1.

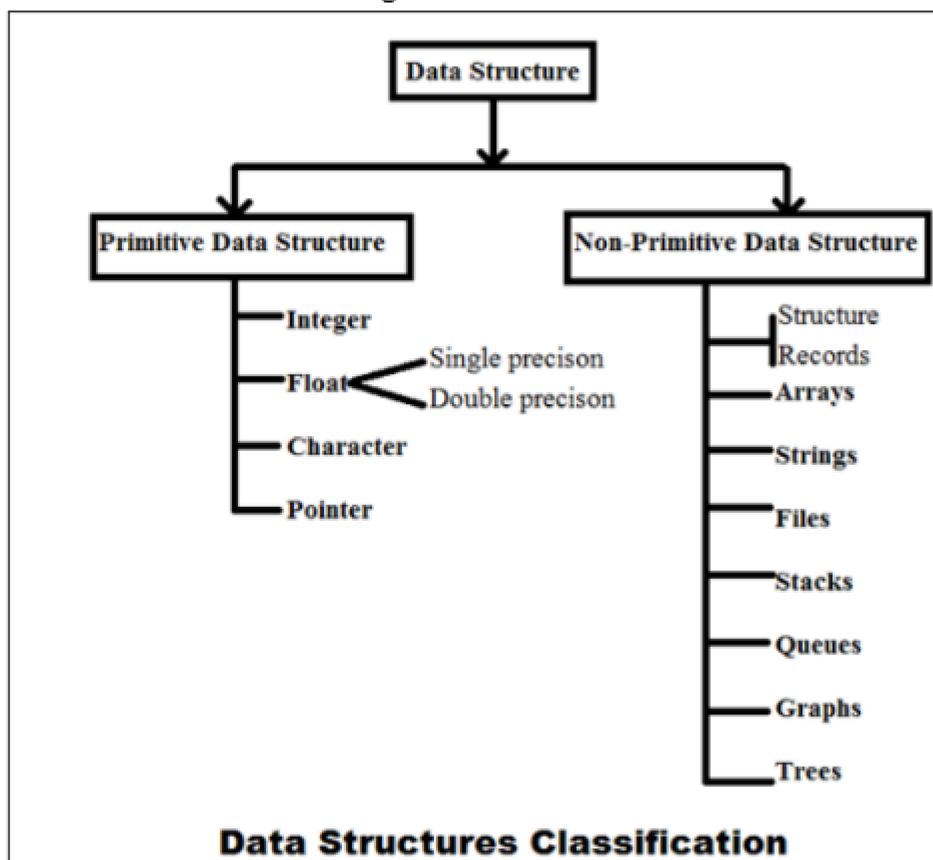
a. Define Data Structures. Explain the various operations on Data Structures. 6 M

Ans:

In terms of computing subjects, Data structure represents name, type, size and format of data in which way it is efficiently organized in computer and accessed for different operations. Some examples of data structures are arrays, stacks, queues, linked lists.

A data structure can be classified in different ways:

· Primitive and non-primitive
· Linear and non-linear
· Homogeneous and non-homogeneous
· Static and dynamic
There is sub-classification also shown in the figure.



Data Structures Classification

The possible operations on the linear data structure are:
· Traversal : Visiting all elements
· Searching: Searching an element
· Sorting: arranging elements in ascending or descending order
· Insertion: inserting after/before searching element or at position
· Deletion: Removing an element by searching/finding
· Merging: Merging more than one collection in a single collection.

b. Define Structures. Explain the types of structures with examples for each.

Structure is a data structure whose individual elements can differ in type. It may contain integer elements, character elements, pointers, arrays and even other structures can also be included as elements within a structure. struct is keyword to define a structure.

Syntax:

```
struct  tag
{
        type member1;
        type member2;
        type member3;
        ......
        type member n;
};
```

New structure type variables can be declared as follows:

```
  struct tag var1, var2, var3, ....... varn, var[10];
```

Example: Structure declaration:

```
    struct student
    {
     int roll;
     char name[20];
     int marks;
};
```

Structure variable declaration:

```
struct student s1,s2;
```

Structure initialization examples:

With declaration:

```
    struct student
    {
     int roll;
     char name[20];
     int marks;
} s1= {111, "Pratap", 76};
```

By individual members:

```
s2.roll = 222;
strcpy(s2.name, "Rubiya");
s2.marks=89;
```

Types of structures:

i) Array of structure:

Whole structure can be an element of the array. A student structure with members roll, name and marks:

```
struct student
{
  int roll;
  char name[30];
  int marks;
};
```

Now we can use this template to create array of 60 students for their individual roll, name and marks.

```
struct student s[60];   /* array of structure */
to access roll of student x : s[x].roll
```

ii) Nested Structure : A structure is defined within a structure or structure variable of another structure is defined within the structure.

```
struct emp {
        int empid; char ename[30]; long in salary;
        struct date { int dd,mm,yy} dob,doj;
} e[500];
```

Here, date structure is defined (with 3 members date, month and year) within emp array of structure.

Members can be access as: structure.memberstructure.member

Example:

e[5].dob.dd = 12;

iii) Self-referential structure:

A member of the structure is pointer to structure itself. It will represent a chained or linked list

structure:

Example:

```
struct list {
    int data;
    struct list *next;
};
```

Here, next is the pointer to similar/same structure. So, it is the xample of a self-referenctial structure.

c. List and explain functions supported in C for Dynamic memory allocation. 7 M

Ans:

Memory can be allocated/ de-allocated at run-time as and when required. C has four in-built functions for the same with header file stdlib.h malloc(), calloc() and realloc() to allocate and free() to de-allocate

malloc():

The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size in bytes and return a pointer of type void which can be casted into pointer of any form.

Syntax of malloc()

ptr=(cast-type*)malloc(byte-size);

example: ptr=(int*)malloc(100*sizeof(int));

calloc()

The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax for calloc():

ptr=(cast-type*)calloc(n,element-size);

ptr=(int*)calloc(25,sizeof(int));

realloc():

If the previously allocated memory is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc().

Syntax/example:

ptr=realloc(ptr, newsize);

free():

Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The
programmer must use free() explicitly to release space.

