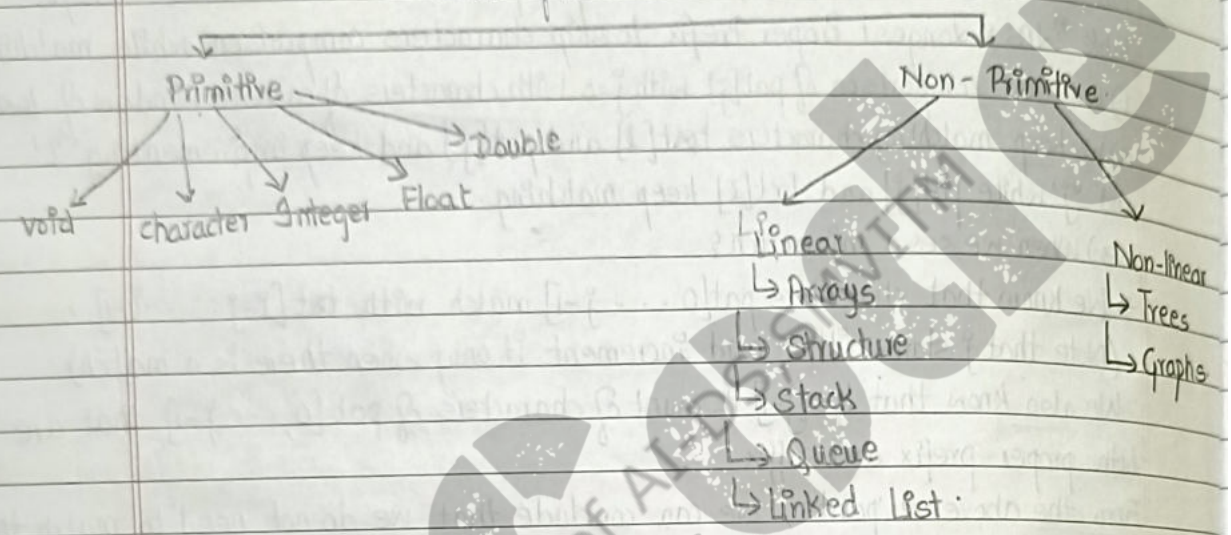
MODEL QP-1 BCS304 - ANSWERS By, VANDANA.U

Q1. a) Define Data Structures. With a neat diagram, explain the classification of data structures with examples.

⇒ A data structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.

The logical or mathematical model of a particular organization of data is called a data structure.



Data Structures is mainly classified into 2 types:

1) Primitive Data structure: These are basic data structures and are directly operated upon by the machine instructions. These data types consists of characters that cannot be divided and hence they are also called as simple data types. Ex: Integers, Floating point numbers, characters & Pointers etc.

2) Non-Primitive Data structure: These are derived from the primitive data structures. Ex: Array, Stack, Graphs etc.

Based on the structure and arrangement of data, non-primitive data structure is divided into:

a) Linear data structure: Here the elements form a sequence or a linear list. One way to represent them is to have relationship b/w elements by means of sequential memory location. Other way is representing elements by means of pointers or links. Ex: Array, Stacks etc.

b) Non-linear data structure: Here the data is not arranged in a sequence or linear fashion. This structure is mainly used to represent data containing a hierarchical relationship b/w elements. Ex: Trees, Graphs etc.

b) What do you mean by pattern matching? Outline the Knuth Morris Pratt (KMP) algorithm and illustrate it to find the occurences of the following

pattern : P : ABCDABD

S : ABC ABCDAB ABCDABCDABDE.

⇒ Pattern matching is the problem of deciding whether or not the given string pattern 'P' appears in a string text 'T'.

Outlining the KMP algorithm:

This algorithm compares the character by character from left to Right. But whenever a mismatch occurs, it uses a pre-processed table called "Prefix Table" LPS : Longest Proper Prefix to skip characters comparison while matching.

1.) We start comparison of pat[j] with j=0 with characters of current window of text.

2.) We keep matching characters txt[i] and pat[j] and keep incrementing 'i' and 'j' while pat[j] and txt[i] keep matching.

3.) When we see a mismatch:

→ We know that characters pat[0 . . . j-1] match with txt[i-j . . . i-1].

(Note that j starts with 0 and increment it only when there is a match).

We also know that lps[j-1] is count of characters of pat[0, . . . j-1] that are both proper prefix and suffix.

From the above 2 points, we can conclude that we do not need to match these lps[j-1] characters with txt[i-j . . . i-1] because we know that these characters will anyway match.

KMP pattern matching:

P=ABCDABD S=ABC ABCDAB ABCDABCDABDE.

Step 1 : Construct Failure Function/LPS Table

0	1	2	3	4	5	6
A	B	C	D	A	B	D
0	0	0	0	1	2	0

Step 2 : Pattern matching using LPS table:

	A	B	C		A	B	C	D	A	B		A	B	C	D	A	B	C	D	A	B	D	E
Step a):	Ă	B̆	C̆	D̸	A	B	D																	
					Ă	B̆	C̆	D̆	Ă	B̆	D̸													
									Ă	B̆	C̸	D	A	B	D									
												Ă	B̆	C̆	D̆	A	B	D̸						
																Ă	B̆	C̆	D̆	A	B	D		

c) Write a C-program in C to implement push, pop and display operations for stacks using arrays.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define max_size 5
int
int
void push();
void pop();
void display();
void main()
{
int choice;
printf("Stack Operations :");
printf("1: Push");
printf("2: Pop");
printf("3. Display");
printf("4. Exit");
while(1)
{
printf("Enter your choice");
scanf("%d", &choice);
switch(choice)
{
case 1: push();
        break;
case 2: pop();
  if (flag)
    printf("The deleted item %d", item);
temp=top;
break;
case 3: display();
        break;
case 4: exit(0);
        break;
default: printf("Invalid choice");
break; }}}
```

```
void push()
{
if(top==(max_size-1))
 {
printf("Stack Overflow");
 }
else
{
printf("Enter the element to be inserted");
scanf("%d", &item);
top=top+1;
stack[top]=item;
 }
temp=top;
}
void pop()
{
if(top==-1)
{
printf("Stack Underflow");
flag=0;
 }
else
item=stack[top];
 top=top-1;
}}
void display()
{
int i;
top=temp;
if(top==-1)
{
printf("Stack is empty");
}
else
{
```

```
printf("The stack elements are");
for(i=top; i>=0; i--)
{
printf("%d", stack[i]);
}}}
```

Q2 a) Explain in brief the different functions of dynamic memory allocation.

⇒ Dynamic memory allocation: The memory allocation and de-allocation is performed during run-time of the program.

Different Dynamic Memory Allocation functions:

a) malloc():

Stands for memory allocation.

Function allocates memory and returns a pointer of type void * to the start of that memory block. If the function fails, it returns NULL. Therefore it is necessary to verify pointer returned is not NULL.

Syntax: ptr=(datatype *)malloc(size_t size);

Here ptr → Pointer Variable of the type datatype, size → no. of bytes required.

b) calloc():

Stands for: Contiguous Allocation.

It is used to dynamically allocate multiple blocks of memory and each block is of the same size.

Syntax: ptr=(datatype *)calloc(n,size);

Here n → number of blocks to be allocated.

Ex: ptr= calloc(20, sizeof(int));

// This function computes the required memory of 20×2 bytes (for int)= 40 bytes of memory block is allocated.

c) realloc():

It is used to modify the size of allocated blocks of

Ex Syntax: ptr=(void *)realloc(void *ptr, size_t size); //Here size → new size.

Ex: int *ptr, *ptr1;

```
ptr=(int *)malloc(num, sizeof(int));
. . . . .
ptr1=(int *)realloc(ptr, num, sizeof(int));
```

d) free():

Deallocate the allocated memory which was done using malloc, calloc and realloc()

Syntax: free(pointer_name);

Ex: free(str);

⇒ i) Compare 2 Strings:

```
void scomp(char str1[10], char str2[10])
{
int i=0, k=0;
while(str1[i]!='\0')
{
if(str1[i]==str2[i])
{
i++;
}
else
{
k=1;
break;
}}
if(k==1 || str2[i]!='\0')
{
printf("Strings are not equal");
}
else
{
printf("Strings are equal");
}}
```

ii) Concatenate two strings:

```
void scat(char str1[], char str2[])
{
int i=0, j;
{
while(str1[i]!='\0')
i++;
j=0;
while(str2[j]!='\0')
{
str1[i++]=str2[j++];
} str1[i++]='\0';
}
printf("String1 after concatenation is %s", str1);
}
```

iii) Reverse a string:

```
void reverse(char str1[10])
{
int i, len, temp;
len=strlen(str1);
for(i=0; i<len/2; i++)
{
temp=str1[i];
str1[i]=str1[len-i-1];
str[len-i-1]=temp;
}}
```

1) Write a function to evaluate the postfix expression. Illustrate the same for the given postfix expression: ABC-D* +EF$+ and assume A=6, B=3, C=2, D=5, E=1 and F=7

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<ctype.h>
float compute(char symbol, float op1, float op2)
{
switch(symbol)
{
case '+': return op1+op2;
          break;
case '-': return op1-op2;
          break;
case '*': return op1*op2;
          break;
case '/': return op1/op2;
          break;
default: return 0;
}}
void main()
{
float s[20], res, op1, op2;
int top, i;
char postfix[20], symbol;
printf("Enter postfix expression");
scanf("%s", postfix);
top=-1;
for(i=0; i<strlen(postfix); i++)
{
symbol=postfix[i];
if(isdigit(symbol))
s[++top]=symbol-'0';
else
{
op2=s[top--];
op1=s[top--];
res=compute(symbol, op1, op2);
s[++top]=res;
}}
res=s[top--];
printf("The result is %f", res);
}
```

