRead = 0;

   while (intput >> k) {
      numRead++;
            sum += k;
   }
   average = (double)sum/numRead;
}
```

Another common use of reference parameters is for a function that swaps two values:

```
void swap( int &j, int &k ) {
   int tmp = j;
   j = k;
   j = tmp;
}
```

This is useful, for example, in sorting an array, when it is often necessary to swap two array elements. The following code swaps the j^{th} and k^{th} elements of array A:

```
swap(A[j], A[k]);
```

Const-Reference Parameters

Another reason to use reference parameters is when you don't want the function to modify an actual parameter, but the actual parameter is very large, and you want to avoid the overhead of creating a copy. Of course, this only works if the function does not modify its formal parameter. To be sure that the actual parameter is not "accidentally" modified, you should use a **const-reference** parameter. Declaring the parameter to be *const* tells the compiler that it should not be changed; if the function does change the parameter, you will get a compile-time warning (possibly an error on some systems). For example:

```
void f(const IntList &L) {
 -- the code here cannot modify L or the compiler will complain --
}
```

The potential use of a const-reference parameter is the reason why member functions that do not modify any data members should be declared *const*. For example, suppose that the *IntList Print* member function was not declared const. Then the following code would cause a compile-time error:

```
void f(const IntList &L) {
  L.Print(cout);
}
```

Because *L* is a const-reference parameter, it is the compiler's job to be sure that *L* is not modified by *f* (and that means that no data members of *L* are modified). The compiler doesn't know how the *Print* function is implemented; it only knows how it was declared, so if it is not declared const, it assumes the worst, and complains that function *f* modifies its const-reference parameter *L*.

Array Parameters

Another unfortunate thing about C++ arrays is that they are *always* passed by reference (even though you don't declare them to be reference parameters). For example:

```
void f(int A[]) {
   A[0] = 5;
}

int main() {
   int B[10];
   B[0] = 2;
   f(B);
   cout << B[0] << endl;  // the output is 5
}
```

Although *f's* parameter looks like it is passed by value (there is no &), since it is an array it is actually passed by reference, so the assignment to *A[0]* is really assigning to *B[0]*, and the program prints 5 (not 2).

If you want to pass an array by value, you should use a **vector**, not a regular C++ array (see the last section in the notes on C++ classes for information about vectors).

7. Inline Functions:-

When a function is declared **inline**, the function is expanded at the calling block. The function is not treated as a separate unit like other normal functions.

But a compiler is free to decide, if a function qualifies to be an inline function. If the inline function is found to have larger chunk of code, it will not be treated as an inline function, but as like other normal functions.

Inline functions are treated like macro definitions by the C++ compiler. They are declared with the keyword inline as follows.

```
//Declaration for C++ Tutorial inline sample:
int add(int x,int y);

//Definition for C++ Tutorial inline sample:
inline int add(int x,int y)
{
return x+y;
}
```

In fact, the keyword inline is not necessary. If the function is defined with its body directly and the function has a smaller block of code, it will be automatically treated as inline by the compiler.

As implied, inline functions are meant to be used if there is a need to repetitively execute a small block of code, which is smaller. When such functions are treated inline, it might result in a significant performance difference.

8.Static Data Members:-

Classes can contain static member data and member functions. When a data member is declared as **static**, only one copy of the data is maintained for all objects of the class. (For more information, see Static Member Functions.)

Static data members are not part of objects of a given class type; they are separate objects. As a result, the declaration of a static data member is not considered a definition. The data member is declared in class scope, but definition is performed at file scope. These static members have external linkage. The following example illustrates this:

```
// static_data_members.cpp
class BufferedOutput
{
public:
// Return number of bytes written by any object of this class.
short BytesWritten()
{
return bytecount;
}
```

```
// Reset the counter.
static void ResetCount()
{
bytecount = 0;
}

// Static member declaration.
static long bytecount;
};

// Define bytecount in file scope.
long BufferedOutput::bytecount;

int main()
{  }
```

In the preceding code, the member bytecount is declared in class BufferedOutput, but it must be defined outside the class declaration.

Static data members can be referred to without referring to an object of class type. The number of bytes written using BufferedOutput objects can be obtained as follows:

```
long nBytes = BufferedOutput::bytecount;
```

For the static member to exist, it is not necessary that any objects of the class type exist. Static members can also be accessed using the member-selection (**.** and **–>**) operators. For example:

```
BufferedOutput Console;


long nBytes = Console.bytecount;
```

In the preceding case, the reference to the object (Console) is not evaluated; the value returned is that of the static object bytecount.

Static data members are subject to class-member access rules, so private access to static data members is allowed only for class-member functions and friends. These rules are described in Member-Access Control. The exception is that static data members must be defined in file scope

regardless of their access restrictions. If the data member is to be explicitly initialized, an initializer must be provided with the definition.

The type of a static member is not qualified by its class name. Therefore, the type of BufferedOutput::bytecount is long.

9. The this pointer:-

The keyword this identifies a special type of pointer. Suppose that you create an object named x of class A, and class A has a non_static member function f(). If you call the function x.f(), the keyword this in the body of f() stores the address of x. You cannot declare the this pointer or make assignments to it.

A static member function does not have this pointer.

The type of this pointer for a member function of a class type X, is X* const. If the member function is declared with the **const** qualifier, the type of this pointer for that member function for **class X, is const X* const.**

A const this pointer can by used only with const member functions. Data members of the class will be constant within that function. The function is still able to change the value, but requires a const_cast to do so:

```
void foo::p() const{
member = 1;                    // illegal
const_cast <int&> (member) = 1;   // a bad practice but legal
}
```

A better technique would be to declare member mutable.

If the member function is declared with the **volatile** qualifier, the type of the this pointer for that member function for class X is volatile X* const. For example, the compiler will not allow the following:

```
struct A {
  int a;
  int f() const { return a++; }
};
```

The compiler will not allow the statement a++ in the body of function f(). In the function f(), the this pointer is of type A* const. The function f() is trying to modify part of the object to which this points.

The this pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions.

For example, you can refer to the particular class object that a member function is called for by using the this pointer in the body of the member function. The following code example produces the output a = 5:

```
#include <iostream>
using namespace std;

struct X {
private:
  int a;
public:
  void Set_a(int a) {

    // The 'this' pointer is used to retrieve 'xobj.a'
    // hidden by the automatic variable 'a'
    this->a = a;
  }
   void Print_a() { cout << "a = " << a << endl; }
};

int main() {
  X xobj;
  int a = 5;
  xobj.Set_a(a);
  xobj.Print_a();
}
```

In the member function Set_a(), the statement this->a = a uses the this pointer to retrieve xobj.a hidden by the automatic variable a.

Unless a class member name is hidden, using the class member name is equivalent to using the class member name with the this pointer and the class member access operator (->).

The example in the first column of the following table shows code that uses class members without the this pointer. The code in the second column uses the variable THIS to simulate the first column's hidden use of the this pointer:

10. Dynamic Memory Allocation and Deallocation:-

Until now, in all our programs, we have only had as much memory available as we declared for our variables, having the size of all of them to be determined in the source code, before the execution of the program. But, what if we need a variable amount of memory that can only be determined during runtime? For example, in the case that we need some user input to determine the necessary amount of memory space.

The answer is *dynamic memory*, for which C++ integrates the operators new and delete.

Operators new and new[]

In order to request dynamic memory we use the operator new. new is followed by a data type specifier and -if a sequence of more than one element is required- the number of these within brackets []. It returns a pointer to the beginning of the new block of memory allocated. Its form is:

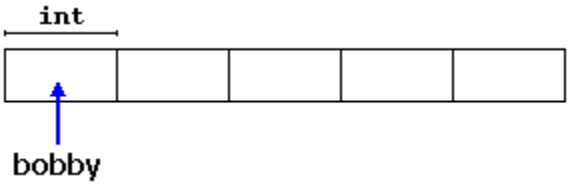
pointer = new type
pointer = new type [number_of_elements]

The first expression is used to allocate memory to contain one single element of type type. The second one is used to assign a block (an array) of elements of type type, where number_of_elements is an integer value representing the amount of these. For example:

Int *bobby;

Bobby = new int[5];

this case, the system dynamically assigns space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to bobby. Therefore, now, bobby points to a valid block of memory with space for five elements of type int.



The first element pointed by bobby can be accessed either with the expression bobby[0] or the expression *bobby. Both are equivalent as has been explained in the section about pointers. The second element can be accessed either with bobby[1] or *(bobby+1) and so on...

You could be wondering the difference between declaring a normal array and assigning dynamic memory to a pointer, as we have just done. The most important difference is that the size of an

array has to be a constant value, which limits its size to what we decide at the moment of designing the program, before its execution, whereas the dynamic memory allocation allows us to assign memory during the execution of the program (runtime) using any variable or constant value as its size.

The dynamic memory requested by our program is allocated by the system from the memory heap. However, computer memory is a limited resource, and it can be exhausted. Therefore, it is important to have some mechanism to check if our request to allocate memory was successful or not.

C++ provides two standard methods to check if the allocation was successful:

One is by handling exceptions. Using this method an exception of type bad_alloc is thrown when the allocation fails. Exceptions are a powerful C++ feature explained later in these tutorials. But for now you should know that if this exception is thrown and it is not handled by a specific handler, the program execution is terminated.

This exception method is the default method used by new, and is the one used in a declaration like:

bobby = new int[5]; // if it fails an exception is thrown

The other method is known as nothrow, and what happens when it is used is that when a memory allocation fails, instead of throwing a bad_alloc exception or terminating the program, the pointer returned by new is a null pointer, and the program continues its execution.

This method can be specified by using a special object called nothrow, declared in header <new>, as argument for new:

```
bobby = new (nothrow) int [5];
```

In this case, if the allocation of this block of memory failed, the failure could be detected by checking if bobby took a null pointer value:

```
int * bobby;
bobby = new (nothrow) int [5];
if (bobby == 0) {
  // error assigning memory. Take measures.

  };
```

This nothrow method requires more work than the exception method, since the value returned has to be checked after each and every memory allocation, but I will use it in our examples due to its simplicity. Anyway this method can become tedious for larger projects, where the

exception method is generally preferred. The exception method will be explained in detail later in this tutorial.

Operator delete and delete[]

Since the necessity of dynamic memory is usually limited to specific moments within a program, once it is no longer needed it should be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of the operator delete, whose format is:

delete pointer;

delete [] pointer;

The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for arrays of elements.

The value passed as argument to delete must be either a pointer to a memory block previously allocated with new, or a null pointer (in the case of a null pointer, delete produces no effect).

```
// rememb-o-matic
#include <iostream>
#include <new>
using namespace std;

int main ()
{
  int i,n;
  int * p;
  cout << "How many numbers would you like to type? ";
  cin >> i;
  p= new (nothrow) int[i];
  if (p == 0)
    cout << "Error: memory could not be allocated";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Enter number: ";
      cin >> p[n];
    }
    cout << "You have entered: ";
    for (n=0; n<i; n++)
      cout << p[n] << ", ";
```

```
   delete[] p;
  }
  return 0;

}
```

Output:-

```
How many numbers would you like to type? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32

You have entered: 75, 436, 1067, 8, 32,
```

Notice how the value within brackets in the new statement is a variable value entered by the user (i), not a constant value:

```
p= new (nothrow) int[i];
```

But the user could have entered a value for i so big that our system could not handle it. For example, when I tried to give a value of 1 billion to the "How many numbers" question, my system could not allocate that much memory for the program and I got the text message we prepared for this case (Error: memory could not be allocated). Remember that in the case that we tried to allocate the memory without specifying the nothrow parameter in the new expression, an exception would be thrown, which if it's not handled terminates the program.

It is a good practice to always check if a dynamic memory block was successfully allocated. Therefore, if you use the nothrow method, you should always check the value of the pointer returned. Otherwise, use the exception method, even if you do not handle the exception. This way, the program will terminate at that point without causing the unexpected results of continuing executing a code that assumes a block of memory to have been allocated when in fact it has not.

11. Exception Handling:-

Exceptions are run-time anomalies, such as division by zero, that require immediate handling when encountered by your program. The C++ language provides built-in support for raising and handling exceptions. With C++ exception handling, your program can communicate unexpected events to a higher execution context that is better able to recover from such abnormal events. These exceptions are handled by code that is outside the normal flow of control.

The try, catch, and throw Statements

The following syntax shows a **try** block and its handlers:

```
try {
  // code that could throw an exception
}
[ catch (exception-declaration) {
  // code that executes when exception-declaration is thrown
  // in the try block
}
[catch (exception-declaration) {
  // code that handles another exception type
} ] . . . ]
// The following syntax shows a throw expression:
throw [expression]
```

⊟ Remarks

The C++ language provides built-in support for handling anomalous situations, known as exceptions, which may occur during the execution of your program. The **try**, **throw**, and **catch** statements implement exception handling. With C++ exception handling, your program can communicate unexpected events to a higher execution context that is better able to recover from such abnormal events. These exceptions are handled by code that is outside the normal flow of control. The Microsoft C++ compiler implements the C++ exception handling model based on the ANSI C++ standard.

C++ also provides a way to explicitly specify whether a function can throw exceptions. You can use exception specifications in function declarations to indicate that a function can throw an exception. For example, an exception specification throw(...) tells the compiler that a function can throw an exception, but doesn't specify the type, as in this example:

```
void MyFunc() throw(...) {
  throw 1;
}
```

For more information, see Exception Specifications.

The code after the **try** clause is the guarded section of code. The throw expression throws (raises) an exception. The code block after the **catch** clause is the exception handler, and catches (handles) the exception thrown by the throw expression. The exception-declaration statement indicates the type of exception the clause handles. The type can be any valid data type, including a C++ class. If the exception-declaration statement is an ellipsis (**...**), the **catch** clause handles any type of exception, including C exceptions and system- or application-generated exceptions such as memory protection, divide by zero, and floating-point violations. Such a handler must be the last handler for its **try** block.

The operand of **throw** is syntactically similar to the operand of a **return** statement.

Execution proceeds as follows:

(2) Control reaches the **try** statement by normal sequential execution. The guarded section within the **try** block is executed.

(3) If no exception is thrown during execution of the guarded section, the **catch** clauses that follow the **try** block are not executed. Execution continues at the statement after the last **catch** clause following the **try** block in which the exception was thrown.

(4) If an exception is thrown during execution of the guarded section or in any routine the guarded section calls (either directly or indirectly), an exception object is created from the object created by the **throw** operand. (This implies that a copy constructor may be involved.) At this point, the compiler looks for a **catch** clause in a higher execution context that can handle an exception of the type thrown (or a **catch** handler that can

handle any type of exception). The **catch** handlers are examined in order of their appearance following the **try** block. If no appropriate handler is found, the next dynamically enclosing **try** block is examined. This process continues until the outermost enclosing **try** block is examined.

(5) If a matching handler is still not found, or if an exception occurs while unwinding, but before the handler gets control, the predefined run-time function terminate is called. If an exception occurs after throwing the exception, but before the unwind begins, terminate is called.

(6) If a matching **catch** handler is found, and it catches by value, its formal parameter is initialized by copying the exception object. If it catches by reference, the parameter is initialized to refer to the exception object. After the formal parameter is initialized, the process of unwinding the stack begins. This involves the destruction of all automatic objects that were constructed (but not yet destructed) between the beginning of the **try** block associated with the **catch** handler and the exception's throw site. Destruction occurs in reverse order of construction. The **catch** handler is executed and the program resumes execution following the last handler (that is, the first statement or construct which is not a **catch** handler). Control can only enter a **catch** handler through a thrown exception, never via a **goto** statement or a **case** label in a **switch** statement.

The following is a simple example of a **try** block and its associated **catch** handler. This example detects failure of a memory allocation operation using the **new** operator. If **new** is successful, the **catch** handler is never executed:

```
// exceptions_trycatchandthrowstatements.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
int main() {
  char *buf;
  try {
    buf = new char[512];
    if( buf == 0 )
      throw "Memory allocation failure!";
```

```
  }
  catch( char * str ) {
    cout << "Exception raised: " << str << '\n';
  }
}
```

The operand of the **throw** expression specifies that an exception of type char * is being thrown. It is handled by a **catch** handler that expresses the ability to catch an exception of type char *. In the event of a memory allocation failure, this is the output from the preceding example:

Exception raised: Memory allocation failure!

The real power of C++ exception handling lies not only in its ability to deal with exceptions of varying types, but also in its ability to automatically call destructor functions during stack unwinding, for all local objects constructed before the exception was thrown.

The following example demonstrates C++ exception handling using classes with destructor semantics:

Example

```
// exceptions_trycatchandthrowstatements2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
void MyFunc( void );
class CTest {
public:
  CTest() {};
  ~CTest() {};
  const char *ShowReason() const {
    return "Exception in CTest class.";
  }
};

class CDtorDemo {
public:
  CDtorDemo();
  ~CDtorDemo();
};

CDtorDemo::CDtorDemo() {
```

```
  cout << "Constructing CDtorDemo.\n";
}

CDtorDemo::~CDtorDemo() {
  cout << "Destructing CDtorDemo.\n";
}

void MyFunc() {
  CDtorDemo D;
  cout<< "In MyFunc(). Throwing CTest exception.\n";
  throw CTest();
}

int main() {
  cout << "In main.\n";
  try {
     cout << "In try block, calling MyFunc().\n";
     MyFunc();
  }
  catch( CTest E ) {
     cout << "In catch handler.\n";
     cout << "Caught CTest exception type: ";
     cout << E.ShowReason() << "\n";
  }
  catch( char *str )    {
     cout << "Caught some other exception: " << str << "\n";
  }
  cout << "Back in main. Execution resumes here.\n";
}
```

```
In main.
In try block, calling MyFunc().
Constructing CDtorDemo.
In MyFunc(). Throwing CTest exception.
Destructing CDtorDemo.
In catch handler.
Caught CTest exception type: Exception in CTest class.
Back in main. Execution resumes here.
```

Comments

Note that in this example, the exception parameter (the argument to the catch clause) is declared in both catch handlers:

```
catch( CTest E )
// ...
catch( char *str )
// ...
```

You do not need to declare this parameter; in many cases it may be sufficient to notify the handler that a particular type of exception has occurred. However, if you do not declare an exception object in the exception-declaration, you will not have access to that object in the **catch** handler clause.

A throw-expression with no operand re-throws the exception currently being handled. Such an expression should appear only in a **catch** handler or in a function called from within a **catch** handler. The re-thrown exception object is the original exception object (not a copy). For example:

```
try {
  throw CSomeOtherException();
}
catch(...) {  // Handle all exceptions
  // Respond (perhaps only partially) to exception
  throw;       // Pass exception to some other handler  }
```

UNIT-II

Function Overloading:-

In function overloading, the function is said to be overloaded when same name is given to different functions. However, the functions will differ at least in any one of these. The number of parameters, the data type of parameters, the order of appearance these three together are referred to as the function signature. While overloading a function, the return types of the function need not differ.

1. Functions differ in function signature.
2. Return types of functions need not differ.

```
#include<conio.h>
```

```
#include<iostream.h>


class arithmetic
{
public:
 void calc(int num1)
{
Cout<<"\n\n Square of a given number:"<<num1*num1<<endl;
}
Void calc(int num1,int num2)
{
 Cout<<"\n\nMultiplication of given number is: "<<num1*num2<<endl;
}
};
Void main()
{
Clrscr();
Arithmetic a;
a.calc(5);
a.calc(6,7);
getch();
}
```

Output :

Square of the given number is : 25

Multiplication of given number is: 42

The code depicts function overloading. There are two functions with the same name calc. In the main function, when the function calc is invoked using the object a, depending up on the type and number of parameters, the compiler binds the call to the function. Hence, when calc(5) is called, the compiler checks for the function matching the parameter type. So calc(int num l) will be invoked and parameter will be passed to the function at runtime and output displayed. Similarly, when calc(6,7) is called, it looks for the same function with two integers as parameter and bind the respective function to the call.

Operator Overloading

Operating overloading allows you to pass different variable types to the same function and produce different results. In this article Ben gives us the low-down on operator overloading in C++ Operator overloading is common-place among many efficient C++ programmers. It allows you to use the same function name, but as different functions.

If this sounds confusing, then just think about it like this: you can use the same function name for as many functions as you like, but you *must* pass different variable types to each function.

In this article I will show you exactly what function overloading is, and how you can get it to work for you in C++. You should have an intermediate knowlede of C++. Any compiler will do, as I will only use ISO-standard compliant syntax.

Operator Overloading - Definition

This can be a weird subject for some, especially those with a strong Java background, or another

language that doesn't support this feature. It can be confusing even for excellent programmers. But it is a strong feature of C++ that, if mastered, can yield some increased productivity in programming.

We all know that an operator can be used in mathematical expressions:

```
int z=x+y;
float g=3.14*g;
```

Now wouldn't it be nice to use operators on our own objects to do what we want? For example, a string class could use + to concatenate, or a Throttle class could use the ++ and -- operators to increase or decrease throttle position. The operators can be programmed to do whatever we want them to.

However, some words of caution. Operator overloading provides NO additional functionality to your code. It just compiles to normal function calls. It's even written out like normal function calls. It is mainly for aesthetics. There is, however, one extremely useful set of operators to overload that makes life much easier: the streaming operators, which I will cover at the end.

Second, you should NOT use operator overloading for unobvious relationships. Using + to concatenate two strings intuitively makes sense to most programmers, so it's easy to use it like that. But how would you define string1*string2? or string1^string2? It isn't very clear what that means. So use caution when considering adding operators to your objects.

Sample Object

For my sample object, I'm going to implement a matrix. This won't be a full-scale implementation of every imaginable matrix operation, but it should be enough to cover the basics of operator overloading, and maybe whet your appetite to complete the implementation for other operations (dot product, inverse, determinant, etc.).

In order to completely encapsulate a matrix within a class, we actually need two classes: Row and Matrix.

So let's start with Row:

```
template<class T>
class Row {
public:
```

```
Row(int cols=0):row(NULL) {SetRowSize(cols);}
~Row() {SetRowSize(0); }
Row(const Row &r):row(NULL) {
SetRowSize(r.numCols);
for (int i=0;i<numCols;i++)
row[i]=r.row[i];
}

void SetRowSize(int n) {
if(row) delete[] row;
if (n>0) {
row=new T[n];
memset(row,0,sizeof(T)*n/sizeof(char));
}
else row=NULL;
numCols=n;
}

int size() { return numCols;}
private:
int numCols;
T* row;
};
```

Let's look at this before continuing on. Notice that I'm making it a template class. This is so you can have a matrix of all the usual numerical types, as well as any type you want to define yourself. The only requirement for the type is that it must have the +, -, and * operators defined on it. We'll get into how to do that. If you don't understand templates, you can think of all of the T's as ints for now.

SetRowSize() deletes any old data, and allocates space for new data, unless we set the number of columns to 0, in which case it merely deletes the data. This lets us use this function for construction, destruction, and dynamic modification in one method. Nifty, eh? The call to memset() just zeroes out the array after figuring out how many bytes the row uses and dividing this by the size of character, because memset() works in terms of chars.

I also defined a copy constructor, which will come in handy quite a bit, as we'll see later on when we copy matrices.

Overloading[]

OK, let's overload our first operator: []

Yes, that's one operator. The array-access operator. It makes perfect sense here, because we have a linear array of objects we would like to access. Let's add this definition to our Row class:

```
T& operator[](int column) {
assert(column<numCols);
return row[column];
}
```

The arguments to our brackets are going to be integers specifying the index of the item we want, so that will be the function's arguments. Notice the syntax: [ReturnType] operator[Op]([argument list]). We do an assertion to make sure we're accessing memory within the array's bounds. If all is OK, we return a reference to the object. Why a reference instead of a value? It won't make much of a difference in a case like this:

```
Row<int> r(1);//1x1 matrix

int a=r[0];
```

a will get the value of r[0] whether a reference or a value is returned. However, if we return a reference, we can then change the value in the row from outside the class, using the [] accessor operator, like so:

```
Row<float> r(1);

r[0]=3.142;
float pi=r[0];
```

Operator Overloading in C++ Overloading =

The only other operator we need to overload is assignment (=). When overloading assignment, we must keep in mind that the object we're assigning to must already exist, and it is that object's operator= method which will be called.

```
Row& operator=(const Row& r) {
SetRowSize(r.numCols);
for (int i=0;i<numCols;i++)
```

```
row[i]=r.row[i];
return *this;
}
```

Again we return a reference, but this time it's a reference to itself. First we set the size of the current row equal to that of the source row, then we copy its values. There is an important note here. Notice that I'm using [] on the primitive T array itself--NOT the overloaded []s of Row. Remember that Row's [] returns a reference, thus if we had written row[i]=r[i], we would get a row that references the exact same data in memory, so that when we changed one the other would change--this isn't what we want at all, so we need to access the raw data in the Row class.

Now we can write code like this:

```
Row<double> r1(5);
Row<double> r2;//creates an empty row
Row<double> r3(2);
r2=r1;
r3=r1;//overwrites previous row information to contain same info as r1
```

Matrices are Made of Many Rows

Now that we have a working Row, we can combine rows into a matrix. Let's start with this basic definition:

```
template<class T>
class Matrix {
public:
Matrix(int rows=0, int cols=0): matrix(NULL) {
SetSize(rows,cols);
}
Matrix(const Matrix& m): matrix(NULL) {
SetSize(m.numRows,m.numCols);
for (int r=0;r<numRows;r++)
matrix[r]=Row<T>(m.matrix[r]);//assign to primitive array, NOT overloaded []--to get a copy
}
void SetSize(int rows, int cols) {
if (rows) delete[]matrix;
if (cols > 0 && rows >0) {
matrix=new Row<T>[rows];
for (int i=0;i<rows;i++)
matrix[i].SetRowSize(cols);
```

```
}
else
rows=NULL;
numCols=cols;numRows=rows;
}
int GetCols() { return numCols;}
int GetRows() { return numRows;}

private:
int numCols, numRows;
Row<T>* matrix;

};
```

This follows very closely the basic form of the Row class. The only item of interest is when we declare and allocate a matrix: we must specify the type, T, after the class name.

First let's implement the same operators we did on the Row class:

```
Row<T>& operator[](int index) {
assert(index<numRows);
return matrix[index];
}

Matrix& operator=(const Matrix& m) {
SetSize(m.numRows,m.numCols);
for (int r=0;r<numRows;r++)
matrix[r]=Row(m.matrix[r]);//assign to primitive array, NOT overloaded []--to get a copy
return *this;
}
```

The most important part of this code is the return type of operator[]. It returns a reference to a Row of type T. This little fact allows us to use the Matrix class like this:

```
Matrix<int> a(2,2);

a[0][0]=2;
a[0][1]=4;
a[1][0]=8;
a[1][1]=16;
```

That is, we can refer to Matrix objects now with exactly the same notation as primitive 2-D arrays in C++: array[row][column]. Our operator overloading is faking it well enough to keep a consistent interface with analogous structures, but add much more functionality and safety. Isn't this cool?

The = operator works the same way as in Row. It sets the size of the current Matrix to that of the source, and then copies all of the objects to the current Matrix. Now we can do the following:

```
Matrix<__int64> m(1000,1000);
Matrix<__int64> n=m;
```

Operator Overloading in C++

Let's do some more interesting things with these matrices now. There are a number of mathematical operations that can be performed on a matrix, the simplest perhaps is addition. Addition of matrices requires that they both have the same dimensions. The resulting matrix is made by simply adding each number in the same position in each matrix and putting the answer in the same position as the two operands.

```
[1 0] [4 3] [5 3]
[2 1] + [-1 0] = [1 1]
```

Since addition creates a new matrix, we don't want to return a reference, but an actual matrix object. Here's what the code looks like:

```
const Matrix operator+( const Matrix& m) {
assert(numCols==m.numCols && numRows==m.numRows);
Matrix theMatrix(numRows,numCols);
for (int r=0;r<numRows;r++)
for (int c=0;c<numCols;c++)
theMatrix[r][c]=matrix[r][c]+m.matrix[r][c];
return theMatrix;
}
```

This adds the current matrix to the matrix in argument m. We first assure that the dimensions are equivalent, then create a new matrix with the same dimensions as the sources. It is then a simple

matter of adding the two sources, and returning the new matrix. Notice that we perform the actual math on the types that make up each row.

