Sorting and Searching

Sorting is the process of arranging a list of elements in a particular order (Ascending or Descending). There are different types sorting techniques. They are

1. Selection Sort:

Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending). In selection sort, the first element in the list is selected and it is compared repeatedly with all the remaining elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped so that first position is filled with the smallest element in the sorted order. Next, we select the element at a second position in the list and it is compared with all the remaining elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated until the entire list is sorted.

In selection sort, element at first location (0th location) is considered as least element, and it is compared with the other elements of the array. If any element is found to be minimum than the element in first location, then that location is taken as minimum and element in that location will be the minimum element.

After completing a set of comparisons, the minimum element is swapped with the element in first location (0th location).

Then again element second location is considered as minimum and it is compared with the other
elements of array and the process continues till the array is sorted in ascending order.
Note: After first pass, smallest element in given list occupies the first position. After second pass, second largest element is placed at second position and so on..

Step by Step Process:

The selection sort algorithm is performed using the following steps...

Step 1 - Select the first element of the list (i.e., Element at first position in the list).
Step 2: Compare the selected element with all the other elements in the list.
Step 3: In every comparison, if any element is found smaller than the selected element (for Ascending order), then both are swapped.
Step 4: Repeat the same procedure with element in the next position in the list till the entire list is sorted.

***Implementation*:**

```
#include<iostream>
using namespace std;
int main()
{
int i, j, n, a[100], t, min;
cout<<"enter range of elements"<<endl;
cin>>n;
cout<<"enter elements:"<<endl;
for(i=0;i<n; i++)
{
cin>>a[i];
}
cout<<"elements before sorting"<<endl;
for(i=0; i<n; i++)
{
```

```
cout<<" "<<a[i];
}

for(i=0; i<n-1; i++)
{
min=i;
for(j=i+1; j<n; j++)
{
if(a[j]<a[min])
min=j;
}
if(min!=i)
{
t=a[i];
a[i]=a[min];
a[min]=t;
}
}
cout<<"\nelements after sorting are"<<endl;
for(i=0; i<n; i++)
{
cout<<" "<<a[i];
}
cout<<endl;
}
```

Complexity of the Selection Sort Algorithm:

To sort an unsorted list with 'n' number of elements, we need to make ((n-1)+(n-2)+(n-3)+. +1) = (n (n-1))/2 number of comparisons in the worst case. If the list is already sorted then it requires 'n' number of comparisons.

Worst Case : $O(n^2)$
Best Case : $\Omega(n^2)$
Average Case : $\Theta(n^2)$

2. Bubble Sort:

In bubble sort each element is compared with its adjacent element. If first element is larger than second one, then both elements are swapped. Otherwise, element are not swapped. Consider the following list of numbers.

Algorithm:

```
begin BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for

   return list

end BubbleSort
```

Implementation:

```
#include<iostream>
using namespace std;
int main()
{
int i, j, n, bubble[20], temp;
cout<<"enter range of elements"<<endl;
cin>>n;
cout<<"enter elements"<<endl;
for(i=0;i<n; i++)
{
cin>>bubble[i];
}
for(i=0; i<n; i++)
{
for(j=0;j<n-1;j++)
{
if(bubble[j] > bubble[j+1])
{
temp=bubble[j];
bubble[j]=bubble[j+1];
bubble[j+1]=temp;
}
}
}
cout<<"after sorting"<<endl;
for(i=0; i<n; i++)
{
cout<<" "<<bubble[i];
}
}
```

Complexity of the Bubble Sort Algorithm:

Worst Case : $O(n^2)$
Best Case : $O(n)$
Average Case : $O(n^2)$

3. Insertion Sort:

Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

Algorithm:

Step 1 - Assume that first element in the list is in sorted portion and all the remaining elements are in unsorted portion.
Step 2: Take first element from the unsorted portion and insert that element into the sorted portion in the order specified.
Step 3: Repeat the above process until all the elements from the unsorted portion are moved into the sorted portion.

Implementation:

```
#include<iostream>
```

```
using namespace std;
int main()
{
int i, j, key,n, a[20];
cout<<"enter range of elements"<<endl;
cin>>n;
cout<<"enter elements"<<endl;
for(i=0;i<n; i++)
{
cin>>a[i];
}
for(i=1; i<n; i++)
{
key=a[i];
j=i-1;
while(j>=0&&a[j]>key)
{
a[j+1]=a[j]
; j=j-1;
}
a[j+1]=key;
}
cout<<"after sorting"<<endl;
for(i=0; i<n; i++)
{
cout<<" "<<a[i];
}
cout<<endl;
}
```

Complexity of the Insertion Sort Algorithm:

To sort an unsorted list with 'n' number of elements, we need to make (1+2+3+. +n-1) = (n (n-1))/2 number of comparisons in the worst case. If the list is already sorted then it requires 'n' number of comparisons.

Worst Case : $O(n^2)$
Best Case : $\Omega(n)$
Average Case : $\Theta(n^2)$

4. Quick Sort:

Quick sort is a fast sorting algorithm used to sort a list of elements. Quick sort algorithm is invented by C. A. R. Hoare.

The quick sort algorithm attempts to separate the list of elements into two parts and then sort each part recursively. That means it use divide and conquer strategy. In quick sort, the partition of the list is performed based on the element called pivot. Here pivot element is one of the elements in the list**.**

The list is divided into two partitions such that "all elements to the left of pivot are smaller than the pivot and all elements to the right of pivot are greater than or equal to the pivot".

Algorithm:

Step 1 - Consider the first element of the list as pivot (i.e., Element at first position in the list).
Step 2 - Define two variables i and j. Set i and j to first and last elements of the list respectively.

Step 3 - Increment i until list[i] > pivot then stop.
Step 4 - Decrement j until list[j] < pivot then stop.
Step 5 - If i < j then exchange list[i] and list[j].
Step 6 - Repeat steps 3,4 & 5 until i > j.
Step 7 - Exchange the pivot element with list[j] element.

Implementation:

```
#include<iostream>
using namespace std;
void QuickSort(int [],int,int);
int main()
{
int i, n, list[20];
cout<<"enter range of elements"<<endl;
cin>>n;
cout<<n<<endl;
cout<<"enter elements"<<endl;
for(i=0;i<n; i++)
{
cin>>list[i];
}
QuickSort(list,0,n-1);
cout<<"after sorting"<<endl;
for(i=0; i<n; i++)
{
cout<<" "<<list[i];
}
}
void QuickSort(int list[],int first,int last)
{
   int pivot,i,j,temp;
    if(first < last)
    {
        pivot = first;
        i = first;
        j = last;

        while(i < j)
        {
            while(list[i] <= list[pivot] && i < last)
                i++;
            while(list[j] > list[pivot])
                j--;
            if(i < j)
            {
                temp = list[i];
                list[i]          =
                list[j];    list[j]
                = temp;
            }
        }

        temp = list[pivot];
        list[pivot]          =
```

```
list[j];    list[j]    =
temp;
```

```
        QuickSort(list,first,j-1);
        QuickSort(list,j+1,last);
    }
}
```

Complexity of the Quick Sort Algorithm:

To sort an unsorted list with 'n' number of elements, we need to make ((n-1)+(n-2)+(n- 3)+. +1) = (n (n-1))/2 number of comparisons in the worst case. If the list is already sorted, then it requires 'n' number of comparisons.