Q3.a) Develop a Cprogram to implement insertion, deletion and display operations on a linear Queue

```
#include<stdio.h>
#define Q_SIZE 5
int ch, r, f, i, item, q[Q_SIZE];
void insertrear();
void deletefront();
void display();
void main()
{
int f=0, r=-1;
for(;;)
```

```
{
printf("1.Insert 2 Delete 3. Display 4. Exit");
printf("Enter the choice");
scanf("%d", &ch);
switch(ch)
{
case 1: insertrear();
        break;
case 2: deletefront();
        break;
case 3: display();
        break;
case 4: exit(0);
        break;
default: printf("Invalid choice");
}}
void insertrear()
{
int item, r=-1;
if(r==Q_SIZE-1)
{
printf("Queue overflow");
return;
}
printf("Enter the item");
scanf("%d",&item);
q[++r]=item;
}
```

```
void deletefront()
{
f=0;
if(f>r) OR if((f==0)&&(r==-1))
{
printf("Queue is empty");
return;
}
printf("Element deleted is %d", q[f++]);
}
void display()
{
int i;
if(f>r) OR if((f==0)&&(r==-1))
{
printf("Queue is empty");
return;
}
printf("Contents are");
for(i=f; i<=r; i++)
printf("%d", q[i]);
}
```

b) Write a program in C to implement a stack of integers using a singly linked list

```
#include<stdio.h>
#include<stdlib.h>
typedef struct node{
int data;
struct node *next;
} NODE;
```

```
Node *top=NULL;
void push(int value)
{
Node *newNODE=(NODE *) malloc (sizeof(Node));
if(newNode==NULL)
{
fprintf(stderr, "Memory allocation failed");
exit(EXIT_FAILURE);
}
newNode->data=value;
newNode->next=top;
top=newNode;
}
int pop() {
if(top==NULL){
fprintf(stderr, "Stack underflow");
exit(EXIT_FAILURE);
}
int value=top->data;
Node *temp=top;
top=top->next;
free(temp);
return value;
}
void display(){
if(top==NULL){
printf("Stack is empty");
return;
}
printf("Stack Contents");
Node *current=top;
while(current!=NULL){
printf("%d", current->data);
current=current->next;
}
printf("\n");
}
```

```
void main()
{
push(1);
push(2);
push(3);
display();
printf("Popped element: %d", pop());
printf("Popped element: %d", pop());
display();
}
```

Q4) a) Write a C program to implement insertion, deletion and display operations in a circular queue.

```
#include<stdio.h>
#include<stdlib.h>
#define max 10
int q[10], front=0, rear=-1;
void main()
{
int ch;
void insert();
void delete();
void display();
printf("Circular queue operations");
printf("1. Insert 2. Delete 3. Display
      4. Exit");
while(1)
{
printf("Enter your choice");
scanf("%d", &ch);
switch(ch)
{
case 1: insert();
      break;
case 2: delete();
      break;
case 3: display();
      break;
case 4: exit(1);
default: printf("Invalid option");
}}
void insert()
{
int x;
if((front==0 && rear==max-1)||(front>0 && rear==
                              front-1))
printf("Queue is overflow");
else
{
printf("Enter the element to be
      inserted");
scanf("%d",&x);
if(rear==max-1 && front>0)
{
rear=0;
q[rear]=x;
}
else
{
if((front==0 && rear==-1) ||
  (rear!=front-1))
q[++rear]=x;
}}
void delete()
{
int a;
if(front==0) && (rear==-1))
{
printf("Queue is underflow");
exit(1);
}
if(front==rear)
{
a=q[front];
rear=-1; front=0;
}
else if(front==max-1)
{
a=q[front];
front=0;
}
else
a=q[front++];
printf("Deleted item is %d",a);
```

```
void display()
{
int i,j;
if(front==0 && rear==-1)
{
printf("Queue Underflow");
exit(1);
}
if(front>rear)
{
for(i=0;i<=rear;i++)
printf("%d",q[i]);
for(j=front;j<=max-1;j++)
printf("\t %d",q[j]);
printf("Rear is at %d",q[rear]);
printf("Front is at %d",q[front]);
}
else
{
for(i=front;i<=rear;i++)
{
printf("\t %d",q[i]); }
printf("Rear is at %d",q[rear]);
printf("Front is at %d",q[front]);
}
printf("\n"); }
```

Q7. b) Write the C function to add two polynomials. Show the linked representation of the below two polynomials and their addition using a circular ~~list~~ singly linked list. P1: $5x^3+4x^2+9x+3$ P2: $6x^2+5$

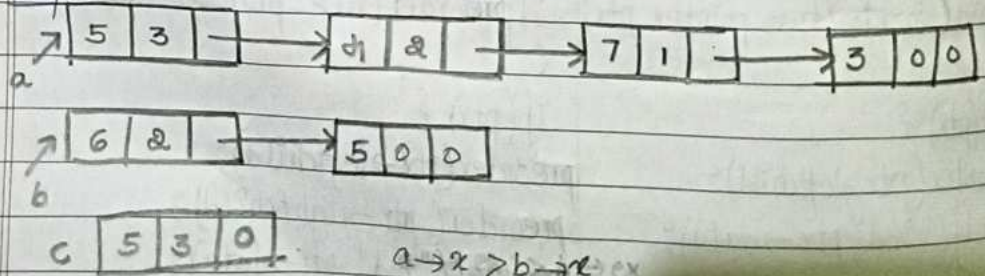
Output: Add the 2 above polynomials & represent them using the linked list.

```
⇒ int compare(NODE a, NODE b)
{
if(a->x > b->x)
return 1;
else if (a->x < b->x)
return -1;
else if (a->y > b->y)
return 1;
else if (a->y < b->y)
return -1;
else if (a->z > b->z)
return 1;
else if (a->z < b->z)
return -1;
return 0;
}
void attach(int cf, int x1, int y1, int z1,
NODE *ptr)
{
NODE temp;
temp=getnode();
temp->coef=cf;
temp->x=x1;
temp->y=y1;
temp->z=z1;
(*ptr)->link=temp;
*ptr=temp;
}
NODE addpoly(NODE a, NODE b)
{
NODE starta, c, lastc;
int sum, done=0;
starta=a;
a=a->link;
b=b->link;
c=getnode();
```

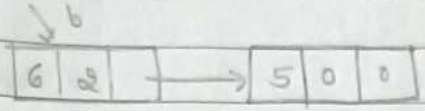
```
c->coef = -1;
c->y = -1;
c->z = -1;
lastc = c;
do {
switch (compare(a,b))
{
case -1: attach(b->coef, b->x, b->y, b->z, &lastc);
b = b->link;
break;
case 0: if(starta == a)
        done = 1;
        else {
    sum = a->coef + b->coef;
     if(sum)
     attach(sum, a->x, a->y, a->z, &lastc);
      a = a->link;
      b = b->link;
    } break;
case 1: if(starta == a)
      done = 1;
    attach(a->coef, a->x, a->y, a->z, &lastc);
    a = a->link;
    break;
    } } while(!done);
    lastc->link = c;
    return c;
    }
```

Representation and Addition of Polynomials:

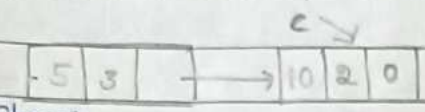
Step 1:



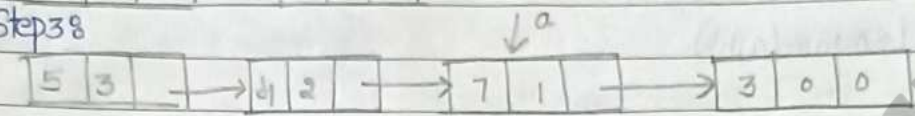
Step 2:

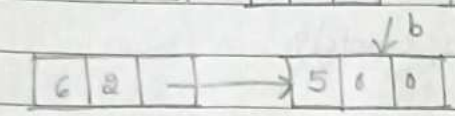


Here $a \to x = b \to x$

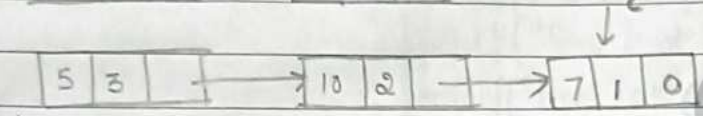


Step 3:

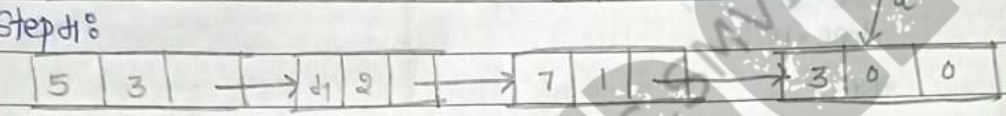


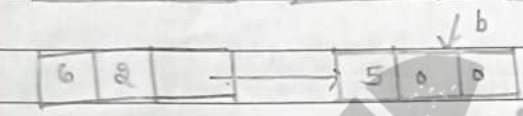


Here $a \to x > b \to x$



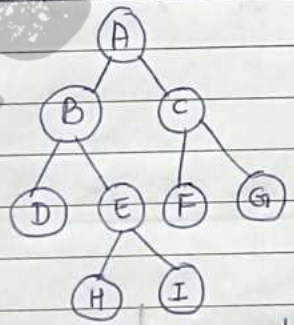
Step 4:





$\Rightarrow 5x^3 + 10x^2 + 7x + 8$ //

Q5) a) Write recursive C functions for inorder, preorder and postorder traversals of a binary tree. Also, find all the traversals for the given tree.