```
Matrix<float> a(2,2);
Matrix<float> b(2,2);

Matrix<float> c(2,3);

Matrix<float> d=a+b;

Matrix<float> e=a+c;//will fail assertion, abort program
```

It is just as easy to define subtraction:

```
const Matrix operator-( const Matrix& m) {
assert(numCols==m.numCols && numRows==m.numRows);
Matrix theMatrix(numRows,numCols);
for (int r=0;r<numRows;r++)
for (int c=0;c<numCols;c++)
theMatrix[r][c]=matrix[r][c]-m.matrix[r][c];
return theMatrix;
}
```

Overloading += and -=

+= and -= are operators that both add and change the current object, so the code to describe it is a combination of +/- and =. We'll return a reference again because we don't want to create a new object, but just modify the existing one, which called the function. We'll just add whatever is currently in it to the other matrix, and return a reference to itself:

```
Matrix& operator+=(const Matrix& m) {
assert(numCols==m.numCols && numRows==m.numRows);
for (int r=0;r<numRows;r++)
for (int c=0;c<numCols;c++)
matrix[r][c]+=m.matrix[r][c];
return *this;
}

Matrix& operator-=( const Matrix& m) {
assert(numCols==m.numCols && numRows==m.numRows);
for (int r=0;r<numRows;r++)
```

```
for (int c=0;c<numCols;c++)
matrix[r][c]-=m.matrix[r][c];
return *this;
}
```

We can now expand our repertoire to include the following possibilities:

```
Matrix<int> a(2,1);
Matrix<int> b(2,1);

a+=b;
a-=b;
```

CLASS TEMPLATES

C++ Class Templates are used where we have multiple copies of code for different data types with the same logic. If a set of functions or classes have the same functionality for different data types, they becomes good candidates for being written as Templates.

One good area where this C++ Class Templates are suited can be container classes. Very famous examples for these container classes will be the STL classes like vector, list etc., Once code is written as a C++ class template, it can support all data types. Though very useful, It is advisable to write a class as a template after getting a good hands-on experience on the logic (by writing the code with normal data types). There are cases where we need specialization for writing optimized code for specific data types. This C++ class template Specialization article gives a brief description.

This article describes how to declare, define and use the C++ Class Templates in practice. This tries to build a very preliminary Queue, using the STL::Vector container class. This code is written and tested with Microsoft Visual C++ 5.00.

Declaring C++ Class Templates:

Declaration of C++ class template should start with the keyword *template*. A parameter should be included inside angular brackets. The parameter inside the angular brackets, can be either the keyword *class* or *typename*. This is followed by the class body declaration with the member data and member functions. The following is the declaration for a sample Queue class.

```
//Sample code snippet for C++ Class Template
template <typename T>
class MyQueue
{
        std::vector<T> data;
    public:
```

```
        void Add(T const &d);
        void Remove();
        void Print();
};
```

The keyword class highlighted in blue color, is not related to the *typename*. This is a mandatory keyword to be included for declaring a template class.

Defining member functions - C++ Class Templates:

If the functions are defined outside the template class body, they should always be defined with the full template definition. Other conventions of writing the function in C++ class templates are the same as writing normal c++ functions.

```
template <typename T> void MyQueue<T> ::Add(T const &d)
{
    data.push_back(d);
}

template <typename T> void MyQueue<T>::Remove()
{
     data.erase(data.begin( ) + 0,data.begin( ) + 1);
}

template <typename T> void MyQueue<T>::Print()
{
    std::vector <int>::iterator It1;
    It1 = data.begin();
    for ( It1 = data.begin( ) ; It1 != data.end( ) ; It1++ )
        cout << " " << *It1<<endl;

}
```

The Add function adds the data to the end of the vector. The remove function removes the first element. These functionalities make this C++ class Template behave like a normal Queue. The print function prints all the data using the iterator.

Full Program - C++ Class Templates:

```
//C++_Class_Templates.cpp

#include <iostream.h>
#include <vector>

template <typename T>
```

```
class MyQueue
{
    std::vector<T> data;
  public:
    void Add(T const &);
    void Remove();
    void Print();
};

template <typename T> void MyQueue<T> ::Add(T const &d)
{
    data.push_back(d);
}

template <typename T> void MyQueue<T>::Remove()
{
    data.erase(data.begin( ) + 0,data.begin( ) + 1);
}

template <typename T> void MyQueue<T>::Print()
{
    std::vector <int>::iterator It1;
    It1 = data.begin();
    for ( It1 = data.begin( ) ; It1 != data.end( ) ; It1++ )
      cout << " " << *It1<<endl;

}
//Usage for C++ class templates
void main()
{
    MyQueue<int> q;
    q.Add(1);
    q.Add(2);

    cout<<"Before removing data"<<endl;
    q.Print();

    q.Remove();
    cout<<"After removing data"<<endl;
    q.Print();
}
```

Advantages of C++ Class Templates:

- One C++ Class Template can handle different types of parameters.

- Compiler generates classes for only the used types. If the template is instantiated for int type, compiler generates only an int version for the c++ template class.
- Templates reduce the effort on coding for different data types to a single set of code.
- Testing and debugging efforts are reduced.

FUNCTION TEMPLATES

C++ Function templates are those functions which can handle different data types without separate code for each of them. For a similar operation on several kinds of data types, a programmer need not write different versions by overloading a function. It is enough if he writes a C++ template based function. This will take care of all the data types.

There are two types of templates in C++, viz., function templates and class templates. This article deals with only the function templates.

C++ templates come to our rescue in such situations. When we use C++ function templates, only one function signature needs to be created. The C++ compiler will automatically generate the required functions for handling the individual data types. This is how a programmer's life is made a lot easier.

C++ Template functions - Details:

Let us assume a small example for Add function. If the requirement is to use this Add function for both integer and float, then two functions are to be created for each of the data type (overloading).

```
int Add(int a,int b) { return a+b;} // function Without C++ template
float Add(float a, float b) { return a+b;} // function Without C++ template
```

If there are some more data types to be handled, more functions should be added.

But if we use a c++ function template, the whole process is reduced to a single c++ function template. The following will be the code fragment for Add function.

```
template <class T>
T Add(T a, T b) //C++ function template sample
{
  return a+b;
}
```

This c++ function template definition will be enough. Now when the integer version of the function, the compiler generates an Add function compatible for integer data type and if float is called it generates float type and so on.

Here T is the typename. This is dynamically determined by the compiler according to the parameter passed. The keyword *class* means, the parameter can be of any type. It can even be a class.

C++ Template functions - Applicability:

C++ function templates can be used wherever the same functionality has to be performed with a number of data types. Though very useful, lots of care should be taken to test the C++ template functions during development. A well written c++ template will go a long way in saving time for programmers.

Abstract classes

An *abstract class* is a class that is designed to be specifically used as a base class. An abstract class contains at least one *pure virtual function*. You declare a pure virtual function by using a *pure specifier* (= 0) in the declaration of a virtual member function in the class declaration.

The following is an example of an abstract class:

```
class AB {
public:
  virtual void f() = 0;
};
```

Function AB::f is a pure virtual function. A function declaration cannot have both a pure specifier and a definition. For example, the compiler will not allow the following:

```
struct A {
  virtual void g() { } = 0;
};
```

You cannot use an abstract class as a parameter type, a function return type, or the type of an explicit conversion, nor can you declare an object of an abstract class. You can, however, declare pointers and references to an abstract class. The following example demonstrates this:

```
struct A {
  virtual void f() = 0;
};

struct B : A {
  virtual void f() { }
};
```

```
// Error:
// Class A is an abstract class
// A g();

// Error:
// Class A is an abstract class
// void h(A);
A& i(A&);

int main() {

// Error:
// Class A is an abstract class
//   A a;

  A* pa;
  B b;

// Error:
// Class A is an abstract class
//   static_cast<A>(b);
}
```

Class A is an abstract class. The compiler would not allow the function declarations A g() or void h(A), declaration of object a, nor the static cast of b to type A.

Virtual member functions are inherited. A class derived from an abstract base class will also be abstract unless you override each pure virtual function in the derived class.

For example:

```
class AB {
public:
  virtual void f() = 0;
};

class D2 : public AB {
  void g();
};

int main() {
  D2 d;
}
```

The compiler will not allow the declaration of object d because D2 is an abstract class; it inherited the pure virtual function f()from AB. The compiler will allow the declaration of object d if you define function D2::g().

Note that you can derive an abstract class from a nonabstract class, and you can override a non-pure virtual function with a pure virtual function.

You can call member functions from a constructor or destructor of an abstract class. However, the results of calling (directly or indirectly) a pure virtual function from its constructor are undefined. The following example demonstrates this:

```
struct A {
  A() {
    direct();
    indirect();
  }
  virtual void direct() = 0;
  virtual void indirect() { direct(); }
};
```

The default constructor of A calls the pure virtual function direct() both directly and indirectly (through indirect()).

The compiler issues a warning for the direct call to the pure virtual function, but not for the indirect call.

Inheritance

- New classes created from existing classes
- Absorb attributes and behaviors
- Derived class
  - Class that inherits data members and member functions from a previously defined base class
- Single inheritance
  - Class inherits from one base class
- Multiple inheritance
  - Class inherits from multiple base classes
- Types of inheritance
  - public: private: protected:

Inheritance: Base and Derived Classes

- Base and derived classes
  - Often an object from a derived class (subclass) is also an object of a base class (superclass)
    - A rectangle is a derived class in reference to a quadrilateral and a base class in reference to a square
- Inheritance examples

Base class	Derived classes
`Student`	`GraduateStudent` `UndergraduateStudent`

`Shape`	`Circle`
	`Triangle`
	`Rectangle`

`Loan`	`CarLoan`
	`HomeImprovementLoan`
	`MortgageLoan`

`Employee`	`FacultyMember` `StaffMember`
`Account`	`CheckingAccount` `SavingsAccount`

Base and Derived Classes

- Implementation of public inheritance
  class CommissionWorker : public Employee {

...

};

  - Class CommissionWorker inherits from class Employee
  - friend functions not inherited
  - private members of base class not accessible from derived class

- protected access
  - Intermediate level of protection between public and private inheritance
  - Derived-class members can refer to public and protected members of the base class simply by using the member names
  - Note that protected data "breaks" encapsulation

- Derived class member functions
  - Cannot directly access **private** members of their base class

    - Maintains encapsulation
  - Hiding **private** members is a huge help in testing, debugging and correctly modifying systems

Base class member access specifier	**Type of inheritance**		
	`public` inheritance	**`protected`** inheritance	**`private`** inheritance
`Public`	**`public`** in derived class. Can be accessed directly by any non-**`static`** member functions, **`friend`** functions and non-member functions.	**`protected`** in derived class. Can be accessed directly by all non-**`static`** member functions and **`friend`** functions.	**`private`** in derived class. Can be accessed directly by all non-**`static`** member functions and **`friend`** functions.
`Protected`	**`protected`** in derived class. Can be accessed directly by all non-**`static`** member functions and **`friend`** functions.	**`protected`** in derived class. Can be accessed directly by all non-**`static`** member functions and **`friend`** functions.	**`private`** in derived class. Can be accessed directly by all non-**`static`** member functions and **`friend`** functions.
`Private`	Hidden in derived class. Can be accessed by non-**`static`** member functions and **`friend`** functions through **`public`** or **`protected`** member functions of the base class.	Hidden in derived class. Can be accessed by non-**`static`** member functions and **`friend`** functions through **`public`** or **`protected`** member functions of the base class.	Hidden in derived class. Can be accessed by non-**`static`** member functions and **`friend`** functions through **`public`** or **`protected`** member functions of the base class.

Overriding Base-Class Members in a Derived Class

- To override a base-class member function
  - In the derived class, supply a new version of that function with the same signature
    - same function name, different definition
  - When the function is then mentioned by name in the derived class, the derived version is automatically called
  - The scope-resolution operator may be used to access the base class version from the derived class

public, private, and protected Inheritance

Direct and Indirect Base Classes

- Direct base class
  - Explicitly listed derived class's header with the colon (**:**) notation when that derived class is declared

    class HourlyWorker : public Employee

    - **Employee** is a direct base class of **HourlyWorker**
- Indirect base class
  - Not listed in derived class's header
  - Inherited from two or more levels up the class hierarchy

    class MinuteWorker : public HourlyWorker

    - **Employee** is an indirect base class of **MinuteWorker**

Using Constructors and Destructors in Derived Classes

- Base class initializer
  - Uses member-initializer syntax
  - Can be provided in the derived class constructor to call the base-class constructor explicitly
    - Otherwise base class's default constructor called implicitly

  - Base-class constructors and base-class assignment operators are not inherited by derived classes
    - Derived-class constructors and assignment operators, however, can call base-class constructors and assignment operators

- A derived-class constructor
  - Calls the constructor for its base class first to initialize its base-class members
  - If the derived-class constructor is omitted, its default constructor calls the base-class' default constructor
- Destructors are called in the reverse order of constructor calls
  - So a derived-class destructor is called before its base-class destructor

Multiple Inheritance

- Multiple Inheritance
  - Derived-class inherits from multiple base-classes
  - Encourages software reuse, but can create ambiguities

Dynamic Polymorphism

Objectives

* Implement the concept of binding
* Use virtual functions
* Use pure virtual functions to create abstract classes
* Implement dynamic polymorphism by using late binding

Dynamic Polymorphism

* Refers to any entity changing its form, depending on circumstances

Binding

* Is the process of associating a function with a class by identifying the type of the object or pointer that is used to invoke the function

Dynamic Binding

* Is done during runtime
* Is also called late binding

Virtual Function

* Is a function that is declared as virtual in a base class and is redefined by a derived class

Using Virtual Functions

Example:

```
class Employee
{
.
.
virtual int calc_net_salary();
.
.
};
class Contract:public Employee
{
.
.
int calc_net_salary();
.
.
.
};
class Direct_Contract: public Contract
```

```
{
.
.
int calc_net_salary();
.
.
 };
```

Pure Virtual Function

* Is a function without a body
* Is created by adding the notation '=0' to the virtual function declaration

Example:

```
virtual int calc_net_salary()=0;
```

Abstract Class

* Is a class containing one or more pure virtual functions
* Is used as a base class for deriving specific classes of the same kind

Static vs Dynamic Polymorphism (Contd.)

* Dynamic polymorphism
  Is considered more flexible

  Is based on overriding principles, which, therefore, is purely class scope and is based on inheritance

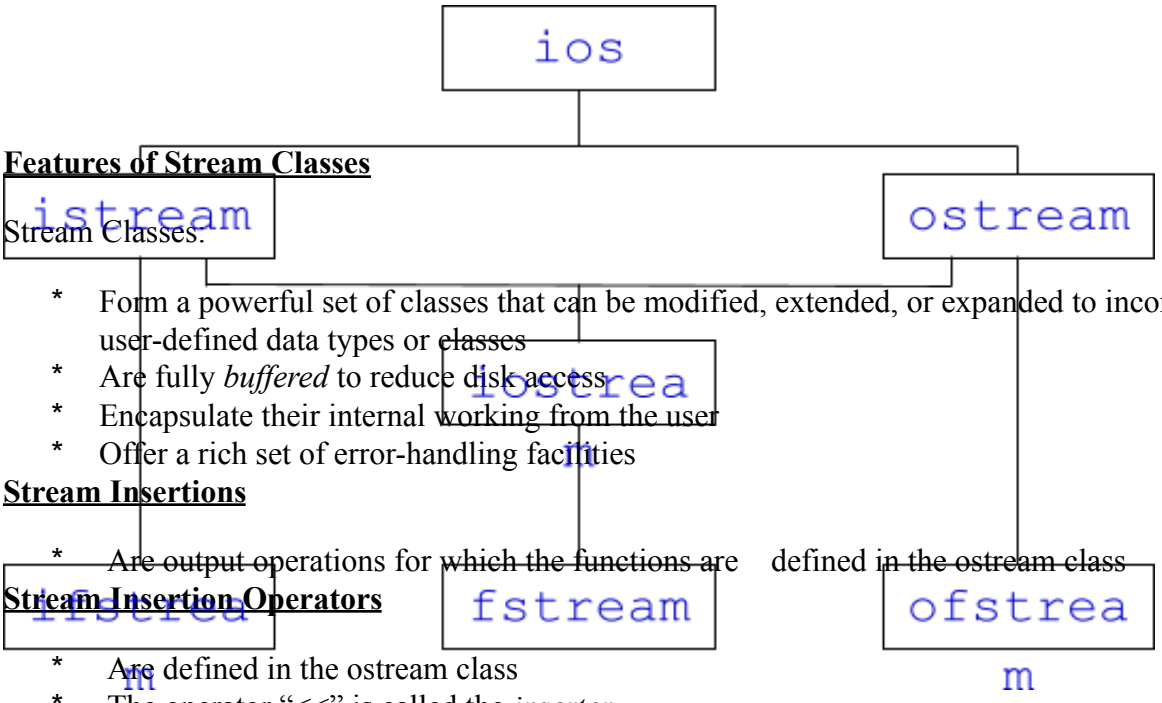
Streams I/O:-

Objectives

In this lesson, you will learn to:

* Define the stream class hierarchy
* Identify the stream insertion and extraction operators
* Use the stream classes for file input and output
* Differentiate between text and binary file input and output
* Apply the following functions for opening and closing files:
  - open()
  - close()
* Use the open mode bits
* Randomly access data files

Stream Class Hierarchy :-



Features of Stream Classes

Stream Classes:

* Form a powerful set of classes that can be modified, extended, or expanded to incorporate user-defined data types or classes
* Are fully *buffered* to reduce disk access
* Encapsulate their internal working from the user
* Offer a rich set of error-handling facilities

Stream Insertions

* Are output operations for which the functions are defined in the ostream class

Stream Insertion Operators

* Are defined in the ostream class
* The operator "<<" is called the *inserter*

Stream Extractions

* Are input operations for which the functions are defined in the istream class

Stream Extraction Operators

* Are defined in the istream class and are used to receive data from the input device
* The operator ">>", called the *extractor*, accepts any built-in data type passed as arguments

The get() and getline() Functions

* Are used to read a complete string from the input stream
* Recognize white spaces
* The get() function

  Syntax:

  cin.get(char *str, int len, char delim = '\n');

* The getline() function

  Syntax:

  cin.getline(char *str, int len, char delim = '\n');

File Input and Output Using Built-in Data Types

* Integer Input and Output

  Example:

```
#include <fstream>

int main()

{

ofstream outobj("INT.TST");

outobj << 25 << ' ' << 4567 << ' ' << 8910;

return 0;

}
```

* Character input and output

  Example:

```
#include <fstream>

        int main()

        {

ofstream out("STR.TST");
```

```
        out << "This is a test string";
return 0;
                }
```

File Input and Output Using Objects

Example:

```
#include<fstream.h>
class student
        {
private:
                int iReg_no;char cName[20];
public:
                void setRegno();
                void setName();
                int getRegno();
                char *getName();
        };
void main()
{
                ofstream Sfil("studfile.dat");
                char ch;
                student Svar;
                Svar.setRegno();
                Svar.setName();
                Sfil<<Svar.getRegno()
                        <<" "<<Svar.getName();
```

```
        Sfil.close(); //Closes the open file
cout<< "\n Do you want to view the                    contents of a file (y/n)?";
        cin>>ch;
        if(ch== 'y')
        {
            ifstream Sfil("studfile.dat");
            char ireg;char nam[20];
            Sfil>>ireg>>nam;
            cout<<"\n Registration Number is "          <<ireg;
            cout<<"\n Student Name is " <<nam;
        }
    }
```

Binary Input and Output (Contd.)

- 3 The write function
  Syntax:

```
write(char* addr, int size)
```

* File input and output using abstract data types
  - 3 read() and write() functions are used to read or write user-defined objects on a file

The open() Function

* Is used to open a file
  Example:

```
ifstream Ifil;   //creates an                    //unopened input stream
Ifil.open("DATA.DAT");      //associates              //the stream to a file
```

The close() Function

* Is used to close a file
  Example:

```
ofstream Ofil;

Ofil.open("DATA.DAT");

...

...

Ofil.close();
```

Open Mode Bits

* Are defined in the ios class
* Are bits that are associated with the opening of files
* Represent the mode in which the file is opened

Mode	Explanation
app	Starts reading or writing at the end of the file. Does not have a meaning with input streams, it is used only for output streams
in	Open for reading
out	Open for writing
ate	Seek to the end of the file
trunc	If the file exists, it is truncated, i.e. all data is erased before writing/reading

The get Pointer

* Specifies the location in the file where the next read operation will occur

The put Pointer

* Specifies the location in the file where the next write operation will occur

The seekg() Function

* Helps to control the get pointer
* Moves the get pointer to an absolute address within the file or to a certain number of bytes from a particular position
* Takes two arguments:
  - The number of bytes to move
  - The reference in the file from where the pointer has to be repositioned

Example:

```
ifstream iFil;
```

```
iFil.seekg(10,ios::beg);
```

The tellg() Function

* Helps to control the get pointer
* Can be used to find the current position of the get file pointer in a file
* Does not take any arguments

Example:

```
int iPosition=iFil.tellg();
```

The seekp() Function

* Helps to control the put pointer
* Moves the put pointer to an absolute address within the file or to a certain number of bytes from a particular position

The tellp() Function

* Helps to control the put pointer
* Can be used to find the current position of the put file pointer in a file

UNIT-III

Algorithm

An algorithm is a sequence of unambiguous instructions for solving a problem

- Obtaining a required output for any legitimate input in a finite amount of time
- Procedural solutions to problems.

Computer is capable of understanding and following the instructions given.

An input specifies an instance of the problem the algorithm solves.

Motivation for Algorithm

- Theoretical importance
  - The study of algorithms represents the core of computer science.
- Practical importance
  - A practitioner's toolkit of known algorithms
  - Framework for designing and analyzing algorithms for new problems.

Asymptotic Notation

Running time of an algorithm, order of growth Worst case

Running time of an algorith increases with the size of the input in the limit as the size of the input increases without bound.

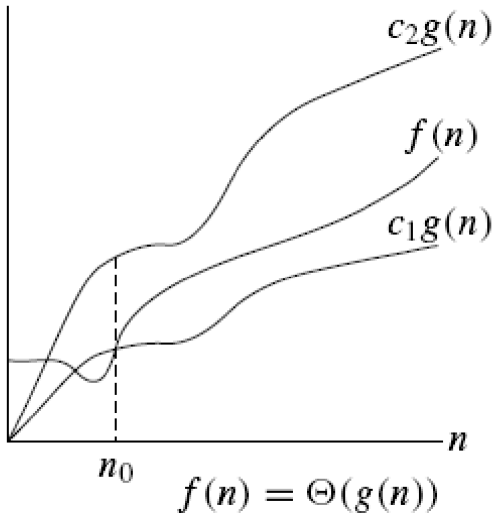
Big-theta notation

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}.^{1}$$

"$f(n) \in \Theta(g(n))$"

"$f(n) = \Theta(g(n))$"

g(n) is an asymptotically tight bound of f(n)



$f(n) = \Theta(g(n))$

(a)

Example

$$\frac{1}{2}n^2 - 3n = \Theta(n^2)$$

$$c_1 n^2 \le \frac{1}{2} n^2 - 3n \le c_2 n^2$$

for all $n \ge n_0$. Dividing by n^2 yields

$$c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2 .$$

n >= 1, c2 >= 1/2

n >= 7, c1 <= 1/14

choose c1 = 1/14, c2 = ½, n0 = 7.

O-notation

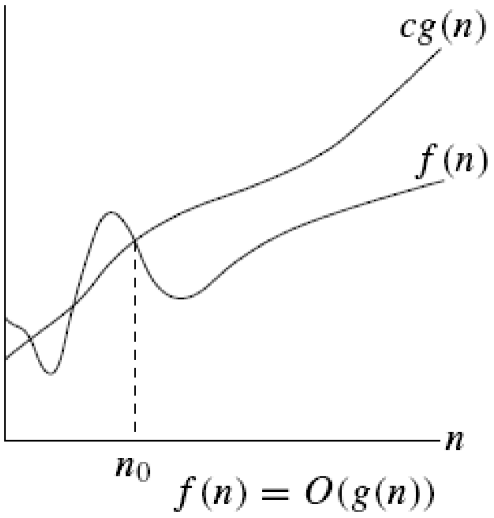
Asymptotic upper bound

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\} .$$

$$f(n) = \Theta(g(n)) \text{ implies } f(n) = O(g(n))$$

$$\Theta(g(n)) \subseteq O(g(n)).$$

f(n) = O(g(n)) some constant multiple of g(n) is an asymptotic upper bound of **f(n),** no claim about how tight an upper bound is.



Example

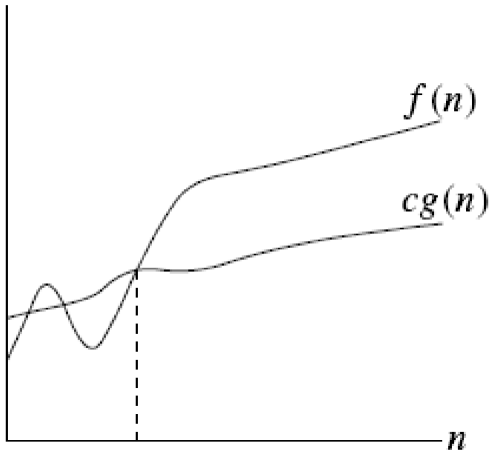
The running time is O (n2) means there is a function f(n) that is O(n2) such that for any value of n, no matter what particular input of size n is chosen, the running time of that input is bounded from above by the value f(n).

Big-Omega notation

Asymptotic lowerbound

Theorem

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$



$f(n) = \Omega(g(n))$

Theorem

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. ■

When we say that the running time (no modifier) of an algorithm is **Ω(g(n)),** we mean that no matter what particular input of size n is chosen for each value of n, the running time on that input is at least a constant times g(n), for sufficiently large n.

Interpretation

$$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$$

$$2n^2 + 3n + 1 = 2n^2 + f(n),$$

not specifying all lower-terms exactly

$$T(n) = 2T(n/2) + \Theta(n)$$

$$\sum_{i=1}^{n} O(i)$$

not the same as $O(1) + O(2) + \cdots + O(n)$

$$2n^2 + \Theta(n) = \Theta(n^2)$$

"No matter how the anonymous functions are chosen on the left of the equal sign,there is a way to choose the anonymous functions on the right of the equal sign to make the equation valid."

for any function **f (n) >= Θ(n),** there is some function **g(n) <=Θ (n2)** such that **2n2 + f (n) = g(n)** for all n.

In other words, the right-hand side of an equation provides a coarser level of detail than the left-hand side.

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2) \ . \end{aligned}$$

o-notation

$$o(g(n)) = \{f(n) : \text{ for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le f(n) < cg(n) \text{ for all } n \ge n_0\} \ .$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

w-notation

$$\omega(g(n)) = \{f(n) : \text{ for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \le cg(n) < f(n) \text{ for all } n \ge n_0\} \ .$$

$f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Analogy to comparison of two real numbers, a, b.

$$
\begin{aligned}
f(n) = O(g(n)) \quad &\approx \quad a \le b\,, \\
f(n) = \Omega(g(n)) \quad &\approx \quad a \ge b\,, \\
f(n) = \Theta(g(n)) \quad &\approx \quad a = b\,, \\
f(n) = o(g(n)) \quad &\approx \quad a < b\,, \\
f(n) = \omega(g(n)) \quad &\approx \quad a > b\,.
\end{aligned}
$$

Transitivity:

$$
\begin{aligned}
f(n) = \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) &\quad \text{imply} \quad f(n) = \Theta(h(n)) \\
f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) &\quad \text{imply} \quad f(n) = O(h(n)) \\
f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) &\quad \text{imply} \quad f(n) = \Omega(h(n)) \\
f(n) = o(g(n)) \text{ and } g(n) = o(h(n)) &\quad \text{imply} \quad f(n) = o(h(n)) \\
f(n) = \omega(g(n)) \text{ and } g(n) = \omega(h(n)) &\quad \text{imply} \quad f(n) = \omega(h(n))
\end{aligned}
$$

Reflexivity:

$$
\begin{aligned}
f(n) &= \Theta(f(n))\,, \\
f(n) &= O(f(n))\,, \\
f(n) &= \Omega(f(n))\,.
\end{aligned}
$$

Symmetry:

$$
f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n))\,.
$$

Transpose symmetry:

$$
\begin{aligned}
f(n) = O(g(n)) &\quad \text{if and only if} \quad g(n) = \Omega(f(n))\,, \\
f(n) = o(g(n)) &\quad \text{if and only if} \quad g(n) = \omega(f(n))\,.
\end{aligned}
$$

Time complexity

In computer science, the **time complexity** of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the size of the input to the problem. The time complexity of an algorithm is commonly expressed using big O notation, which suppresses multiplicative constants and lower order terms. When expressed this way, the time complexity is said to be described *asymptotically*, i.e., as the input size goes to infinity. For example, if the time required by an algorithm on all inputs of size n is at most $5n^3 + 3n$, the asymptotic time complexity is O (n^3).

Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, where an elementary operation takes a fixed amount of time to perform. Thus the amount of time taken and the number of elementary operations performed by the algorithm differ by at most a constant factor.

Since an algorithm may take a different amount of time even on inputs of the same size, the most commonly used measure of time complexity, the worst-case time complexity of an algorithm, denoted as ***T(n)***, is the maximum amount of time taken on any input of size n. Time complexities are classified by the nature of the function $T(n)$. For instance, an algorithm with $T(n) = O(n)$ is called a linear time algorithm, and an algorithm with $T(n) = O(2^n)$ is said to be an exponential time algorithm.

Table of common time complexities

The following table summarizes some classes of commonly encountered time complexities. In the table, poly(x) = $x^{O(1)}$, i.e., polynomial in x.

Name	Complexity class	Running time ($T(n)$)	Examples of running times	Examples of algorithms
constant time		$O(1)$	10	Determining if a number is even or odd
inverse Ackermann		$O(\alpha(n))$		Amortized time per operation using a disjoint set
iterated logarithmic		$O(\log^* n)$		Distributed coloring of cycles
log-logarithmic		$O(\log \log n)$		Amortized time per operation using a bounded priority queue[1]

logarithmic time	DLOGTIME	$O(\log n)$	$\log n$, $\log(n^2)$	Binary search
polylogarithmic time		$\text{poly}(\log n)$	$(\log n)^2$	
fractional power		$O(n^c)$, $0<c<1$	$n^{1/2}$, $n^{2/3}$	Searching in a kd-tree
linear time		$O(n)$	n	Finding the smallest item in an unsorted array
"n log star n"		$O(n \log^* n)$		Seidel's polygon triangulation algorithm.
linearithmic time		$O(n \log n)$	$n \log n$, $\log n!$	Fastest possible comparison sort
quadratic time		$O(n^2)$	n^2	Bubble sort; Insertion sort
cubic time		$O(n^3)$	n^3	Naive multiplication of two $n \times n$ matrices. Calculating partial correlation.
polynomial time	P	$2^{O(\log n)} = \text{poly}(n)$	n, $n \log n$, n^{10}	Karmarkar's algorithm for linear programming; AKS primality test
quasi-polynomial time	QP	$2^{\text{poly}(\log n)}$	$n^{\log \log n}$, $n^{\log n}$	Best-known $O(\log^2 n)$-approximation algorithm for the directed Steiner tree problem.
sub-exponential time (first definition)	SUBEXP	$O(2^{n^\varepsilon})$ for all $\varepsilon > 0$	$O(2^{\log n \log \log n})$	Assuming complexity theoretic conjectures, BPP is contained in SUBEXP.[2]
sub-exponential time (second definition)		$2^{o(n)}$	$2^{n1/3}$	Best-known algorithm for integer factorization and graph isomorphism
exponential time	E	$2^{O(n)}$	1.1^n, 10^n	Solving the traveling salesman problem using dynamic programming
exponential time	EXPTIME	$2^{\text{poly}(n)}$	$n!$, n^n, 2^{n2}	

factorial time		$O(n!)$	$n!$	Solving the traveling salesman problem via brute-force search
double exponential time	2-EXPTIME	$2^{2^{poly(n)}}$	2^{3n}	Deciding the truth of a given statement in Presburger arithmetic

Space Complexity

Space Complexity S (P) =C+SP (I)

- ***Fixed Space Requirements (C)***
  Independent of the characteristics of the inputs and outputs
  - **Instruction space**
  - **Space for simple variables, fixed-size structured variable, constants**

- ***Variable Space Requirements (SP (I))***
  depend on the instance characteristic I
  - **Number, size, values of inputs and outputs associated with I**
  - **Recursive stack space, formal parameters, local variables, return address**

Program 1.9: Simple arithmetic function