Worst Case : $O(n^2)$
Best Case : O (n log n)
Average Case : O (n log n)

5. Merge Sort:

Merge sort is a sorting technique based on **divide and conquer technique**. With worst-case time complexity being O(n log n), it is one of the most respected algorithms. Merge sort first divides the array into equal halves and then combines them in a sorted manner.

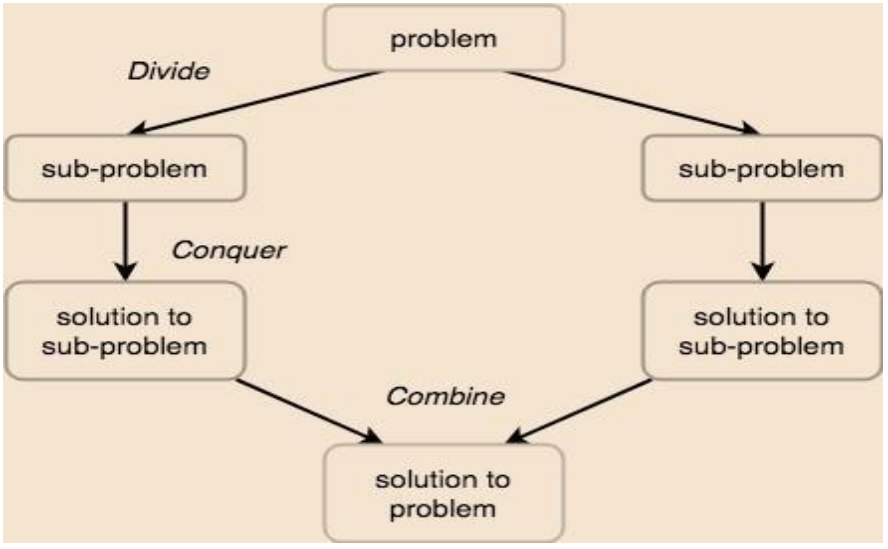
What is the rule of Divide and Conquer?

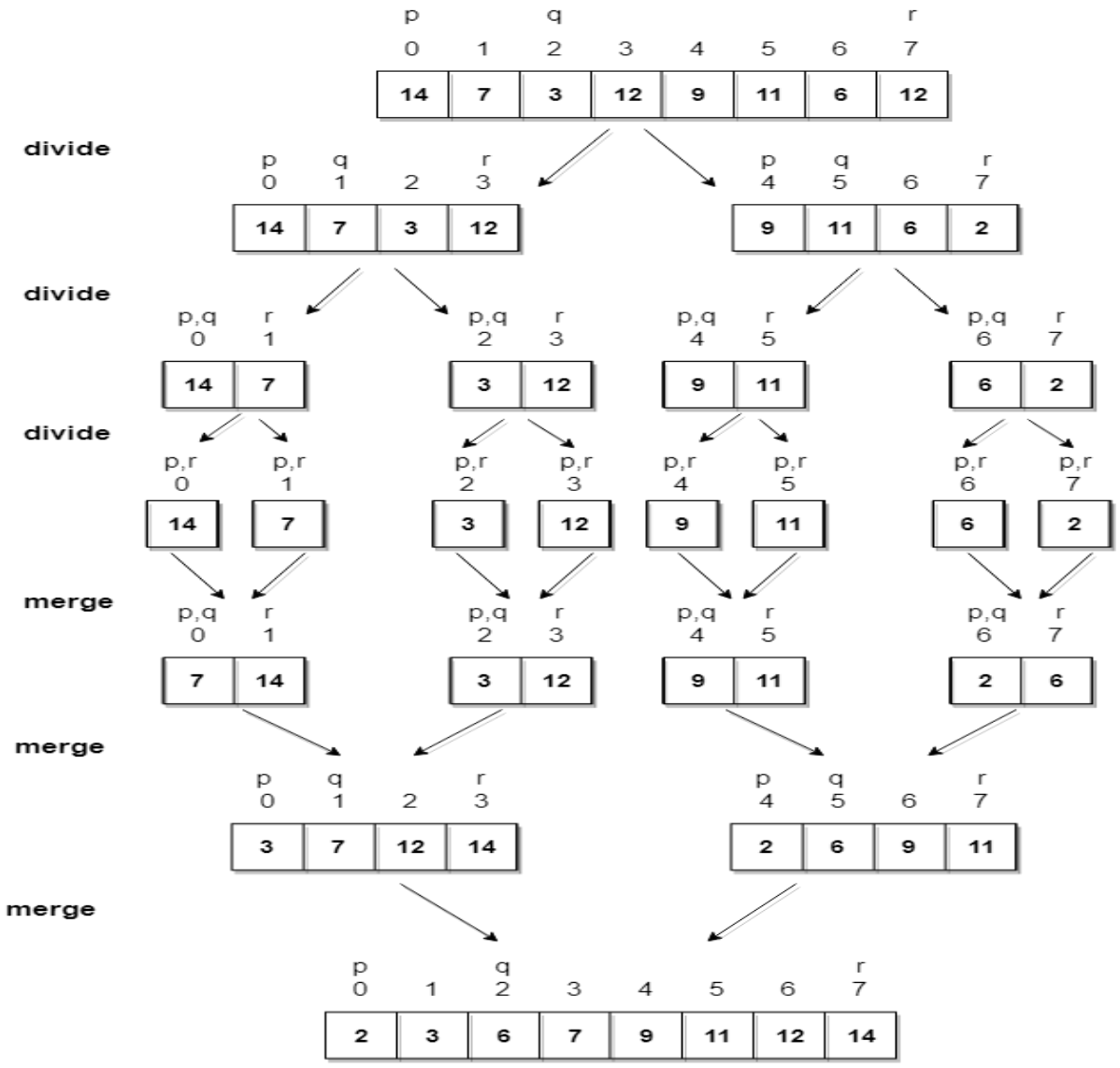
If we can break a single big problem into smaller sub-problems, solve the smaller sub-problems and combine their solutions to find the solution for the original big problem, it becomes easier to solve the whole problem.

In Merge Sort, the given unsorted array with n elements, is divided into n subarrays, each having one element, because a single element is always sorted in itself. Then, it repeatedly merges these subarrays, to produce new sorted subarrays, and in the end, one complete sorted array is produced.

The concept of Divide and Conquer involves three steps:

1. Divide the problem into multiple small problems.
2. Conquer the subproblems by solving them. The idea is to break down the problem into atomic subproblems, where they are actually solved.
3. Combine the solutions of the subproblems to find the solution of the actual problem.