Syntax/example:

free(ptr);

2.

a. Define pattern matching. Write the Knuth Morris Pratt pattern maching algorthin and apply the same \to search the pattern ‘abcdabcy’ in the text ‘abcxabcdabxabcdabcy’. 10M

Ans:

Pattern matching: It is very common to search a sub-string within a string or to search a pattern in given text. The process of searching apattern within given text is know as pattern matching. There are 2 popular methods i.e. Brute-Force and KMP (Knuth Morris Pratt)

```
/* KMP pattern searching algorithm - Implementation  */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void calcLPS(char pat[], int M, int lps[])
{
   int len = 0,i;
   lps[0] = 0; /* lps[0] is always 0 */
   i = 1;
   while (i < M) { /* the loop calculates lps[i] for i = 1 to M-1 */
  if (pat[i] == pat[len]) { len++;  lps[i] = len; i++; }
  else  /* This is tricky. Consider the example for patter "AAACAAAA"*/
  if (len != 0) len = lps[len - 1];/* i is not incremented here */
  else { lps[i] = 0; i++; }/* i is incremented here */
   }
}

void KMPSearch(char* pat, char* txt)
{
   int M = strlen(pat), N=strlen(txt), i,j, found=0;
   int *lps;
   lps=malloc(M*sizeof(int)); for(i=0;i<M;i++) lps[i]=0;
   /* lps[] will hold the longest prefix/ proper suffix values for pattern
     Preprocess the pattern (calculate lps[] array) */
   calcLPS(pat, M, lps);
   printf("lps[] = {"); for(i=0;i<M;i++)printf(" %d,",lps[i]); printf("};\n");
   i=0; j=0;
   while (i < N) {
  if (pat[j] == txt[i]) { j++; i++; }
  printf("i=%d j=%d\n",i,j);
  if (j == M) {
```

```
    found=1;
    printf("Pattern %s is found at index %d in %s\n",pat,i-j,txt);

    j = lps[j - 1];
  }

  /* mismatch after j matches */
  else if (i < N && pat[j] != txt[i]) {
    /* Do not match lps[0..lps[j-1]] characters,
       they will match anyway next */
    if (j != 0) j = lps[j - 1];
    else i = i + 1;
  }

   }
   if(!found) printf("Pattern %s is not found in %s\n",pat,txt);
}

int main()
{
   char txt[] = "abcxabcdabxabcdabcy";
   char pat[] = "abcdabcy";
   KMPSearch(pat,txt);
   return (0);
}
```

Simulation of output with lps array for the pattern ‘abcdabcy’ in the text ‘abcxabcdabxabcdabcy’

lps[] = { 0, 0, 0, 0, 1, 2, 3, 0,};
i=1 j=1
i=2 j=2
i=3 j=3
i=3 j=0
i=5 j=1
i=6 j=2
i=7 j=3
i=8 j=4
i=9 j=5
i=10 j=6
i=10 j=2
i=10 j=0
i=12 j=1
i=13 j=2
i=14 j=3
i=15 j=4
i=16 j=5
i=17 j=6
i=18 j=7

i=19 j=8
Pattern abcdabcy is found at index 11 in abcxabcdabxabcdabcy

b. Write the fast Transpose algorithm to transpose the given sparse matrix. Express the given sparse matrix as triplets and find its transpose. 10M

$$\begin{pmatrix} 10 & 0 & 0 & 25 & 0 \\ 0 & 23 & 0 & 0 & 45 \\ 0 & 0 & 0 & 0 & 32 \\ 42 & 0 & 0 & 31 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 30 & 0 & 0 \end{pmatrix}$$

Ans:

First converting Sparse matrix into triplets:
Total rows: 6
Total Columns: 5
Total Non-zero elements: 8

6	5	8
0	0	10
0	3	25
1	1	23
1	4	45
2	4	32
3	0	42
3	3	31
5	2	30

Transposing

5	6	8
0	0	10
0	3	42
1	1	23
2	5	30
3	0	25
3	3	31
4	1	45
4	2	32

/*Fast Transpose algorithm-routine implement */
(a) void transpose(int trip1[][3],int trip2[][3])

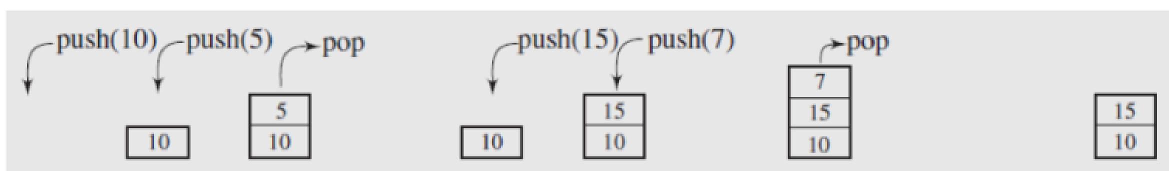
(b) {
(i) int x,y,z,n;
(ii) trip2[0][0] = trip1[0][1];
(iii) trip2[0][1] = trip1[0][0];
(iv) trip2[0][2] = trip1[0][2];
(v) z=1;
(vi) n=trip1[0][2];
(vii) for(x=0;x<trip1[0][1];x++)
1. for(y=1;y<=n;y++)
a. /*if a column number of current triple==x
b. then insert the current triple */
c. if(x==trip1[y][1])
d. {
i. trip2[z][0]=x;
ii. trip2[z][1]=trip1[y][0];
iii. trip2[z][2]=trip1[y][2];
iv. z++;
e. }
}

3.
a. Define stacks. List and explain the various operations on stacks using arrays with stack overflow and stack underflow conditions. 10M

Ans:

A stack is an ordered list in which the insertion (also called push and add) and deletion (also called pop and remove) are made at one end called the top.
Given a stack S=(a0, …, an-1), we say that a0 is the bottom element, an-1 is the top element and ai is on top of ai-1 for 0 < I < n. Stack is also known as a Last-In-First-Out (LIFO) list.

Inserting and deleting elements in a stack.
A sequence of push and pop operations on a stack which is initially empty is shown in the figure. Stack contents are shown immediately right to the operations performed.



Stacks can be implemented using arrays or linked list. In array implementation of stack, stack size has to be specified when the array is created. Stacks can be implemented using dynamic arrays as well where the array size can be expanded when the need arises.