```
float abc(float a, float b, float c)
{
   return a + b + b * c + (a + b - c) / (a + b) + 4.00;
}
```

$S_{abc}(I) = 0$

Program 1.10: Iterative function for summing a list of numbers (p.20)

```
float sum(float list[ ], int n)
{
 float tempsum = 0;
 int i;
 for (i = 0; i<n; i++)
    tempsum += list [i];
 return tempsum;
}
```

$S_{sum}(I) = 0$

Recall: pass the address of the first element of the array and pass by value

Program 1.11: Recursive function for summing a list of numbers (p.20)

```
float rsum(float list[ ], int n)
```

```
{
  if (n) return rsum(list, n-1) + list[n-1];
  return 0;
}
```

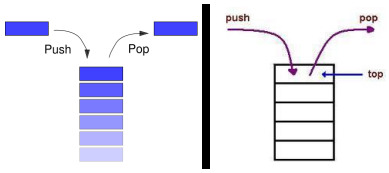
$S_{sum}(I)=S_{sum}(n)=6n$

***Figure 1.1: Space needed for one recursive call of Program 1.11**

Type	Name	Number of bytes
parameter: floatparameter: integerreturn address:(used internally)	list [ ] n	2 2 2(unless a far address)
TOTAL per recursive call		6

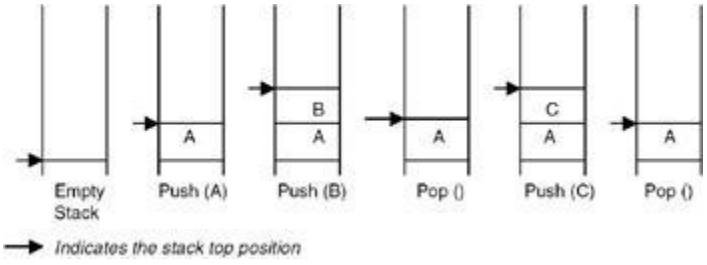
STACKS

Definition: A stack is an ordered set of elements in which additions and deletions are should be taken place from only one end this end is called top.. That means that it is possible to remove elements from a stack in reverse order from the insertion of elements into the stack. Thus, a stack data structure exhibits the LIFO (last in first out) property. Push and pop are the operations that are provided for insertion of an element into the stack and the removal of an element from the stack, respectively. Shown in Figure 19.1 are the effects of push and pop operations on the stack.



- Insertion of an element in to the stack is called “push ()” operation
- Deletion of an element from the stack is called “pop ()” operation
- Displaying of elements in stack called “display()” operation

Stack is follows the LIFO method (Last in first out) notation



It is an example for linear linked list its follows the two conditions while performing the operations

1) Underflow condition

2) Overflow condition

- Overflow condition occurs when the stack is full

If(top>=n)

- Underflow condition occurs when the stack is empty

If(top==-1)

Stack can be represented in two ways

1 array representation (sequential)

2 liner list representation (pointer)

Array representation of stack:

Push() operation code:

```
Int x;
Cout<<" enter values";
Cin>>x;
top++;
If(top>=n)
{
Cout<<"stack is overflow";
return;
```

```
}
S[top]=x;
```

Pop() operation code:

```
Int x;
If(top==-1)
{
  Cout<<"stack is underflow";
 Return;
}
X=s[top];
Top--;
```

Display() operation code:

```
If(top==-1)
 {
Cout<<"stack is empty";
Return;
}
 For(i=top;i>=0;i--)
 Cout<<s[i];
}
```

Stacks are commonly used Data Structures while writing code. It's concept is really simple which makes it even simpler to write it in code. Consider this situation. There are **a pile of 5 Books** on a Table. You want to add one book to the pile. What do you do??? You simply add the book on the TOP of the pile. What if you want the third book from the new 6 book pile? You then lift each book one by one from the TOP until the third book reaches the top. Then you take

the third book and replace all the othe  rs back into the pile by adding them from the TOP.

If you've noticed I've mentioned the word TOP in Caps. Yes, TOP is the most important word as far as stacks are concerned. Data is stored in a Stack where adding of data is permitted only from the top. moving/Deleting Data is also done from the top. As Simple as That.

Now you may ask where Stacks are used?

Stacks are infact used on every Processor. Each processor has a stack where data and addresses are pushed or added to the stack. Again the TOP rule is followed here. The ESP Register adds as a Stack Pointer that refers to the top of the stack in the Processor. Anyway, since the explaination of how the Processor Stack works is beyond the subject of this Tutorial, let's write our Stack Data Structure. Remember some Stack Terminology before continuing. Adding Data to the Stack is known as Pushing and deleting data from the stack is known as Popping.

Clearly we can see that the last data pushed is the first one to be popped out. That's why a Stack is also known as a LIFO Data Structure which stands for "Last In,First Out" and I guess you know why.

Let us see how we implemented the stack. We first created a variable called top that points to the top of the stack. It is initialised to -1 to indicate that the stack is empty. As Data is entered, the value in top increments itself and data is stored into an array arr. Now there's one drawback to this Data Structure. Here we state the Maximum number of elements as 10. What if we need more than 10 Data Elements? In that case we combine a Stack along with a Linked List which will be explained later.

QUEUES

There's a huge crowd at your local grocery store. There are too many people trying to buy their respective items and the Shopkeeper doesn't know from where to start. Everyone wants their job done quickly and the shopkeeper needs an efficient method to solve this problem. What does he do? He introduces a Queue System based on the First Come, First Serve System. The Last Person trying to buy an item stands behind the last person at the END of the queue. The Shopkeeper however is present at the FRONT end of the queue. He gives the item to the person in FRONT of the queue and after the transaction is done, the person in FRONT of the Queue Leaves. Then the person second in queue becomes the First person in the Queue.

Do you get the point here? *A Queue is similar to a stack except that addition of data is done in the BACK end and deletion of data is done in the FRONT.* Writing a queue is a lot harder than writing a stack. We maintain 2 Integers in our Queue Data Structure, one signifying the FRONT end of the Queue and the other referring to the BACK end of the Queue.

Let us use the same coding style as we used for the Stack. We first initialize both the ends to -1 to indicate an empty queue. When Data is added to the queue both ends get respective positive values. When New Data is added, the back End is incremented and when data is deleted the front end is decremented. This works fine but it has a major drawback. What if the Maximum capacity of the Queue is 5 Items. Suppose the User has added 4 items, deleted 3 items and adds 2 again.

The Stack won't permit him to add the last half of the data as it will report that the stack is full. The Reason is that we are blindly incrementing/decrementing each end depending on addition/deletion not realizing that both the ends are related to each other. I leave this as an exercise for you to answer.

Why will our proposed Stack report the Stack as Full even though it's actually vacant?

So we need another approach. In this method we focus more on the data than on the addition end and the deletion end. What we now use is the grocery store example again. Suppose there are 5 items in a queue and we want to delete them one by one. We first delete the first data item pointed by the deletion end. Then we shift all data one step ahead so that the second item becomes the first, third item becomes second and so on...

Another method would be to maintain the difference between the two ends which is not practical. Hence we stick to our previous method. It might be slow in Big Queues, but it certainly works great. Here's the code.

```
#include <iostream>
using namespace std;
#define MAX 5          // MAXIMUM CONTENTS IN QUEUE
class queue
{
private:
int t[MAX];
int al;      // Addition End
int dl;      // Deletion End
public:
queue()
```

```
{
dl=-1;
al=-1;
}
void del()
{
int tmp;
if(dl==-1)
{
cout<<"Queue is Empty";
}
else
{
for(int j=0;j<=al;j++)
{
if((j+1)<=al)
{
tmp=t[j+1];
t[j]=tmp;
}
else
{
al--;

if(al==-1)
dl=-1;
else
dl=0;
}
}
}
}

void add(int item)
{
if(dl==-1 && al==-1)
{
dl++;
al++;
}
else
{
al++;
if(al==MAX)
{
cout<<"Queue is Full\n";
```

```
al--;
return;
}
}
t[al]=item;

}

void display()
{
if(dl!=-1)
{
for(int i=0;i<=al;i++)
cout<<t[i]<<" ";
}
else
cout<<"EMPTY";
}

};

int main()
{
queue a;
int data[5]={32,23,45,99,24};

cout<<"Queue before adding Elements: ";
a.display();
cout<<endl<<endl;

for(int i=0;i<5;i++)
{
a.add(data[i]);
cout<<"Addition Number : "<<(i+1)<<" : ";
a.display();
cout<<endl;
}
cout<<endl;
cout<<"Queue after adding Elements: ";
a.display();
cout<<endl<<endl;

for(i=0;i<5;i++)
{
a.del();
cout<<"Deletion Number : "<<(i+1)<<" : ";
```

```
a.display();
cout<<endl;
}
return 0;
}
```

OUTPUT:

```
Queue before adding Elements: EMPTY
Addition Number : 1 : 32
Addition Number : 2 : 32 23
Addition Number : 3 : 32 23 45
Addition Number : 4 : 32 23 45 99
Addition Number : 5 : 32 23 45 99 24

Queue after adding Elements: 32 23 45 99 24

Deletion Number : 1 : 23 45 99 24
Deletion Number : 2 : 45 99 24
Deletion Number : 3 : 99 24
Deletion Number : 4 : 24
Deletion Number : 5 : EMPTY
```

As you can clearly see through the output of this program that addition is always done at the end of the queue while deletion is done from the front end of the queue. Once again the Maximum limit of Data will be extended later when we learn Linked Lists.

BINARY TREES

Until now all data structures that we have covered (Stack, Queue, Linked List) are linear in nature ie. they have a definite order of placement. Now we shall study Binary Trees which requires a different thought process as it is a non linear data structure.

A Binary Tree consists of a main node known as the Root. The Root then has two sub-sections, ie. The left and the right half. The data subsequently stored after the root is created depends on its value compared to the root.

Suppose the root value is 10 and the Value to be added is 15, then the data is added to the right section of the root. The Basic idea is that every node can be thought of a binary tree itself. Each node has two pointers, one to the left and the other to the right. Depending on the value to be stored, it is placed after a node's right pointer if the value of the node is lesser than the one to be added or the node's left pointer if vice versa.

Let's take an Example. To add the Following List of Numbers, we end up with a Binary Tree like this:

32 16 34 1 87 13 7 18 14 19 23 24 41 5 53

Here's How:

**: KEEP ADDING DATA IN THE TREE ON PAPER AFTER EACH STEP BELOW TO UNDERSTAND HOW THE TREE IS FORMED.

1) Since 32 is the First Number to be added, 32 becomes the root of the tree.

2) Next Number is 16 which is lesser than 32 Hence 16 becomes left node of 32.
3) 34. Since 34 > 32 , 34 becomes the right node of the ROOT.

4) 1. Since 1 < 32 we jump to the left node of the ROOT. But since the left node has already been taken we test 1 once again. Since 1 < 16, 1 becomes the left node of 16.
5) 87. Since 87 > 32 we jump to the right node of the root. Once again this space is occupied by 34. Now since 87 > 34, 87 becomes the right node of 34.

6) 13. Since 13 < 32 we jump to left node of the root. There, 13 < 16 so we continue towards the left node of 16. There 13 > 1, so 13 becomes the right node of 1.
7) Similarly work out addition till the end ie. before Number 53.
8) 53. Since 53 > 32 we jump to the right node of the root. There 53 > 34 so we continue to the right node of 34. There 53 < 87 so we continue towards the left node of 87. There 53 > 41 so we jump to the right node of 41. Since the Right node of 41 is empty 53 becomes the right node of 41.

This should give you an idea of how a Binary Tree works. You must know that:
1) The linking of nodes to nodes in a Binary Tree is one to one in nature ie. a node cannot be pointed by more than 1 node.
2) A Node can point to two different sub-nodes at the most.
Here in the binary tree above there are a few nodes whose left and right pointers are empty ie. they have no sub-node attached to them. So Nodes 5,14,18, 19,23,24,41 have their left nodes empty.
There are three popular ways to display a Binary Tree. Displaying the trees contents is known as transversal. There are three ways of transversing a tree iw. in inorder,preorder and postorder transversal methods. Description of each is shown below:

PREORDER:
1) Visit the root.
2) Transverse the left leaf in preorder.
3) Transverse the right leaf in preorder.

INORDER:
1) Transverse the left leaf in inorder.
2) Visit the root.
3) Transverse the right leaf in inorder.

POSTORDER:
1) Transverse the left leaf in postorder.
2) Transverse the right leaf in postorder.
3) Visit the root.
Writing code for these three methods are simple if we understand the recursive nature of a binary tree. Binary tree is recursive, as in each node can be thought of

a binary tree itself. It's just the order of displaying data that makes a difference for transversal.
Deletion from a Binary Tree is a bit more difficult to understand. For now just remember that for deleting a node, it is replaced with it's next inorder successor. I'll explain everything after the Binary Tree code.
Now that you've got all your Binary Tree Fundas clear, let's move on with the Source code.

```
#include <iostream>
using namespace std;
#define YES 1
#define NO 0
class tree
{
private:
struct leaf
{
int data;
leaf *l;
leaf *r;
};
struct leaf *p;
public:
tree();
~tree();
void destruct(leaf *q);
tree(tree& a);
void findparent(int n,int &found,leaf* &parent);
void findfordel(int n,int &found,leaf *&parent,leaf* &x);
void add(int n);
void transverse();
void in(leaf *q);
void pre(leaf *q);
void post(leaf *q);
void del(int n);
};

tree::tree()
{
p=NULL;
}

tree::~tree()
{
destruct(p);
}
```

```
void tree::destruct(leaf *q)
{
if(q!=NULL)
{
destruct(q->l);
del(q->data);
destruct(q->r);
}
}
void tree::findparent(int n,int &found,leaf *&parent)
{
leaf *q;
found=NO;
parent=NULL;
if(p==NULL)
return;
q=p;
while(q!=NULL)
{
if(q->data==n)
{
found=YES;
return;
}
if(q->data>n)
{
parent=q;
q=q->l;
}
else
{
parent=q;
q=q->r;
}
}
}
void tree::add(int n)
{
int found;
leaf *t,*parent;
findparent(n,found,parent);
if(found==YES)
cout<<"\nSuch a Node Exists";
else
{
t=new leaf;
```

```
t->data=n;
t->l=NULL;
t->r=NULL;
if(parent==NULL)
p=t;
else
parent->data > n ? parent->l=t : parent->r=t;
}
}
void tree::transverse()
{
int c;
cout<<"\n1.InOrder\n2.Preorder\n3.Postorder\nChoice: ";
cin>>c;
switch(c)
{
case 1:
in(p);
break;
case 2:
pre(p);
break;
case 3:
post(p);
break;
}
}
void tree::in(leaf *q)
{
if(q!=NULL)
{
in(q->l);
cout<<"\t"<<q->data<<endl;
in(q->r);
}
}
void tree::pre(leaf *q)
{
if(q!=NULL)
{
cout<<"\t"<<q->data<<endl;
pre(q->l);
pre(q->r);
}
}
void tree::post(leaf *q)
```

```
{
if(q!=NULL)
{
post(q->l);
post(q->r);
cout<<"\t"<<q->data<<endl;
}
}
void tree::findfordel(int n,int &found,leaf *&parent,leaf *&x)
{
leaf *q;
found=0;
parent=NULL;
if(p==NULL)
return;
q=p;
while(q!=NULL)
{
if(q->data==n)
{
found=1;
x=q;
return;
}
if(q->data>n)
{
parent=q;
q=q->l;
}
else
{
parent=q;
q=q->r;
}
}
}

void tree::del(int num)
{
leaf *parent,*x,*xsucc;
int found;

// If EMPTY TREE
if(p==NULL)
{
cout<<"\nTree is Empty";
```

```
return;
}
parent=x=NULL;
findfordel(num,found,parent,x);
if(found==0)
{
cout<<"\nNode to be deleted NOT FOUND";
return;
}
// If the node to be deleted has 2 leaves
if(x->l != NULL && x->r != NULL)
{
parent=x;
xsucc=x->r;

while(xsucc->l != NULL)
{
parent=xsucc;
xsucc=xsucc->l;
}
x->data=xsucc->data;
x=xsucc;
}
// if the node to be deleted has no child
if(x->l == NULL && x->r == NULL)
{
if(parent->r == x)
parent->r=NULL;
else
parent->l=NULL;

delete x;
return;
}
// if node has only right leaf
if(x->l == NULL && x->r != NULL )
{
if(parent->l == x)
parent->l=x->r;
else
parent->r=x->r;
delete x;
return;
}
// if node to be deleted has only left child
if(x->l != NULL && x->r == NULL)
```

```
{
if(parent->l == x)
parent->l=x->l;
else
parent->r=x->l;
delete x;
return;
}
}
int main()
{
tree t;
int data[]={32,16,34,1,87,13,7,18,14,19,23,24,41,5,53};
for(int i=0;i<15;i++)
t.add(data[i]);

t.transverse();
t.del(16);
t.transverse();
t.del(41);
t.transverse();
return 0;
}
```

OUTPUT:

```
1.InOrder
2.Preorder
3.Postorder
Choice: 1
1
5
7
13
14
16
18
19
23
24
32
34
41
53
87
1.InOrder
2.Preorder
3.Postorder
```

```
Choice: 2
32
18
1
13
7
5
14
19
23
24
34
87
41
53
1.InOrder
2.Preorder
3.Postorder
Choice: 3
5
7
14
13
1
24
23
19
18
53
87
34
32
```

UNIT-IV

Dictionary

- A dictionary is a collection of elements
- Each element has a field called key
  - (key, value)

- Every key is usually distinct
- Dictionary Operations
  - Determine whether or not the dictionary is empty
  - Determine the dictionary size (i.e., # of pairs)
  - Insert a pair into the dictionary
  - Search the pair with a specified key
  - Delete the pair with a specified key

Accessing Dictionary Elements

- Random Access
  - Any element in the dictionary can be retrieved by simply performing a search on its key

- Sequential Access
  - Elements are retrieved one by one in ascending order of the key field
  - Sequential Access Operations:
    - Begin – retrieves the element with smallest key
    - Next – retrieves the next element

Dictionary with Duplicates

- Keys are not required to be distinct
- Word dictionary is such an example
  - Pairs are of the form (word, meaning)
  - May have two or more entries for the same word
  - For example, the meanings of the word, rank:
    - (rank, a relative position in a society)
    - (rank, an official position or grade)
    - (rank, to give a particular order or position to)
    - etc.

Application of Dictionary

- Collection of student records in a class
  - (key, value) =
    (student-number, a list of assignment and exam marks)
  - All keys are distinct
- Get the element whose key is Tiger Woods
- Update the element whose key is Seri Pak
- Read Examples 10.1, 10.2 & 10.3
- Exercise: Give other real-world applications of dictionaries and/or dictionaries with duplicates

Dictionary as an Ordered Linear List

- L = (e1, e2, e3, …, en)
- Each ei is a pair (key, value)
- Array or chain representation
  - unsorted array: O(*n*) search time
  - sorted array: O(log*n*) search time
  - unsorted chain: O(*n*) search time
  - sorted chain: O(*n*) search time

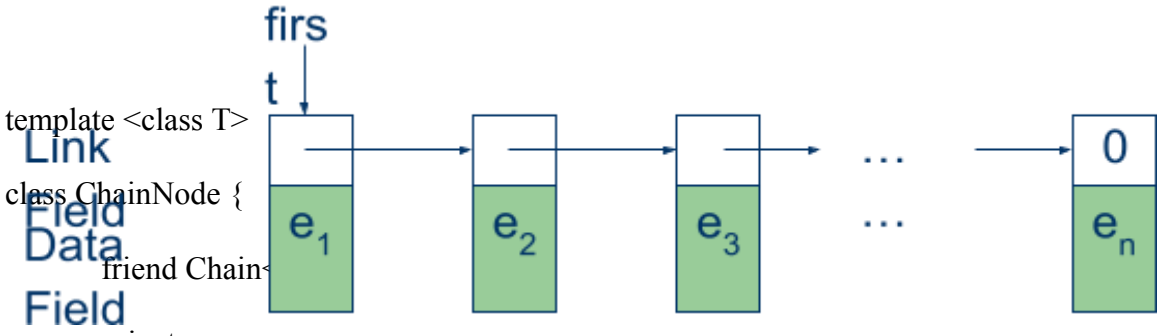
Linked Representation of Linear List

- Each element is represented in a cell or node.
- Each node keeps explicit information about the location of other relevant nodes.
- This explicit information about the location of another node is called a link or pointer.

Singly Linked List

- Let *L = (e1,e2,…,en)*
  - Each element *ei* is represented in a separate node
  - Each node has exactly one link field that is used to locate the next element in the linear list
  - The last node, *en*, has no node to link to and so its link field is NULL.

- This structure is also called a chain**.**




```
template <class T>
class ChainNode {
        friend Chain<
        private:
                T data;
                ChainNode<T> *link;
};
```

- Since Chain<T> is a friend of ChainNode<T>, Chain<T> has access to all members (including private members) of ChainNode<T>.

Class ‘Chain’

```
template <class T>
class Chain {
public:         Chain() { first = 0; }
                ~Chain();
                bool isEmpty() const { return first == 0; }
                int Length() const;
                bool Find(int k, T& x) const;
                int Search(const T& x) const;
                Chain<T>& Delete(int k, T& x);
                Chain<T>& Insert(int k, const T& x);
                void Output(ostream& out) const;
private:    ChainNode<T> *first;
                int listSize;
};
```

Operation 'Length'

```
template <class T>
int Chain<T>::Length() const
{
ChainNode<T> *current = first;
int len = 0;
while (current) {
len++;
current = current->link;
```

```
}
return len;
}
```

- The time complexity is

Q(n), where n is the length of the chain

Operation 'Find'

```
template <class T>
bool Chain<T>::Find(int k, T& x) const
{// Set x to the k'th element in the list if it exists
// Throw illegal index exception if no such element exists
checkIndex(k);
// move to desired node
ChainNode<T>* current = first;
for (int i = 0; i < k; i++)
current = current->link;
x = current->data;
return true;
}
```

- The time complexity is

O(k)

Exercise – write the code for checkIndex() operation & determine its time complexity

Operation 'Search'

```
template <class T>
```

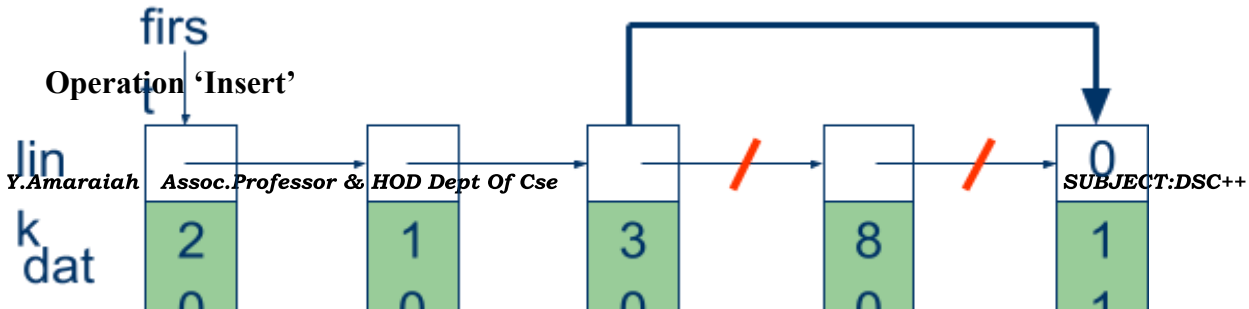
```
int Chain<T>::Search(const T& x) const
{// Locate x and return its position if found else return -1