Algorithm:

Step 1 − if it is only one element in the list it is already sorted, return.
Step 2 − divide the list recursively into two halves until it can no more be divided.
Step 3 − merge the smaller lists into new list in sorted order.

Implementation:

```
#include<iostream>
using namespace std;
void mergesort(int a[],int lb,int mid, int ub)
{
int i=lb,j=mid+1,k=0,b[50];
        while(i<=mid&&j<=ub)
        {
                if(a[i]<a[j])
                {
                        b[k++]=a[i++];
                }
                else
                {
```

```
                        b[k++]=a[j++];
                }
        }
        while(i<=mid)
        {
        b[k++]=a[i++];
        }
        while(j<=ub)
        {
        b[k++]=a[j++];
        }
        for(k=0;k<=ub-lb;k++)
        {
        a[k+lb]=b[k];
        }
}
void merge(int a[],int low,int high)
{
int mid;
        if(low<high)
        {
        mid=(low+high)/2;
        merge(a,low,mid);
        merge(a,mid+1,high);
        mergesort(a,low,mid,high);
        }
}
int main()
{
int i, a[30],n;
cout<<"Enter the number of elements:";
cin>>n;
cout<<"Enter elements:"<<endl;
for(i=0;i<n;i++)
{
        cin>>a[i];
}
cout<<"\n\nBefore sorting:";
for(i=0;i<n;i++)
{
        cout<<a[i]<<" ";
}
merge(a,0,n-1);
cout<<"\n\nAfter sorting:";
for(i=0;i<n;i++)
{
        cout<<a[i]<<" ";
}
return 0;
}
```

Complexity of the Merge Sort Algorithm:

Worst Case Time Complexity : O(n*log n)

Best Case Time Complexity : O(n*log n)

Average Time Complexity : O(n*log n)

6. Heap Sort:

Heap sort is one of the sorting algorithms used to arrange a list of elements in order. Heapsort algorithm uses one of the tree concepts called Heap Tree. In this sorting algorithm, we use Max Heap to arrange list of elements in Descending order and Min Heap to arrange list elements in Ascending order.

Algorithm:

Step 1 - Construct a Binary Tree with given list of Elements.
Step 2 - Transform the Binary Tree into Min Heap.
Step 3 - Delete the root element from Min Heap using Heapify method.
Step 4 - Put the deleted element into the Sorted list.
Step 5 - Repeat the same until Min Heap becomes empty.
Step 6 - Display the sorted list.

Implementation:

For example please go through the heap tree concept...

```
#include <iostream>
using namespace std;
void heapify(int arr[], int n, int i)
{
   int smallest = i;
   int l = 2 * i + 1;
   int r = 2 * i + 2;
     if (l < n && arr[l] < arr[smallest])
       smallest = l;
   if (r < n && arr[r] < arr[smallest])
       smallest = r;
     if (smallest != i) {
       swap(arr[i], arr[smallest]);
       heapify(arr, n, smallest);
   }
}
void heapSort(int arr[], int n)
{
 for (int i = n / 2 - 1; i >= 0; i--)
       heapify(arr, n, i);
 for (int i = n - 1; i >= 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
   }
}

int main()
{
int a[50],n,i;
cout<<"Enter the size of the array:";
cin>>n;
cout<<"\nEnter the elements:";
for(i=0;i<n;i++)
cin>>a[i];
cout<<"\nThe list before sorting:";
for(i=0;i<n;i++)
cout<<a[i]<<" ";
```

```
heapSort(a,n);
cout<<"\nThe list after sorting:";
for(i=0;i<n;i++)
cout<<a[i]<<" ";
cout<<"\n\n";
}
```

Complexity of the Heap Sort Algorithm:
To sort an unsorted list with 'n' number of elements, following are the complexities...
Worst Case : O(n log n)
Best Case : O(n log n)
Average Case : O(n log n)

Comparison among all the sorting techniques:

	Time					
Sort	Average	Best	Worst	Space	Stability	Remarks
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Constant	Stable	Always use a modified bubble sort
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Constant	Stable	Even a perfectly sorted input requires scanning the entire array
Insertion Sort	$O(n^2)$	O(n)	$O(n^2)$	Constant	Stable	In the best case (already sorted), every insert requires constant time
Heap Sort	O(n*log(n))	O(n*log(n))	O(n*log(n))	Constant	Instable	By using input array as storage for the heap, it is possible to achieve constant space
Merge Sort	O(n*log(n))	O(n*log(n))	O(n*log(n))	Depends	Stable	On arrays, merge sort requires O(n) space; on linked lists, merge sort requires constant space
Quicksort	O(n*log(n))	O(n*log(n))	O(n^2)	Constant	Stable	Randomly picking a pivot value (or shuffling the array prior to sorting) can help avoid worst case scenarios such as a perfectly sorted array.

Searching:

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.
Gathering any information (or) trying to find any data is said to be a Searching process.
Searching technique can be used more efficiently if the data is present in an ordered manner.
Most widely used Searching methods are:-
1) Linear Search (Sequential Search) 2) Binary Search

1. Linear Search:

Suppose an array is given, which contains „n‟ elements. If no other information is given and we are asked to search for an element in array, than we should compare that element, with all the elements present in the array. This method which is used to Search the element in the array is known as Liner Search. Since the key element/ the element which is to be searched in array, if found out by comparing with every element of array one-by-one, this method is also known as Sequential Search.

Algorithm:

Step 1 - Read the search element from the user.
Step 2 - Compare the search element with the first element in the list.
Step 3 - If both are matched, then display "Given element is found!!!" and terminate the function
Step 4 - If both are not matched, then compare search element with the next element in the list.
Step 5 - Repeat steps 3 and 4 until search element is compared with last element in the list.
Step 6 - If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

Implementation:

```
#include<iostream>
using namespace std;
int main()
{
int linear[20],n,i,k,temp=0;
//clrscr();
cout<<"enter range of elements"<<endl;
cin>>n;
cout<<"enter elements into array"<<endl;
 for(i=0; i<n; i++)
{
cin>>linear[i];
}
cout<<"enter search key:"<<endl;
cin>>k;
for(i=0;i<n; i++)
{
if(k==linear[i])
{
temp=1;
cout<<k<<"is found at location"<<i+1<<endl;
break;
}
}
if(temp!=1)
{
cout<<"element not found"<<endl;
}
}
```

- The time complexity of Linear search is:

a. Best case = O(1)
b. Average case = n(n+1)/2n = O(n)
c. Worst case = O(n)

- The space complexity of Linear Search is O(1).

2. Binary Search:

Efficient search method for large arrays. Liner search, will required to compare the key element, with every element in array. If the array size is large, liner search requires more time for execution.
In such cases, binary search technique can be used.

To perform binary search:-
i) Elements should be entered into array in Ascending Order
ii) Middle element of the array must be found. This is done as follows

Find the lowest position & highest position of the array i.e., if an array contains „n" elements then:-
Low=0
High =n-1
Mid =(low+high)/2

Note: We are calculating mid position of the array not the middle element. The element present in the mid position is considered as middle element of array.