Array implementation of a stack
A C program to implement a stack is shown below.

```
#include <stdio.h>
#define MAX_SIZE 100
int stack[MAX_SIZE];
int top = -1;
int isEmpty() {   if (top < 0) return 1;   else return 0; }
```

```
int isFull() {   if (top >= MAX_SIZE-1) return 1;   else return 0; }

void push(int item) {
  if (top >= MAX_SIZE -1)    printf("Stack is Full. Push Failed\n");
  else    // increment top and then store the item.
    stack[++top] = item;
}

int pop() {
  int item;
  if (top < 0)     printf("Stack is Empty. Pop Failed\n");
  else    // get the item to return and then decrement top.
eturn stack[top--];
}

int main() {
  int item, option=1;
  while (option !=0)   {
    printf("Enter option (1: push, 2: pop, 0:Exit):");
    scanf("%d", &option);
    switch (option)  {
      case 1: printf("Enter item to be pushed:");
         scanf("%d", &item);
         push(item);
         break;
      case 2: item = pop();
         printf("Poped %d\n", item);
         break;
      case 0: printf("Exiting\n");
         break;
      default: printf("Invalid option. Retry\n");
    }
  }
}
```

b. Write an algorithm to convert an infix expression to postfix expression and also trace the same for the expression: (a+b) *d + e/f + c 10M

Ans:

The algorithm:
1) Initialize the Stack.
2) Scan the operator from left to right in the infix expression.
3) If the leftmost character is an operand, set it as the current output to the Postfix string.
4) And if the scanned character is the operator and the Stack is empty or contains the '(', ')' symbol, push the operator into the Stack.
5) If the scanned operator has higher precedence than the existing precedence operator in the Stack or if the Stack is empty, put it on the Stack.

6) If the scanned operator has lower precedence than the existing operator in the Stack, pop all the Stack operators. After that, push the scanned operator into the Stack.
7) If the scanned character is a left bracket '(', push it into the Stack.
8) If we encountered right bracket ')', pop the Stack and print all output string character until '(' is encountered and discard both the bracket.
9) Repeat all steps from 2 to 8 until the infix expression is scanned.
10) Print the Stack output.
11) Pop and output all characters, including the operator, from the Stack until it is not empty.

Step by step output for **"(a+b) *d + e/f + c "** expression

Input String	Output Stack	Operator Stack
(a+b) *d + e/f + c		
(a+b) *d + e/f + c		(
(a+b) *d + e/f + c	a	(
(a+b) *d + e/f + c	a	(+
(a+b) *d + e/f + c	ab	(+

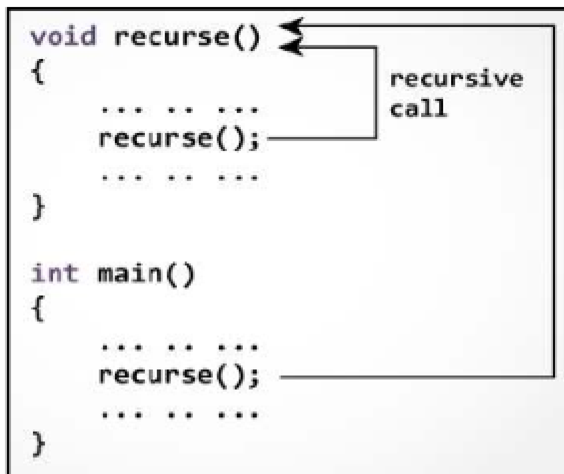
Input String	Output Stack	Operator Stack
(a+b) *d + e/f + c	ab+	
(a+b) *d + e/f + c	ab+	
(a+b) *d + e/f + c	ab+	*
(a+b) *d + e/f + c	ab+ d	*
(a+b) *d + e/f + c	ab+ d	*
(a+b) *d + e/f + c	ab+ d *	+
(a+b) *d + e/f + c	ab+ d *	+
(a+b) *d + e/f + c	ab+ d * e	+
(a+b) *d + e/f + c	ab+ d * e	+/
(a+b) *d + e/f + c	ab+ d * ef	+/
(a+b) *d + e/f + c	ab+ d * ef	+/
(a+b) *d + e/f + c	ab+ d * ef /+	+
(a+b) *d + e/f + c	ab+ d * ef /+	+
(a+b) *d + e/f + c	ab+ d * ef /+ c	+
(a+b) *d + e/f + c	ab+ d * ef /+ c+	

4.

a. Define Recursion. Explain the types of recursion. Write the recursive functions for i) factorial of a number ii) Tower of Hanoi 10M

Ans:

- Recursion is the process which comes into existence when a function calls a copy of itself to work on a smaller problem.
- Any function which calls itself is called recursive function, and such function calls are called recursive calls.
- Factorial of 5 = 5*4*3*2*1
- Factorial of 5 = 5*Factorial(4)
- Factorial of n = n*(Factorial(n-1))



Types of recursion:

· Direct Recursion: When a function calls itself within the same function repeatedly, it is called the direct recursion.

· Indirect Recursion: When a function is mutually called by another function in a circular manner, the function is called an indirect recursion function.

· Tail Recursion: Recursive call at the end of the function

· Non Tail/ Head Recursion: the recursive call will be the first statement in the function.

Linear recursion: A function is called the linear recursive if the function makes a single call to itself at each time the function runs and grows linearly in proportion to the size of the problem.

· Tree Recursion: A function is called the tree recursion, in which the function makes more than one call to itself within the recursive function (Two calls of Fibinacci).

```
/* Recursive function for factorial */
long int factorial(int n)
{
   if (n==0 || n==1)  return (1);
   else    return (n*factorial(n-1));
}

/* Recursive function for Tower of Hanoi */
long int k=0;
void hanoi(int d,char l[],char r[],char m[])
{
    if(d>0)
    {
     hanoi(d-1,l,m,r);
     printf("%7ld. Move disk %d from %s to %s\n",++k,d,l,r);
     hanoi(d-1,m,r,l);
```

```
    }
}
```

b. Give the Ackerman function and apply the same to evaluate A(1,2)   4M

Ans:

- Ackerman's function is a classic example of recursive function. It is well defined total computable function but not primitive recursive.
- Earlier before 1995 the concept was that all computable functions are primitive recursive also.
- Ackerman's function grows exponentially:
- Ackerman's A(x,y) function is defined for integer x and y:

$$A(x,y)= \begin{cases} y+1 & \text{if } x = 0 \\ A(x-1,1) & \text{if } y = 0 \\ A(x-1,A(x,y-1)) & \text{otherwise} \end{cases}$$

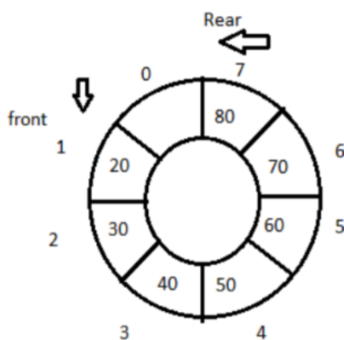
```
/*Ackerman's function in C */
int A(int m, int n)
{
 if (m == 0) return n+1;
 if (n == 0) return A( m - 1, 1 );
 return A( m - 1, A( m, n - 1 ) );
}
```

The result of A(1,2) will be 4 and total function calls in recursion will be 10 times.

c. Explain the various operations of Circular queue using array.   6M

Ans:

- Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle logically
- Disadvantages of linear queue:
    - o Items cannot be inserted from front end even the space is available
    - o Items cannot be deleted from rear or middle
- This can be solved using a circular queue where front and rear are adjacent when queue is full. If any item is deleted then variable rear can be used to insert item to utilize it.