  // search the chain for x
  ChainNode<T>* current = first;
  int index = 0;  // index of current
  while (current != NULL && current->data != x)
  {
    // move to next node
    current = current->link;
    index++;
  }
  // make sure we found matching element
  if (current == NULL)
    return -1;
  else
    return index;
}
```

The time complexity is O(n)

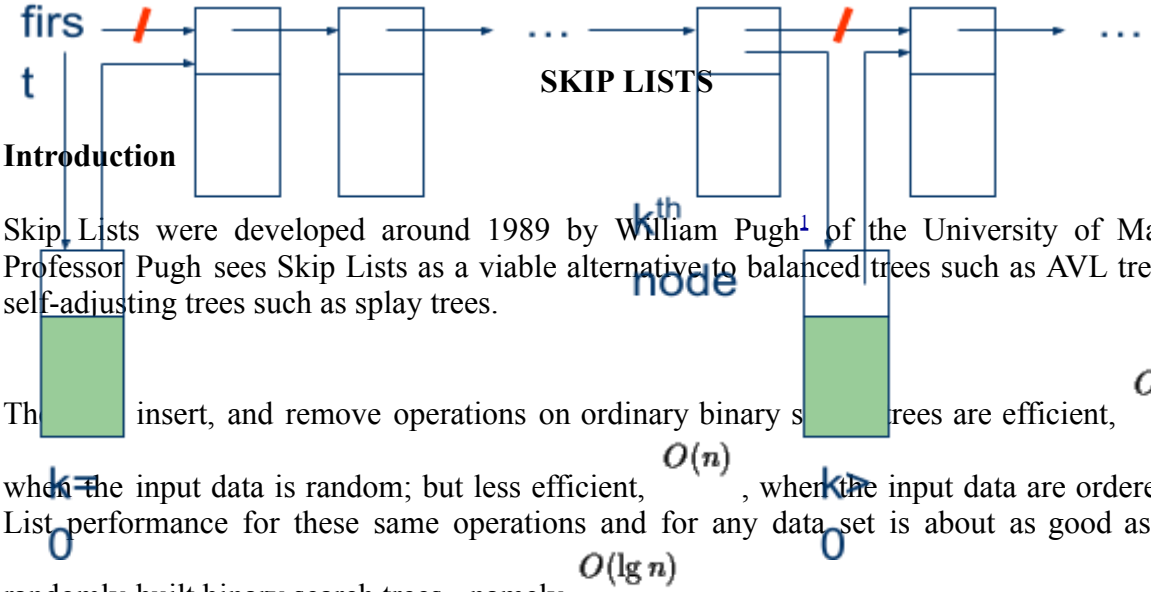
Operation 'Delete'

- To delete the fourth element from the chain, we
  - locate the third and fourth nodes
  - link the third node to the fifth
  - free the fourth node so that it becomes available for reuse

Operation 'Insert'



- To insert an element following the *k*th in chain, we
  - locate the *k*th node
  - new node's link points to *k*th node's link
  - *k*th node's link now points to the new node



SKIP LISTS

Introduction

Skip Lists were developed around 1989 by William Pugh[1] of the University of Maryland. Professor Pugh sees Skip Lists as a viable alternative to balanced trees such as AVL trees or to self-adjusting trees such as splay trees.

Th insert, and remove operations on ordinary binary s rees are efficient, $O(\lg n)$, when the input data is random; but less efficient, $O(n)$, when the input data are ordered. Skip List performance for these same operations and for any data set is about as good as that of randomly-built binary search trees - namely $O(\lg n)$

In an ordinary sorted linked list, find, insert, and remove are in $O(n)$ because the list must be scanned node-by-node from the head to find the relevant node. If somehow we could scan down the list in bigger steps (``skip" down, as it were), we would reduce the cost of scanning. This is the fundamental idea behind Skip Lists.

In simple terms, Skip Lists are sorted linked lists with two differences:

1. the nodes in an ordinary list have one 'next' reference. The nodes in a Skip List have many 'next' references (called *forward* references).
2. the number of forward references for a given node is determined probabilistically.

We speak of a Skip List node having *level*s, one level per forward reference. The number of levels in a node is called the *size* of the node.

In an ordinary sorted list, insert, remove, and find operations require sequential traversal of the list. This results in $O(n)$ performance per operation. Skip Lists allow intermediate nodes in the list to be ``skipped" during a traversal - resulting in an *expected* performance of $O(\lg n)$ per operation.

The Basic Idea

To introduce the Skip List data structure, let's look at three list data structures that allow skipping, but are not truly Skip Lists. The first of these, shown in Figure 1 allows every other node to be skipped in a traversal. The second, shown in Figure 2 additionally allows every fourth node to be skipped. The third, shown in Figure 3 additionally allows every eighth node to be skipped, but suggests further development of the idea to skipping every 2^i -th node.

For all three pseudo-skip-lists, there is a header node and the nodes do not all have the same number of forward references. Every node has a reference to the next node, but some have additional references to nodes further along the list.

Here is pseudo-code for the find operation on each list. This same algorithm will be used with real Skip Lists later.

```
Comparable & find(const Comparable & X)
{
node = header node
for (reference level of node from (nodesize - 1) down to 0)
while (the node referred to is less than X)
node = node referred to
if (node referred to has value X)
return node's value
else
return item_not_found
}
```

We start at the highest level of the header node and follow the references along this level until the value at the node referred to is equal to or larger than X. At this point, we switch to the next lower level and continue the search. Eventually, we shall be dealing with a reference at the lowest level of a node. If the next node has the value X, then we return that value; otherwise we return a value that signals unsuccessful search.

Skip Every Second Node

Figure 1 shows a 16-node list in which every second node has a reference two nodes ahead. The value stored at each node is shown below it (and corresponds in this example to the position of the node in the list). The header node has two levels; it's no smaller than largest node in the list. Node 2 has a reference to node 4, two nodes ahead. Similarly for nodes 4, 6, 8, *etc* - every second node has a reference two nodes ahead.

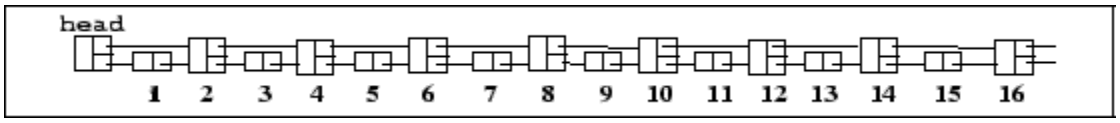


Figure 1: Skip Every 2nd Node

It's clear that the find operation does not need to visit each node. It can skip over every other node, then do a final visit at the end. The number of nodes visited is therefore no more than $\lceil \frac{n}{2} \rceil + 1$. For example, the nodes visited in scanning for the node with value 15 would be 2, 4, 6, 8, 10, 12, 14, 16, 15, a total of $(\lceil \frac{16}{2} \rceil + 1) = 9$.

Follow the algorithm for find on page ❑ for this simple example to be sure you understand it thoroughly.

Skip Every Second and Fourth Node

The second example is a list in which every second node has a reference two nodes ahead and additionally every fourth node has a reference four nodes ahead. Such a list is shown in Figure 2. The header is still no smaller than the largest node in the list.

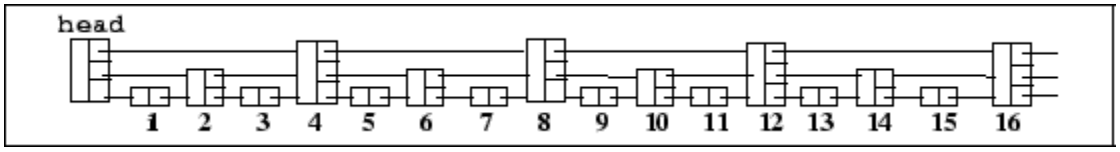


Figure 2: Skip Every 2nd and 4th Node

The find operation can now make bigger skips than those for the list in Figure 1. Every fourth node is skipped until the search is confined between two nodes of size 3. At this point, as many as three nodes may need to be scanned. It is also possible that some nodes will be visited more than once using the algorithm on page ❑. The number of nodes visited[2] is no more than $\lceil \frac{n}{4} \rceil + 3$. As an example, look at the nodes visited in scanning for the node with value of 15. These are 4, 8, 12, 16, 14, 16, 15 for a total of $(\lceil \frac{16}{4} \rceil + 3) = 7$.

Skip Every 2^i -th Node

This final example is for a list in which some skip lengths are even larger.

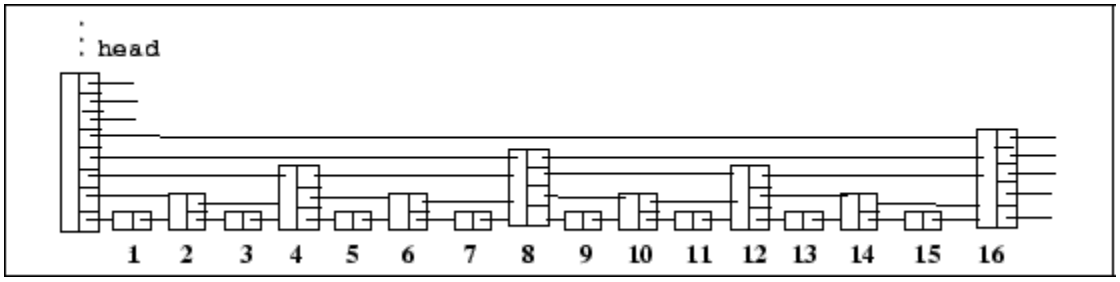


Figure 3: Skip Every 2^i -th Node

Every 2^i -th node, $i = 1, \ldots \lceil \lg n \rceil$, has a forward reference 2^i nodes ahead. For example, every 2^{nd} node has a reference 2 nodes ahead; every 8^{th} node has a reference 8 nodes ahead, *etc.* Figure 3 shows a short list of this type. Once again, the header is no smaller than the largest node on the list. It is shown arbitrarily large in the Figure.

Suppose the Skip List in Figure 3 contained 32 nodes and consider a search in it. Working down from the highest level, we first encounter node 16 and have cut the search in half. We then search again, one level down in either the left or right half of the list, again cutting the remaining search in half. We continue in this manner till we find the sought-after node (or not). This is quite reminiscent of binary search in an array and is perhaps the best way to intuitively understand why the maximum number of nodes visited in this list is in $O(\lg n)$.

This data structure is looking pretty good, but there's a serious problem with it for the insert and remove operations. The work required to reorganize the list after an insertion or deletion is in $O(n)$. For example, suppose that the first element is removed in Figure 3. Since it is necessary to maintain the strict pattern of node sizes, values from 2 to the end must be moved toward the head and the end node must be removed. A similar situation occurs when a new element is added to the list.

This is where the probabilistic approach of a true Skip List comes into play. A Skip List is built with the same *distribution* of node sizes, but without the requirement for the rigid pattern of node sizes shown. It is no longer necessary to maintain the rigid pattern by moving values around after a remove or insert operation. Pugh shows that *with high probability* such a list still exhibits $O(\lg n)$ behavior. The probability that a given Skip List will perform badly is very small.

Skip Lists - the Probabilistic Approach

Figure 4 shows the list of Figure 3 with the nodes reorganized. The distribution of node sizes is exactly the same as that of Figure 3; the nodes just occur in a different pattern. In this example, the pattern would require that 50% of the nodes have just one reference, 25% have just two

references, 12.5% have just three references, *etc*. The distribution of node sizes is maintained, but the strict order is not required.

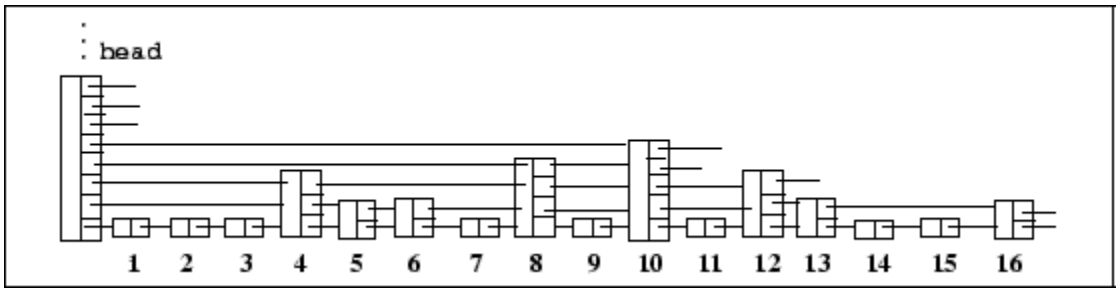


Figure 4: A Probabilistic Skip List

Figure 4 shows one way this might work out. Of course, this is probabilistic, so there are many other possible node sequences that would satisfy the required probability distribution.

When inserting new nodes, we choose the size of the new node probabilistically. Every Skip List has an associated (and fixed) probability p that determines the distribution of nodes. A fraction p of the nodes that have at least r references also have $r+1$ references. The Skip List does not have to be reorganized when a new element is inserted.

An Aside on Node Distribution

Suppose we have an infinitely-long Skip List with associated probability p. This means that a fraction, p, of the nodes with a forward reference at level i *also* have a forward reference at level $i+1$.

Let L_k be the fraction of nodes having *precisely* k forward references (*i.e.*, L_k is the fraction of nodes of size k). Then, $L_k = pL_{k-1}$ and

$$L_1 = 1 - \sum_{i=2}^{\infty} L_i$$

Since

$$\sum_{i=2}^{\infty} L_i = L_2 + L_3 + \cdots = L_2 + pL_2 + p^2 L_2 + \cdots$$

we can write L_1 in terms of L_2 as

$$L_1 = 1 - L_2 \sum_{i=0}^{\infty} p^i$$

Recall that $\sum_{i=0}^{\infty} p^i$ is the sum of the geometric progression with first term 1 and common ratio p. Thus,

$$\sum_{i=0}^{\infty} p^i = \frac{1}{1-p}$$

Therefore, $L_1 = 1 - \frac{L_2}{1-p}$. Since $L_2 = pL_1$, we can write

$$L_1 = 1 - \frac{pL_1}{1-p} \Rightarrow L_1 = 1 - p$$

Example: In the situation shown in Figure 4, $p = 0.5$. Therefore, $1 - \frac{1}{2} = \frac{1}{2}$ of the nodes with at least one reference have two references; one-half of those with at least two references have three references, *etc*. You should work out the distribution for a SkipList with associated probability of $\frac{1}{4}$ to be sure you understand how distributions are computed.

Choosing a Node Probabilistically

When a new node is needed for a insert operation, the size of the node is chosen by consulting a random number generator r. Here is one way to do this for a Skip List with associated probability p and maximum node level maxLevel

```
int generateNodeLevel(double p, int maxLevel)
{
int level = 1;

while (drand48() < p)
```

```
level++;
return (level > maxLevel) ? maxLevel : level;
}
```

Note that the level of the new node is independent of the number of nodes already in the Skip List. Each node is chosen only on the basis of the Skip List's associated probability.

When the associated probability is p, the average number of comparisons that must be done for find, is $1 + \frac{\log_{\frac{1}{p}} n}{p} + \frac{1}{1-p}$. For example, for a list of size 65,536, the average number of nodes to be examined is 34.3 for $p = \frac{1}{4}$ and 35 for $p = \frac{1}{2}$. This is a tremendous improvement over an ordinary sorted list for which the average number of comparisons is $\frac{n}{2} = 32768$.

Determining the Header's Level

The level of the header node is the maximum allowed level in the SkipList and is chosen at construction. Pugh shows that the maximum level should be chosen as $\log_{\frac{1}{p}} n$. Thus, for $p = \frac{1}{2}$, the maximum level for a SkipList of up to 65,536 elements should be chosen no smaller than $\lg_2 65536 = 16$.

Performance Considerations

The *expected* time to find an element (and therefore to insert or delete an element) is in $O(\lg n)$. It is possible for the time to be substantially larger if the configuration of node levels is unfavorable for a particular operation. Since the node sizes are generated randomly, it is possible to get a ``bad'' run of sizes. For example, it is possible that each node will be generated at the same size, producing the equivalent of an ordinary sorted list. A bad run of sizes will result in longer-than expected search (and therefore insert or remove) times; the Skip List will simply not be as efficient as expected. Obviously, a bad run will be proportionately less important in a long Skip List than in a short one. The probability of poor performance decreases rapidly with the number of nodes (i.e., list size).

The probability that an operation will take longer than expected is a function of the probability associated with the list. For example, Pugh calculates that for a list with $p = 0.5$ and 4096 elements, the probability that the actual time will exceed the expected time by a factor of 3 is less than one in 200 million.

The relative time and space performance of a Skip List depends on the probability level associated with the list. Pugh suggests that a probability of 0.25 be used in most cases. If the variability of performance is important, he suggests using a probability of 0.5 (variability decreases with increasing probability). Interestingly, the average number of references per node is only 1.33 when a probability of 0.25 is used. A binary search tree, of course, has 2 references per node, so Skip Lists can be more space-efficient.

Implementation of Skip Lists

The implementation embeds the SkipListNode class as a private member of SkipList.

```
template <class Comparable>

class SkipList
{
private:
  // embedded node
  class SkipListNode
  {
  public:
    void setDatum(const Comparable & datum);
    void setForward(int i, SkipListNode * f);
    void setSize(int sz);
    SkipListNode(); // 16 levels, Comparable() for datum
    SkipListNode(const Comparable & datum, int levels);
    SkipListNode(const SkipListNode &);
    ~SkipListNode();

    const Comparable & getDatum() const;
    int getSize() const;
    SkipListNode * getForward(int level);
  private:
    int _levels;  // number of levels in this node
    vector<SkipListNode *> _forward;  // forward pointers
    Comparable _datum;  // datum stored at this node
  }; // SkipListNode
```

continued below

SkipList implementation continued

```
public:
  SkipList(); // use max_node_size of 16, prob of 0.25
  SkipList(int max_node_size, double probab);
  SkipList(const SkipList &);
```

```
  ~SkipList();

  int getHighNodeSize() const; // size of the node with most levels
  int getMaxNodeSize() const; // max node size allowed in this list
  double getProbability() const; // prob for this list

  // insert item into this skiplist.
  void insert(const Comparable & item);

  // Search for item in skiplist.  If found, return the item with
  // success equal true.  Otherwise, return a dummy item with
  // success equal false.
  const Comparable & find(const Comparable & item, bool & success);

  // remove the first occurrence of the specified item
  void remove(const Comparable & item);
private:
  // Search for item in this skiplist, starting at startnode
  // startnode will usually be the header.  If item is found,
  // return the node at which it is found with success equal true.
  // Otherwise, return the preceding node with success equal false.
  // Precondition:  startnode is not NULL
  SkipListNode * find(const Comparable & item,
                      SkipListNode * startnode);

  // Accessor for _head
  SkipListNode * getHeader() const; // return the header node
```

continued below

SkipList implementation continued

```
  // Finds the point at which item would be inserted using a node
  // of the given size.
  // Returns a "lookback" node that has each of its forward
  // pointers set to the closest node (toward the header) at that level.
  SkipListNode * findInsertPoint(const Comparable & item,
                                 int nodesize);

  // insert item into this skiplist, using a node of the
  // specified size. success returned true if insertion succeeds
  void insert(const Comparable & item, int nodesize, bool & success);

  int _high_node_size;  // highest-level node now in this list
  int _max_node_size;  // highest-level node allowed in this list
  double _prob;  // probability associated with this list
```

```
  SkipListNode * _head;  // header node of this list
};  // SkipList
```

Overview of SkipList Methods

We will examine SkipList methods insert, and remove in some detail below. Pseudocode for find was given on page ❑.

Insertion and deletion involve searching the SkipList to find the insertion or deletion point, then manipulating the references to make the relevant change.

When inserting a new element, we first generate a node that has had its level selected randomly. The SkipList has a maximum allowed level set at construction time. The number of levels in the header node is the maximum allowed. For convenience in searching, the SkipList keeps track of the maximum level actually in the list. There is no need to search levels above this actual maximum.

FindInsertPoint Method

In an ordinary linked list, insertion and deletion require having a pointer to the previous node. Insertion is done after this previous node, deletion deletes the node following the previous node. We call the point in the list after the previous node, the *insertion point*, even if we are doing deletions. Finding the insertion point is the first step in doing an insertion or a deletion.

In Skip Lists, we need pointers to all the *see-able* previous nodes between the insertion point and the header. Imagine standing at the insertion point, looking back toward the header. All the nodes you can see are the see-able nodes. Some nodes may not be see-able because they are blocked by higher nodes. Figure 5 shows an example.

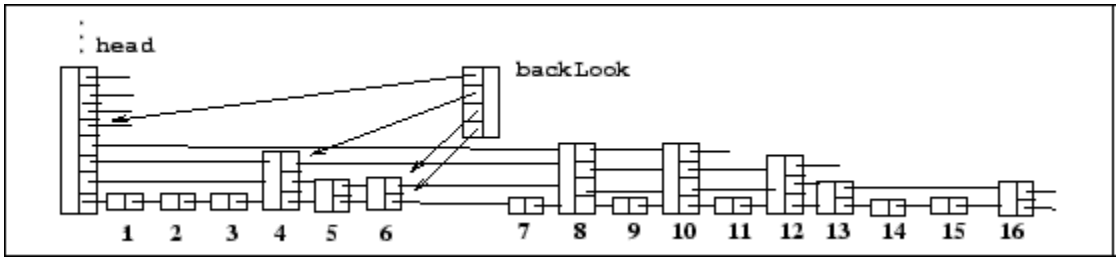


Figure 5: Update Node Example

In the figure, the insertion point is between nodes 6 and 7. Looking back toward the header, the nodes you can see at the various levels are

```
level  node seen
```

0 6
1 6
2 4
3 header

We construct a backLook node that has its forward pointers set to the relevant see-able nodes. This is the type of node returned by the findInsertPoint method.

Insert Method

Once we have the backLook node returned by findInsertPoint and have constructed the new node to be inserted, doing an insertion is easy. Here's how.

The public insert(const Comparable &) method decides on the new node's size by random choice, then calls the overloading private method
insert(const Comparable &,int,bool &) to do all the work.

```
template<class T>
void
SkipList<T>::insert(const T & item)
{
  int nodesize = randomNodeSize(getProbability(), getMaxNodeSize());
  insert(item, nodesize);
}
```

This is the private insert method.

```
template<class T>
void
SkipList<T>::insert(const T & item, int nodesize)
{
  SkipList<T>::SkipListNode * backLook = findInsertPoint(item, nodesize);
  SkipList<T>::SkipListNode * newnode =
    new SkipList<T>::SkipListNode(item, nodesize);
  // newnode may have more levels than are currently in list
  // If so, adjust high_node_size of list.
  if (nodesize > getHighNodeSize())
    setHighNodeSize(nodesize);
  //  Rearrange the relevant previous pointers using information
  //  in the backLook node.
  for (int i = 0; i < nodesize; i++)
    {
```

```
    SkipList<T>::SkipListNode * prev = backLook->getForward(i);
    SkipList<T>::SkipListNode * next = prev->getForward(i);
    newnode->setForward(i, next);
    prev->setForward(i, newnode);
   }
  incrementListSize();
}
```

Remove Method

The remove method is entirely analogous to the insert method. It uses the backLook node returned by findInsertPoint to reassign the previous nodes' pointers around the node to be removed.

Instructions:

- **Insert 10 random nodes**: inserts 10 nodes into the list with randomly generated key and value
- **Reset** : deletes all nodes in list
- **The key and value text field**: are used to show and read data of a node
- Insert : inserts a new node into the list using the key and value entered in textfields
- Search: searches the node in the list with the key entered in key-textfield
- Delete: deletes the node with the key entered in key-textfield out of the list

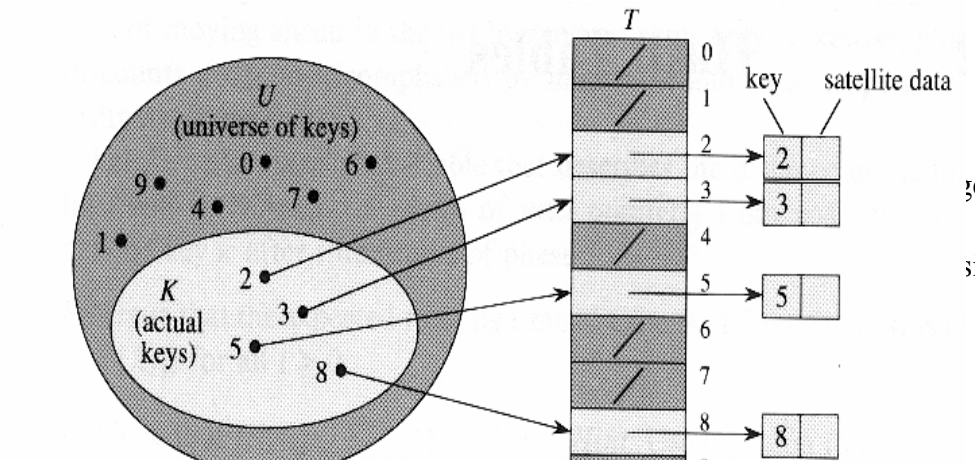
- If you select a node in the display, its key and value is automatically shown in the text fields

HASHING

- Support the following operations
  - Find
  - Insert
  - Delete. (deletions may be unnecessary in some applications)
- Unlike binary search tree, AVL tree and B+-tree, the following functions cannot be done:
  - Minimum and maximum
  - Successor and predecessor
  - Report data within a given range
  - List out the data in order
  -

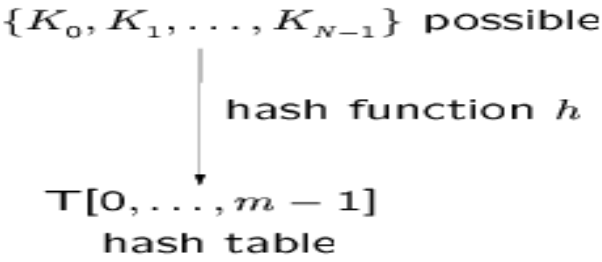
Unrealistic solution

- Each position (slot) corresponds to a key in the universe of keys
  - T[k] corresponds to an element with key k
  - If the set contains no element with key k, then T[k]=NULL



ge compared with the

size is 108, but we



Compilers use hash tables (symbol table) to keep track of declared variables.

- On-line spell checkers. After prehashing the entire dictionary, one can check each word in constant time and print out the misspelled word in order of their appearance in the document.
- Useful in applications when the input keys come in sorted order. This is a bad case for binary search tree. AVL tree and B+-tree are harder to implement and they are not necessarily more efficient
- With hashing, an element of key k is stored in T[h(k)]

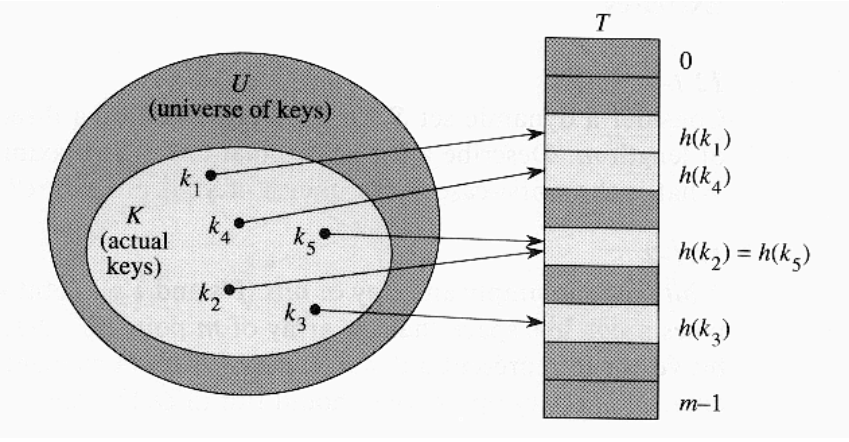


table T[0,1,...,m-1]

- two keys may hash to the same slot
- can we ensure that any two distinct keys get different cells?
  - No, if |U|>m, where m is the size of the hash table

 Design a good hash function

- that is fast to compute and
- can minimize the number of collisions

 Design a method to resolve the collisions when they occur

Hash Function:-

- The division method
  - h(k) = k mod m
  - e.g. m=12, k=100, h(k)=4

Requires only a single division operation (quite fast)

- Certain values of m should be avoided
  - e.g. if m=2p, then h(k) is just the p lowest-order bits of k; the hash function does not depend on all the bits
  - Similarly, if the keys are decimal numbers, should not set m to be a power of 10
- It's a good practice to set the table size m to be a prime number
- Good values for m: primes not too close to exact powers of 2
  - e.g. the hash table is to hold 2000 numbers, and we don't mind an average of 3 numbers being hashed to the same entry
    - choose m=701
- Can the keys be strings?
- Most hash functions assume that the keys are natural numbers
  - if keys are not natural numbers, a way must be found to interpret them as natural numbers

Method 1

- Add up the ASCII values of the characters in the string
- Problems:
  - Different permutations of the same set of characters would have the same hash value
  - If the table size is large, the keys are not distribute well. e.g. Suppose m=10007 and all the keys are eight or fewer characters long. Since ASCII value <= 127, the hash function can only assume values between 0 and 127*8=1016

Method 2