⇒ Inorder traversal:

```
void inorder(tree_pointer ptr)
{
if(ptr) {
inorder(ptr→leftchild);
printf("%d", ptr→data);
inorder(ptr→rightchild);
}
```

Postorder traversal:

```
void postorder(tree_pointer ptr)
{
if(ptr) {
postorder(ptr→leftchild);
postorder(ptr→rightchild);
printf("%d", ptr→data);
}
```

preorder traversal:

```
void preorder(tree_pointer ptr)
{
if(ptr){
printf("%d",ptr->data);
preorder(ptr->leftchild);
preorder(ptr->rightchild);
}}
```

Inorder Traversal: DBHEIAFCG
Preorder Traversal: ABDEHICFG
Postorder Traversal: DHIEBFGCA.

b) Write C-functions for the following:

i) Search an element in the singly linked list:

search(info, link, start, item, loc)
1. set ptr=start
2. Repeat step-3 while ptr!=NULL
3. if item=ptr->info then
   set loc:=ptr and EXIT
   else
   ptr=ptr->link
   [end of if]
   [end of step 2 loop]
4. Search is unsuccessful set loc:=NULL
5. Exit

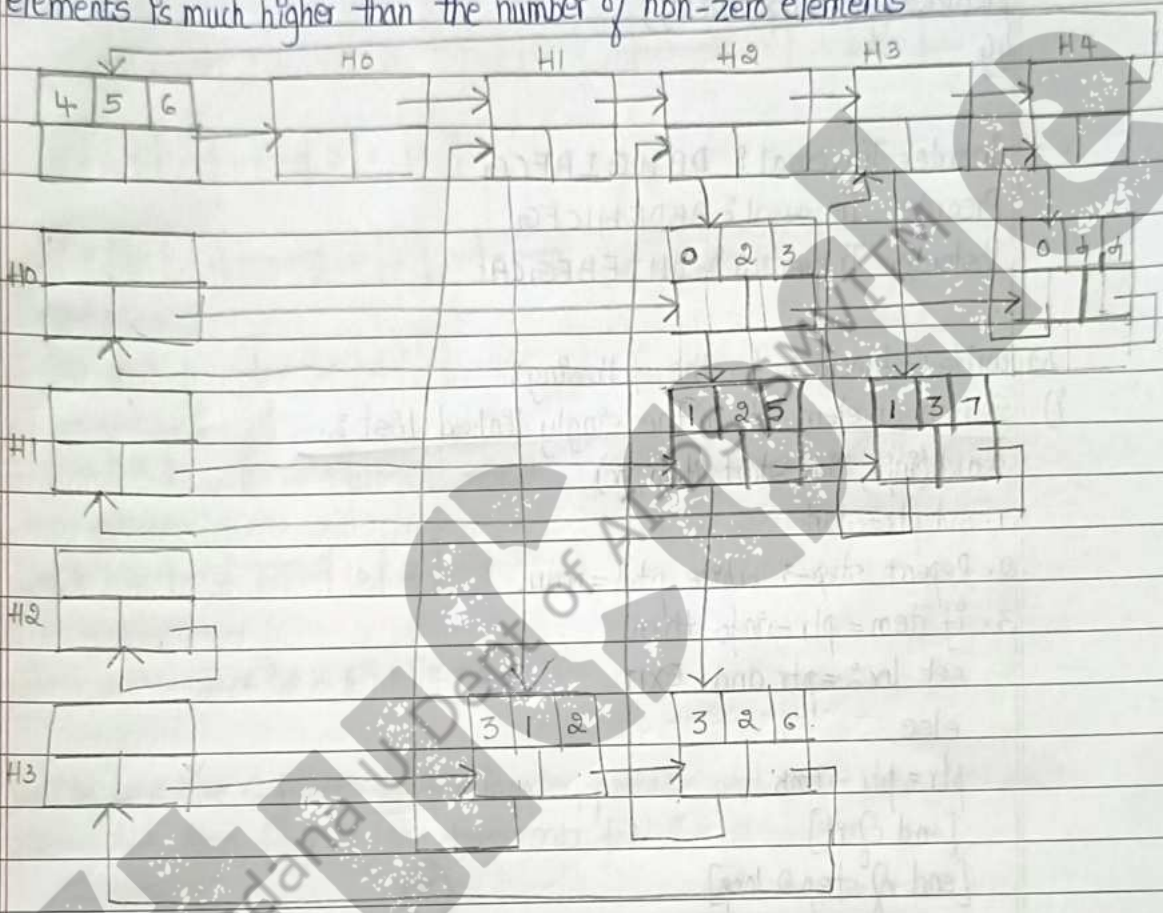
ii) Concatenation of two singly linked list:

```
listpointer concatenate(listpointer ptr1, listpointer ptr2)
{
if(!ptr1) return ptr2;
if(!ptr2) return ptr1;
for(temp=ptr1; temp->link; temp=temp->link);
temp->link=ptr2;
}
```

c. Define Sparse matrix. For the given sparse matrix, give the linked list representation.

$$A = \begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$

⇒ A sparse matrix is a special case of a matrix in which the number of zero elements is much higher than the number of non-zero elements.



Q6. a) Write C-functions for the following:

i) Inserting a node at the beginning of a Doubly Linked list.

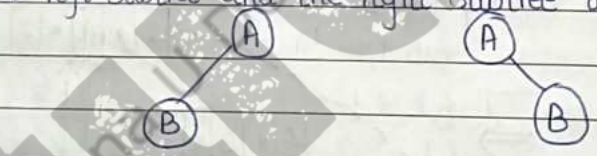
⇒
```
void insertbeg()
{
if(first == NULL)
{
create();
first = last = temp;
}
else
{
temp->next = first;
first->prev = temp;
first = temp;
}
```

ii) Deleting a node at the end of the Doubly linked list:

```
void deleteend()
{
struct node *temp, *temp2;
temp = first;
if(first == NULL)
{
printf("List is empty");
return;
}
if(temp -> next == NULL)
{
printf("The item deleted is %d", temp -> info);
free(temp);
first = last = NULL;
}
else
{
temp2 = last -> prev;
temp2 -> next = NULL;
printf("Item deleted is %d",
        last -> info);
free(last);
last = temp2;
}}
```

b) Define Binary Tree. Explain the representation of a binary tree with a suitable example.

⟹ A Binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left subtree & right subtree. Thus, the left subtree and the right subtree are different.



Binary Tree Representation:

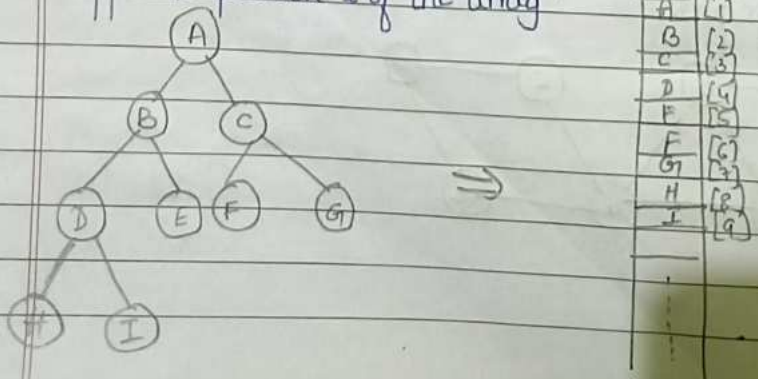
The storage representation of binary trees can be classified as:

1) Array Representation:

→ A tree can be represented using an array, which is called Sequential representation.

→ The nodes are numbered from 1 to n, and one-dimensional array can be used to store the nodes.

→ Position 0 of this array is left empty and the node numbered i is mapped to position i of the array.



2) Linked Representation:

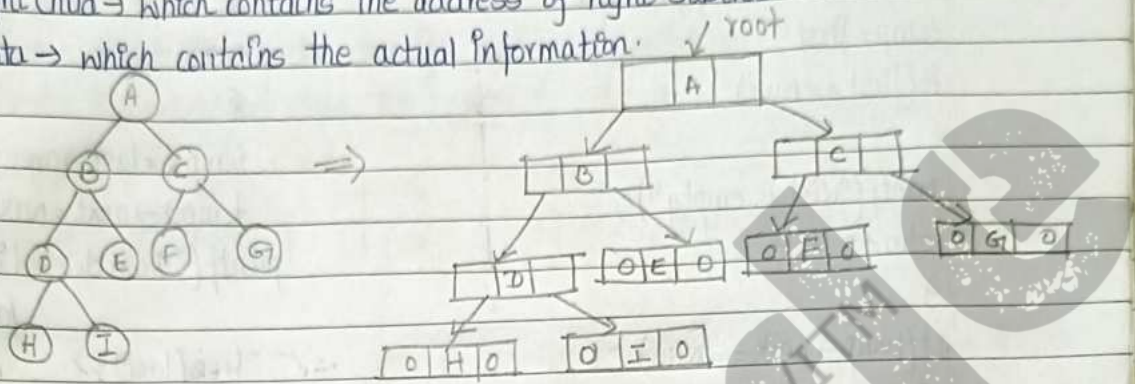
The problem with array representation is that memory is wasted.