Now the search key element is compared with middle element of array. Three cases arises
Case 1 : If middle element is equal to key, then search is end.
Case 2 : If middle element is greater than key, then search is done, before the middle element of array.
Case 3 : If middle element is less than key, then search is done after the middle element of array.

This process is repeated till we get the key element (or) till the search comes to an end, since key element is not in the list.

Algorithm:

Step 1 - Read the search element from the user.
Step 2 - Find the middle element in the sorted list.
Step 3 - Compare the search element with the middle element in the sorted list.
Step 4 - If both are matched, then display "Given element is found!!!" and terminate the function.
Step 5 - If both are not matched, then check whether the search element is smaller or larger than the middle element.
Step 6 - If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
Step 7 - If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
Step 8 - Repeat the same process until we find the search element in the list or until sublist contains only one element.
Step 9 - If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.

Implementation:

```
#include<iostream>
using namespace std;
int main()
{
int binary[20],n, i, k, low, mid, high;
cout<<"enter range of elements"<<endl;
cin>>n;
cout<<"enter elements into array"<<endl;
for(i=0;i<n; i++)
```

```
{
cin>>binary[i];
}
cout<<"enter search key"<<endl;
cin>>k;
low=0;
high=n-1;
while(low<=high)
{
mid=(low+high)/2;
if(binary[mid]<k)
{
low=mid+1;
}
else if(binary[mid]>k)
{
high=mid-1;
}
else
break;
}
if(binary[mid]==k)
{
cout<<"element is found at location"<<mid+1<<endl;
}
else
cout<<"element not found"<<endl;
}
```

- The time complexity of Linear search is:

a. Best case = O(1)
b. Average case = O(logn)
c. Worst case = O(logn)

- The space complexity of Binary Search is O(1).

Hashing:

In all search techniques like linear search, binary search and search trees, the time required to search an element depends on the total number of elements present in that data structure. In all these search techniques, as the number of elements increases the time required to search an element also increases linearly.

Hashing is another approach in which time required to search an element doesn't depend on the total number of elements. Using hashing data structure, a given element is searched with constant time complexity. Hashing is an effective way to reduce the number of comparisons to search an element in a data structure.

Hashing is defined as follows...

“Hashing is the process of indexing and retrieving element (data) in a data structure to provide a faster way of finding the element using a hash key”.

Here, the hash key is a value which provides the index value where the actual data is likely to be stored in the data structure.

In this data structure, we use a concept called Hash table to store data. All the data values are inserted into the hash table based on the hash key value. The hash key value is used to map the data with an index in the hash table. And the hash key is generated for every data using a hash function.

That means every entry in the hash table is based on the hash key value generated using the hash function.

Hash Table is defined as follows...

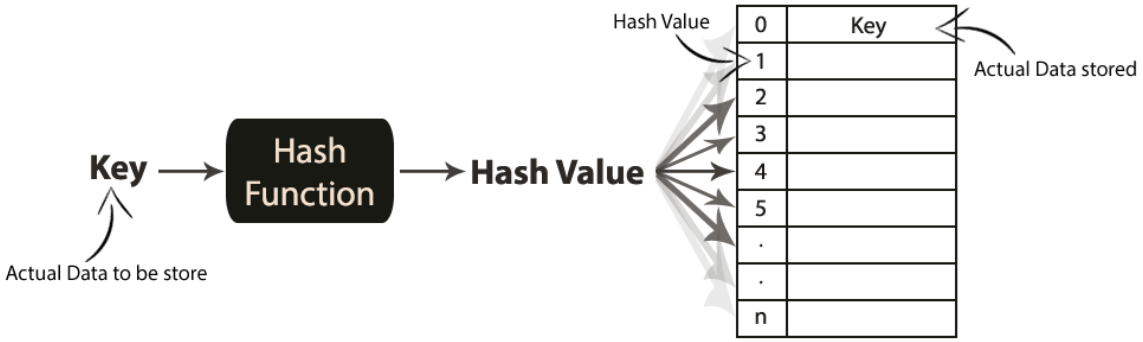
"Hash table is just an array which maps a key (data) into the data structure with the help of hash function such that insertion, deletion and search operations are performed with constant time complexity (i.e. O(1))".

Hash tables are used to perform insertion, deletion and search operations very quickly in a data structure. Using hash table concept, insertion, deletion, and search operations are accomplished in constant time complexity. Generally, every hash table makes use of a function called hash function to map the data into the hash table.

A hash function is defined as follows...

"Hash function is a function which takes a piece of data (i.e. key) as input and produces an integer (i.e. hash value) as output which maps the data to a particular index in the hash table".

Basic concept of hashing and hash table is shown in the following figure...



Collision:

Since a hash function gets us a small number for a key which is a big integer or string, there is possibility that two keys result in same value.

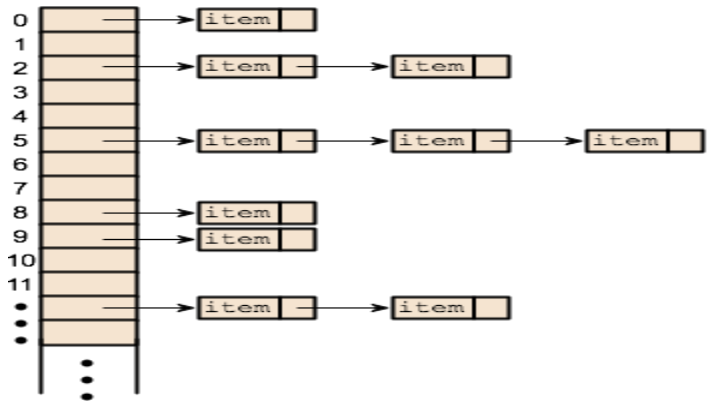
The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some **collision handling technique.**

There are mainly two methods to handle collision:

1. Separate Chaining
2. Open Addressing

1. Separate Chaining:

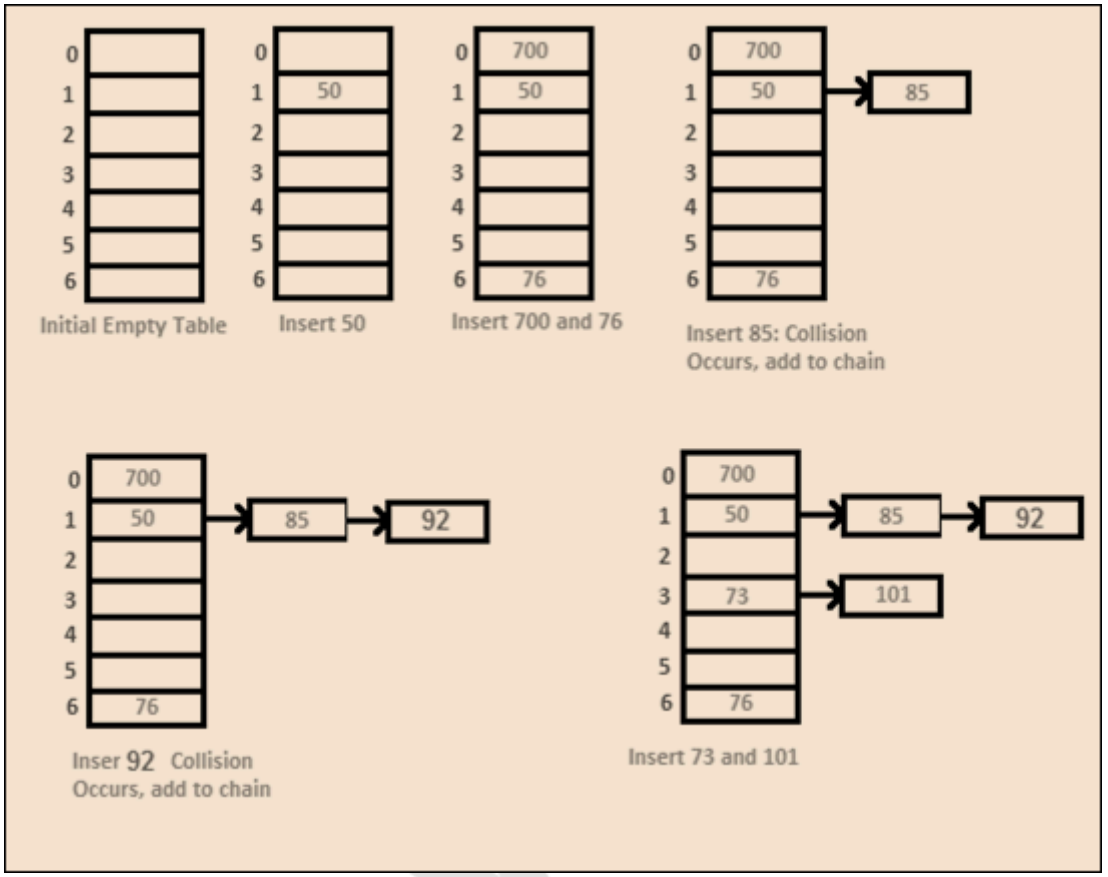
Separate Chaining is also called as closed addressing and open hashing. Separate chaining is one of the most commonly used collision resolution techniques. It is usually implemented using linked lists. In separate chaining, each element of the hash table is a linked list. To store an element in the hash table you must insert it into a specific linked list. If there is any collision (i.e. two different elements have same hash value) then store both the elements in the same linked list.



The cost of a lookup is that of scanning the entries of the selected linked list for the required key. If the distribution of the keys is sufficiently uniform, then the average cost of a lookup depends only on the average number of keys per linked list. For this reason, chained hash tables remain effective even when the number of table entries (N) is much higher than the number of slots.

For separate chaining, the worst-case scenario is when all the entries are inserted into the same linked list. The lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number (N) of entries in the table.

Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Advantages:

- Simple to implement.
- Hash table never fills up, we can always add more elements to chain.
- Less sensitive to the hash function or load factors.
- It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

- Cache performance of chaining is not good as keys are stored using linked list. Open addressing provides better cache performance as everything is stored in same table.
- Wastage of Space (Some Parts of hash table are never used)
- If the chain becomes long, then search time can become O(n) in worst case.
- Uses extra space for links.

2. Open Addressing:

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. In open addressing, instead of in linked lists, all entry records are stored in the array itself. When a new entry has to be inserted, the hash index of the hashed value is computed and then the array is examined (starting with the hashed index). If the slot at the hashed index is unoccupied, then the entry record is inserted in slot at the hashed index else it proceeds in some probe sequence until it finds an unoccupied slot.

Open addressing is again three types:

a. Linear probing:

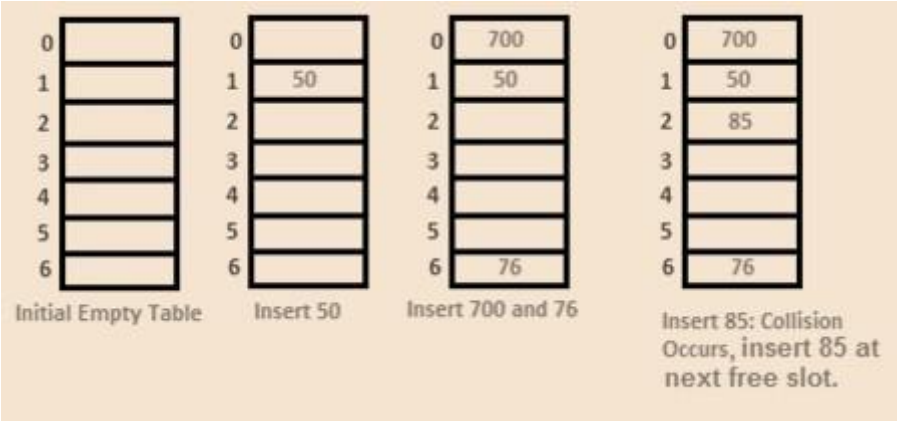
When searching for an entry, the array is scanned in the same sequence until either the target element is found or an unused slot is found. This indicates that there is no such key in the table. The name "open addressing" refers to the fact that the location or address of the item is not determined by its hash value.

Linear probing is when the interval between successive probes is fixed (usually to 1). Let's assume that the hashed index for a particular entry is index. The probing sequence for linear probing will be:

index = index % hashTableSize
index = (index + 1) % hashTableSize
index = (index + 2) % hashTableSize
index = (index + 3) % hashTableSize

In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

Let us consider a simple hash function as "key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



While inserting 85,there is a collision with slot 1,so we are using linear probing approach,

index = (index + 1) % hashTableSize

i.e., index=(85+1)%7

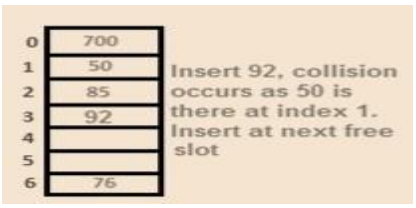
=86%7

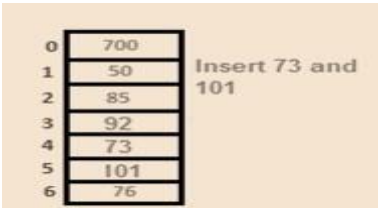
=2 so now 85 will insert in slot 2 which is empty.

Next element 92, While inserting 92,there is a collision with slot 1so again we are using linear probing approach

index = (index + 1) % hashTableSize=(92+1)%7=1, there is a collision with slot 2

index = (index + 2) % hashTableSize=(92+2)%7=3 so now 92 will insert in slot 3 which is empty.