```
int hash( const string & key, int tableSize )
{
    return ( key[ 0 ] + 27 * key[ 1 ] + 729 * key[ 2 ] ) % tableSize;
}
```

- If the first 3 characters are random and the table size is 10,0007 => a reasonably equitable distribution
- Problem
  - English is not random
  - Only 28 percent of the table can actually be hashed to (assuming a table size of 10,007)

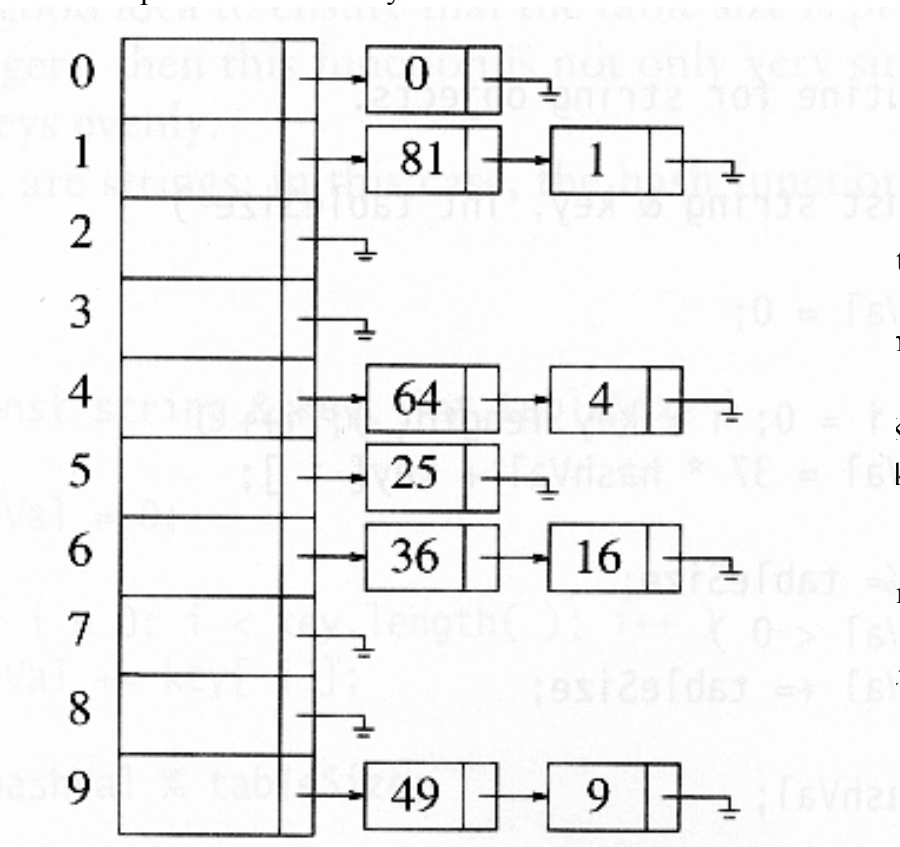
Method 3

- computes
- involves all characters in the key and be expected to distribute well

Collision Handling:

(1) Separate Chaining

- Instead of a hash table, we use a table of linked list
- keep a linked list of keys that hash to the same value



to point to a linked list that

ıg of this list.

ς)]. Delete K if it is found.

ke *m* the next larger prime

in each linked list will be a

n can be done in constant

tion will slow down the

Collision Handling:

(2) Open Addressing

- Open addressing:
  - relocate the key K to be inserted if it collides with an existing key. That is, we store K at an entry different from T[h(K)].

- Two issues arise
  - what is the relocation scheme?
  - how to search for K later?
- Three common methods for resolving a collision in open addressing
  - Linear probing
  - Quadratic probing

Open Addressing

- To insert a key K, compute h0(K). If T[h0(K)] is empty, insert it there. If collision occurs, probe alternative cell h1(K), h2(K), ... until an empty cell is found.
- hi(K) = (hash(K) + f(i)) mod m, with f(0) = 0
  - f: *collision resolution strategy*

Linear Probing :-

- f(i) =i
  - cells are probed sequentially (with wraparound)
  - hi(K) = (hash(K) + i) mod m
- Insertion:
  - Let K be the new key to be inserted. We compute hash(K)
  - For i = 0 to m-1
    - compute L = (hash(K) + I) mod m
    - T[L] is empty, then we put K there and stop.
  - If we cannot find an empty entry to put K, it means that the table is full and we should report an error.
- hi(K) = (hash(K) + i) mod m
- E.g, inserting keys 89, 18, 49, 58, 69 with hash(K)=K mod 10

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						

To insert 69 probe at t[9],t[0],t[1],t[2].

UNIT-V

Priority queue

A priority queue **is an abstract data type in computer programming that supports the following three operations:**

- insertWithPriority: add an element to the queue with an associated priority
- getNext: remove the element from the queue that has the *highest priority*, and return it (also known as "PopElement(Off)", or "GetMinimum")
- peekAtNext (optional): look at the element with *highest priority* without removing it.

Similarity to queues

One can imagine a priority queue as a modified queue, but when one would get the next element off the queue, the highest-priority one is retrieved first.

Stacks and queues may be modeled as particular kinds of priority queues. In a stack, the priority of each inserted element is monotonically increasing; thus, the last element inserted is always the first retrieved. In a queue, the priority of each inserted element is monotonically decreasing; thus, the first element inserted is always the first retrieved.

Simple implementations

There are a variety of simple, usually inefficient, ways to implement a priority queue. They provide an analogy to help one understand what a priority queue is:

- *Sorted list implementation*: Like a checkout line at the supermarket, but where important people get to "cut" in front of less important people. (O(n) insertion time, O(1) get-next time, O(n*log(n)) to build)
- *Unsorted list implementation*: Keep a list of elements as the queue. To add an element, append it to the end. To get the next element, search through all elements for the one with the highest priority. (O(1) insertion time, O(n) get-next due to search)

These implementations are usually inefficient, though can be good depending on the workload (for example, if one does very few GetNext operations, the unsorted list approach may work well). In practice, priority queues blend these two approaches, so any operation takes roughly O(log(n)) time or less.

Usual implementation

To get better performance, priority queues typically use a heap as their backbone, giving O(log *n*) performance for inserts and removals (and O(1) for peekAtNext). Alternatively, if a self-balancing binary search tree is used, all three operations take O(log *n*) time; this is a popular solution where one already has access to these data structures, such as through third-party or standard libraries.

Heap (data structure)

In computer science, a **heap** is a specialized tree-based data structure that satisfies the *heap property:* if *B* is a child node of *A*, then key(*A*) ≥ key(*B*). This implies that an element with the greatest key is always in the root node, and so such a heap is sometimes called a *max-heap*. (Alternatively, if the comparison is reversed, the smallest element is always in the root node, which results in a *min-heap*.) The several variants of heaps are the prototypical most efficient implementations of the abstract data type priority queues. Priority queues are useful in many applications. In particular, heaps are crucial in several efficient graph algorithms.

Heaps are usually implemented in an array, and do not require pointers between elements.

The operations commonly performed with a heap are:

- *find-max* or *find-min*: find the maximum item of a max-heap or a minimum item of a min-heap, respectively
- *delete-max* or *delete-min*: removing the root node of a max- or min-heap, respectively
- *increase-key* or *decrease-key*: updating a key within a max- or min-heap, respectively
- *insert*: adding a new key to the heap
- *merge*: joining two heaps to form a valid new heap containing all the elements of both.

Heaps are used in the sorting algorithm heapsort.

What is a Heap Data Structure?

• A Heap data structure is a binary tree with the following properties:

1. It is a *complete binary tree*; that is, each level of the tree is completely filled, except possibly the bottom level. At this level, it is filled from left to right.

2. It satisfies the heap-order property: The data item stored in each node is greater than or equal to the data items stored in its children.

External Sorting

Outline

- Background
- Basic external sorting.
    - Balanced *n*-way sort merge.
- Balance vs. unbalance.
- Initial block distributions.
    - Polyphase sort merge.

Massive Sorting

- How would you sort lots of data?

  Using a pair of 5-year-old computers, two home DSL connections, 42 hours of computer time, and 5 man hours, I now had documents describing the reading preferences of 260,000 U.S. citizens.

  I downloaded all the files to an external 120 GB Firewire drive in UFS format. The raw data occupied little more than 5 GB.

External Sorting

- How do you sort gigabytes of data?
    - This much data won't fit in physical store, and maybe not in virtual memory either.
- Most of the data's going to be stored on external devices.
    - Disk, tapes, perhaps the network.
- Sorting in such cases is called **external sorting**.

Is Sorting Necessary?

- Sorting makes practical massive data storage and retrieval applications.

    - Sorting improves indexing and access to data.
    - Sorting simplifies and speeds data maintenance.
- Without sorting, databases themselves would not be possible.

External Devices

- External devices have several complicating characteristics.
    - Read-write access to external devices is slow (at least x10).
    - Access patterns may be restricted (sequential tape access).
- Successful external sorting algorithms must accommodate device characteristics.
    - Effective external sorts are device dependent.

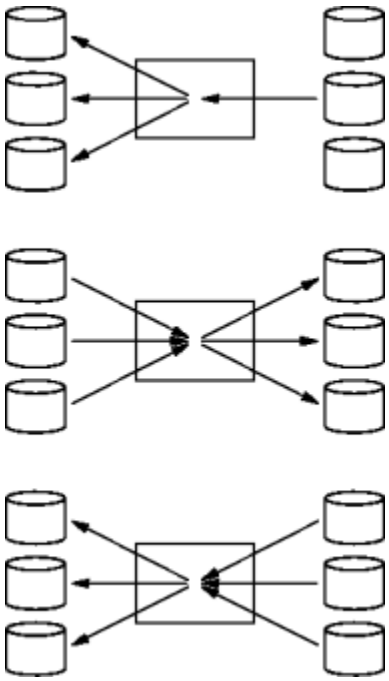
Objectives

- Stick to general external sorting principles.
- Fast sorting by minimizing I-O.
- Assume devices are disk- and tape-like.
    - Faster random access and slower sequential access.
    - However, sustained sequential access always wins.
- Assume a primary-secondary virtual-memory hierarchy.

General Approach.

- Minimize device I-O by reading and writing big blocks.
    - Big blocks in the system can be manipulated by the sort.
- The data sloshes back and forth between input and output devices.

- The sort works on data as it sluices through the system.

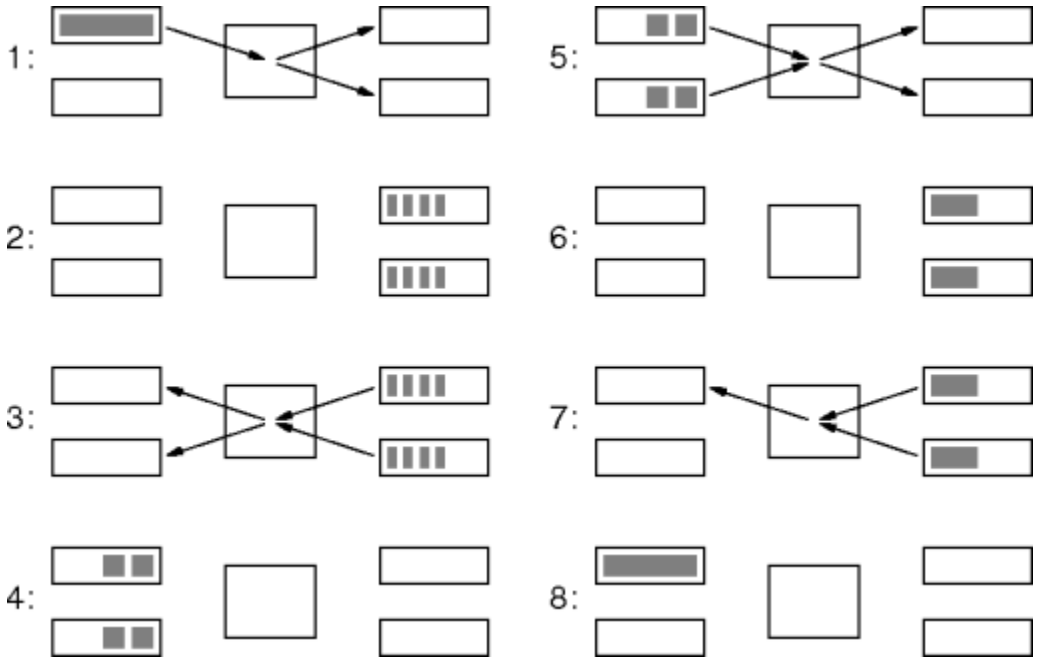


- The complete data transfer from input to output is called a **pass**.
- External sort costs can be measured in passes.
  - As well devices and the usual time and space measures.

Basic Idea

- Group data into blocks; data within a block is sorted.
- Merge two or more sorted blocks into a single sorted block.
- One pass through the data halves the number of blocks while doubling their size.
- Eventually you're left with a single, sorted block.
- This is known as the **sort-merge strategy**.

Basic Idea Illustrated



Implementation Details

- Assume $2n > 2$ devices.
- The initial block size is determined by available primary storage.
- The initial block sort is an internal sort.
    - The initial data ordering makes the choice of an internal sort tricky.
- The remaining passes combine and sort blocks with a merge (as in merge sort).

Balanced *n*-way Sort Merge

- A sort-merge algorithm is
    - **balanced** if it uses *n* input and *n* output devices.
    - ***n*-way** if it merges *n* blocks into one block.
- The result is a **balanced, *n*-way sort merge**.

Analysis

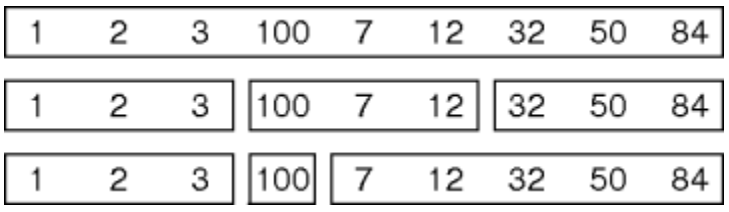
- Assume a balanced, *n*-way sort merge.
- A file of *R* records can be split into $S = R/B$ blocks, with each block holding (at most) *B* records.
    - *B* determined by primary-storage availability.
- Each *n*-way merge reduces *M* blocks to *M*/*n* blocks.
- Data are sorted in $O(\log_n S)$ passes.

The Initial Distribution

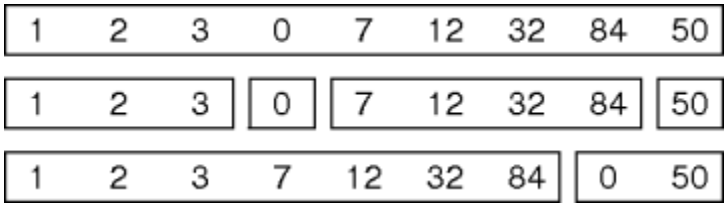
- The initial distribution pass blocks, sorts, and distributes the data.
- A simple initial distribution pass has problems:
    - The block size is limited by available storage.
        - Larger block sizes mean fewer passes.
    - It ignores any sorting that exists in the data.
        - Long ordered runs are broken up into blocks.

Exploiting Initial Order

- Use ordered-run length to determine block size.



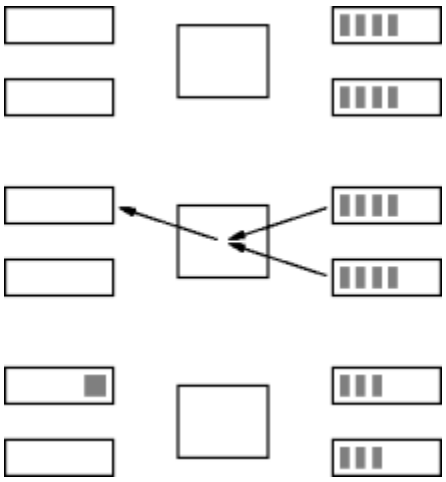
- Buffer out-of-order data to maintain ordered runs.



Replacement Selection

- Respecting order when building runs is called **replacement selection**.
- The primary-storage sort buffer becomes a cache for out-of-order data.
    - The cache is flushed (a new block is started) when it's full of out-of-order data.
- Merging is tricker given the unequal block sizes.

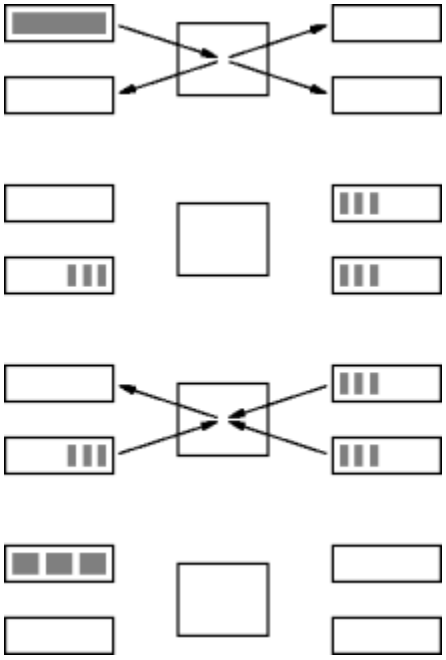
Balance Problems



- An *n*-way balanced sort merge wastes devices.
    - *n* - 1 output devices are unused during a pass.

- More devices in motion means faster sorting.

Unbalancing The Sort

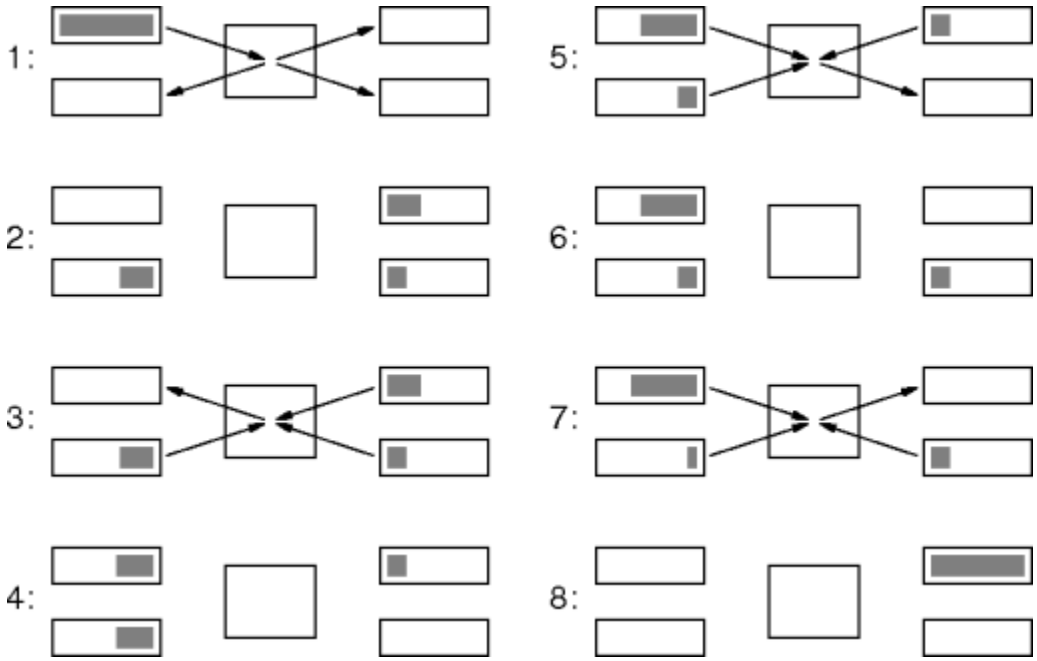


- An unbalanced sort merge can use all but one device during a merge.
  - But requires a distribute pass after each merge pass.

Improved Unbalancing

- Combine the merging and distribution passes.
  1. Merge the blocks until one of the input devices empties.
  2. Switch the output device to an input device.
  3. Switch the empty input device to the output device.
  4. Continue the merge.
- This strategy is called **polyphase merging**.

Polyphase Merge Illustrated



Properties

- For best results, polyphase merging requires
    - Nonuniform block sizes.
    - Nonuniform block distribution.
- Without nonuniformity, polyphase merging degenerates to merge-distributed passes.
- Replacement selection produces nonuniform block sizes.

Nonuniform Block Distribution

- Work backwards through a polyphase merge sort to determine initial block distributions.

D_1	D_2	D_3	D_4
0	13	11	7
7	6	4	0
3	2	0	4

- A four device example.
- These numbers are known as **generalized Fibonacci numbers**.

1	0	2	2
0	1	1	1
1	0	0	0

- **Dummy blocks** make the numbers come out right.

Analysis

- Analyzing polyphase merge sort is complicated.
- In general, polyphase merging is better than balanced *n*-way merging when there's a small (> 8) number of devices.
    - But an external sort is written for a specific system and task, not "in general".

UNIT-VI

Binary Search Trees

Binary Search Trees

1. Definition
   - All keys are distinct – why?
   - Key in the root of the left subtree is less than the root.
   - Key in the root of the right subtree is greater than the root.
   - Left and right subtrees are binary search trees.

- Figure 1 is an example:

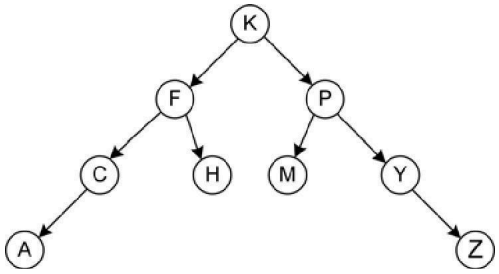


Figure 1 Binary Search Tree

- Where would you add X, B, and E?

- At seats, build tree from the nodes: B R L M T C A N P D

- What about deletion?
  - Since it is easier to delete from a leaf, we first replace the node with its inorder successor, and then delete the node that contained the inorder successor.

- The code for insertion is as follows. I like to create the new node outside the insert

routine.

- **Operations**:

- Search – O(*h*) , where *h* is the height of the tree. In the best case (balanced), *h* is

  - log *n* . In the worst case, *h* is *n* (the number of nodes in the tree).

- Insert – O(*h*) , where *h* is the height of the tree.

Delete – O(*h*) , where *h* is the height of the tree. What makes deletion difficult?

A *binary tree* is a special kind of tree, one that limits each node to no more than two children. A *binary search tree*, or BST, is a binary tree whose nodes are arranged such that for every node *n*, all of the nodes in *n*'s left subtree have a value less than *n*, and all nodes in *n*'s right subtree have a value greater than *n*. As we discussed, in the average case BSTs offer $\log_2 n$ asymptotic time for inserts, deletes, and searches. ($\log_2 n$ is often referred to as *sublinear* because it outperforms linear asymptotic times.)

The disadvantage of BSTs is that in the worst-case their asymptotic running time is reduced to linear time. This happens if the items inserted into the BST are inserted in order or in near-order. In such a case, a BST performs no better than an array. As we discussed at the end of Part 3, there exist self-balancing binary search trees, ones that ensure that, regardless of the order of the

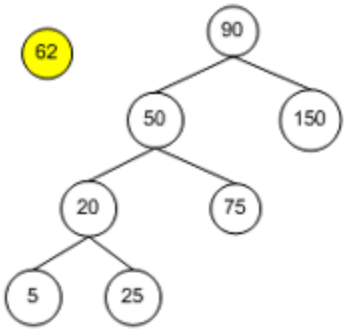
data inserted, the tree maintains a $\log_2 n$ running time. In this article, we'll briefly discuss two self-balancing binary search trees: AVL trees and red-black trees. Following that, we'll take an in-depth look at skip lists. Skip lists are a really neat data structure that is much easier to implement than AVL trees or red-black trees, yet still guarantees a running time of $\log_2 n$.

Self-Balancing Binary Search Trees

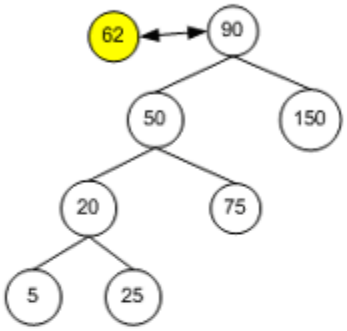
Recall that new nodes are inserted into a binary search tree at the leaves. That is, adding a node to a binary search tree involves tracing down a path of the binary search tree, taking left's and right's based on the comparison of the value of the current node and the node being inserted, until the path reaches a dead end. At this point, the newly inserted node is plugged into the tree at this reached dead end. Figure 1 illustrates the process of inserting a new node into a BST.

As Figure 1 shows, when making the comparison at the current node, the node to be inserted travels down the left path if its value is less than the current node, and down the right if its value is greater than the current's. Therefore, the structure of the BST is relative to the order with which the nodes are inserted. Figure 2 depicts a BST after nodes with values 20, 50, 90, 150, 175, and 200 have been added. Specifically, these nodes have been added in ascending order. The result is a BST with no breadth. That is, its topology consists of a single line of nodes rather than having the nodes fanned out.

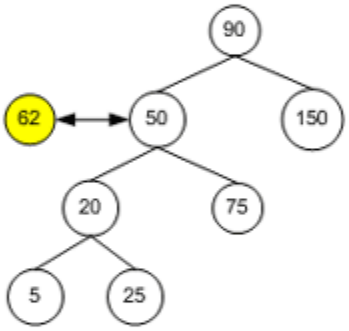
Given the following BST, we want to insert a node with the value 62...



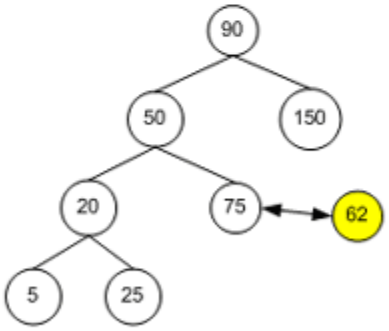
We start by comparing the node to insert (62) with the root (90). We see that 62 is less than 90, so we know 62 must be added somewhere to the root's left subtree.



We next compare 62 to 50. Since 62 is greater than 50, 62 must belong somewhere in 50's right subtree.



We next compare 62 to 75. Since 75 is greater than 62, 62 must exist somewhere in 75's left subtree.



Since 75's left child is a null reference, we have found the new location for node 62!

All that's left to do is set 75's left child to 62, which adds 62 to the BST tree and maintains the binary search tree property.

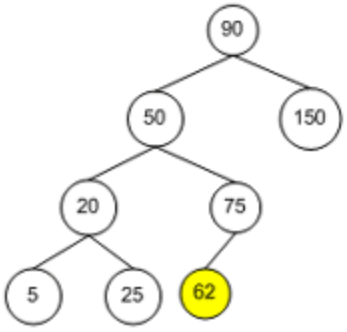


Figure 1. Inserting a new node into a BST

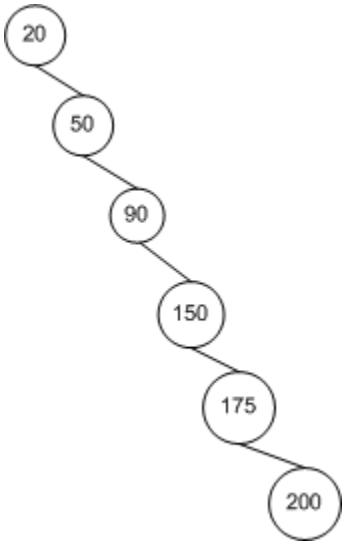


Figure 2. A BST after nodes with values of 20, 50, 90, 150, 175, and 200 have been added

BSTs—which offer sublinear running time for insertions, deletions, and searches—perform optimally when their nodes are arranged in a fanned out manner. This is because when searching for a node in a BST, each single step down the tree reduces the number of nodes that need to be potentially checked by one half. However, when a BST has a topology similar to the one in Figure 2, the running time for the BST's operations are much closer to linear time because each step down the tree only reduces the number of nodes that need to be searched by one. To see why, consider what must happen when searching for a particular value, such as 175. Starting at the root, 20, we must navigate down through each right child until we hit 175. That is, there is no savings in nodes that need to be checked at each step. Searching a BST like the one in Figure 2 is identical to searching an array—each element must be checked one at a time. Therefore, such a structured BST will exhibit a linear search time.

It is important to realize that the running time of a BST's operations is related to the BST's *height*. The height of a tree is defined as the length of the longest path starting at the root. The height of a tree can be defined recursively as follows:

- The height of a node with no children is 0.
- The height of a node with one child is the height of that child plus one.
- The height of a node with two children is one plus the greater height of the two children.

To compute the height of a tree, start at its leaf nodes and assign them a height of 0. Then move up the tree using the three rules outlined to compute the height of each leaf nodes' parent. Continue in this manner until every node of the tree has been labeled. The height of the tree, then, is the height of the root node. Figure 3 shows a number of binary trees with their height

computed at each node. For practice, take a second to compute the heights of the trees yourself to make sure your numbers match up with the numbers presented in the figure below.

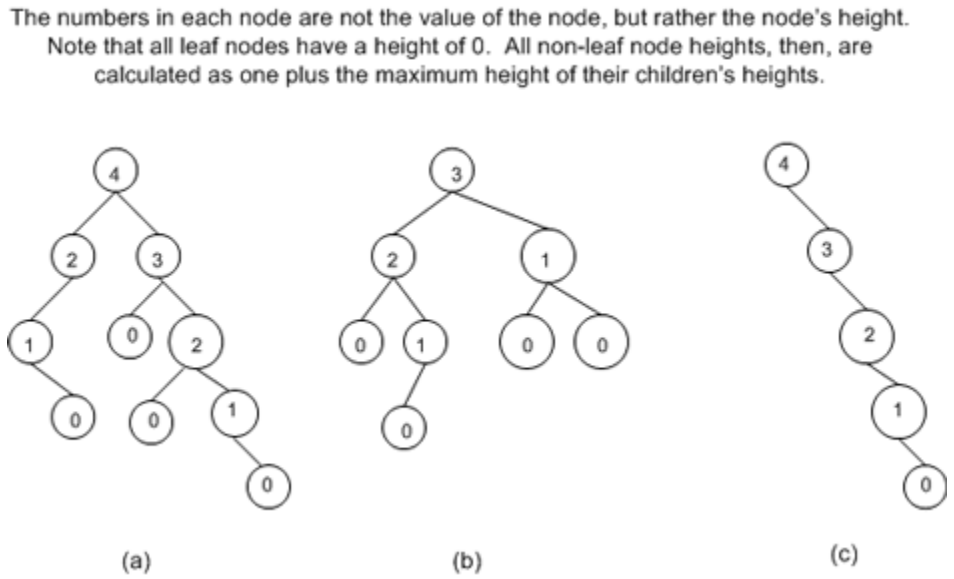


Figure 3. Example binary trees with their height computed at each node

A BST exhibits $\log_2 n$ running times when its height, when defined in terms of the number of nodes, n, in the tree, is near the floor of $\log_2 n$. (The floor of a number x is the greatest integer less than x. So, the floor of 5.38 would be 5 and the floor of 3.14159 would be 3. For positive numbers x, the floor of x can be found by simply truncating the decimal part of x, if any.) Of the three trees in Figure 3, tree (b) has the best height to number of nodes ratio, as the height is 3 and the number of nodes present in the tree is 8. As we discussed in Part 1 of this article series, $\log_a b = y$ is another way of writing $a^y = b$. $\log_2 8$, then, equals 3, because $2^3 = 8$. Tree (a) has 10 nodes and a height of 4. $\log_2 10$ equals 3.3219 and change, the floor of that being 3. So, 4 is not the ideal height. Notice that by rearranging the topology of tree (a)—by moving the far-bottom right node to the child of one of the non-leaf nodes with only one child—we could reduce the tree's height by one, thereby giving the tree an optimal height to node ratio. Finally, tree (c) has the worst height to node ratio. With its 5 nodes it could have an optimal height of 2, but due to its linear topology is has a height of 4.

The challenge we are faced with, then, is ensuring that the topology of the resulting BST exhibits an optimal ratio of height to the number of nodes. Because the topology of a BST is based upon the order in which the nodes are inserted, intuitively you might opt to solve this problem by ensuring that the data that's added to a BST is not added in near-sorted order. While this is possible if you know the data that will be added to the BST beforehand, it might not be practical. If you are not aware of the data that will be added—like if it's added based on user input, or is added as it's read from a sensor—then there is no hope of guaranteeing the data is not inserted in near-sorted order. The solution, then, is not to try to dictate the order with which the data is inserted, but to ensure that after each insertion the BST remains *balanced*. Data structures that are designed to maintain balance are referred to as *self-balancing binary search trees*.

A *balanced tree* is a tree that maintains some predefined ratio between its height and breadth. Different data structures define their own ratios for balance, but all have it close to $\log_2 n$. A self-balancing BST, then, exhibits $\log_2 n$ asymptotic running time. There are numerous self-balancing BST data structures in existence, such as AVL trees, red-black trees, 2-3 trees, 2-3-4 trees, splay trees, B-trees, and others. In the next two sections, we'll take a brief look at two of these self-balancing trees—AVL trees and red-black trees.

Balanced Trees

- Binary search trees are a good technique for finding elements in a dictionary.

- But it has a disadvantage:
  - In the worst case, the operations are still O(n) .

- There are applications that are time critical, and this amount of time is not acceptable.

- We will consider balanced trees (AVL, red-black, and AA trees) where the worst case operation is O(log n) .

- Go over recursion handout showing trace of recursion.

AVL Trees

- Balanced tree – the height of left subtree and height of right subtree differ by at most one.

- AVL trees are height balanced trees.

- The name comes from the people that came up with the idea – Adelson-Velskii and Landis.

- Definition:
  - An empty binary tree is an AVL tree.

  - If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is an AVL tree iff

    T_L and T_R are AVL trees and

$|h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_R, respectively.

- For several examples determine if AVL or not.

- How bad can a tree be and still be AVL?

- An *AVL search tree* is a binary search tree that is also an AVL tree.
- See Figure 2 for some possible examples. For each tree:
  - Is it an AVL tree?

  - Is it an AVL search tree?

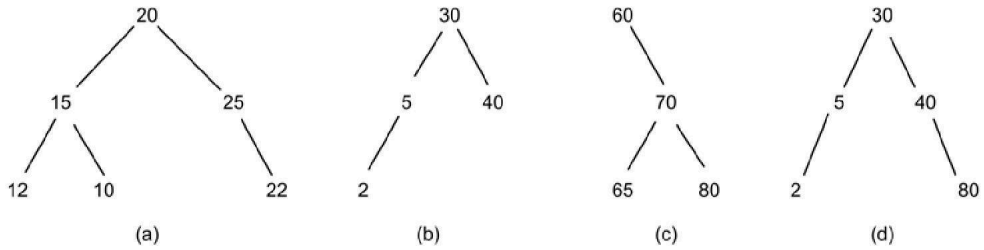


Figure 2 Possible AVL Trees and AVL Search Trees

- Properties of AVL trees:

  - The height of an AVL tree with n elements/node is $O(\log n)$.

  - For every value of n, $n \geq 0$, there exists an AVL tree.

  - An n-element AVL search tree can be searched in O(*height*) ● $O(\log n)$ time.

  - A new element can be inserted into an n-element AVL search tree so that the result is an $n+1$ element AVL tree and such an insertion can be done in $O(\log n)$

time.

- An element can be deleted from an n-element AVL search tree so that the result is an $n-1$ element AVL tree and such a deletion can be done in O(log n) time.

- From here on out, when I refer to an AVL tree, I mean an AVL search tree.

- The bound in the height of an AVL tree can be shown by the recursive routine shown below:

- worst_tree (h) = /* Worst tree of height h */
- root left = worst_tree(h-1)
- root right = worst_tree(h-2)

Searching

- How is this done? (Same way as with the binary search tree.)

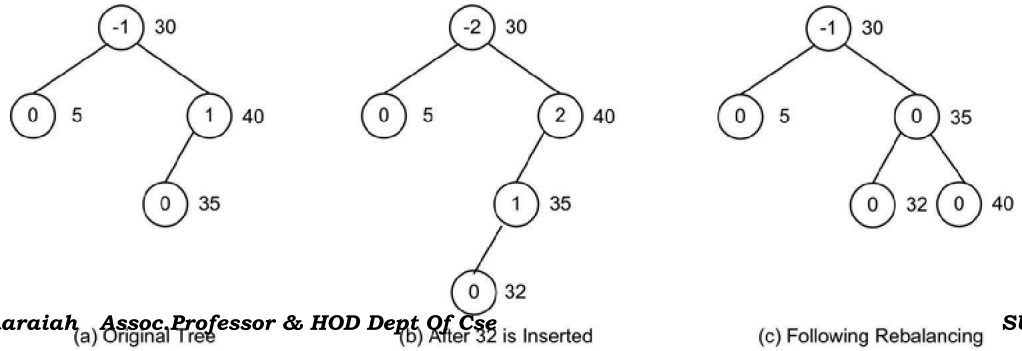
Insertion

- Insertions – just like BST, find the leaf where the node needs to go. After that node is placed in the AVL tree, the tree is rebalanced if necessary.

- We need to have a *balance factor* associated with each node x in the AVL tree.

height of left subtree of x – height of right subtree of x

- See Figure 3 for an example of inserting 32 into an AVL tree and the subsequent rebalancing. Notice that the balance factor is contained in each of the nodes shown in the tree.



(a) Original Tree (b) After 32 is Inserted (c) Following Rebalancing

Figure 3 Example of Inserting into an AVL-Tree

- Observations about the unbalanced tree that results from an insertion.
    - In the unbalanced tree, the balance factors are limited to –2, –1, 0, 1, and 2.
    - A node with balance factor 2 has a balance factor of 1 before the insertion. Similarly, a node with balance factor –2, has a balance factor of –1 before the insertion.
    - The balance factors of only those nodes on the path from the root to the newly inserted node can change as a result of the insertion.
    - Let *A* denote the nearest ancestor of the newly inserted node whose balance factor is either –2 or 2. The balance factor of all nodes on the path from *A* to the newly inserted node was 0 prior to the insertion.

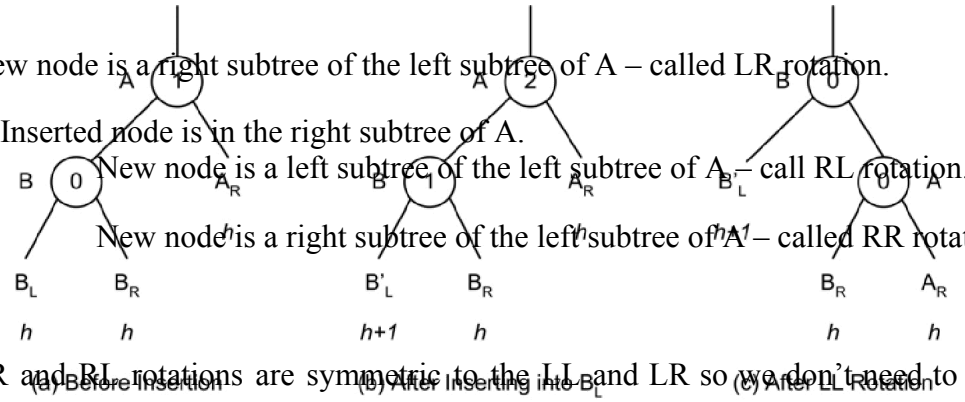
- In Figure 2(b), what is *A*? (Node 40.)
- There are four cases have to be consider when inserting into an AVL tree. We categorize these starting from A.
    - Inserted node is in the left subtree of A.
      New node is a left subtree of the left subtree of A – call LL rotation.

New node is a right subtree of the left subtree of A – called LR rotation.

o Inserted node is in the right subtree of A.

New node is a left subtree of the left subtree of A – call RL rotation.

New node is a right subtree of the left subtree of A – called RR rotation.



- The RR and RL rotations are symmetric to the LL and LR so we don't need to discuss them.

- A generic LL type imbalance appears in Figure 4.

Figure 4 LL Rotation

The balance factor b in the tree is defined below:

$b \bullet 0 \Rightarrow bf(B) \bullet bf(A) \bullet 0$ after rotation.

$b \bullet 1 \Rightarrow bf(B) \bullet 0$ and $bf(A) \bullet -1$ after rotation.

$b \bullet -1 \Rightarrow bf(B) \bullet 1$ and $bf(A) \bullet 0$ after rotation.

- The complexity of insertion into an AVL tree is O(*height*) ● O(log *n*) .

- *Notice that a single rotation is sufficient to restore balance if the insertion causes imbalance.*

- Consider the tree in Figure 6.
  - o Insert 70
  - o Insert 22

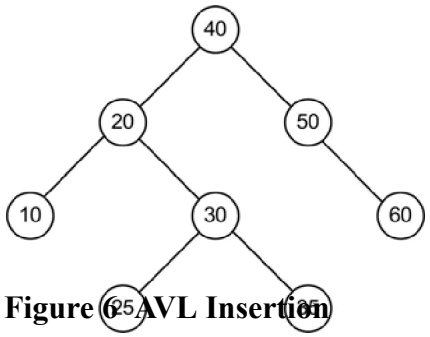


Figure 6 AVL Insertion

Deletion

- Deletions: Replace with inorder successor, delete, and rebalance.

- An R0 imbalance at A is rectified by performing the rotation shown in Figure 7.

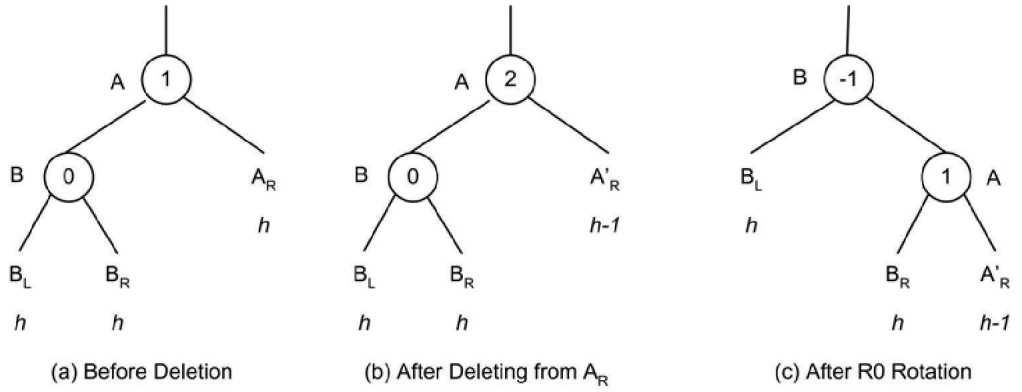


Figure 7 R0 Rotation

- Figure 8 shows how to handle an R1 imbalance

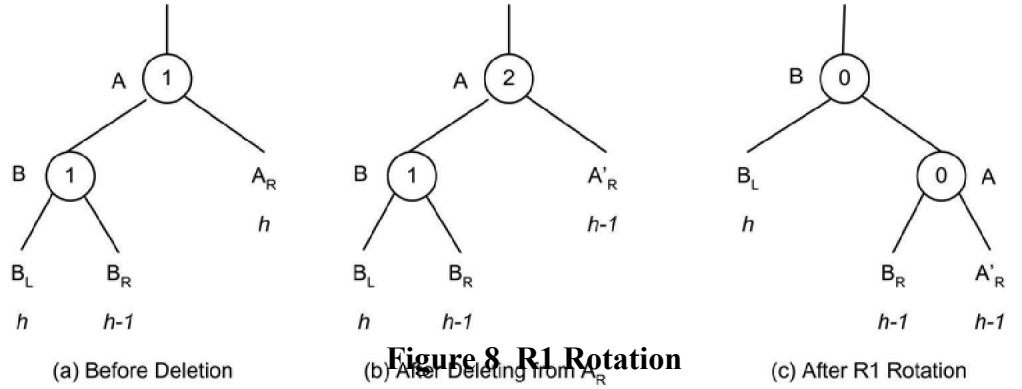


Figure 8 R1 Rotation

- Following an R1 rotation, we must continue to examine nodes on the path to the root.

  Unlike the case of an insertion, one rotation may not suffice to restore balance following a deletion. The number of rotations needed is $O(\log n)$.

- The transformation needed when the imbalance is of type R–1 appears in Figure 9.

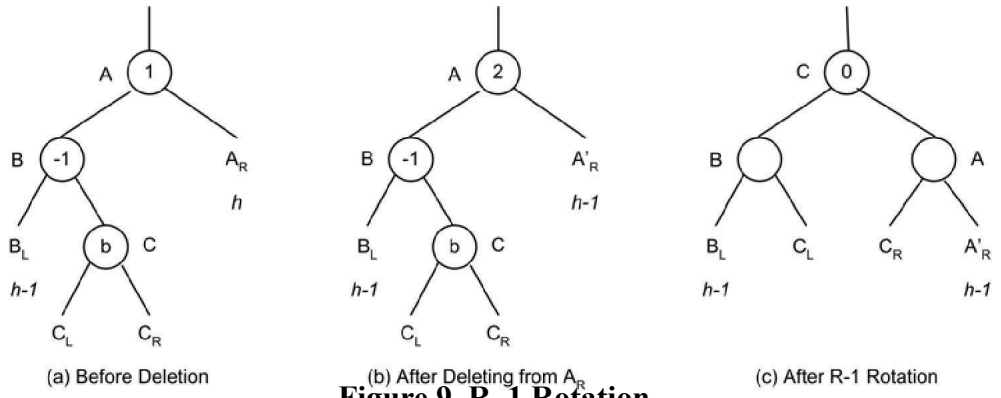


Figure 9 R–1 Rotation

- R-1 rotations may possibly need to continue to the root.

- See the LR rotation for the definition of b.

- What happens if AVL property is extended so left and right differ by 2 or 3 or *K*?
  - Termed height balanced tree, HB[*K*].

  - As *K* increases, worst case height increases, but time to rebalance is less.
  - There are symmetric rotations

- Consider the tree shown in Figure 10. We want to delete node 60

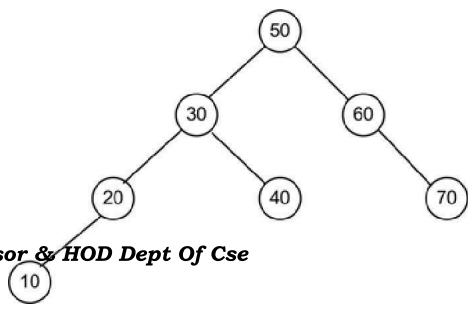


Figure 10 Delete node 60

UNIT-VII

B-tree

Definition

A B-tree of order m (the maximum number of children for each node) is a tree which satisfies the following properties:

1. Every node has at most m children.
2. Every node (except root) has at least $\lceil m/2 \rceil$ children.
3. The root has at least two children if it is not a leaf node.
4. All leaves appear in the same level, and carry information.
5. A non-leaf node with k children contains $k-1$ keys.

Each internal node's elements act as separation values which divide its subtrees. For example, if an internal node has three child nodes (or subtrees) then it must have two separation values or elements a_1 and a_2. All values in the leftmost subtree will be less than a_1 , all values in the middle subtree will be between a_1 and a_2, and all values in the rightmost subtree will be greater than a_2.

Internal nodes in a B-tree – nodes which are not leaf nodes – are usually represented as an ordered set of elements and child pointers. Every internal node contains a **maximum** of U children and – other than the root – a **minimum** of L children. For all internal nodes other than the root, the number of elements is one less than the number of child pointers; the number of elements is between $L-1$ and $U-1$. The number U must be either $2L$ or $2L-1$; thus each internal node is at least half full. This relationship between U and L implies that two half-full nodes can be joined to make a legal node, and one full node can be split into two legal nodes (if there is room to push one element up into the parent). These properties make it possible to delete and insert new values into a B-tree and adjust the tree to preserve the B-tree properties.

Leaf nodes have the same restriction on the number of elements, but have no children, and no child pointers.

The root node still has the upper limit on the number of children, but has no lower limit. For example, when there are fewer than $L-1$ elements in the entire tree, the root will be the only node in the tree, and it will have no children at all.

A B-tree of depth $n+1$ can hold about U times as many items as a B-tree of depth n, but the cost of search, insert, and delete operations grows with the depth of the tree. As with any balanced tree, the cost grows much more slowly than the number of elements.

Some balanced trees store values only at the leaf nodes, and so have different kinds of nodes for leaf nodes and internal nodes. B-trees keep values in every node in the tree, and may use the same structure for all nodes. However, since leaf nodes never have children, a specialized structure for leaf nodes in B-trees will improve performance.

Best case and worst case heights

The best case height of a B-Tree is:

$$\log_m n.$$

The worst case height of a B-Tree is:

$$\log_{m/2} n$$

Where *m* is the maximum number of children a node can have.

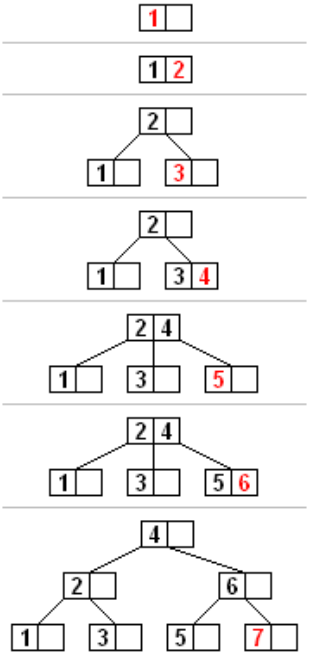
Algorithms

Warning: the discussion below uses "element", "value", "key", "separator", and "separation value" to mean essentially the same thing. The terms are not clearly defined. There are some subtle issues at the root and leaves.

Search

Searching is similar to searching a binary search tree. Starting at the root, the tree is recursively traversed from top to bottom. At each level, the search chooses the child pointer (subtree) whose separation values are on either side of the search value.

Binary search is typically (but not necessarily) used within nodes to find the separation values and child tree of interest.



Insertion

A B Tree insertion example with each iteration.

All insertions start at a leaf node. To insert a new element

Search the tree to find the leaf node where the new element should be added. Insert the new element into that node with the following steps:

1. If the node contains fewer than the maximum legal number of elements, then there is room for the new element. Insert the new element in the node, keeping the node's elements ordered.
2. Otherwise the node is full, so evenly split it into two nodes.
    1. A single median is chosen from among the leaf's elements and the new element.
    2. Values less than the median are put in the new left node and values greater than the median are put in the new right node, with the median acting as a separation value.
    3. Insert the separation value in the node's parent, which may cause it to be split, and so on. If the node has no parent (i.e., the node was the root), create a new root above this node (increasing the height of the tree).

If the splitting goes all the way up to the root, it creates a new root with a single separator value and two children, which is why the lower bound on the size of internal nodes does not apply to the root. The maximum number of elements per node is $U-1$. When a node is split, one element moves to the parent, but one element is added. So, it must be possible to divide the maximum number $U-1$ of elements into two legal nodes. If this number is odd, then $U=2L$ and one of the

new nodes contains $(U-2)/2 = L-1$ elements, and hence is a legal node, and the other contains one more element, and hence it is legal too. If $U-1$ is even, then $U=2L-1$, so there are $2L-2$ elements in the node. Half of this number is $L-1$, which is the minimum number of elements allowed per node.

An improved algorithm supports a single pass down the tree from the root to the node where the insertion will take place, splitting any full nodes encountered on the way. This prevents the need to recall the parent nodes into memory, which may be expensive if the nodes are on secondary storage. However, to use this improved algorithm, we must be able to send one element to the parent and split the remaining $U-2$ elements into two legal nodes, without adding a new element. This requires $U = 2L$ rather than $U = 2L-1$, which accounts for why some textbooks impose this requirement in defining B-trees.

Deletion

There are two popular strategies for deletion from a B-Tree.

- locate and delete the item, then restructure the tree to regain its invariants

or

- do a single pass down the tree, but before entering (visiting) a node, restructure the tree so that once the key to be deleted is encountered, it can be deleted without triggering the need for any further restructuring

The algorithm below uses the former strategy.

There are two special cases to consider when deleting an element:

1. the element in an internal node may be a separator for its child nodes
2. deleting an element may put its node under the minimum number of elements and children.

Each of these cases will be dealt with in order.

Deletion from a leaf node

- Search for the value to delete.
- If the value is in a leaf node, it can simply be deleted from the node,
- If underflow happens, check siblings to either transfer a key or fuse the siblings together.
- if deletion happened from right child retrieve the max value of left child if there is no underflow in left child
- in vice-versa situation retrieve the min element from right

Deletion from an internal node

Each element in an internal node acts as a separation value for two subtrees, and when such an element is deleted, two cases arise. In the first case, both of the two child nodes to the left and right of the deleted element have the minimum number of elements, namely $L-1$. They can then be joined into a single node with $2L-2$ elements, a number which does not exceed $U-1$ and so is a legal node. Unless it is known that this particular B-tree does not contain duplicate data, we must then also (recursively) delete the element in question from the new node.

In the second case, one of the two child nodes contains more than the minimum number of elements. Then a new separator for those subtrees must be found. Note that the largest element in the left subtree is still less than the separator. Likewise, the smallest element in the right subtree is the smallest element which is still greater than the separator. Both of those elements are in leaf nodes, and either can be the new separator for the two subtrees.

- If the value is in an internal node, choose a new separator (either the largest element in the left subtree or the smallest element in the right subtree), remove it from the leaf node it is in, and replace the element to be deleted with the new separator.
- This has deleted an element from a leaf node, and so is now equivalent to the previous case.

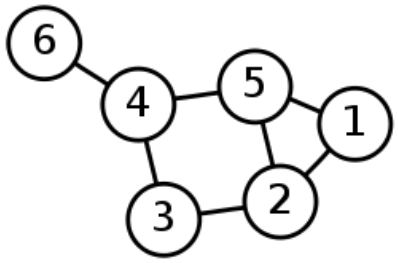
Rebalancing after deletion

If deleting an element from a leaf node has brought it under the minimum size, some elements must be redistributed to bring all nodes up to the minimum. In some cases the rearrangement will move the deficiency to the parent, and the redistribution must be applied iteratively up the tree, perhaps even to the root. Since the minimum element count doesn't apply to the root, making the root be the only deficient node is not a problem. The algorithm to rebalance the tree is as follows.

- If the right sibling has more than the minimum number of elements
    - Add the separator to the end of the deficient node.
    - Replace the separator in the parent with the first element of the right sibling.
    - Append the first child of the right sibling as the last child of the deficient node
- Otherwise, if the left sibling has more than the minimum number of elements.
    - Add the separator to the start of the deficient node.
    - Replace the separator in the parent with the last element of the left sibling.
    - Insert the last child of the left sibling as the first child of the deficient node
- If both immediate siblings have only the minimum number of elements
    - Create a new node with all the elements from the deficient node, all the elements from one of its siblings, and the separator in the parent between the two combined sibling nodes.
    - Remove the separator from the parent, and replace the two children it separated with the combined node.
    - If that brings the number of elements in the parent under the minimum, repeat these steps with that deficient node, unless it is the root, since the root is permitted to be deficient.

The only other case to account for is when the root has no elements and one child. In this case it is sufficient to replace it with its only child.

Graph (data structure)



A labeled graph of 6 vertices and 7 edges.

In computer science, a **graph** is an abstract data structure that is meant to implement the graph concept from mathematics.

A graph data structure consists mainly of a finite (and possibly mutable) set of ordered pairs, called **edges** or **arcs**, of certain entities called **nodes** or **vertices**. As in mathematics, an edge (x,y) is said to **point** or **go from** x **to** y. The nodes may be part of the graph structure, or may be external entities represented by integer indices or references.

A graph data structure may also associate to each edge some **edge value**, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).

Operations

The basic operations provided by a graph data structure G usually include:

- adjacent(G, x,y): tests whether there is an edge from node x to node y.
- neighbors(G, x): lists all nodes y such that there is an edge from x to y.
- add(G, x,y): adds to G the edge from x to y, if it is not there.
- delete(G, x,y): removes the edge from x to y, if it is there.
- get_node_value(G, x): returns the value associated with the node x.
- set_node_value(G, x, a): sets the value associated with the node x to a.

Structures that associate values to the edges usually also provide:

- get_edge_value(G, x,y): returns the value associated to the edge (x,y).
- set_edge_value(G, x,y,v): sets the value associated to the edge (x,y) to v.

Representations

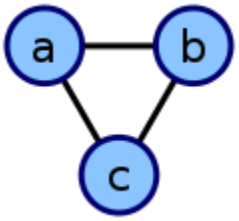
Different data structures for the representation of graphs are used in practice:

- **Adjacency list** - An adjacency list is implemented as an array of lists, with one list of destination nodes for each source node.
- **Incidence list** - A variant of the adjacency list that allows for the description of the edges at the cost of additional edges.
- **Adjacency matrix** - A two-dimensional Boolean matrix, in which the rows and columns represent source and destination vertices and entries in the matrix indicate whether an edge exists between the vertices associated with that row and column.
- **Incidence matrix** - A two-dimensional Boolean matrix, in which the rows represent the vertices and columns represent the edges. The array entries indicate if both are related, i.e. incident.

Adjacency lists are preferred for sparse graphs; otherwise, an adjacency matrix is a good choice.

For graphs with some regularity in the placement of edges, a symbolic graph is a possible choice of representation.

Adjacency list



This undirected cyclic graph can be described by the list {a,b}, {a,c}, {b,c}.

In graph theory, an **adjacency list** is the representation of all edges or arcs in a graph as a list.

If the graph is undirected, every entry is a set (or multiset) of two nodes containing the two ends of the corresponding edge; if it is directed, every entry is a tuple of two nodes, one denoting the source node and the other denoting the destination node of the corresponding arc.

Typically, adjacency lists are unordered.

Adjacency matrix

In mathematics and computer science, an **adjacency matrix** is a means of representing which vertices of a graph are adjacent to which other vertices. Another matrix representation for a graph is the incidence matrix.

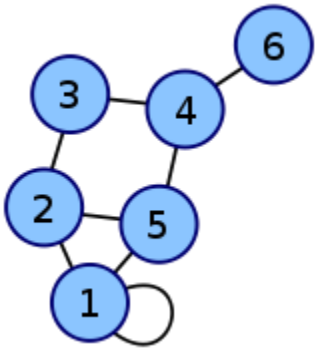
Specifically, the adjacency matrix of a finite graph **G** on n vertices is the $n \times n$ matrix where the non-diagonal entry a_{ij} is the number of edges from vertex i to vertex j, and the diagonal entry a_{ii}, depending on the convention, is either once or twice the number of edges (loops) from vertex i to itself. Undirected graphs often use the former convention of counting loops twice, whereas directed graphs typically use the latter convention. There exists a unique adjacency matrix for each isomorphism class of graphs (up to permuting rows and columns), and it is not the adjacency matrix of any other isomorphism class of graphs. In the special case of a finite simple graph, the adjacency matrix is a (0,1)-matrix with zeros on its diagonal. If the graph is undirected, the adjacency matrix is symmetric.

- The relationship between a graph and the eigenvalues and eigenvectors of its adjacency matrix is studied in spectral graph theory.

Examples

- Here is an example of a labeled graph and its adjacency matrix. The convention followed here is that an adjacent edge counts 1 in the matrix for an undirected graph. (X,Y coordinates are 1-6)

Labeled graph **Adjacency matrix**



$$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

- The adjacency matrix of a complete graph is all 1's except for 0's on the diagonal.
- The adjacency matrix of an empty graph is a zero matrix.

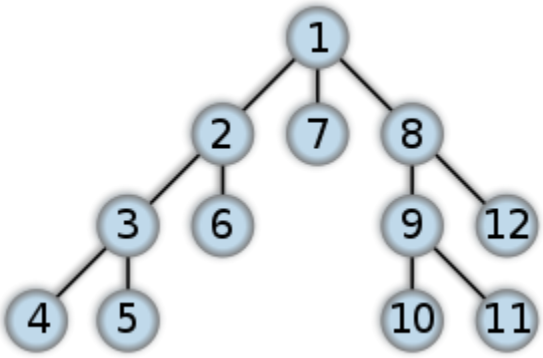
Algorithms

Graph algorithms are a significant field of interest within computer science. Typical higher-level operations associated with graphs are: finding a path between two nodes, like depth-first search and breadth-first search and finding the shortest path from one node to another, like Dijkstra's algorithm. A solution to finding the shortest path from each node to every other node also exists in the form of the Floyd-Warshall algorithm.

A directed graph can be seen as a flow network, where each edge has a capacity and each edge receives a flow. The Ford-Fulkerson algorithm is used to find out the maximum flow from a source to a sink in a graph.

Depth-first search

Depth-first search



Order in which the nodes are expanded

Class	Search algorithm
Data structure	Graph
Worst case performance	$O(\|V\| + \|E\|)$ for explicit graphs traversed without repetition, $O(b^d)$ for implicit graphs with branching factor b searched to depth d
Worst case space complexity	$O(\|V\|)$ if entire graph is traversed without repetition, O(longest path length searched) for implicit graphs without elimination of duplicate nodes

- **Depth-first search** (**DFS**) is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking

Formal definition

Formally, DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hasn't finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a stack for exploration.

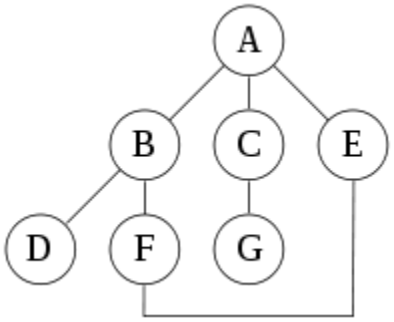
The time and space analysis of DFS differs according to its application area. In theoretical computer science, DFS is typically used to traverse an entire graph, and takes time O(|V| + |E|), linear in the size of the graph. In these applications it also uses space O(|V|) in the worst case to store the stack of vertices on the current search path as well as the set of already-visited vertices. Thus, in this setting, the time and space bounds are the same as for breadth first search and the choice of which of these two algorithms to use depends less on their complexity and more on the different properties of the vertex orderings the two algorithms produce.

For applications of DFS to search problems in artificial intelligence, however, the graph to be searched is often either too large to visit in its entirety or even infinite, and DFS may suffer from non-termination when the length of a path in the search tree is infinite. Therefore, the search is only performed to a limited depth, and due to limited memory availability one typically does not use data structures that keep track of the set of all previously visited vertices. In this case, the time is still linear in the number of expanded vertices and edges (although this number is not the same as the size of the entire graph because some vertices may be searched more than once and others not at all) but the space complexity of this variant of DFS is only proportional to the depth limit, much smaller than the space needed for searching to the same depth using breadth-first search.

For such applications, DFS also lends itself much better to heuristic methods of choosing a likely-looking branch. When an appropriate depth limit is not known a priori, iterative deepening depth-first search applies DFS repeatedly with a sequence of increasing limits; in the artificial intelligence mode of analysis, with a branching factor greater than one, iterative deepening increases the running time by only a constant factor over the case in which the correct depth limit is known due to the geometric growth of the number of nodes per level.

Example

For the following graph:



a depth-first search starting at A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously-visited nodes and will not repeat them (since this is a small graph), will visit the nodes in the following order: A, B, D, F, E, C, G.

Performing the same search without remembering previously visited nodes results in visiting nodes in the order A, B, D, F, E, A, B, D, F, E, etc. forever, caught in the A, B, D, F, E cycle and never reaching C or G.

Iterative deepening prevents this loop and will reach the following nodes on the following depths, assuming it proceeds left-to-right as above:

- 0: A
- 1: A (repeated), B, C, E

(Note that iterative deepening has now seen C, when a conventional depth-first search did not.)

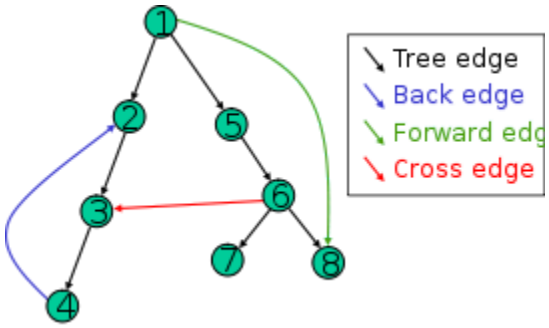
- 2: A, B, D, F, C, G, E, F

(Note that it still sees C, but that it came later. Also note that it sees E via a different path, and loops back to F twice.)

- 3: A, B, D, F, E, C, G, E, F, B

For this graph, as more depth is added, the two cycles "ABFE" and "AEFB" will simply get longer before the algorithm gives up and tries another branch.

Output of a depth-first search



The four types of edges defined by a spanning tree

The most natural result of a depth first search of a graph (if it is considered as a function rather than a procedure) is a spanning tree of the vertices reached during the search. Based on this spanning tree, the edges of the original graph can be divided into three classes: **forward edges**, which point from a node of the tree to one of its descendants, **back edges**, which point from a node to one of its ancestors, and **cross edges**, which do neither. Sometimes **tree edges**, edges which belong to the spanning tree itself, are classified separately from forward edges. It can be shown that if the original graph is undirected then all of its edges are tree edges or back edges.

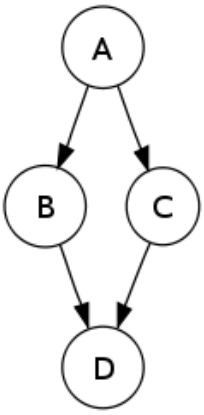
Vertex orderings

It is also possible to use the depth-first search to linearly order the vertices of the original graph (or tree). There are three common ways of doing this:

- A **preordering** is a list of the vertices in the order that they were first visited by the depth-first search algorithm. This is a compact and natural way of describing the progress of the search, as was done earlier in this article. A preordering of an expression tree is the expression in Polish notation.

- A **postordering** is a list of the vertices in the order that they were *last* visited by the algorithm. A postordering of an expression tree is the expression in reverse Polish notation.

- A **reverse postordering** is the reverse of a postordering, i.e. a list of the vertices in the opposite order of their last visit. When searching a tree, reverse postordering is the same as preordering, but in general they are different when searching a graph. For example, when searching the directed graph



beginning at node A, one visits the nodes in sequence, to produce lists either A B D B A C A, or A C D C A B A (depending upon the algorithm chooses to visit B or C first). Note that repeat visits in the form of backtracking to a node, to check if it has still unvisited neighbours, are included here (even if it is found to have none). Thus the possible preorderings are A B D C and A C D B (order by node's leftmost occurrence in above list), while the possible reverse postorderings are A C B D and A B C D (order by node's rightmost occurrence in above list). Reverse postordering produces a topological sorting of any directed acyclic graph. This ordering is also useful in control flow analysis as it often represents a natural linearization of the control flow. The graph above might represent the flow of control in a code fragment like