In linked representation, each node has three fields:

Left Child → which contains the address of left subtree.

Right Child → which contains the address of right subtree.

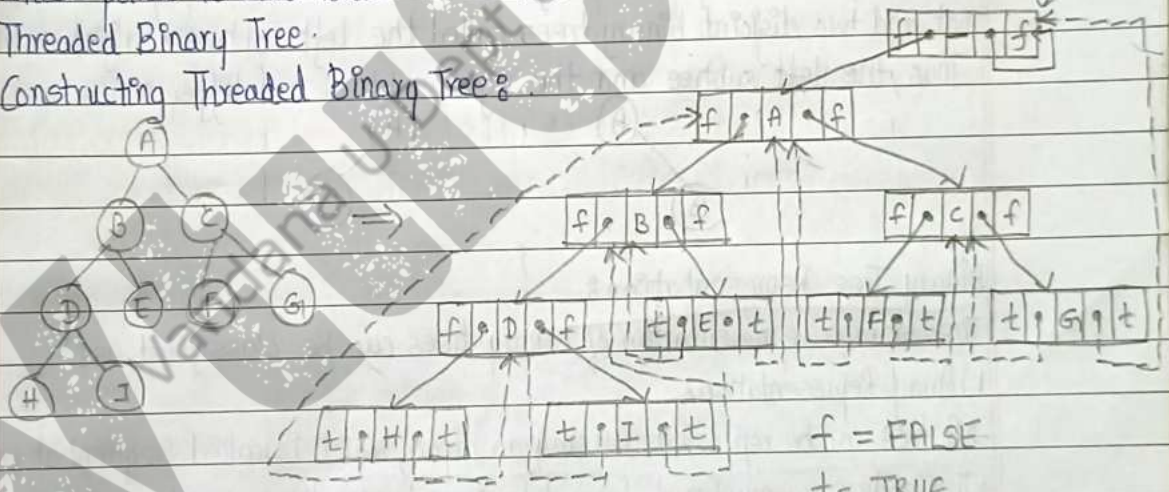
Data → which contains the actual information.



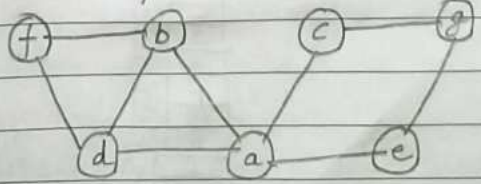
c) Define the Threaded binary tree. Construct Threaded binary for the following elements: A, B, C, D, E, F, G, H, I.

⇒ In the linked representation of any binary tree, there are more null links than actual pointers. These null links are replaced by the pointers, called Threads, which points to other nodes in the tree. Such a tree can be called as Threaded Binary Tree.

Constructing Threaded Binary Tree:



Q7. a) Design an algorithm to traverse a graph using Depth First Search [DFS]. Apply DFS for the given graph:



Function for DFS:

```
void dfs(int v)
{
int i;
visited[v]=1;
s[++top]=v;
for(i=1; i<=n; i++)
{
if(a[v][i]==1 && visited[i]==0)
{
printf("%d", i);
dfs(i);
}}}
```

DFS Algorithm:

1) Start by putting any one of the graph's vertices on top of a stack.
2) Take the top item of the stack and add it to the visited list.
3) Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4) Keep repeating steps 2 and 3 until the stack is empty.

Applying DFS for the following graph:

	Stack	v=adj(s[top])	Nodes visited s	Pop(stack)
Initial	a	–	a	–
1	a	b	a,b	
2	a,b	d	a,b,d	
3	a,b,d	f	a,b,d,f	
4	a,b,d,f	–	a,b,d,f	f
5	a,b,d	–	a,b,d,f	d
6	a,b	–	a,b,d,f	b
7	a	c	a,b,d,f,c	
8	a,c	g	a,b,d,f,c,g	
9	a,c,g	e	a,b,d,f,c,g,e	
10	a,c,g,e	–	a,b,d,f,c,g,e	e
11	a,c,g	–	a,b,d,f,c,g,e	g
12	a,c	–	a,b,d,f,c,g,e	c
13	a	–	a,b,d,f,c,g,e	a

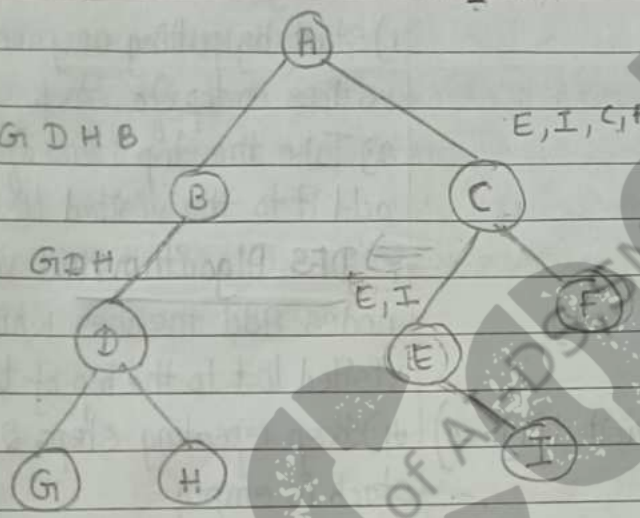
b) Construct a binary tree from the Post-order and Inorder sequence given below:

In-order: GDHBAEICF

Postorder: GHDBIEFCA

(LDR) Root (LR Data) Root

Inorder : GDHBAEICF Postorder : GHDBIEFCA

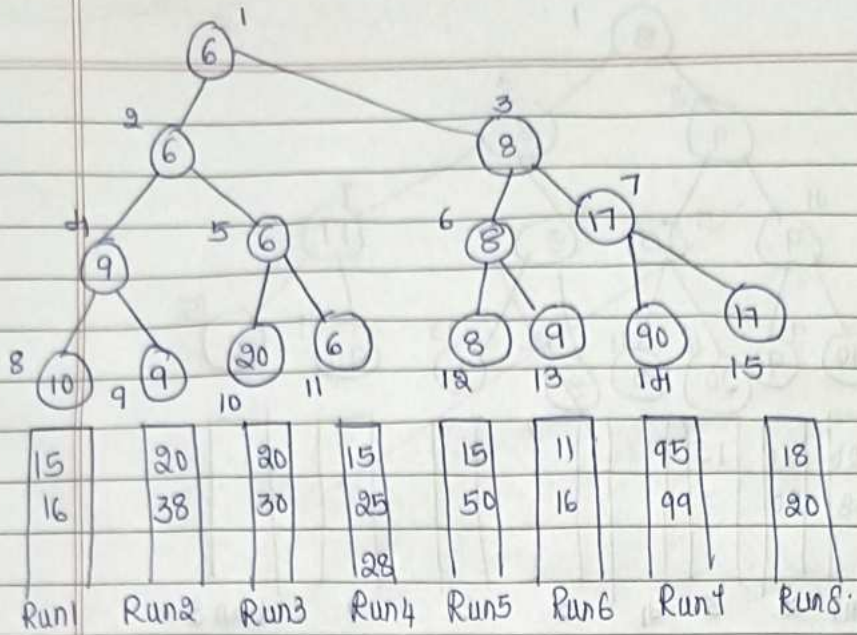


c) Define selection Tree. Construct min-winner tree for the runs of a game given below. Each run consists of values of players. Find the first 5 winners.

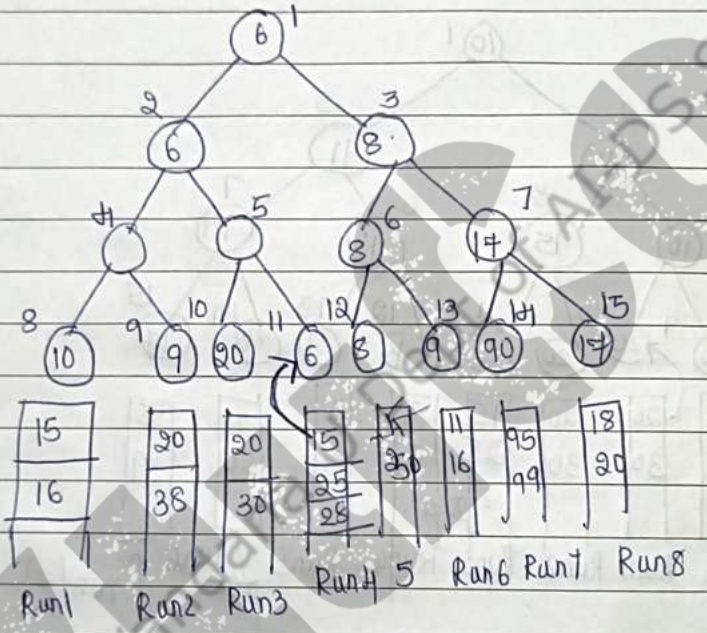
10	9	20	6	8	9	90	17
15	20	20	15	15	11	95	18
16	38	30	25	50	16	99	20
			28				

⟹ The tournament tree is a complete binary tree with 'n' external nodes and n-1 internal nodes. The external nodes represent the players, and the internal nodes are representing the winner of the match between the 2 players. This tree is also known as selection tree.