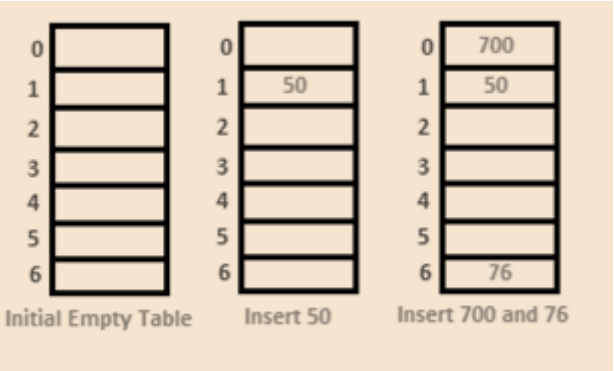
b. Quadratic probing:

Quadratic probing is similar to linear probing and the only difference is the interval between successive probes or entry slots. Here, when the slot at a hashed index for an entry record is already occupied, you must start traversing until you find an unoccupied slot. The interval between slots is computed by adding the successive value of an arbitrary polynomial in the original hashed index.

Let us assume that the hashed index for an entry is index and at index there is an occupied slot. The probe sequence will be as follows:

index = index % hashTableSize
index = (index + 1^2) % hashTableSize
index = (index + 2^2) % hashTableSize
index = (index + 3^2) % hashTableSize
and so on…

Let us consider a simple hash function as "key mod 7" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



While inserting 85,there is a collision with slot 1,so we are using quadratic probing approach,
index = (index + 1^2) % hashTableSize
i.e., index=(85+1^2)%7=(85+1)%7
=86%7
=2 so now 85 will insert in slot 2 which is empty.

Next element 92, While inserting 92,there is a collision with slot 1so again we are using linear probing approach
index = (index + 1^2) % hashTableSize=(92+1^2)%7=1, there is a collision with slot 2.
index = (index + 2^2) % hashTableSize=(92+2^2)%7==(92+4)%7=5 so now 92 will insert in slot 5 which is empty.

0	700
1	50
2	85
3	
4	
5	**92**
6	76

Next element 73,
index = index % hashTableSize
index=73%7=3, so now 73 will insert in slot 3 which is empty.

0	700
1	50
2	85
3	**73**
4	
5	92
6	76

Next element 101,
index = index % hashTableSize=101%7=3, there is a collision with slot 3.
index = (index + 1^2) % hashTableSize=(101+1^2)%7=4, so now 101 will insert in slot 4 which is empty.

0	700
1	50
2	85
3	73
4	**101**
5	92
6	76

c. Double Hashing:

Double hashing is a collision resolving technique in Open Addressed Hash tables. Double hashing is similar to linear probing and the only difference is the interval between successive probes. Here, the interval between probes is computed by using two hash functions. Double hashing uses the idea of applying a second hash function to key when a collision occurs.

Double hashing can be done using :

(hash1(index) + i * hash2(index)) % hashTableSize

Here hash1() and hash2() are hash functions and hashTableSize is size of hash table. (We repeat by increasing i when collision occurs)

First hash function is typically **hash1(index) = index % hashTableSize**
Second hash function is : **hash2(index) = PRIME – (index % PRIME)** where PRIME is a prime smaller than the hashTableSize.

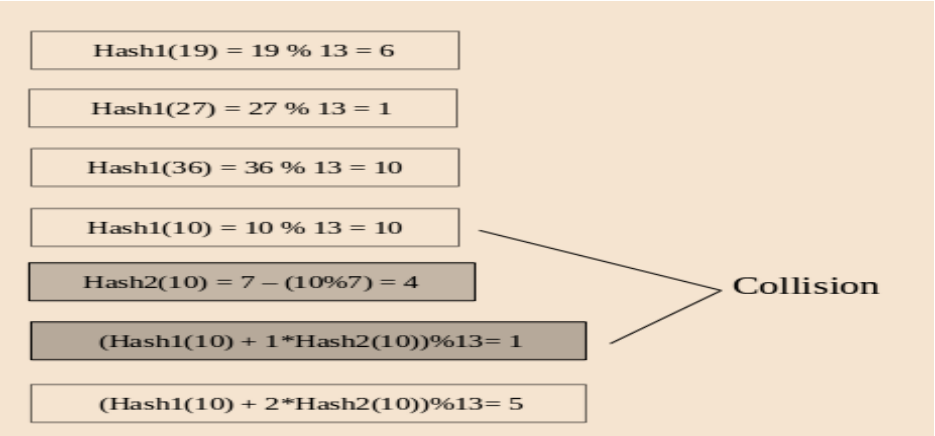
There are a couple of requirements for the second function:

- It must never evaluate to 0
- Must make sure that all cells can be probed

Let us consider a simple hash function as “key mod 13” and sequence of keys as 19,27,36,10. Assume size of the hash table is 13.

Let say Hash1(index)=index%13

Hash2(index)=7-(index%7) ,where 7 is the random prime number.



The hash table is

0	
1	27
2	
3	
4	
5	10
6	19
7	
8	
9	
10	36
11	
12	
13	

Graphs:

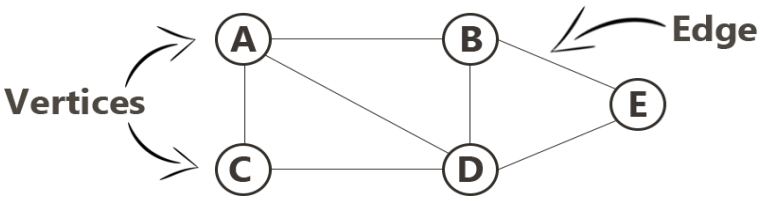
Graph is a non-linear data structure. It contains a set of points known as nodes (or vertices) and a set of links known as edges (or Arcs). Here edges are used to connect the vertices. A graph is defined as follows...

"Graph is a collection of vertices and arcs in which vertices are connected with arcs" or "Graph is a collection of nodes and edges in which nodes are connected with edges".

Generally, a graph G is represented as G = (V , E), where V is set of vertices and E is set of edges.

Example:

The following is a graph with 5 vertices and 6 edges.
This graph G can be defined as G = (V , E)
Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



Basic Graph Terminologies:

1. ***Vertex:***
   Individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.
2. ***Edge:***
   An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (startingVertex, endingVertex). For example, in above graph the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

   - Edges are three types.