```
if (A) then {
  B
} else {
  C
}
D
```

and it is natural to consider this code in the order A B C D or A C B D, but not natural to use the order A B D C or A C D B.

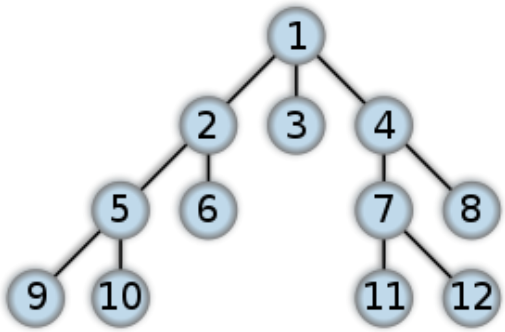
Applications

Algorithms that use depth-first search as a building block include:

- Finding connected components.
- Topological sorting.
- Finding 2-(edge or vertex)-connected components.
- Finding strongly connected components.
- Planarity Testing
- Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)
- bi connectivity.

Breadth-first search

Breadth-first search



Order in which the nodes are expanded

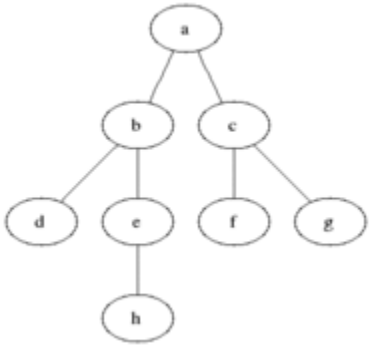
Class	Search algorithm
Data structure	Graph
Worst case performance	$O(\|V\| + \|E\|) = O(b^d)$
Worst case space complexity	$O(\|V\| + \|E\|) = O(b^d)$

In graph theory, **breadth-first search** (**BFS**) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.

How it works

BFS is an uninformed search method that aims to expand and examine all nodes of a graph or combination of sequences by systematically searching through every solution. In other words, it exhaustively searches the entire graph or sequence without considering the goal until it finds it. It does not use a heuristic algorithm.

From the standpoint of the algorithm, all child nodes obtained by expanding a node are added to a FIFO (i.e., First In, First Out) queue. In typical implementations, nodes that have not yet been examined for their neighbors are placed in some container (such as a queue or linked list) called "open" and then once examined are placed in the container "closed".



Animated example of a breadth-first search

Algorithm (informal)

1. Enqueue the root node.
2. Dequeue a node and examine it.
    - If the element sought is found in this node, quit the search and return a result.
    - Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.
3. If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
4. Repeat from Step 2.

Note: Using a stack instead of a queue would turn this algorithm into a depth-first search.

Pseudocode