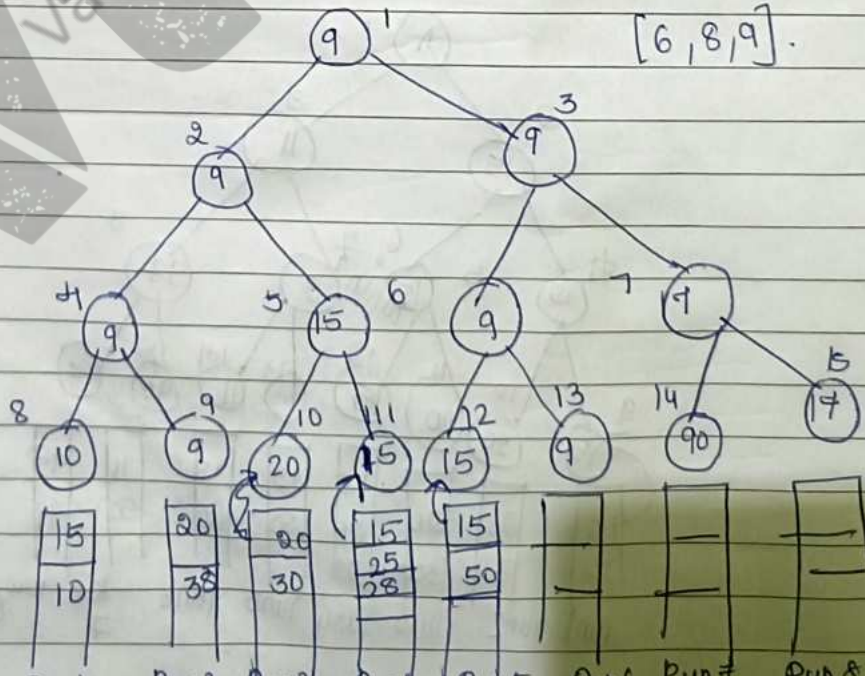
2 types : Winner Trees and Loser Trees.



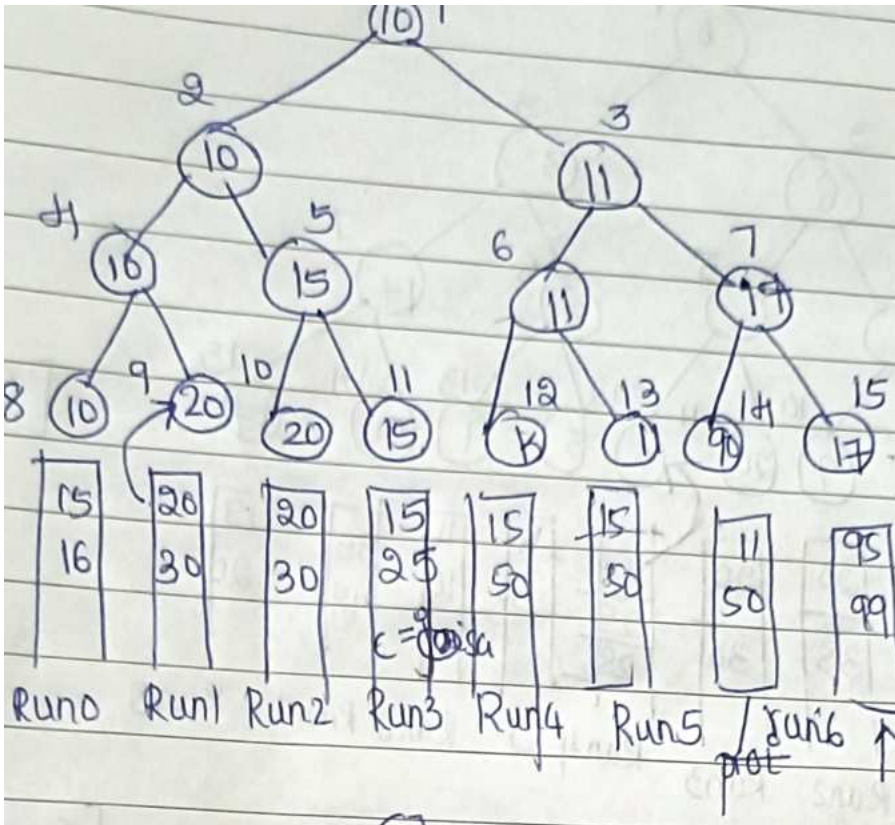
6

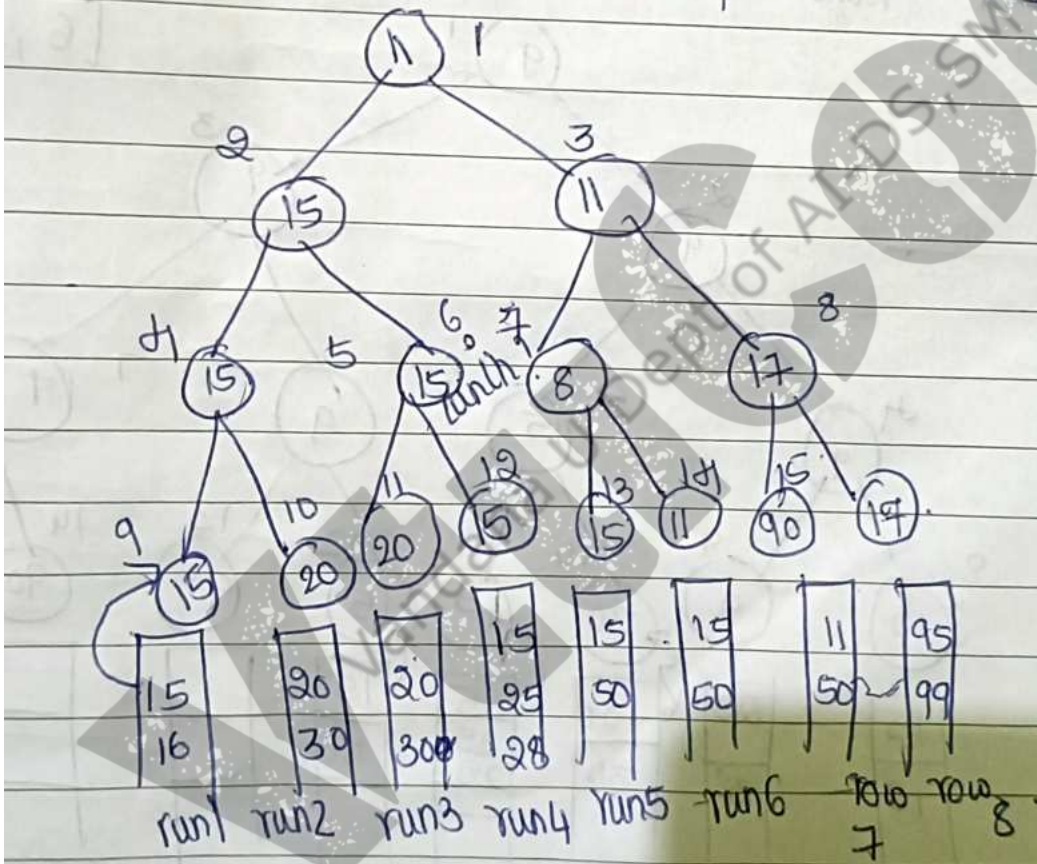


[6, 8]



[6, 8, 9].





8, 9, 10, 11 are the first 5 winners

Q8. a) Define Binary Search Tree. Construct a BST for the following elements: 100, 85, 45, 55, 120, 20, 70, 90, 115, 65, 130, 45. Traverse using inorder, preorder & postorder traversal techniques. Write recursive C-functions for the same.

⇒ A Binary search tree has:

- Every element has a unique key.
- The keys in a non-empty left subtree (right subtree) are smaller (larger) than the key in the root of subtree.
- The left and right subtrees are also Binary Search Trees.

BST Construction:

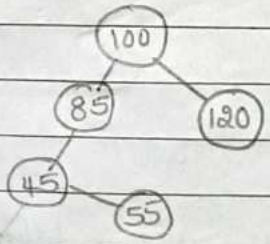
1.) Insert 100:

2) Insert 85: 85 < 100, so insert it to left of 100

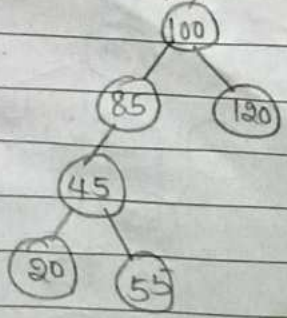
3) Insert 45: 45 < 100, to the left
45 < 85 to the left of 85

4) Insert 55: 55 < 100 → left, 55 < 85 → left, 45 > 55, So Right

5) Insert 120: 120 > 100, so Right

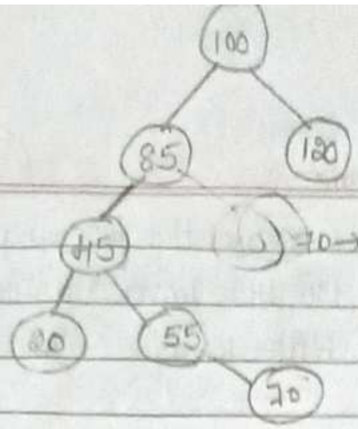


6) Insert 20: 20 < 100 → left, 20 < 85 → left, 20 < 45 → left

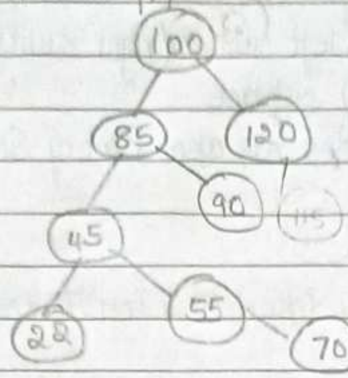


7) Insert 70 : 70<100 → left, 70<85 → left

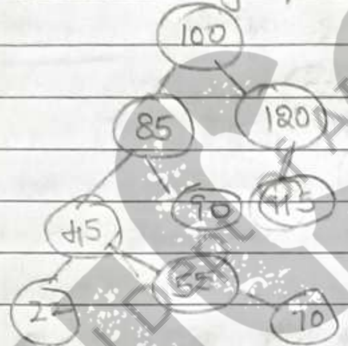
(45) 70 → Right 70>55 → Right



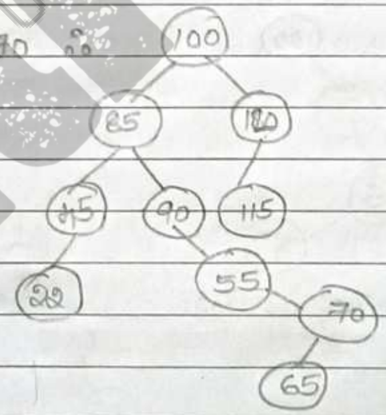
8) Insert 90 : 90<100 → left, 90>85 → Right