Operations on Circular Queue:

- front: to delete item from front position.

- rear: to insert item at rear position.
- insertQ(value) This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.

Steps:Check whether queue is Full

if ((rear == SIZE-1 && front == 0) || (rear == front-1)).
◦ If it is full then display Queue is full.
If queue is not full then, check if (rear == SIZE – 1 && front != 0)
if it is true then set rear=0 and insert element.
• delQ() This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.
Steps:Check whether queue is Empty means check (front==-1).

- If it is empty then display Queue is empty. If queue is not empty then step 3
- Check if (front==rear) if it is true then set front=rear= -1 else check if (front==size-1), if it is true then set front=0 and return the element.
- To calculate from and rear:

front = (front + 1) % size
rear = (rear + 1) % size

5.
a. Give the node structure of create a single linked list of integers and write the functions to perform the
following operations.
i) Create a list containing 3 nodes with data 10, 20, 30 using front insertion
ii) Insert a node with data 40 at the end of the list
iii) Delete a node whose data is 30
iv) Display the list contents 10M

Ans:

```
/* Solution by a complete program- Driver function main() is written at last of the answer */
#include<stdlib.h>
#include <stdio.h>

struct llist
{
  int info;
  struct llist *next;
};
typedef struct llist node;
node *start=NULL;

void insert_beg(int data) /* I. to insert from the beginning */
{
  node *temp;
  temp=(node *)malloc(sizeof(node));
  temp->info=data;
  temp->next =NULL;
  if(start==NULL) start=temp;
```

```
 else {   temp->next=start; start=temp;}
}

void insert_end(int data) /* ii. to insert at the end */
{
 node *temp,*ptr;
 temp=(node *)malloc(sizeof(node));
 temp->info=data;
 temp->next =NULL;
 if(start==NULL) start=temp;
 else
 {
  ptr=start;
  while(ptr->next !=NULL) ptr=ptr->next;
  ptr->next =temp;
 }
}

void delete_data(int data) /* iii. Deleting searching data */
{
 node *temp,*ptr;
 temp=start;
 while(temp->next!=NULL) {
    if(temp->next->info==data) {
        ptr=temp->next;
        temp->next=temp->next->next;
        printf("\nThe node containing %d is deleted.",ptr->info);
        free(ptr);
        break;
    }
    temp=temp->next;
 }

}

void display() /* iv. Displaying during and after insertion and deletion */
{
 node *ptr;
 if(start==NULL) printf("The List is empty.\n");
else
 {
  ptr=start;
  printf("\nThe Linked List:\n");
  while(ptr!=NULL)
  {
   printf("%d -->",ptr->info );
   ptr=ptr->next ;
  }
 }
}
```

```
int main()
{
  insert_beg(30);
  insert_beg(20);
  insert_beg(10);
  insert_end(40);
  display();
  delete_data(30);
  display();
  return (0);

}
```

Expected output:
The Linked List:
10 -->20 -->30 -->40 -->
The node containing 30 is deleted.
The Linked List:
10 -->20 -->40 -->

b. Write the functions for:
i) Finding length of the list
ii) Concatenate two lists
iii) Reverse a list 10M

Ans:

i) Finding length of the list

```
/*recursive function to find length */
int length(node *temp){
   if(temp==NULL)
      return (0);
   else
       return (1+length(temp->next));

}
```

ii) Concatenating 2 lists

```
/*Concatenating two linked lists
 list1 & list2 are two singly linked list and their pointer name is next */
void concatenate(struct node *list1, struct node *list2)
{
   if(a->next == NULL)
     list1->next =list2;
       else
    concatenate (list1->next, list2); #recursion
}
```

iii) Reversing a list

```
struct LIST { int data;  struct LIST *next; };
typedef struct LIST node;
static void reverse(node **list)
{
    node *prev = NULL,*current=*list,*next=NULL;
    while (current != NULL)
    {
  next = current->next;
  current->next = prev;
  prev = current;
  current = next;
    }
    *list = prev;
}
```

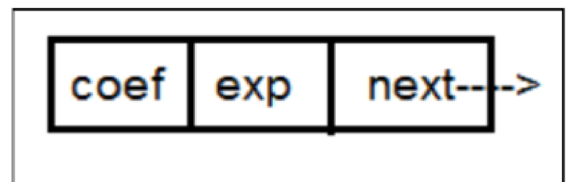
6.

a. Write the node representation for the linked representation of a polynomial. Explain the algorithm to add 2 polynomials represented as linked list 8M

Ans:

Node Representation for the linked representation of a polynomial

```
struct polynode {
      int coef;
      int exp;
      struct polynode * next;
  };
  typedef struct polynode *polyptr;
```



Algorithm to add 2 polynomials represented as linked list

Adding polynomials p1 & p2 storing the result in p3

To do this, we have to break the process down to cases:

1) Case 1: exponent of p1 > exponent of p2

a. Copy node of p1 to end of p3.

b. [go to next node]

2) Case 2: exponent of p1 < exponent of p2

a. Copy node of p2 to end of p3.

b. [go to next node]

3) Case 3: exponent of p1 = exponent of p2

a. Create a new node in p3 with the same exponent and with the sum of the coefficients of p1 and p2.

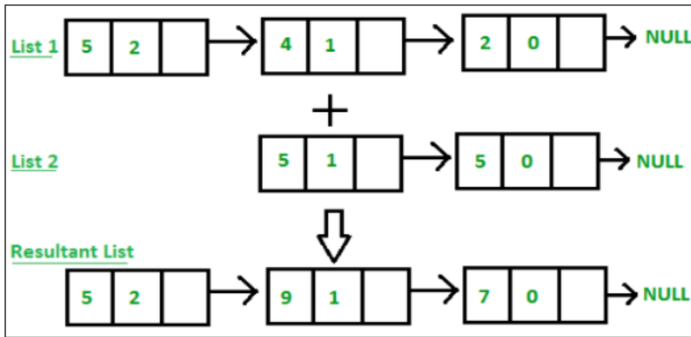
Explaining:

Input:

p1 = 5x^2 + 4x^1 + 2x^0

p2 = 5x^1 + 5x^0

Output = 5x^2 + 9x^1 + 7x^0



b. For the given sparse matrix, write the diagrammatic linked list representation. 4M

$$A = \begin{pmatrix} 3 & 0 & 0 & 0 \\ 5 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 8 \\ 0 & 0 & 9 & 0 \end{pmatrix}$$

Ans:

• A sparse matrix has relative number of elements 0. Representing a sparse matrix using 2-D array takes substantial amount of space with no use.