```
1  procedure BFS(Graph,source):
2      create a queue Q
3      enqueue source onto Q
4      mark source
5      while Q is not empty:
6          dequeue an item from Q into v
7          for each edge e incident on v in Graph:
8              let w be the other end of e
9              if w is not marked:
10                 mark w
11                 enqueue w onto Q
```

[

Space complexity

Since all of the nodes of a level must be saved until their child nodes in the next level have been generated, the space complexity is proportional to the number of nodes at the deepest level. Given a branching factor b and graph depth d the asymptotic space complexity is the number of nodes at the deepest level, $O(b^d)$. When the number of vertices and edges in the graph are known ahead of time, the space complexity can also be expressed as $O(|E| + |V|)$ where $|E|$ is the cardinality of the set of edges (the number of edges), and $|V|$ is the cardinality of the set of vertices. In the worst case the graph has a depth of 1 and all vertices must be stored. Since it is exponential in the depth of the graph, breadth-first search is often impractical for large problems on systems with bounded space.

Time complexity

Since in the worst case breadth-first search has to consider all paths to all possible nodes the time complexity of breadth-first search is $1 + b + b^2 + b^3 + \cdots + b^d$ which is $O(b^d)$. The time complexity can also be expressed as $O(|E| + |V|)$ since every vertex and every edge will be explored in the worst case.

Completeness

Breadth-first search is complete. This means that if there is a solution, breadth-first search will find it regardless of the kind of graph. However, if the graph is infinite and there is no solution breadth-first search will diverge.

Proof of completeness

If the shallowest goal node is at some finite depth say d, breadth-first search will eventually find it after expanding all shallower nodes (provided that the branching factor b is finite).[1]

Optimality

For unit-step cost, breadth-first search is optimal. In general breadth-first search is not optimal since it always returns the result with the fewest edges between the start node and the goal node. If the graph is a weighted graph, and therefore has costs associated with each step, a goal next to the start does not have to be the cheapest goal available. This problem is solved by improving breadth-first search to uniform-cost search which considers the path costs. Nevertheless, if the graph is not weighted, and therefore all step costs are equal, breadth-first search will find the nearest and the best solution.

Bias towards nodes of high degree

It has been empirically observed (and analytically shown for random graphs) that incomplete breadth-first search is biased towards nodes of high degree. This makes a breadth-first search sample of a graph very difficult to interpret. For example, a breadth-first sample of 1 million nodes in Facebook (less than 1% of the entire graph) overestimates the average node degree by 240%.[2]

Applications

Breadth-first search can be used to solve many problems in graph theory, for example:

- Finding all nodes within one connected component
- Copying Collection, Cheney's algorithm
- Finding the shortest path between two nodes *u* and *v*
- Testing a graph for bipartiteness
- (Reverse) Cuthill–McKee mesh numbering
- Ford–Fulkerson method for computing the maximum flow in a flow network
- Serialization/Deserialization of a binary tree vs serialization in sorted order, allows the tree to be re-constructed in an efficient manner.

Finding connected components

The set of nodes reached by a BFS (breadth-first search) form the connected component containing the starting node.

Testing bipartiteness

BFS can be used to test bipartiteness, by starting the search at any vertex and giving alternating labels to the vertices visited during the search. That is, give label 0 to the starting vertex, 1 to all its neighbours, 0 to those neighbours' neighbours, and so on. If at any step a vertex has (visited) neighbours with the same label as itself, then the graph is not bipartite. If the search ends without such a situation occurring, then the graph is bipartite.

VIII

PATTERN MATCHING ALGORITHMS

Brute Force algorithm

Main features

- no preprocessing phase;
- constant extra space needed;
- always shifts the window by exactly 1 position to the right;
- comparisons can be done in any order;
- searching phase in $O(mn)$ time complexity;
- $2n$ expected text characters comparisons.

Description

The brute force algorithm consists in checking, at all positions in the text between 0 and n-m, whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern by exactly one position to the right.

The brute force algorithm requires no preprocessing phase, and a constant extra space in addition to the pattern and the text. During the searching phase the text character comparisons can be done in any order. The time complexity of this searching phase is $O(mn)$ (when searching for $a^{m-1}b$ in a^n for instance). The expected number of text character comparisons is $2n$.

The C code

```
void BF(char *x, int m, char *y, int n) {
  int i, j;

  /* Searching */
  for (j = 0; j <= n - m; ++j) {
    for (i = 0; i < m && x[i] == y[i + j]; ++i);
    if (i >= m)
      OUTPUT(j);
  }
}
```

This algorithm can be rewriting to give a more efficient algorithm in practice as follows:

```
#define EOS '\0'

void BF(char *x, int m, char *y, int n) {
 char *yb;
 /* Searching */
 for (yb = y; *y != EOS; ++y)
  if (memcmp(x, y, m) == 0)
    OUTPUT(y - yb);
}
```

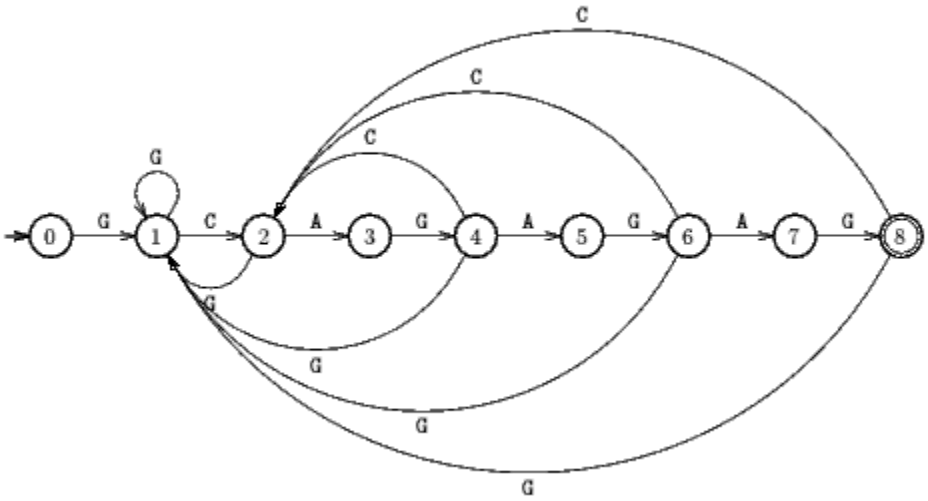
Research with an automaton

Main features

- builds the minimal deterministic automaton recognizing the language $\Sigma^* x$;
- extra space in $O(m\sigma)$ if the automaton is stored in a direct access table;
- preprocessing phase in $O(m\sigma)$ time complexity;
- searching phase in $O(n)$ time complexity if the automaton is stored in a direct access table, $O(n\log(\sigma))$ otherwise.

The example

Preprocessing phase



The states are labelled by the length of the prefix they are associated with.
Missing transitions are leading to the initial state 0.

Searching phase

Boyer-Moore

. Computer hardware has changed very considerably since that time yet the fundamental logic is still sound in a different world under very different conditions. Modern personal computers now have the capacity to work on very large sources and often have enough memory to handle those sources without requiring any form of tiling scheme and this capacity very well suits the design of the Boyer Moore exact pattern matching algorithm.

Usage would include tasks like recursively searching files for virus patterns, searching databases for keys or data, text and word processing and any other task that requires handling large amounts of data at very high speed.

How does it work ?

The following example is set up to search for the pattern within the source.

Source "This is a test of the Boyer Moore algorithm."
Pattern "algorithm"

Constructing A 256 member table is constructed that is initially filled with the length of the pattern which in this case is 9 characters. The 256 members represent the full range of characters in the ASCII character set. A second pass is then made on the table that places a descending count from the original length of the pattern in the ASCII table for each character that occurs.

algorithm <- pattern
87654321 <- shift values for each character

The table constructed in this manner allows the algorithm to determine in one access if the character being compared is within the search pattern or not. The first character compared is the end character of the pattern "m" to the corresponding position in the source.

```
          |
Source  "This is a test of the Boyer Moore algorithm."
Pattern "algorithm"
          |
```

The character being compared is "a" which is within the characters that are in the pattern. Character "a" has a shift of 8 so the pattern is shifted 8 characters right.

```
                |
Source  "This is a test of the Boyer Moore algorithm."
Pattern       "algorithm"
                |
```

The GOOD SUFFIX shift
The shift that was just performed has normally been called the GOOD SUFFIX shift. The next character being compared is "f" which is not within the patern and this requires a different strategy to handle the shift. The logic of the Boyer Moore design is that if a character is compared that is not within the characters that are in the pattern, no match can be found by comparing any further characters at this position so the pattern can be shifted completely past the mismatching character.

```
                    |
Source  "This is a test of the Boyer Moore algorithm."
Pattern             "algorithm"
                    |
```

The BAD CHARACTER shift
This shift is usually called the BAD CHARACTER shift and it is calculated in a different manner. The table for the BAD CHARACTER shift occurs in this form.

```
Pattern "algorithm"
        123456789
```

Some of the older implementation have constructed a second table but there is a far more efficient way to do it. The loop that does the reverse comparisons must have a loop counter and this loop counter produces the descending shift value that can be used to perform the BAD CHARACTER shift. The above shift that mismatched on the character "f" occurred on the first comparison after the shift so the shift past the mismatching character is 9.

The following character compared in the source is "e" which is also not in the table and it produces a BAD CHARACTER shift on the first position of the comparison so the shift is also 9.

```
                        |
Source  "This is a test of the Boyer Moore algorithm."
Pattern                 "algorithm"
                        |
```

The next mismatch is "a" which occurs in the search pattern. This lines up with the first character of the pattern being searched for. The shift value in the table for the character "a" is 8 which will produce the match being searched for.

```
                            |
Source  "This is a test of the Boyer Moore algorithm."
Pattern                     "algorithm"
                            |
```

The reverse comparison loop will compare all of the characters in the pattern to the position in the source and will produce a match at the end.

The additional heuristic
The is the need to produce an addition heuristic to handle the occurrence of repeated sequences of characters which is common in both plain text and binary files. What will happen if there is a

repeated sequence of characters that are within the characters used in the search pattern is that the value in the table for the GOOD SUFFIX shift will produce a shift that goes past the correct match.

```
    |
xxxxBooooxxxx
 Boooo
    |
```

When the GOOD SUFFIX shift table is constructed from the characters in a pattern that has repeated sequences, the repeat character values are overwritten at the same position in the table so you do not have an incremental decrementing of the values for each character position.

```
Boooo
4111
```

If the GOOD SUFFIX shift is applied with these values in the table, the pattern will be shifted past the correct match.

```
      |
xxxxBooooxxxx
   Boooo
      |
```

The additional heuristic calculates the number of comparisons made in the current position and subtracts that from the GOOD SUFFIX shift. This will produce the correct match in most instances but the subtraction can also produce 0 so to ensure that a shift is applied in the worst case, if the calculated value is less than 1, a minimum shift of one is performed.

```
   |
xxxxBooooxxxx
 Boooo
   |
```

Two comparisons have been made at this position so the GOOD SUFFIX shift of 4 for "B" has 2 subtracted from it to produce a shift of 2 characters.

```
      |
xxxxBooooxxxx
  Boooo
      |
```

The two variations

The two variations supplied with the complete Boyer Moore algorithm are based on the two different shifts that the original uses, the SBM.ASM version uses the GOOD SUFFIX shift from the table without the BAD CHARACTER shift, the BMH.ASM uses the BAD CHARACTER shift and simply increments the location if the character occurs within the table.

Both will produce reasonable performance because they have less overhead than the original. Their performance in loop speed must be offset against the lack of the extra logic that is used in the original algorithm.

The SBM.ASM version is generally faster on older machines and some AMD machines because the shorter pipeline with a lower penalty for register stalls better suits the code design.

The BMH.ASM version is faster on a late model Intel machine because it better suits the longer pipeline of later Intel processors and is less prone to branch prediction buffer penalties. This version will show much slower speeds if the character range is limited to characters that are within the search pattern as it only does a single increment in that situation.

The original algorithm BM.ASM averages better across x86 processors and is less vulnerable in worst case situations. The range of variation between all three versions is about 10%. The original version is within a few percent of the fastest tests on both types of machines and this reflects its dependence on logic rather than just fast loop code.

Boyer-Moore algorithm

Main features

- performs the comparisons from right to left;
- preprocessing phase in $O(m+\sigma)$ time and space complexity;
- searching phase in $O(mn)$ time complexity;
- $3n$ text character comparisons in the worst case when searching for a non periodic pattern;
- $O(n / m)$ best performance.

Description

The Boyer-Moore algorithm is considered as the most efficient string-matching algorithm in usual applications. A simplified version of it or the entire algorithm is often implemented in text editors for the «search» and «substitute» commands.

The algorithm scans the characters of the pattern from right to left beginning with the rightmost one. In case of a mismatch (or a complete match of the whole pattern) it uses two precomputed functions to shift the window to the right. These two shift functions are called the *good-suffix shift* (also called matching shift and the *bad-character shift* (also called the occurrence shift).

Assume that a mismatch occurs between the character $x[i]=a$ of the pattern and the character $y[i+j]=b$ of the text during an attempt at position j.
Then, $x[i+1 .. m-1]=y[i+j+1 .. j+m-1]=u$ and $x[i] \neq y[i+j]$. The good-suffix shift consists in aligning the segment $y[i+j+1 .. j+m-1]=x[i+1 .. m-1]$ with its rightmost occurrence in x that is preceded by a character different from $x[i]$ (*see figure*).

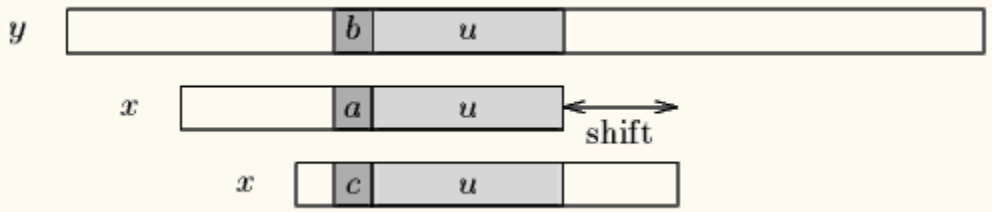


Figure . The good-suffix shift, *u* re-occurs preceded by a character *c* different from *a*.

If there exists no such segment, the shift consists in aligning the longest suffix v of $y[i+j+1 .. j+m-1]$ with a matching prefix of x (*see figure*).

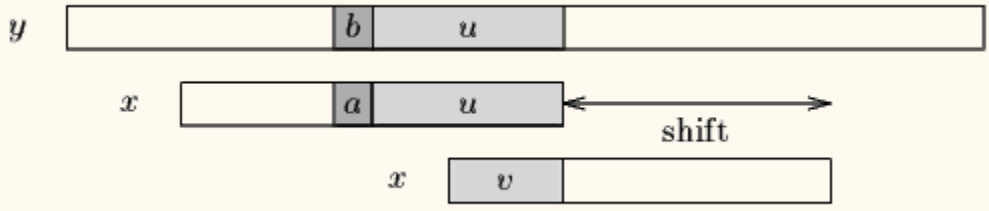


Figure . The good-suffix shift, only a suffix of *u* re-occurs in *x*.

The bad-character shift consists in aligning the text character $y[i+j]$ with its rightmost occurrence in $x[0 .. m-2]$. (*see figure*)

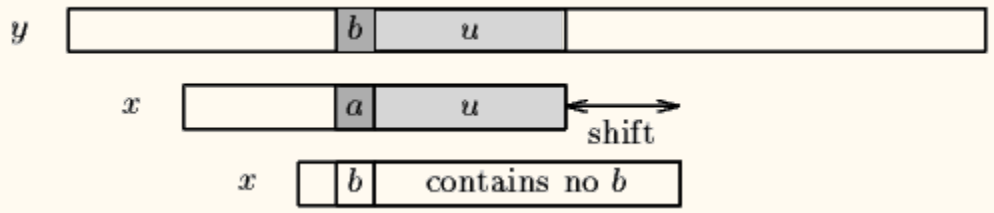


Figure . The bad-character shift, *a* occurs in *x*.

If *y*[*i*+*j*] does not occur in the pattern *x*, no occurrence of *x* in *y* can include *y*[*i*+*j*], and the left end of the window is aligned with the character immediately after *y*[*i*+*j*], namely *y*[*i*+*j*+1] (*see figure*).

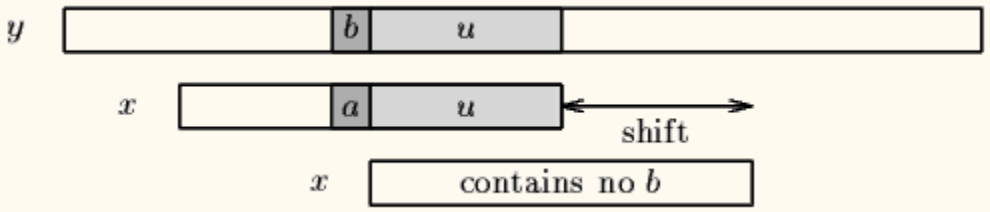


Figure . The bad-character shift, *b* does not occur in *x*.

Note that the bad-character shift can be negative, thus for shifting the window, the Boyer-Moore algorithm applies the maximum between the the good-suffix shift and bad-character shift. More formally the two shift functions are defined as follows.

The good-suffix shift function is stored in a table *bmGs* of size *m*+1.

Let us define two conditions:

i, *s*): for each *k* such that $i < k < m$, $s \geqslant k$ or $x[k-s]=x[k]$

i, *s*): if $s < i$ then $x[i-s] \neq$
]

Then, for $0 \leqslant i < m$: $bmGs[i+1]=\min\{s>0 : Cs(i, s)$ and $Co(i, s)$ hold$\}$
and we define *bmGs*[0] as the length of the period of *x*. The computation of the table *bmGs* use a table *suff* defined as follows: for $1 \leqslant i < m$, $suff[i]=\max\{k : x[i-k+1 .. i]=x[m-k .. m-1]\}$

The bad-character shift function is stored in a table *bmBc* of size σ. For *c* in Σ: $bmBc[c] = \min\{i : 1 \leqslant i < m-1$ and $x[m-1-i]=c\}$ if *c* occurs in *x*, *m* otherwise.

Tables *bmBc* and *bmGs* can be precomputed in time $O(m+\sigma)$ before the searching phase and require an extra-space in $O(m+\sigma)$. The searching phase time complexity is quadratic but at most $3n$ text character comparisons are performed when searching for a non periodic pattern. On large alphabets (relatively to the length of the pattern) the algorithm is extremely fast. When searching for $a^{m-1}b$ in b^n the algorithm makes only $O(n / m)$ comparisons, which is the absolute minimum for any string-matching algorithm in the model where the pattern only is preprocessed.

The C code