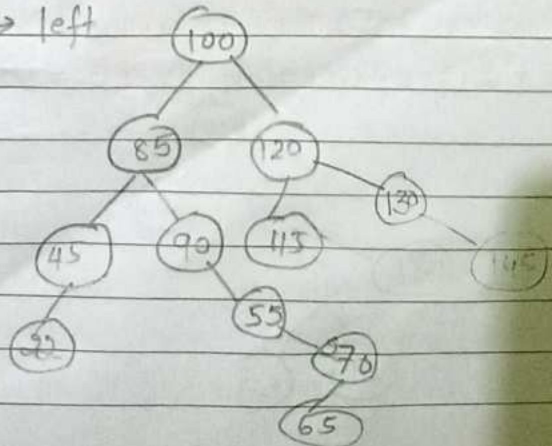
9) Insert 115 : 115>100 → Right, 115<120 → left.



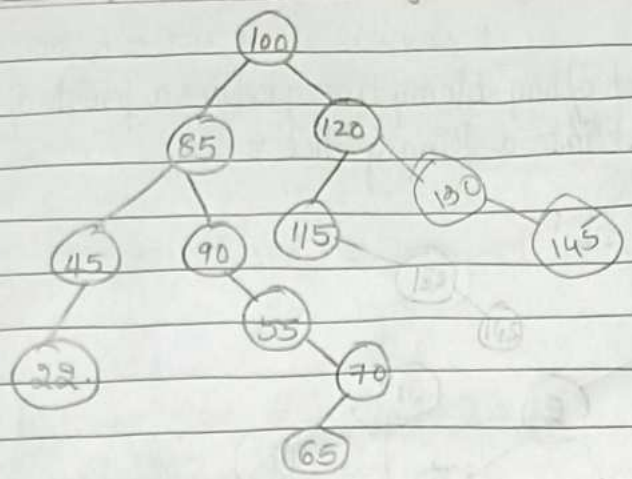
10) Insert 65 : 65<70 :



11) Insert 130 : 115>130 → left



12) Insert 145 : 145 > 130 → Right



Functions : 1) Inorder functions

```
void in(NODE IN)
{
 if (IN != NULL)
 {
  in(IN → ltree);
  printf("%d \t", IN → value);
  in(IN → rtree);
 }
}
```

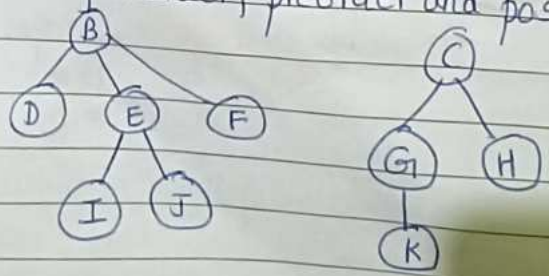
2) Preorder functions

```
void pre(NODE PRE)
{
if (PRE != NULL)
{
printf("%d", PRE → value);
pre(PRE → ltree);
pre
post(PREE → rtree;
}
```

3) Postorder functions

```
void post(NODE POST)
{
if(POST != NULL)
{
post(POST → ltree);
post(POST → rtree);
printf("%d\t", POST → value);
}}
```

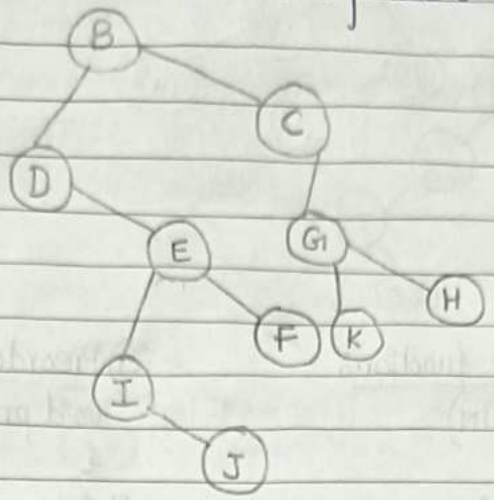
b) Define Forest. Transform the given Forest into a Binary Tree and traverse using inorder, preorder and postorder traversal.



A Forest is a set of $n>0$ disjoint trees. When we remove the root of a tree we obtain a forest.

Ex: Removing the root of any binary tree produces a forest of 2 trees.

Transforming a Forest into a Binary Tree:



Traversal:

a) Pre-order traversal: B D E I J F C G K H

b) Post-order traversal: J I F E D K H G C B

c) In-order traversal: D I J E F B K G H C

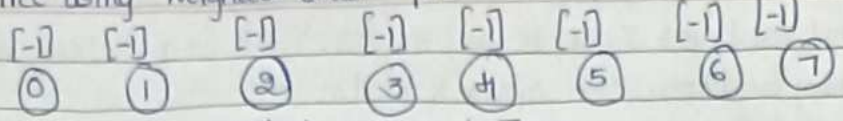
c. Define the Disjoint set. Consider the tree created by the weighted union function on the sequence of unions: union(0,1), union(2,3), union(4,5), union(6,7), union(0,2), union(4,6), and union(0,4). Process the simple find and the collapsing find on eight finds and compare which is efficient.

⟹ Definition: It is a data structure that contains a collection of disjoint or non-overlapping sets.

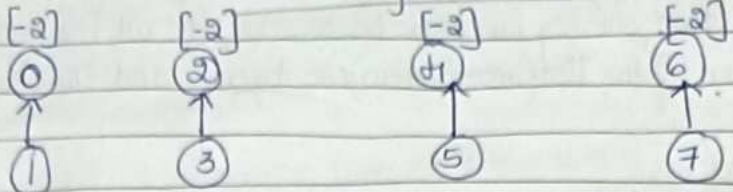
If sets S_i and S_j are two sets and $i \neq j$, then there is no element that is both in S_i and S_j.

Ex: If we have 10 elements numbered from 0 to 9, we can partition them as $S_1 = \{0,1,2\}$ $S_2 = \{3,4,5,6,7\}$ $S_3 = \{8,9\}$.

Tree using weighted union fns

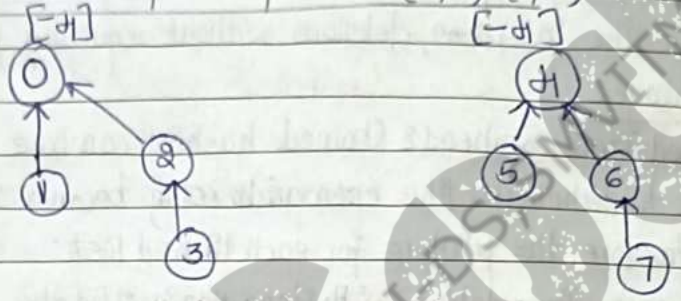


Initial height - 1 Trees

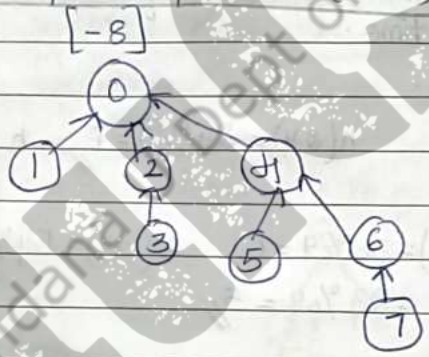


Height - 2 trees following Union (0,1) (2,3) (4,5) (6,7)

Height - 3 trees following union (0,2) (4,6)



Height - 4 Tree following union (0,4)



Alm for simple-find

```
int simple find(int i)
{ for (; parent[i]>=0; i=parent[i])
    ;
  return i;
}
void
```

The time complexity of simple find is $\sum_{i=2}^{n} i = O(n^2)$

Time complexity of collapsing find is less

Alm for collapsing find:

```
int collapsing find(int i)
{
int root, trail, lead;
for(root=i; parent[root]>=0;
 root=parent[root])
  ;
for(trial=i; trial!=root;
 trial=lead) {
 lead = parent[trial];
parent[trial]=root;
  }
return root;
}
```

Q9 a) What is chained hashing? Discuss its pros and cons. Construct the hash table to insert the keys: 7, 24, 18, 52, 36, 54, 11, 23 in a chained hash table of 9 memory locations. Use $h(k) = k \bmod m$

⇒ Definition: Chaining technique avoids collision using an array of linked lists (run time). If more than one key has same hash value, the all the keys will be inserted at the end of the list (insert rear) one by one and thus collision is avoided.