   - **Undirected Edge** - An undirected egde is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
   - **Directed Edge** - A directed egde is a unidirectional edge. If there is directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
   - **Weighted Edge** - A weighted egde is a edge with value (cost) on it.
3. ***Undirected Graph***: A graph with only undirected edges is said to be undirected graph.
4. ***Directed Graph:*** A graph with only directed edges is said to be directed graph.
5. ***Mixed Graph:*** A graph with both undirected and directed edges is said to be mixed graph.
6. ***End vertices or Endpoints:*** The two vertices joined by edge are called end vertices (or endpoints) of that edge.
7. ***Origin:*** If a edge is directed, its first endpoint is said to be the origin of it.
8. ***Destination:*** If a edge is directed, its first endpoint is said to be the origin of it and the other endpoint is said to be the destination of that edge.
9. ***Adjacent:*** If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, vertices A and B are said to be adjacent if there is an edge between them.
10. ***Incident:*** Edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.
11. ***Outgoing Edge:*** A directed edge is said to be outgoing edge on its origin vertex.
12. ***Incoming Edge:*** A directed edge is said to be incoming edge on its destination vertex.
13. ***Degree:*** Total number of edges connected to a vertex is said to be degree of that vertex.
14. ***Indegree:*** Total number of incoming edges connected to a vertex is said to be indegree of that vertex.
15. ***Outdegree:*** Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.
16. ***Parallel edges or Multiple edges:*** If there are two undirected edges with same end vertices and two directed edges with same origin and destination, such edges are called parallel edges or multiple edges.
17. ***Self-loop:*** Edge (undirected or directed) is a self-loop if its two endpoints coincide with each other.

18. ***Simple Graph:*** A graph is said to be simple if there are no parallel and self-loop edges.
19. ***Path:*** A path is a sequence of alternate vertices and edges that starts at a vertex and ends at other vertex such that each edge is incident to its predecessor and successor vertex.

Graph Representations:

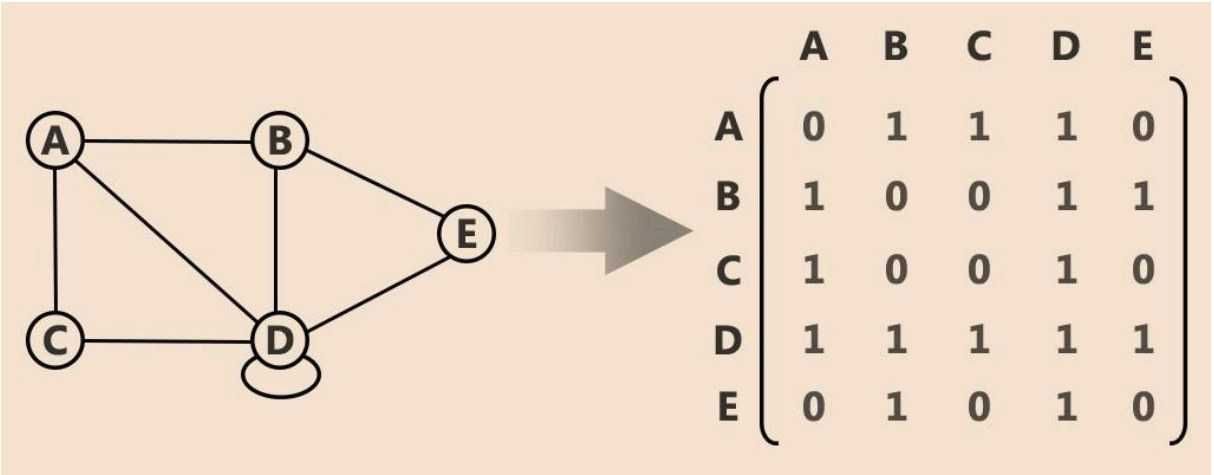
Graph data structure is represented using following representations...

1. Adjacency Matrix
2. Incidence Matrix
3. Adjacency List

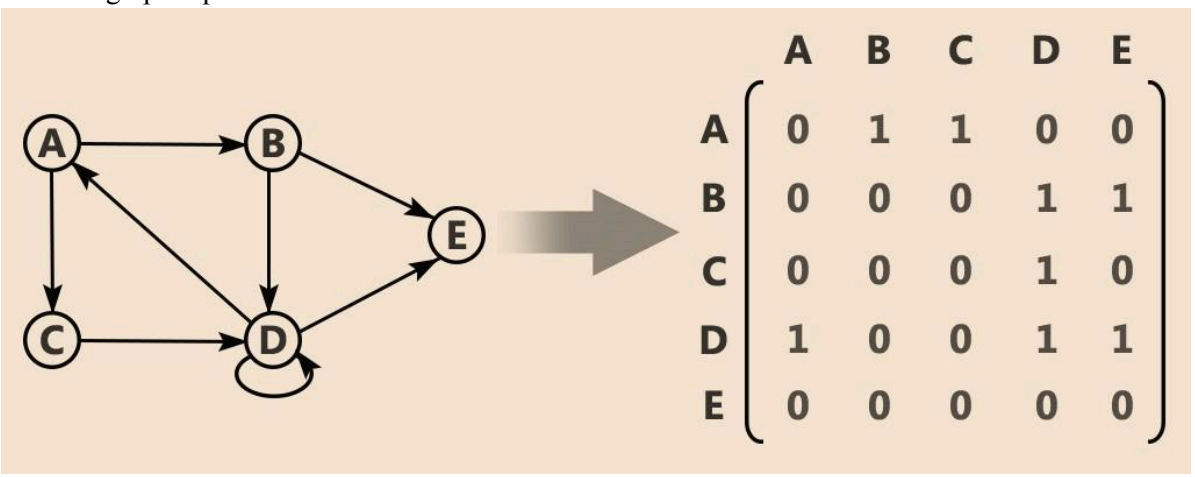
1. Adjacency Matrix:

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

For example, consider the following undirected graph representation...



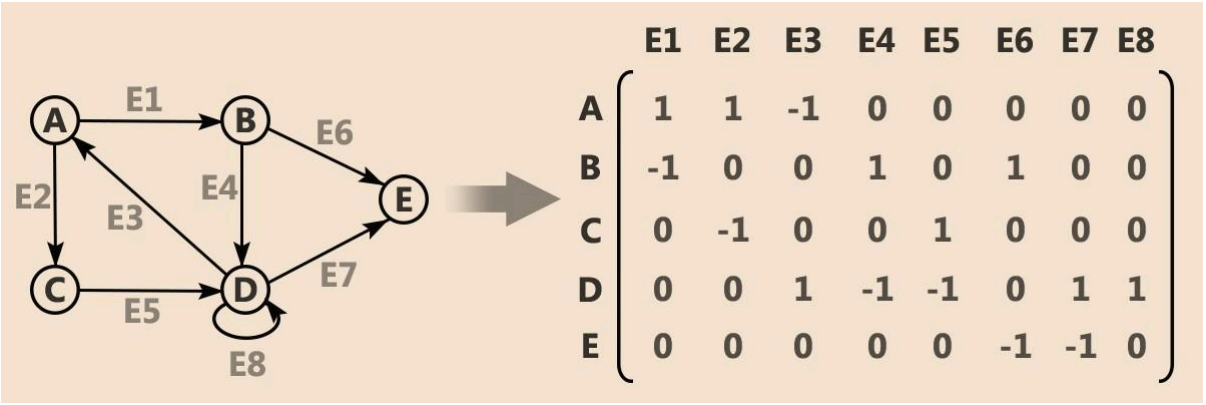
Directed graph representation...