```
void preBmBc(char *x, int m, int bmBc[]) {
  int i;

  for (i = 0; i < ASIZE; ++i)
    bmBc[i] = m;
  for (i = 0; i < m - 1; ++i)
    bmBc[x[i]] = m - i - 1;
}


void suffixes(char *x, int m, int *suff) {
  int f, g, i;

  suff[m - 1] = m;
  g = m - 1;
  for (i = m - 2; i >= 0; --i) {
    if (i > g && suff[i + m - 1 - f] < i - g)
      suff[i] = suff[i + m - 1 - f];
    else {
      if (i < g)
        g = i;
      f = i;
      while (g >= 0 && x[g] == x[g + m - 1 - f])
        --g;
      suff[i] = f - g;
    }
  }
}

void preBmGs(char *x, int m, int bmGs[]) {
  int i, j, suff[XSIZE];

  suffixes(x, m, suff);

  for (i = 0; i < m; ++i)
    bmGs[i] = m;
  j = 0;
```

```
  for (i = m - 1; i >= 0; --i)
    if (suff[i] == i + 1)
      for (; j < m - 1 - i; ++j)
        if (bmGs[j] == m)
          bmGs[j] = m - 1 - i;
  for (i = 0; i <= m - 2; ++i)
    bmGs[m - 1 - suff[i]] = m - 1 - i;
}


void BM(char *x, int m, char *y, int n) {
  int i, j, bmGs[XSIZE], bmBc[ASIZE];

  /* Preprocessing */
  preBmGs(x, m, bmGs);
  preBmBc(x, m, bmBc);

  /* Searching */
  j = 0;
  while (j <= n - m) {
    for (i = m - 1; i >= 0 && x[i] == y[i + j]; --i);
    if (i < 0) {
      OUTPUT(j);
      j += bmGs[0];
    }
    else
      j += MAX(bmGs[i], bmBc[y[i + j]] - m + 1 + i);
  }
}
```

The example

Preprocessing phase

c	A	C	G	T
$bmBc[c]$	1	6	2	8

i	0	1	2	3	4	5	6	7
$x[i]$	G	C	A	G	A	G	A	G
$suff[i]$	1	0	0	2	0	4	0	8
$bmGs[i]$	7	7	7	2	7	4	7	1

bmBc and *bmGs* tables used by Boyer-Moore algorithm

Knuth-Morris-Pratt string matching

The problem: given a (short) pattern and a (long) text, both strings, determine whether the pattern appears somewhere in the text. Last time we saw how to do this with finite automata. This time we'll go through the Knuth-Morris-Pratt (KMP) algorithm, which can be thought of as an efficient way to build these automata. I also have some working C++ source code which might help you understand the algorithm better.

First let's look at a naive solution.

suppose the text is in an array: char T[n]
and the pattern is in another array: char P[m].

One simple method is just to try each possible position the pattern could appear in the text.

Naive string matching:

```
for (i=0; T[i] != '\0'; i++)
{
for (j=0; T[i+j] != '\0' && P[j] != '\0' && T[i+j]==P[j]; j++) ;
if (P[j] == '\0') found a match
}
```

There are two nested loops; the inner one takes O(m) iterations and the outer one takes O(n) iterations so the total time is the product, O(mn). This is slow; we'd like to speed it up.

In practice this works pretty well -- not usually as bad as this O(mn) worst case analysis. This is because the inner loop usually finds a mismatch quickly and move on to the next position without going through all m steps. But this method still can take O(mn) for some inputs. In one bad example, all characters in T[] are "a"s, and P[] is all "a"'s except for one "b" at the end. Then it takes m comparisons each time to discover that you don't have a match, so mn overall.

Here's a more typical example. Each row represents an iteration of the outer loop, with each character in the row representing the result of a comparison (X if the comparison was unequal). Suppose we're looking for pattern "nano" in text "bananananobano".

```
 0 1 2 3 4 5 6 7 8 9 10 11
  T: b a n a n a n o b a n o

i=0: X
i=1:   X
i=2:     n a n X
i=3:       X
i=4:         n a n o
i=5:           X
i=6:             n X
```

```
i=7:                X
i=8:                  X
i=9:                    n  X
i=10:                     X
```

Some of these comparisons are wasted work! For instance, after iteration i=2, we know from the comparisons we've done that T[3]="a", so there is no point comparing it to "n" in iteration i=3. And we also know that T[4]="n", so there is no point making the same comparison in iteration i=4.

Skipping outer iterations

The Knuth-Morris-Pratt idea is, in this sort of situation, after you've invested a lot of work making comparisons in the inner loop of the code, you know a lot about what's in the text. Specifically, if you've found a partial match of j characters starting at position i, you know what's in positions T[i]...T[i+j-1].

You can use this knowledge to save work in two ways. First, you can skip some iterations for which no match is possible. Try overlapping the partial match you've found with the new match you want to find:

```
i=2: n  a  n
i=3:    n  a  n  o
```

Here the two placements of the pattern conflict with each other -- we know from the i=2 iteration that T[3] and T[4] are "a" and "n", so they can't be the "n" and "a" that the i=3 iteration is looking for. We can keep skipping positions until we find one that doesn't conflict:

```
i=2: n  a  n
i=4:       n  a  n  o
```

Here the two "n"'s coincide. Define the *overlap* of two strings x and y to be the longest word that's a suffix of x and a prefix of y. Here the overlap of "nan" and "nano" is just "n". (We don't allow the overlap to be all of x or y, so it's not "nan"). In general the value of i we want to skip to is the one corresponding to the largest overlap with the current partial match:

String matching with skipped iterations:

```
i=0;
while (i<n)
{
for (j=0; T[i+j] != '\0' && P[j] != '\0' && T[i+j]==P[j]; j++) ;
if (P[j] == '\0') found a match;
i = i + max(1, j-overlap(P[0..j-1],P[0..m]));
}
```

Skipping inner iterations

The other optimization that can be done is to skip some iterations in the inner loop. Let's look at the same example, in which we skipped from i=2 to i=4:

```
i=2: n  a  n
i=4:       n  a  n  o
```
In this example, the "n" that overlaps has already been tested by the i=2 iteration. There's no need to test it again in the i=4 iteration. In general, if we have a nontrivial overlap with the last partial match, we can avoid testing a number of characters equal to the length of the overlap.

This change produces (a version of) the KMP algorithm:

KMP, version 1:

```
i=0;
o=0;
while (i<n)
{
for (j=o; T[i+j] != '\0' && P[j] != '\0' && T[i+j]==P[j]; j++) ;
if (P[j] == '\0') found a match;
o = overlap(P[0..j-1],P[0..m]);
i = i + max(1, j-o);
}
```
The only remaining detail is how to compute the overlap function. This is a function only of j, and not of the characters in T[], so we can compute it once in a *preprocessing* stage before we get to this part of the algorithm. First let's see how fast this algorithm is.

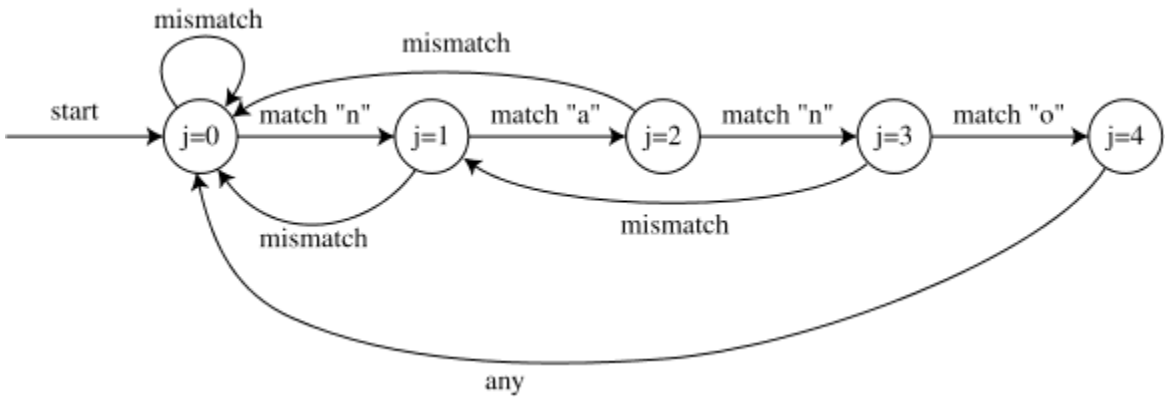
KMP time analysis

We still have an outer loop and an inner loop, so it looks like the time might still be O(mn). But we can count it a different way to see that it's actually always less than that. The idea is that every time through the inner loop, we do one comparison T[i+j]==P[j]. We can count the total time of the algorithm by counting how many comparisons we perform.

We split the comparisons into two groups: those that return true, and those that return false. If a comparison returns true, we've determined the value of T[i+j]. Then in future iterations, as long as there is a nontrivial overlap involving T[i+j], we'll skip past that overlap and not make a comparison with that position again. So each position of T[] is only involved in one true comparison, and there can be n such comparisons total. On the other hand, there is at most one false comparison per iteration of the outer loop, so there can also only be n of those. As a result we see that this part of the KMP algorithm makes at most 2n comparisons and takes time O(n).

KMP and finite automata

If we look just at what happens to j during the algorithm above, it's sort of like a finite automaton. At each step j is set either to j+1 (in the inner loop, after a match) or to the overlap o (after a mismatch). At each step the value of o is just a function of j and doesn't depend on other information like the characters in T[]. So we can draw something like an automaton, with arrows connecting values of j and labeled with matches and mismatches.



The difference between this and the automata we are used to is that it has only two arrows out of each circle, instead of one per character. But we can still simulate it just like any other automaton, by placing a marker on the start state (j=0) and moving it around the arrows. Whenever we get a matching character in T[] we move on to the next character of the text. But whenever we get a mismatch we look at the same character in the next step, except for the case of a mismatch in the state j=0.

So in this example (the same as the one above) the automaton goes through the sequence of states:

```
j=0
      mismatch T[0] != "n"
j=0
      mismatch T[1] != "n"
j=0
      match T[2] == "n"
j=1
      match T[3] == "a"
j=2
      match T[4] == "n"
j=3
      mismatch T[5] != "o"
j=1
      match T[5] == "a"
j=2
      match T[6] == "n"
j=3
      match T[7] == "o"
j=4
      found match
j=0
      mismatch T[8] != "n"
j=0
      mismatch T[9] != "n"
j=0
```

match T[10] == "n"

j=1

mismatch T[11] != "a"

j=0

mismatch T[11] != "n"

This is essentially the same sequence of comparisons done by the KMP pseudocode above. So this automaton provides an equivalent definition of the KMP algorithm.

As one student pointed out in lecture, the one transition in this automaton that may not be clear is the one from j=4 to j=0. In general, there should be a transition from j=m to some smaller value of j, which should happen on any character (there are no more matches to test before making this transition). If we want to find all occurrences of the pattern, we should be able to find an occurrence even if it overlaps another one. So for instance if the pattern were "nana", we should find both occurrences of it in the text "nanana". So the transition from j=m should go to the next longest position that can match, which is simply j=overlap(pattern,pattern). In this case overlap("nano","nano") is empty (all suffixes of "nano" use the letter "o", and no prefix does) so we go to j=0.

Alternate version of KMP

The automaton above can be translated back into pseudo-code, looking a little different from the pseudo-code we saw before but performing the same comparisons.

KMP, version 2:

```
j = 0;
for (i = 0; i < n; i++)
for (;;) {      // loop until break
   if (T[i] == P[j]) { // matches?
   j++;       // yes, move on to next state
   if (j == m) {   // maybe that was the last state
      found a match;
      j = overlap[j];
   }
   break;
   } else if (j == 0) break;   // no match in state j=0, give up
   else j = overlap[j];    // try shorter partial match
}
```

The code inside each iteration of the outer loop is essentially the same as the function match from the C++ implementation I've made available. One advantage of this version of the code is that it tests characters one by one, rather than performing random access in the T[] array, so (as in the implementation) it can be made to work for stream-based input rather than having to read the whole text into memory first.

The overlap[j] array stores the values of overlap(pattern[0..j-1],pattern), which we still need to show how to compute.

Since this algorithm performs the same comparisons as the other version of KMP, it takes the same amount of time, O(n). One way of proving this bound directly is to note, first, that there is one true comparison (in which T[i]==P[j]) per iteration of the outer loop, since we break out of the inner loop when this happens. So there are n of these total. Each of these comparisons results in increasing j by one. Each iteration of the inner loop in which we don't break out of the loop results in executing the statement j=overlap[j], which decreases j. Since j can only decrease as many times as it's increased, the total number of times this happens is also O(n).

Computing the overlap function

Recall that we defined the *overlap* of two strings x and y to be the longest word that's a suffix of x and a prefix of y. The missing component of the KMP algorithm is a computation of this overlap function: we need to know overlap(P[0..j-1],P) for each value of j>0. Once we've computed these values we can store them in an array and look them up when we need them.

To compute these overlap functions, we need to know for strings x and y not just the longest word that's a suffix of x and a prefix of y, but all such words. The key fact to notice here is that if w is a suffix of x and a prefix of y, and it's not the longest such word, then it's also a suffix of overlap(x,y). (This follows simply from the fact that it's a suffix of x that is shorter than overlap(x,y) itself.) So we can list all words that are suffixes of x and prefixes of y by the following loop:

```
while (x != empty) {
x = overlap(x,y);
output x;
}
```

Now let's make another definition: say that shorten(x) is the prefix of x with one fewer character. The next simple observation to make is that shorten(overlap(x,y)) is still a prefix of y, but is also a suffix of shorten(x).

So we can find overlap(x,y) by adding one more character to some word that's a suffix of shorten(x) and a prefix of y. We can just find all such words using the loop above, and return the first one for which adding one more character produces a valid overlap:

Overlap computation:

```
z = overlap(shorten(x),y)
while (last char of x != y[length(z)])
{
if (z = empty) return overlap(x,y) = empty
else z = overlap(z,y)
}
return overlap(x,y) = z
```

So this gives us a recursive algorithm for computing the overlap function in general. If we apply this algorithm for x=some prefix of the pattern, and y=the pattern itself, we see that all recursive calls have similar arguments. So if we store each value as we compute it, we can look it up instead of

computing it again. (This simple idea of storing results instead of recomputing them is known as *dynamic programming*; we discussed it somewhat in the first lecture and will see it in more detail next time.)

So replacing x by P[0..j-1] and y by P[0..m-1] in the pseudocode above and replacing recursive calls by lookups of previously computed values gives us a routine for the problem we're trying to solve, of computing these particular overlap values. The following pseudocode is taken (with some names changed) from the initialization code of the C++ implementation I've made available. The value in overlap[0] is just a flag to make the rest of the loop simpler. The code inside the for loop is the part that computes each overlap value.

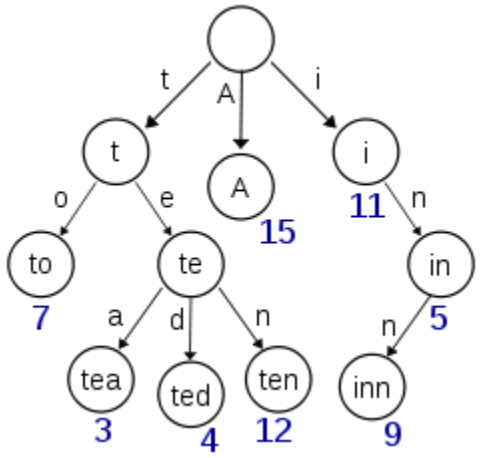
KMP overlap computation:

```
overlap[0] = -1;
for (int i = 0; pattern[i] != '\0'; i++) {
overlap[i + 1] = overlap[i] + 1;
while (overlap[i + 1] > 0 &&
      pattern[i] != pattern[overlap[i + 1] - 1])
   overlap[i + 1] = overlap[overlap[i + 1] - 1] + 1;
}
return overlap;
```

Let's finish by analyzing the time taken by this part of the KMP algorithm. The outer loop executes m times. Each iteration of the inner loop decreases the value of the formula overlap[i+1], and this formula's value only increases by one when we move from one iteration of the outer loop to the next. Since the number of decreases is at most the number of increases, the inner loop also has at most m iterations, and the total time for the algorithm is O(m).

The entire KMP algorithm consists of this overlap computation followed by the main part of the algorithm in which we scan the text (using the overlap values to speed up the scan). The first part takes O(m) and the second part takes O(n) time, so the total time is O(m+n).

Trie



A trie for keys "A", "to", "tea", "ted", "ten", "i", "in", and "inn".

In computer science, a **trie**, or **prefix tree**, is an ordered tree data structure that is used to store an associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree shows what key it is associated with. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are normally not associated with every node, only with leaves and some inner nodes that correspond to keys of interest.The term trie comes from "retrieval." Following the etymology, the inventor, Edward Fredkin, pronounces it "tree". However, it is pronounced "try" by other authors.

In the example shown, keys are listed in the nodes and values below them. Each complete English word has an arbitrary integer value associated with it. A trie can be seen as a deterministic finite automaton, although the symbol on each edge is often implicit in the order of the branches.It is not necessary for keys to be explicitly stored in nodes. (In the figure, words are shown only to illustrate how the trie works.)

- Though it is most common, tries need not be keyed by character strings. The same algorithms can easily be adapted to serve similar functions of ordered lists of any construct, e.g., permutations on a list of digits or shapes. In particular, a **bitwise trie** is keyed on the individual bits making up a short, fixed size of bits such as an integer number or pointer to memory.

    - Unlike most other algorithms, tries have the peculiar feature that the time to insert, or to delete or to find is almost identical because the code paths followed for each are almost identical. As a result, for situations where code is inserting, deleting and finding in equal measure tries can handily beat binary search trees or even hash tables, as well as being better for the CPU's instruction and branch caches.

The following are the main advantages of tries over binary search trees (BSTs):

- Looking up keys is faster. Looking up a key of length *m* takes worst case O(*m*) time. A BST performs O(log(*n*)) comparisons of keys, where *n* is the number of elements in the tree, because lookups depend on the depth of the tree, which is logarithmic in the number of keys if the tree is balanced. Hence in the worst case, a BST takes O(*m* log *n*) time. Moreover, in the worst case log(*n*) will approach *m*. Also, the simple operations tries use during lookup, such as array indexing using a character, are fast on real machines.
- Tries can require less space when they contain a large number of short strings, because the keys are not stored explicitly and nodes are shared between keys with common initial subsequences.
- Tries facilitate longest-prefix matching, helping to find the key sharing the longest possible prefix of characters all unique.

The following are the main advantages of tries over hash tables:

- Tries can perform a "closest fit" find almost as quickly as an exact find[*citation needed*]. Hash tables can only perform an exact find as they store no relational state between keys.
- Tries tend to be faster on average at insertion than hash tables because hash tables must rebuild their index when it becomes full - which is a very expensive operation. Tries therefore have much better bounded worst case time costs which is important for latency sensitive code.
- Tries can be implemented in a way which avoids the need for additional (dynamic) memory i.e. an **in-place** implementation. Hash tables must always have additional memory in which to store the hash indexation table.
- Looking up keys can be much faster if a hash function can be avoided. Tries can key successfully on arbitrary integer or pointer sized keys without applying a hash function beforehand (i.e. a bitwise trie) to ensure entropy distribution. This makes them faster than hash tables for integer or pointer sized keys in almost all cases as good quality hash functions tend to be a large overhead when hashing just four to eight bytes of data.
- Tries support longest-prefix matching but hashing does not.

Applications

As replacement of other data structures

As mentioned, a trie has a number of advantages over binary search trees.[4] A trie can also be used to replace a hash table, over which it has the following advantages:

- Looking up data in a trie is faster in the worst case, O(m) time, compared to an imperfect hash table. An imperfect hash table can have key collisions. A key collision is the hash function mapping of different keys to the same position in a hash table. The worst-case lookup speed in an imperfect hash table is O(N) time, but far more typically is O(1), with O(m) time spent evaluating the hash.

- There are no collisions of different keys in a trie.
- Buckets in a trie which are analogous to hash table buckets that store key collisions are only necessary if a single key is associated with more than one value.
- There is no need to provide a hash function or to change hash functions as more keys are added to a trie.
- A trie can provide an alphabetical ordering of the entries by key.

Tries do have some drawbacks as well:

- Tries can be slower in some cases than hash tables for looking up data, especially if the data is directly accessed on a hard disk drive or some other secondary storage device where the random access time is high compared to main memory.[5]
- It is not easy to represent all keys as strings, such as floating point numbers - a straightforward encoding using the bitstring of their encoding leads to long chains and prefixes that are not particularly meaningful. Nevertheless a bitwise trie can handle standard IEEE single and double format floating point numbers.

Dictionary representation

A common application of a trie is storing a dictionary, such as one found on a mobile telephone. Such applications take advantage of a trie's ability to quickly search for, insert, and delete entries; however, if storing dictionary words is all that is required (i.e. storage of information auxiliary to each word is not required), a minimal acyclic deterministic finite automaton would use less space than a trie.[*citation needed*]

Tries are also well suited for implementing approximate matching algorithms, including those used in spell checking and hyphenation[2] software.

Algorithms

We can describe trie lookup (and membership) easily. Given a recursive trie type, storing an optional value at each node, and a list of children tries, indexed by the next character, (here, represented as a Haskell data type):

```
data Trie a =
    Trie { value    :: Maybe a
         , children :: [(Char,Trie a)] }
```

We can lookup a value in the trie as follows:

```
find :: String -> Trie a -> Maybe a
find []     t = value t
find (k:ks) t = case lookup k (children t) of
                  Nothing  -> Nothing
                  Just t'  -> find ks t'
```

In an imperative style, and assuming an appropriate data type in place, we can describe the same algorithm in Python (here, specifically for testing membership). Note that children is map of a node's children; and we say that a "terminal" node is one which contains a valid word.

```
def find(node, key):
    for char in key:
        if char not in node.children:
            return None
        else:
            node = node.children[char]
    return node.value
```

Sorting

Lexicographic sorting of a set of keys can be accomplished with a simple trie-based algorithm as follows:

- Insert all keys in a trie.
- Output all keys in the trie by means of pre-order traversal, which results in output that is in lexicographically increasing order. Pre-order traversal is a kind of depth-first traversal. In-order traversal is another kind of depth-first traversal that is more appropriate for outputting the values that are in a binary search tree rather than a trie.

Full text search

A special kind of trie, called a suffix tree, can be used to index all suffixes in a text in order to carry out fast full text searches.

Bitwise tries

Bitwise tries are much the same as a normal character based trie except that individual bits are used to traverse what effectively becomes a form of binary tree. Generally, implementations use a special CPU instruction to very quickly find the first set bit in a fixed length key (e.g. GCC's __builtin_clz() intrinsic). This value is then used to index a 32 or 64 entry table which points to the first item in the bitwise trie with that number of leading zero bits. The search then proceeds by testing each subsequent bit in the key and choosing child[0] or child[1] appropriately until the item is found.

Compressing tries

When the trie is mostly static, i.e. all insertions or deletions of keys from a prefilled trie are disabled and only lookups are needed, and when the trie nodes are not keyed by node specific data (or if the node's data is common) it is possible to compress the trie representation by merging the common branches. This application is typically used for compressing lookup tables when the total set of stored keys is very sparse within their representation space.

For example it may be used to represent sparse bitsets (i.e. subsets of a much fixed enumerable larger set) using a trie keyed by the bit element position within the full set, with the key created from the string of bits needed to encode the integral position of each element. The trie will then have a very degenerate form with many missing branches, and compression becomes possible by storing the leaf nodes (set segments with fixed length) and combining them after detecting the repetition of common patterns or by filling the unused gaps.

Such compression is also typically used, in the implementation of the various fast lookup tables needed to retrieve Unicode character properties (for example to represent case mapping tables, or lookup tables containing the combination of base and combining characters needed to support Unicode normalization). For such application, the representation is similar to transforming a very large unidimensional sparse table into a multidimensional matrix, and then using the coordinates in the hyper-matrix as the string key of an uncompressed trie. The compression will then consist of detecting and merging the common columns within the hyper-matrix to compress the last dimension in the key; each dimension of the hypermatrix stores the start position within a storage vector of the next dimension for each coordinate value, and the resulting vector is itself compressible when it is also sparse, so each dimension (associated to a layer level in the trie) is compressed separately.

Some implementations do support such data compression within dynamic sparse tries and allow insertions and deletions in compressed tries, but generally this has a significant cost when compressed segments need to be split or merged, and some tradeoff has to be made between the smallest size of the compressed trie and the speed of updates, by limiting the range of global lookups for comparing the common branches in the sparse trie.

The result of such compression may look similar to trying to transform the trie into a directed acyclic graph (DAG), because the reverse transform from a DAG to a trie is obvious and always possible, however it is constrained by the form of the key chosen to index the nodes.

Another compression approach is to "unravel" the data structure into a single byte array. This approach eliminates the need for node pointers which reduces the memory requirements substantially and makes memory mapping possible which allows the virtual memory manager to load the data into memory very efficiently.

Another compression approach is to "pack" the trie. Liang describes a space-efficient implementation of a sparse packed trie applied to hyphenation, in which the descendants of each node may be interleaved in memory.