Advantage:

1.) Simple Implementation: Chained hashing is relatively simple to implement compared to other collision ~~rele~~ resolution techniques like open addressing.

2.) Support Dynamic Operations: Chained hashing can easily support dynamic operations like insertions, deletions without requiring frequent rehashing.

Disadvantage:

1.) Increased Space Overhead: Chained hashing can have higher space overhead compared to techniques like open addressing because it requires additional memory to store the pointers for each linked list.

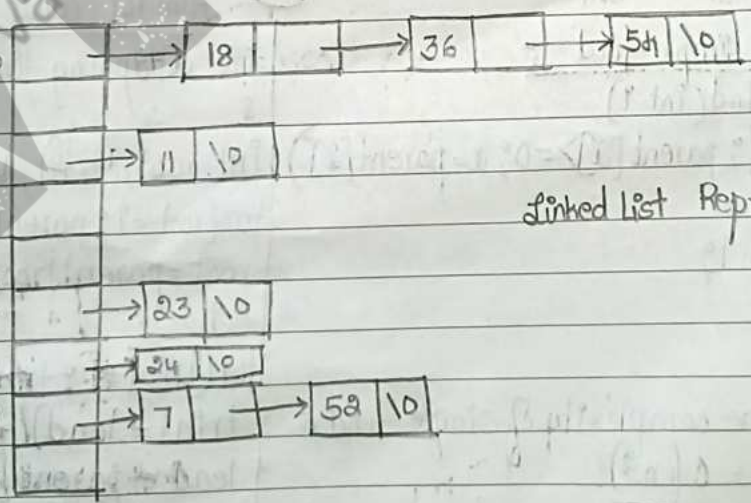
2.) Performance Degradation with long chains: As the length of the chains increases due to collisions, the performance of chained hashing can degrade, leading to longer search time.

Constructing Hash Table:

$h(7) = h(k) = k \bmod m = 7 \% 9 = 7$ // $h(24) = 24 \% 9 = 6$ // $h(18) = 18 \% 9 = 0$ //

$h(52) = 52 \% 9 = 7$ // $h(36) = 36 \% 9 = 0$ // $h(54) = 54 \% 9 = 0$ //

$h(11) = 11 \% 9 = 2$ // $h(23) = 23 \% 9 = 5$ //



Linked List Representation

b) Define the leftist tree. Give its declaration in C. Check whether the given binary tree is a leftist tree or not. Explain your answer.

⇒ Definition: Leftist trees, are defined using the concept of an extended binary tree, which is a binary tree in which all empty binary subtrees have been replaced by a square node called as External nodes.

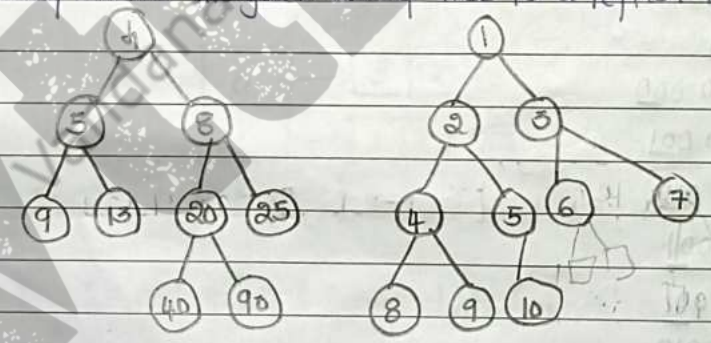
2 Types: Height Based Leftist Tree
Weight " "

A leftist tree is a binary tree such that if it is not empty, then,
shortest(left_child(x)) ≥ shortest(right_child(x)) for every internal node 'x'.

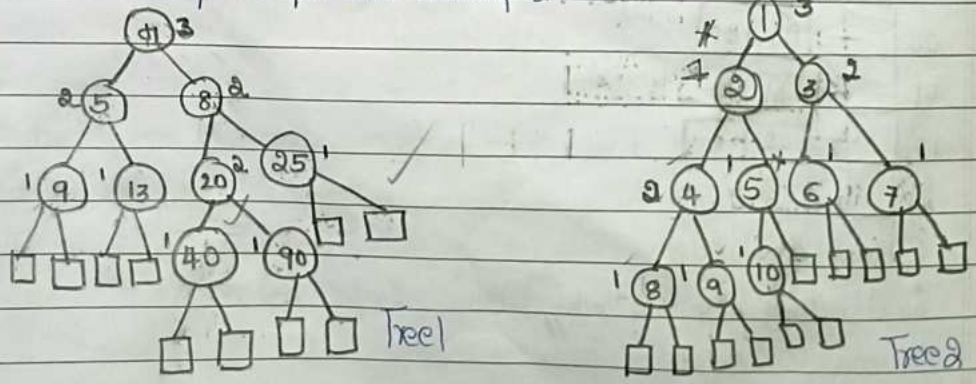
Declaration:

```
typedef struct {
        int key;
        /* Other fields */
        } element;
typedef struct leftist *leftisttree;
        struct {
            leftisttree leftchild;
            element data;
            leftisttree rightchild;
            int shortest;
            } leftist;
```

Checking whether the given binary Tree is a leftist tree or not:



⇒ Extended Binary Tree for both binary tree:



As per the definition of the leftist tree is shortest (leftchild(x)) ≥ shortest (rightchild(x)).

We see that in both the trees at every node, the above condition is satisfied.

For ex: In Tree1, shortest (leftchild(8)) = 3 and shortest(rightchild(8)) = 2 which satisfies the condition.

In tree(2), shortest(leftchild(1)) < shortest (rightchild(1))

i.e 1 < 2 which violates the condition

∴ tree(2) is not a leftist tree.

Tree(1) is a leftist tree and Tree(2) is not a leftist tree.

---

c) What is dynamic Hashing? Explain the following techniques with examples:

i) Dynamic Hashing using Directories.

ii) Directory less Dynamic Hashing.

⇒ Dynamically increases the size of the hash table as the collision occurs.

There are 2 types:.

i) Dynamic Hashing using Directories: /Extendible Hashing:

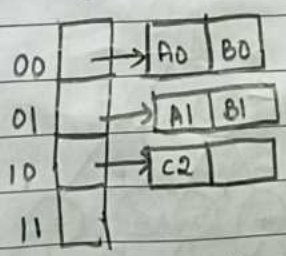
Uses a directory that grows or shrinks depending on the data distribution.

No overflow buckets.

Uses a directory of pointers to buckets/bins which are collection of records. The number of buckets are doubled by doubling the directory, and splitting just the bin that overflowed.

Ex:

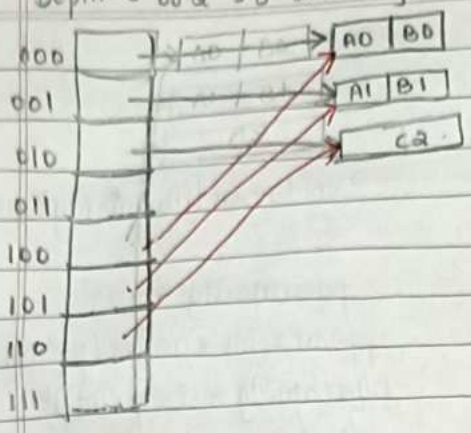
K	h(K)
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C1	110 001
C2	110010



Here depth is 2

∴ directory size is $2^2 = 4$ //

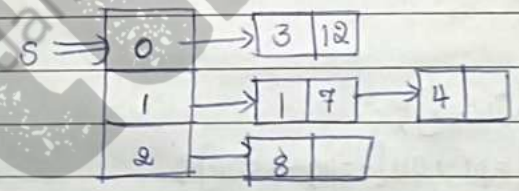
Depth = 3 ∴ $2^3 = 8$ directory size.



ii) Directory less Dynamic Hashing: No directory, Splits the buckets in linear order, uses overflow buckets.
Here pages are split when overflow occurs. Splitting occurs in turns, in a round robin fashion, one by one from the ~~last to~~ first to the last bucket.
Ex: Insert inorder using Linear Hashing: 1, 7, 3, 8, 12, 4

After Insertion till 12:

When 4 ~~is~~ is being inserted, overflow occured, so we split the bucket (no matter it is full or partially empty). And increment pointer



10 a) What is a Priority Queue? Demonstrate functions in C to implement the max priority queue with an example:
i) Insert into the Max priority queue
ii) Delete "
iii) Display "

⇒ A Priority Queue is a collection of elements such that each element has an associated priority.
2 Types: Single ended priority queue
Double ended priority queue

Insert: void insert(MaxPriorityQueue *pq, int val)

```
{
if(pq->size==pq->capacity)
  {
    resize(pq);
  }
pq->arr[pq->size++]=val;
}
```

→ void resize(MaxpriorityQueue *pq)

```
{
pq->capacity *=2;
pq->arr=(int *)realloc(pq->arr,
pq->capacity * sizeof(int));
}
```

Delete: int delMax(MaxPriorityQueue *pq)

```
{
if(pq->size==0)
{
printf("Priority queue underflow");
return -1;
}
int maxIndex=0;
for(int i=1; i<pq->size; i++)
{
if(pq->arr[i]>pq->arr[maxIndex])
  {
    maxIndex=i;
  }}
int max=pq->arr[maxIndex];
pq->arr[maxIndex]=pr->arr[--pq->size];
return max;
}
```

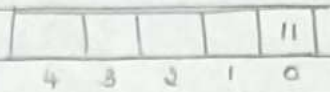
Display: void display(MaxPriorityQueue *pq)

```
{
if(pq->size==0)
{
printf("Priority Queue is empty");
return;
}
printf("Contents of max priority queue");
```

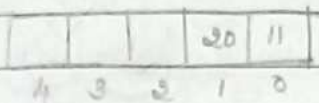
```
for(int i=0; i<pqsize; i++)
{
printf("%d", pq->arr[i]);
}
printf("\n");
```

Illustrating with an example: 11, 20, 30, 40, 5.