2. *Incidence Matrix*

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of edges. That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6. In this matrix, rows represent vertices and columns represents edges. This matrix is filled with 0 or 1 or -1. Here, 0 represents that the row edge is not connected to column vertex, 1 represents that the row edge is connected as the outgoing edge to column vertex and -1 represents that the row edge is connected as the incoming edge to column vertex.

For example, consider the following directed graph representation...

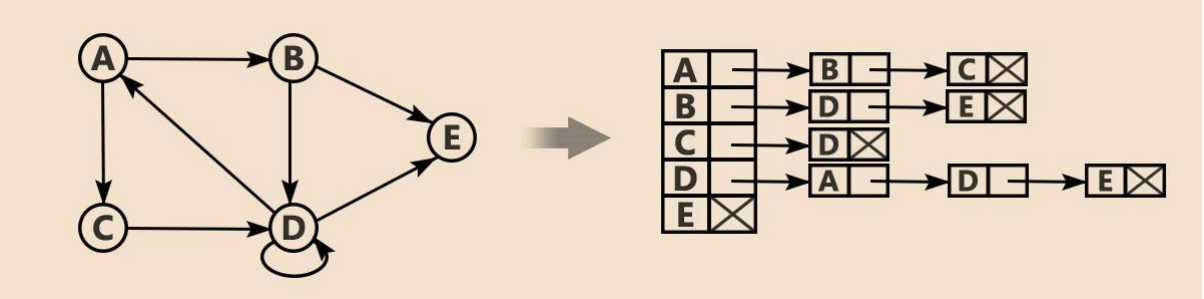


	E1	E2	E3	E4	E5	E6	E7	E8
A	1	1	-1	0	0	0	0	0
B	-1	0	0	1	0	1	0	0
C	0	-1	0	0	1	0	0	0
D	0	0	1	-1	-1	0	1	1
E	0	0	0	0	0	-1	-1	0

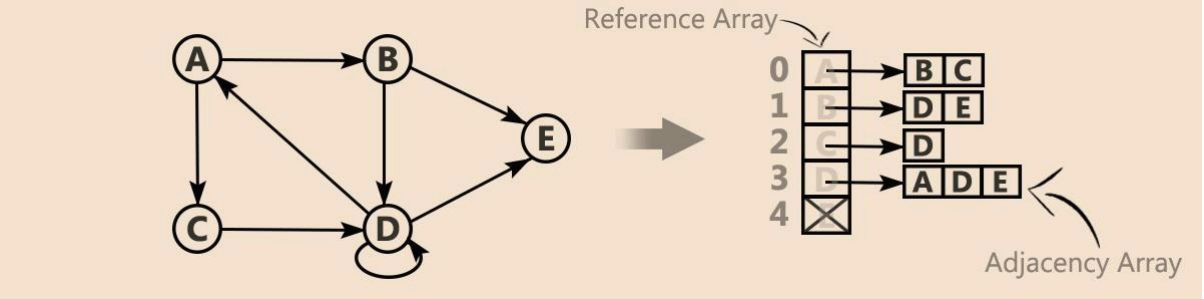
3. *Adjacency List:*

In this representation, every vertex of a graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using an array as follows..



Graph Search and traversal algorithms:

Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the

edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.

There are two graph traversal techniques and they are as follows...

1. DFS (Depth First Search)
2. BFS (Breadth First Search)

1. DFS (Depth First Search):

DFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal.

Algorithm:

Step 1 - Define a Stack of size total number of vertices in the graph.

Step 2 - Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

Step 3 - Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it on to the stack.

Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

Step 5 - When there is no new vertex to visit then use back tracking and pop one vertex from the stack.

Step 6 - Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph.

"Back tracking is coming back to the vertex from which we reached the current vertex".

Implementation:

```
#include<iostream>
using namespace std;
class graph
{

 int cost[10][10],i,j,k,n,m,stack[10],top,v;
 bool visited[10];
 public:
    graph()
      {
        top=-1;
        for ( int i=1;i<=10;i++) visited[i]=false;
        for (int i=0;i<10;i++)
          for(j=0;j<10;j++)
             cost[i][j]=0;
      }
    void accept();
    void DFS();
};
```

```
void graph::accept()
  {
    cout<<"enter the no of vertices"<<endl;
    cin>>n;
    cout<<"enter the no of edges"<<endl;
    cin>>m;
    cout<<"\nEnter EDGES"<<endl;
    for(k=1;k<=m;k++)
       {
        cin>>i>>j;
        cost[i][j]=1;
        cost[j][i]=1;
       }
}
void graph::DFS()
{
  cout<<"enter the initial vertex"<<endl;
  cin>>v;
  cout<<"visited vertices\n";
  stack[++top]=v;
  while(top!=-1)
    {
      v=stack[top--];
      if(visited[v]==false)
       {
         cout<<v;
         visited[v]=true;
       }
      else
        continue;
        for(j=n;j>=1;j--)
        if(!visited[j]&& cost[v][j]==1)
             stack[++top]=j;
    }
}
int main()
{
 graph ob;
 ob.accept();
 ob.DFS();
 return 0;
}
```

Complexity:

Time complexity O(V+E), when implemented using an adjacency list.

2. BFS (Breadth First Search):

BFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal.

Algorithm:

Step 1 - Define a Queue of size total number of vertices in the graph.

Step 2 - Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.

Step 3 - Visit all the non-visited adjacent vertices of the vertex which is at front of the Queue and insert them into the Queue.

Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

Step 5 - Repeat steps 3 and 4 until queue becomes empty.

Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph.

Implementation:

```
#include<iostream>
using namespace std;
class graph
{

 int cost[10][10],i,j,k,n,m,queue[10],front,rear,v;
 bool visited[10];
 public:
    graph()
     {
       front=-1;rear=-1;
       for ( int i=1;i<=10;i++) visited[i]=false;
       for (int i=0;i<10;i++)
         for(j=0;j<10;j++)
            cost[i][j]=0;
     }

    void accept();
    void BFS();
};

void graph::accept()
 {
   cout<<"enter the no of vertices"<<endl;
   cin>>n;
   cout<<"enter the no of edges"<<endl;
   cin>>m;
   cout<<"\nEnter EDGES"<<endl;
   for(k=1;k<=m;k++)
     {
      cin>>i>>j;
      cost[i][j]=1;
      cost[j][i]=1;
     }
}

void graph::BFS()
{
  cout<<"enter the initial vertex"<<endl;
  cin>>v;
```

```
cout<<"visited vertices\n";

visited[v]=true;
queue[++rear]=v;
```

```
  while(front!=rear)
    {
      v=queue[++front];
      cout<<v;
      for(j=1;j<=n;j++)
        if(!visited[j]&& cost[v][j]==1)
          {
            queue[++rear]=j;
            visited[j]=true;
          }
    }
}

int main()
{
 graph ob;
 ob.accept();
 ob.BFS();
 return 0;
}
```

Complexity:

The time complexity of BFS is O(V + E), where V is the number of nodes and E is the number of edges.