• A sparse matrix can be represented by triplets with maximum column of 3 and each row will have corresponding row number, column number of non-zero elements and non-zero element itself.

• Those triples can also be represented by a singly linked list having members row, col, val (non-zero) and address of next node:

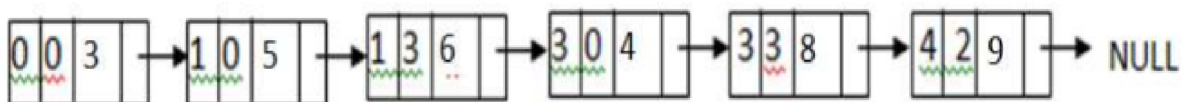
• Structure definition:

```
struct spmlist  {
  int row,col,val;
  struct spmlist *next;
};
```

• Triplets in given sparse matrix (row, colum and non-zero element):

• 0 0 3, 1 0 5, 1 3 6, 3 0 4, 3 3 8, and 4 2 9

• Now linked representation



c. List out the difference between singly linked list and doubly linked list. Write the functions to perform
on following operations on doubly linked list.
i) Insert a node at rear end of the list
ii) Delete a node at the rear end of the list
iii) Search a node with the given key value 8M

Ans:

i) Inserting a node at the rear end of a doubly linked list

void add_end (struct dnode **s, int num)/* adding at end */

```
{
  struct dnode *r, *q = *s ;
  if ( *s == NULL )  /* if the linked list is empty, create new */
  {
    *s = malloc ( sizeof ( struct dnode ) ) ;
    (*s)-> prev = NULL ;(*s)-> data = num ;(*s) -> next = NULL ;
  }
  else /* traverse the linked list till the last node is reached */
  {
    while ( q -> next != NULL ) q = q -> next ;
    r = malloc ( sizeof ( struct dnode ) ) ;/ allocate & create*/
    r -> data = num ; r -> next = NULL ;
     r -> prev = q ; q -> next = r ;
  }
}
```

ii) delete a node at the rear end of a doubly linked list

```
int delete_end(node *h)
{
    node *temp, *temp1, *temp2;
    temp=h;
    if(temp->next==NULL)
    {
      free(temp);
      h=NULL;
      return 0;
    }
    else
    {
      temp2=temp1->prev;
      temp2->next=NULL;
      printf("deleting %d\n", temp1->data);
      free(temp1);
    }
    return 0;
}
```

Iii) Search a node with the given key value

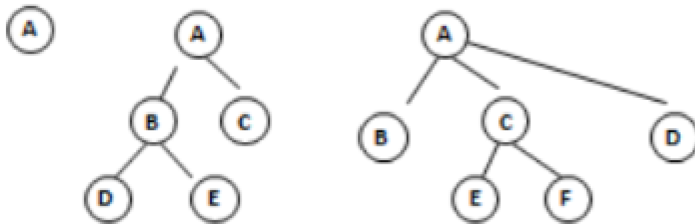
```
void search(struct node *head)  {
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)      {        printf("\nEmpty List\n");      }
    else {
          printf("\nEnter item which you want to search?\n");
          scanf("%d",&item);
          while (ptr!=NULL)       {
              if(ptr->data == item)    {
                    printf("\nitem found at location %d ",i+1);
                    flag=0;
```

```
break;
            }
            else    flag=1;
                   i++;
            ptr = ptr -> next;
        }
        if(flag==1)    printf("\nItem not found\n");
    }
}
```

7.
a. Define a tree? With a suitable example explain:
i) Complete Binary tree
ii) Strict Binary tree
iii) Skewed Binary tree

Ans:

A tree: A tree is a finite set of elements that is either empty or is partitioned in to different disjoint subsets. A Tree is non-linear data structure represents the nodes connected by edges.

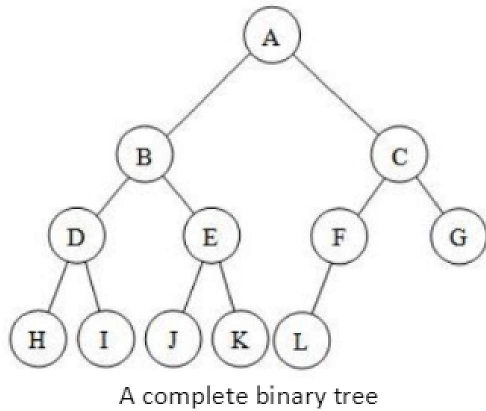


The simplest form of tree is a binary tree. A binary tree consists of a node (called the root node) and
left and right sub-trees. Since each element in a binary tree can have only 2 children, we typically
name them the left and right child.
Both the sub-trees are themselves binary trees.
A Binary Tree node contains following parts.
◦ Data
◦ Pointer to left child
◦ Pointer to right child

```
struct bintree
{
  struct bintree *left;
  int data;
  struct bintree *right;
}
```

i) Complete Binary Tree

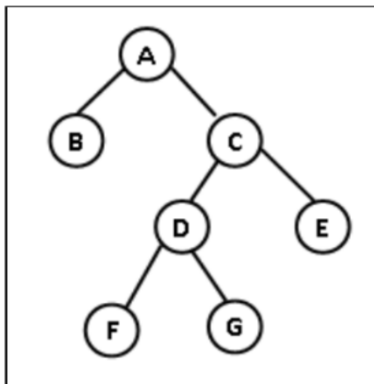
A complete binary tree is almost complete binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.



A complete binary tree

ii) Strict Binary Tree

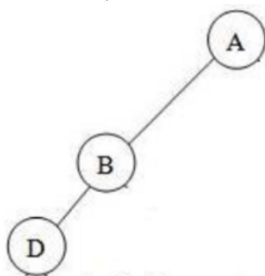
If every non-leaf node in a binary has nonempty left and right subtrees, the tree is termed a strictly binary tree. All Internal nodes will have 0 or two children. A strictly binary tree with n leaves always contains 2n -1nodes. Or, to put it another way, all of the nodes in a strictly binary tree are of degree zero or two, never degree one.