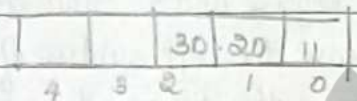
Step 1: Insert 11



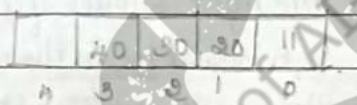
Step 2: Insert 20



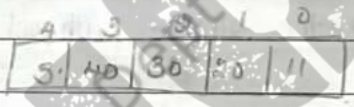
Step 3: Insert 30



Step 4: Insert 40



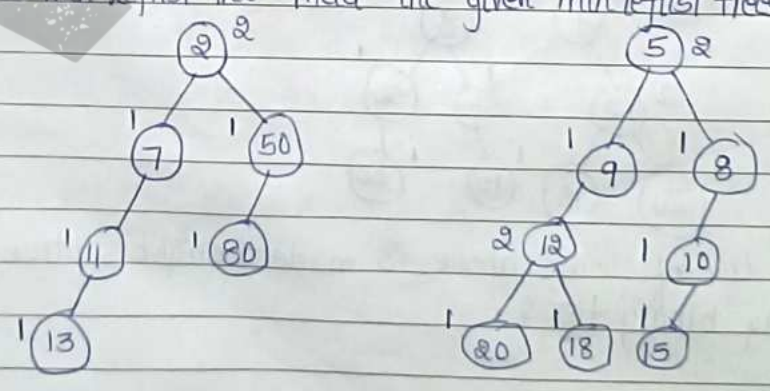
Step 5: Insert 5



Step 6: Delete an element: Since priority is taken into consideration elements are deleted first followed by 30, 20, 11 and 5

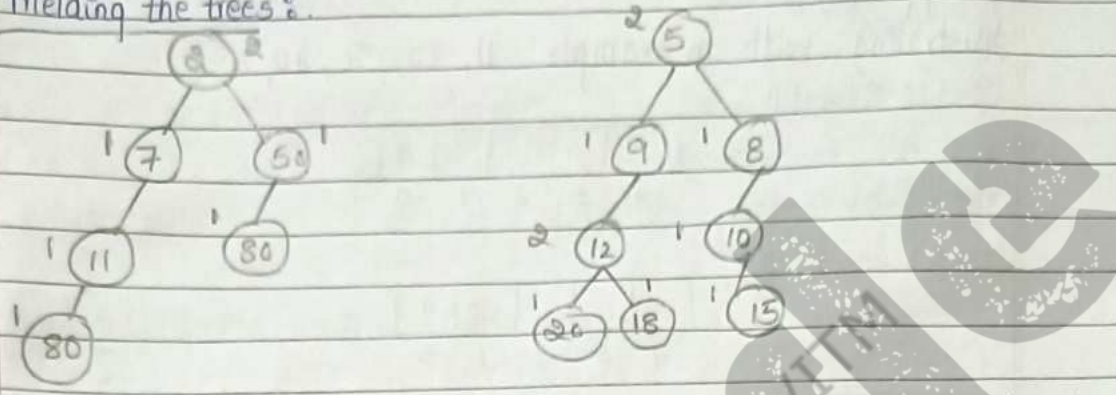
Step 7: We can also display the elements in a similar way.

b) Define min leftist Tree. Meld the given min leftist trees.



⇒ Definition: A min leftist tree (max leftist tree) is a leftist tree in which the keyvalue in each node is no larger (smaller) than the keyvalues in its children (if any).

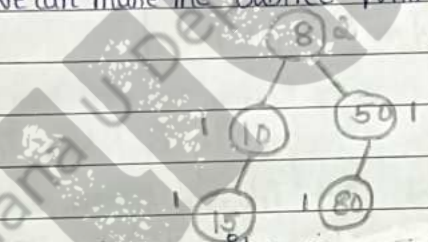
Melding the trees:



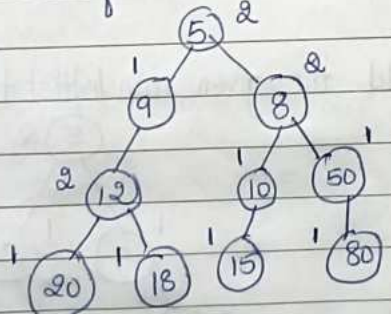
⇒ Step 1: Compare the root keys 2 and 5. Since 2<5, the binary tree should have 2 in its root. We shall leave the left subtree of 2 unchanged and meld 2's right subtree with the entire binary tree rooted at 5.

Step 2: When melding the right subtree of 2 and the binary tree rooted at 5, we notice that 5<50. So, 5 should be in the root of the melded tree.

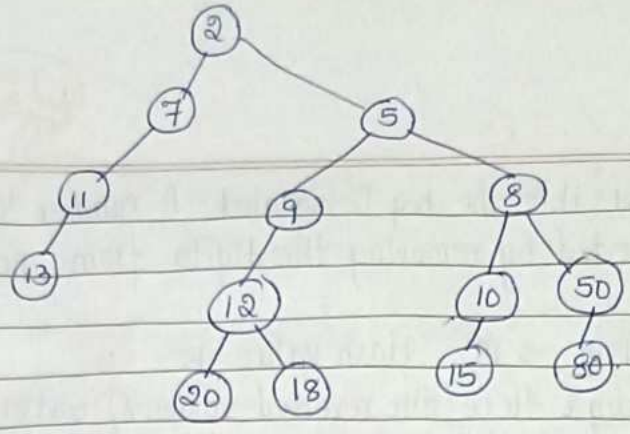
Step 3: Now meld the subtrees with root 8 and 50. Since 8<50 and 8 has no right subtree, we can make the subtree with root 50 the right subtree of 8.



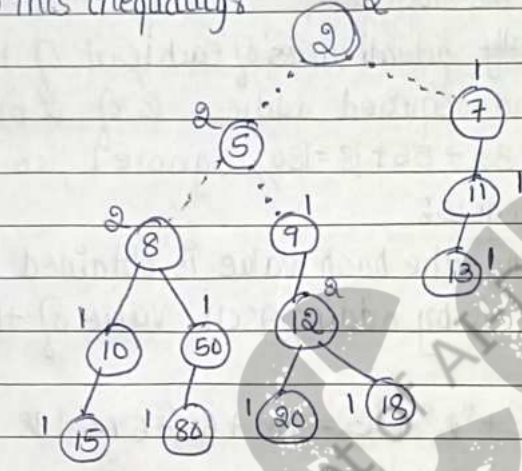
Step 4: Hence the result of melding the right subtree of 2 and the tree rooted at 5 is the tree as shown below:



Step 5:
When the tree as shown above is made the right subtree of 2, we get the following binary tree:



Step 6: To convert the above tree into a leftist tree, we begin at the last modified root ensuring shortest(leftchild()) >= shortest(rightchild()). This inequality holds at 8. Simply interchange the left and right subtree to stop this inequality:



c) Define Hashing. Explain different hashing functions with examples. Discuss the properties of a good hashing function.

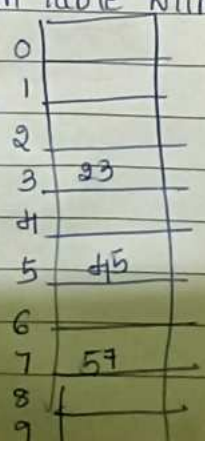
=> Hashing is an effective way to store and retrieve data in some data structure. Hashing technique is designed to use a special function called the hash function which is used to map a given value with a particular key for faster access of elements.

Different Hashing Functions:

1) Division method: The hash function depends on the reminder of the division. Typically, the divisor is the table length.

Ex: To store 57, 23 and 45 in a hash table with memory size 10.

57 % 10 = 7 // 23 % 10 = 3 // 45 % 10 = 5 //



2:) Midsquare method: Here the key is squared. A number 'l' in the middle of k^2 is selected by removing the digits from both ends.

$h(k) = l$

Ex: $k = 72$, $k^2 = 5184$ ⇒ 18// Hash value = 18

3:) Fold Boundary: Here the reversed values of outer parts of keys are added.

Ex: key: 12345678 and required address is of two digits.

so 12, 34, 56, 78. Reverse and add ⇒ 21 + 34 + 56 + 87 = 198.

Ignore 1, we get 98 as the location

Fold shifting: Here ~~actually~~ actual values of each part of key is added

Ex: key: 12345678 and required address is of 2 bytes.

12, 34, 56, 78 ⇒ 12 + 34 + 56 + 78 = 180// Ignore 1, so value = ~~78~~ 80.

4:) Converting keys into integers:

Here if key 'k' is a string, the hash value is obtained by converting the string into integers. i.e by adding ASCII value of the characters

Ex: Key = ABC then

$h("ABC") = 'A' + 'B' + 'C' = 65 + 66 + 67 = 198$

Properties of a Good Hash function:

a:) Must be simple to compute.

b:) Number of collisions should be less while placing the record in the hash table.

c:) Hash function should produce such keys which will be distributed uniformly over an array.

d:) The hash function should depend on every bit of the key. Thus the hash function that simply extracts a portion of the key is not suitable