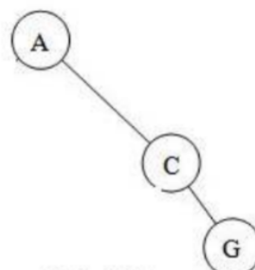


A strict Binary Tree

iii) Skewed Binary Tree

A skewed binary tree is special type of binary tree where a node will have 1 child or 0 (no) child. Only leaf node will not have any child.





A Binary Search tree can be skewed as left skewed (decreasing order) or right skewed (increasing order)

b. Write the routines to traverse the given tree using:
i) Pre-order traversal
ii) Post-order traversal 6M

Ans:

i) Recursive routine of Pre-order traversal: Root-Left-Right

```
void preorder(TREE *tree) {
   if(tree)  {
printf(" %d",tree->data);
preorder(tree->left);
preorder(tree->right);   }
}
```

ii) Recursive routine of Pre-order traversal: Left-Root-Right

```
void inorder(TREE *tree){
   if(tree)  {
 inorder(tree->left);
 printf(" %d",tree->data);
inorder(tree->right);   }
}
```

c. Write the Recursive search algorithm for a Binary Search Tree   04Marks

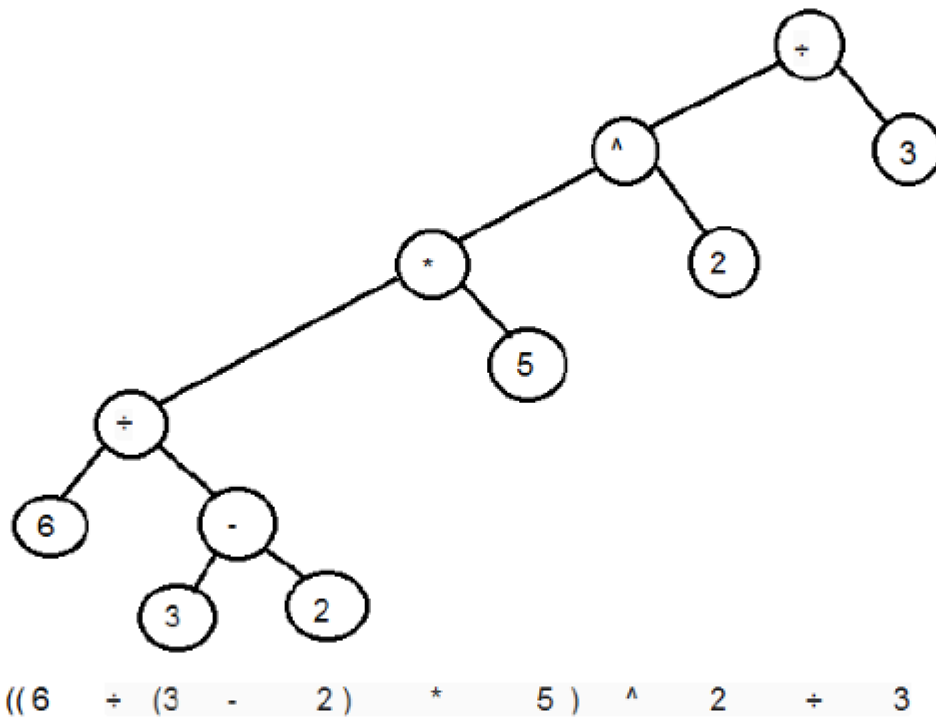
Ans:
A recursive algorithm to search for a key in a BST follows immediately from the recursive structure: If the tree is empty, we have a search miss; if the search key is equal to the key at the root, we have a search hit. Otherwise, we search (recursively) in the appropriate left or right sub-tree.

(a) Start
(b) Input node BST and the key to search
(c) if (root == NULL || root->key == key)
(i) return root;
(d) if (root->key < key)
(i) return search(root->right, key);
(e) return search(root->left, key);
(f) End.

8.
a. Draw the Binary Tree for the following expression. ((6 ÷ (3-2)*5) ^2 ÷ 3 Traverse the above generated tree using Pre-order and post order traversal and also write their respective functions.   10M

Ans:
÷ symbol is old school division symbol so, expression is:
((6 / (3-2)*5) ^2 / 3

The Binary tree for the expression:



Pre-order traversal:
÷ ^ * ÷ 6 – 3 2 5 2 3

Pre-order traversal:

6 3 2 - ÷ 5 * 2 ^ 3 ÷

Recursive functions

```
void preorder(TREE *tree) {
   if(tree)  {
printf(" %d",tree->data);
preorder(tree->left);
preorder(tree->right);   }
}
```

b. Write the routines for
i) Copying of Binary trees
ii) Testing equality of binary trees 10M

Ans:
i) Copying of Binary Tree

```
typedef struct node {     int data;     struct node *left;     struct node *right; } node;

/* copying passed tree as root to newtree using recursion  */
node* copybintree(node *root){
```

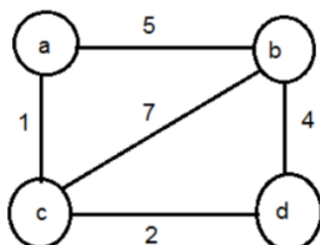
```
    if(root == NULL)
        return NULL;
    /* create a copy of root node */
     node *newtree;
    newtree = (node *) malloc(sizeof(node));
    newtreee->data = data;
    newtree->left = NULL;
newtree->right = NULL;
    /* Recursively create copy of left and right sub tree */
    newtree->left = copybintree(root->left);
    newtree->right = copybintree(root->right);
    /* Return root of copuied tree */
    return newtree;
}
```

ii) Testing equality of Binary Tree

```
// Function/Routine to check if two Binary Search Trees are identical
int isIdentical(struct Node* root1,struct Node* root2)
{
    // Check if both the trees are empty
    if (root1 == NULL && root2 == NULL)
        return 1;
    // If any one of the tree is non-empty  and other is empty, return false
    else if (root1 == NULL || root2 == NULL)
        return 0;
     else { // Check if current data of both trees equal and recursively check for left and right
subtrees
        if (root1->data == root2->data && isIdentical(root1->left, root2->left)
            && isIdentical(root1->right, root2->right))
            return 1;
        else
            return 0;
    }
}
```

9.

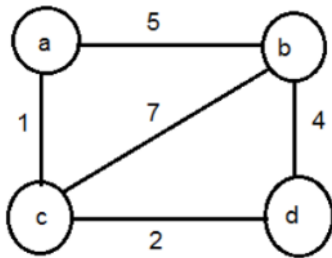
a. Define graph. Give the adjacency matrix and Adjacency list representation for the following graph. 8M



Ans:

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as,

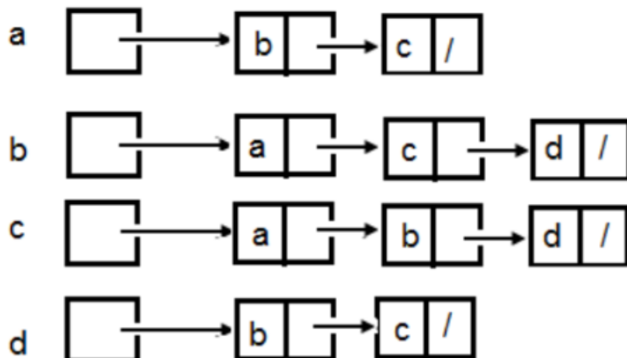
A graph is a set of vertices and edges, which connect them. We write: G = (V,E) where V is the set of vertices and the set of edges, E = { (vi,vj) } where vi and vj are in V.



The adjacency matrix of a weighted graph can be used to store the weights of the edges We make a matrix of nodes x nodes for corresponding row and column and fill the cells with weights of the connecting edges/arcs.

	a	b	c	d
a	0	5	1	0
b	5	0	7	4
c	1	7	0	2
d	0	4	2	0

Adjacency list:
An array of linked lists is used. The size of the array is equal to the number of vertices.



b. Write the algorithm for following graph traversal method. 8M
i) Breadth First Search
ii) Depth first search

Ans:

i) Breadth First Search (BFS)
Breadth First Traversal of a graph is roughly analogous to level by level traversal of an ordered tree. If the traversal has visited the vertex v, then it next visits all the vertices adjacent to v, putting the various adjacent to these in a waiting list to be traversed after all vertices adjacent to v have been visited.
Breadth first search: (Algorithm)
1. Begin
2. unmark all vertices
3. choose some starting vertex x
4. mark x

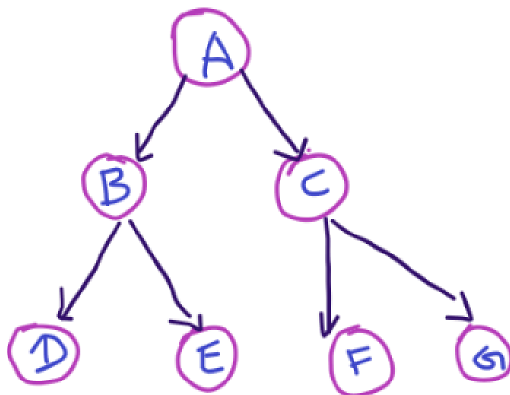
5.   list L = x
6.   tree T = x
7.   while L nonempty
8.   choose some vertex v from front of list
9.   visit v
10.   for each unmarked neighbor w
        mark w
                    add it to end of list
      add edge vw to T
11. End

ii) Depth First Search (DFS)

Depth first search is another way of traversing graphs, which is closely related to pre-order traversal of a tree. Stack is used to implement DFS traversal by recursion

Implementation in C using recursion.

```
int G[10][10],visited[10]={0},n;
/*n is number of vertices and graph is stored in array G[10][10] */
void DFS(int i)
{
   int j;
   printf("\n%d",i);
   visited[i]=1;
   for(j=0;j<n;j++)
     if(!visited[j]&&G[i][j]==1)
        DFS(j);
}
```



BFS - ABCDEFG

DFS - ABDECFG

c. Write an algorithm for insertion sort.   4M

Ans:

Insertion Sort is a sorting algorithm where the array is sorted by taking one element at a time. The principle behind insertion sort is to take one element, iterate through the sorted array & find its correct position in the sorted array.

Start
Step 1- Input Array
Step 2 − If it is the first element, it is already sorted. return 1;

Step 3 − Pick next element
Step 4 − Compare with all elements in the sorted sub-list
Step 5 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted
Step 6 − Insert the value
Step 7 − Repeat until list is sorted
End.

10.
a. Define Hashing. Explain any 3 hashing functions. 8M

Ans:
Hashing is a technique to convert a range of key values into a range of indexes of an array. In general we use modulo operator to get a range of key values. Hashing technique use a specialfunction called the Hash function which is used to map a given value with a particular key for faster access of elements. Each Item can be located in a Bin/Bucket/Shelf according to its key value (number)
Most simple Hash Method/Function
```
int hash(int key) {  return key % SIZE;  }
```

A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes. There are many hash functions that use numeric or alphanumeric keys. Here 3 hash functions are described with examples:

A. Division Method.
B. Mid Square Method.
C. Folding Method.

A) Modular method /Division Method :
It is the most simple method of hashing an integer x. This method divides x by size of hash table M and then uses the remainder obtained. In this case, the hash function can be given as
$h(x) = x \bmod M$
Example: To calculate the hash values of keys 1234 and 5462. Solution Setting M = 97, hash values can be calculated as:
h(1234) = 1234 % 97 = 70
h(5642) = 5642 % 97 = 16

B) Mid-square Method:
The mid-square method is a good hash function which works in two steps:
Step 1: Square the value of the key. That is, find k2.
Step 2: Extract the middle r digits of the result obtained in Step 1.
In the mid-square method, the same r digits must be chosen from all the keys. Therefore, the hash function can be given as: h(k) = s where s is obtained by selecting r digits from k2.
Example: To calculate the hash value for keys 1234 and 5642 using the mid-square method:
The hash table has 100 memory locations.
Solution: Note that the hash table has 100 memory locations whose indices vary from 0 to 99.

This means that only two digits are needed to map the key to a location in the hash table, so r = 2.
When k = 1234, k2 = 1522756, h (1234) = 27
When k = 5642, k2 = 31832164, h (5642) = 21

C) Folding Method:
The folding method works in the following two steps:
Step 1: Divide the key value into a number of parts. That is, divide k into parts k1, k2, ..., kn, where each part has the same number of digits except the last part which may have lesser digits than the other parts.
Step 2: Add the individual parts. That is, obtain the sum of k1 + k2 + ... + kn. The hash value is produced by ignoring the last carry, if any. Note that the number of digits in each part of the key will vary depending upon the size of the hash table. .
Example: Given a hash table of 100 locations, calculate the hash value using folding method for keys 5678 and 34567.
Solution: Since there are 100 memory locations to address, we will break the key into parts where each part (except the last) will contain two digits. The hash values can be obtained as shown below:
Key: 34567, Parts: 34,56 and 7, Sum:97 so, Hash value: 97
Key: 5678, Parts: 56 and 78, Sum:134 Hash value: 34 (ignore carry)

b. Explain static and dynamic hashing in detail : 8M

Ans:

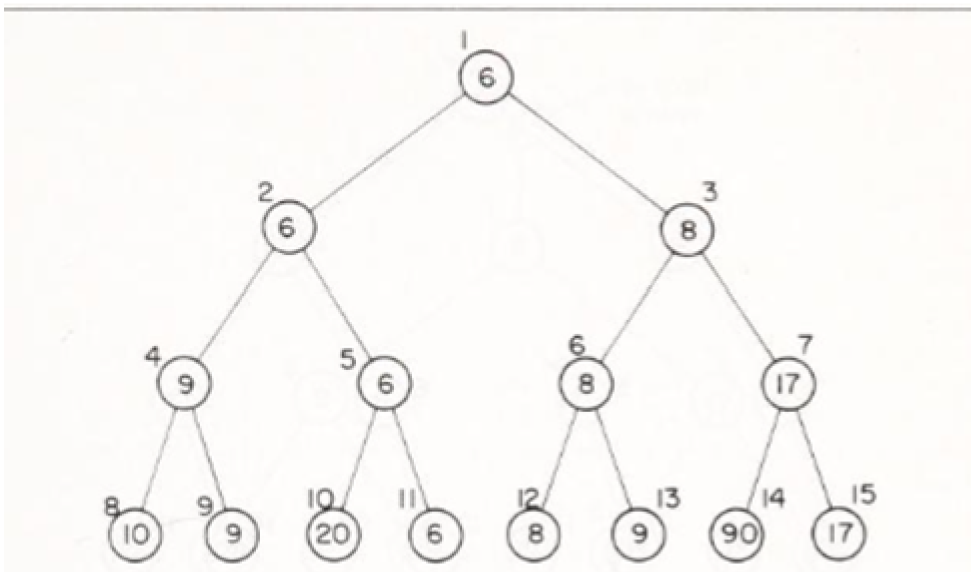
Static hashing:
- Single hash function h(k) on key k
- Desirable properties of a hash function
- Uniform: Total domain of keys is distributed uniformly over the range
- Random: Hash values should be distributed uniformly irrespective of distribution of keys
- Example of hash functions: h(k) = k MOD m (k is key & m is size of hash table)

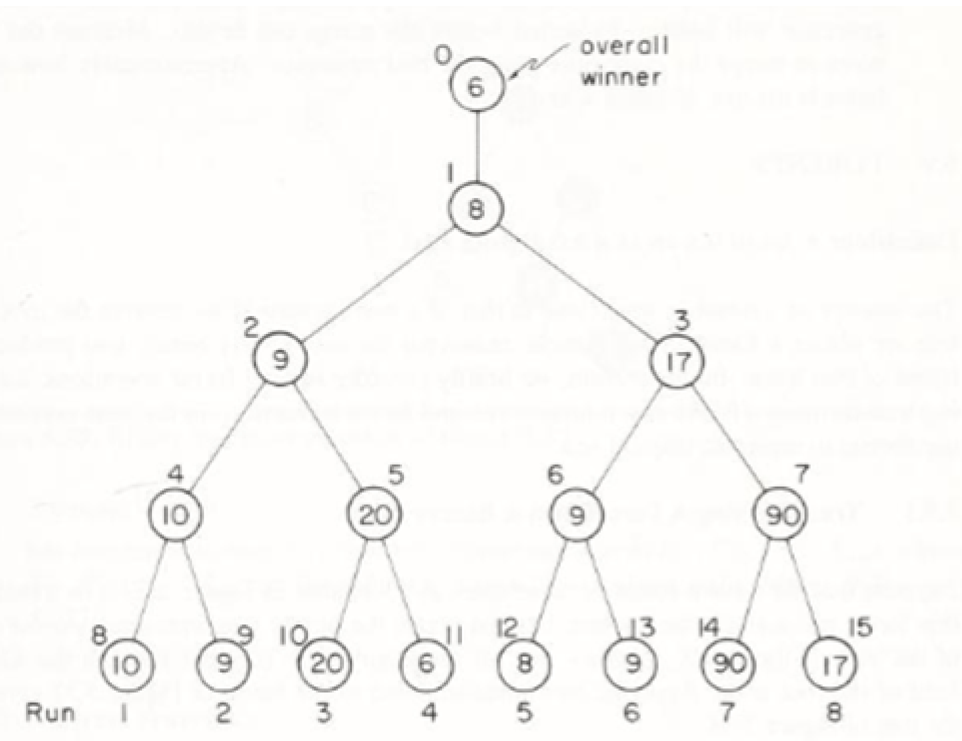
Collision resolution
- Chaining
  - Load factor
  - Primary pages and overflow pages (or buckets)
- Search time more for overflow buckets
- Open addressing
  - Linear probing
  - Quadratic probing
  - Double hashing

Problems of static hashing
- Fixed size of hash table due to fixed hash function
- Primary/secondary Clustering (We should keep size of hash table a prime number to reduce clustering)
- May require rehashing of all keys when chains or overflow buckets are full

Dynamic hashing:

- In this hashing scheme the set of keys can be varied & the address space is allocated dynamically.
- If a file F is collection of record a record R is key+data stored in pages (buckets) then space utilization:
  Number of records/(Number of pages*Page capacity)
- Hash function modified dynamically as number of records grow
- Needs to maintain determinism
- Extendible hashing
- Linear hashing
  Again: Dynamic hashing Hash function modified dynamically as number of records grow
  Needs to maintain determinism Extendible hashing and Linear hashing
- Organize overflow buckets as binary trees
- m binary trees for m primary pages
- h0(k) produces index of primary page
- Particular access structure for binary trees
- Family of functions : g(k) = { h1(k),....hi(k),....}
- Each hi(k) produces a bit
- At level i, hi(k) is 0, take left branch, otherwise right branch Example: bit representation





overall winner
0
6
1
8
2
9
3
17
4
10
5
20
6
9
7
90
8
10
9
9
10
20
11
6
12
8
13
9
14
90
15
17
Run
1
2
3
4
5
6
7
8