ASSIGNMENT - 2

Page No.:
Date: / /

(11) explain following :
① DQUEUE ② priority queue ③ circular queue.

① DQUEUE :

Dqueue stands for double ended queue. In a linear queue, the usual practice is for insertion of elements. we use one end called rear for insertion of elements where front pointer is used for deletion of elements

Defination : double ended queue is a type of queue for which we can insert the elements from both front as well as rear similarly we can delete the elements from both front and rear.

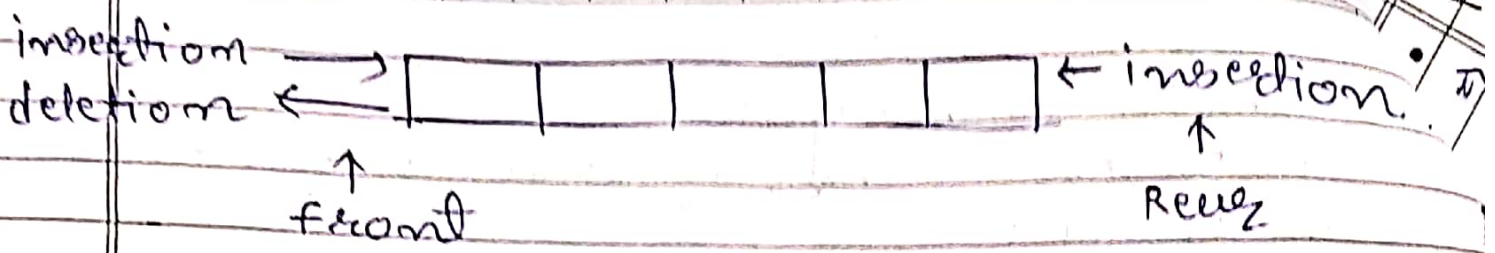
Double ended queue are basically of two types.
1) Insertion restricted queue (input Restricted)
2) Deletion restricted queue (output Restricted)

1) Input Restricted queue :

In input Restricted queue rear pointer performs both insertion and deletion operation whereas the front pointer performs only deletion operation.

(2) output Restricted queue :

In output restricted queue, front pointer performs both insertion and deletion where rear pointer performs only insertion.



The most common way of representation of dequeue are:

1) Using a doubly linked list
2) Using a circular array

The operations that can be performed on dequeue are:

1) Add an element at rear end
2) Add an element at front end
3) delete the element from rear end
4) delete the element from front end.

- procedure DQINSERT_FRONT (Q, F, R, N, Y)

⇒ Given F and R pointers to the front and rear elements of a queue, a queue consisting of N element and element Y, this procedure inserts Y at the front of the queue

1) [overflow]
   If f = 0
   then write ("empty")
   return
2) [Decrement front pointer]
   F ← F - 1
3) [Insert element]
   Q[F] ← F
   Return.

• procedure DQDELETE - REAR (Q, F, R)

=> Given F & R pointers to the front & rear elements of a queue, & a queue Q to which they correspond, this fn deletes and returns the last element from the front end of a queue And y is temporary variable.

1) [Underflow]
   If R = 0
   then write ('Underflow')
   Return (0)
2) [Delete element]
   y ← Q[R]
3) [queue empty ?]
   If R = F
   Then R ← F ← 0
   else R ← R - 1
4) [Return element]
   Return (y)

• procedure DQUEUE - DISPLAY (F, R, Q)

Given F & R are pointers to the front and rear elements of a queue, a queue consist of N elements. this procedure display queue contents

1) [check for empty]
   If F >= R
   then Write ('queue is empty')
   Return
2) [Display content]
   for (I = front ; I <= Rear ; I++)
   write (Q[I])
3) [Return statement]
   Return

(2) priority queue :-

A queue in which we are able to insert remove items from any position based on some property (such as priority of the tasks to be processed) is often refered as priority queue

⇒ Below figure represent a priority queue of jobs of real time, waiting to use a computer
⇒ priorites of 1, 2, 3 have been attached with jobs of real time, online and batch respectively. therefore if a job is initiated with priority i, it is inserted immediately at the end of list of other jobs with priorities i, Here, jobs are always removed from the front of queue.

Task identification

R_1	R_2	----	R_{i-1}	O_1	O_2	---	O_{j-1}	B_1	B_2	--	B_{k-1}	---
1	1	----	1	2	2	---	2	3	3	--	3	---

priority ↑ R_i ↑ O_j ↑ B_K

fig(a) priority queue viewed as a single queue with insertion allowed at any position

priority 1

R_1	R_2	----	R_{i-1}	---	← R_i

priority 2

O_1	O_2	-----	O_{j-1}	---	← O_j

priority 3

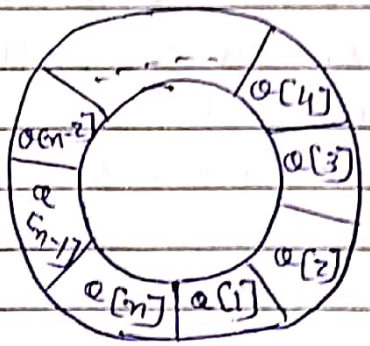
B_1	B_2	----	B_{k-1}	----	← B_K

fig(b) : priority queue viewed as a viewed set of queue.

(3) CIRCULAR QUEUE :

consider an example where the size of the queue is four elements initially the queue is empty. It is required to insert symbols 'A' 'B' and 'C', delete 'A' and 'B' and insert 'D' & 'E'. A trace of the contents of the queue is given. Note that an overflow occurs on trying to insert symbol 'E' even though the first two locations are not being used.

A more suitable method of representing a queue, which prevents an excessive use of memory, is to arrange the elements q[1], q[2], ..., q[n] in a circular fashion. with q[1] following q[n] the insertion and deletion algorithems are as under:
pictorically circular queue can be shown as



fig(a): A vector representation of a circular

procedure: (CQINSERT (F, R, q, N, Y) Given pointers to the front and rear of a circular queue, F & R, a vector q consisting of N elements and an element Y, this procedure inserts Y at the rear of the queue initially, f and R are set to zero

1) [Reset rear pointer]
If R = N
then R ← 1
else R ← R+1
2) [overflow ?]
If F = R
then write ('overflow')
Return
3) [Insert element]
Q[R] ← y
4) [Is front pointer properly set ?]
If F = 0
then F ← 1
Return

procedure : CQ DELETE (F, R, Q, N)
Given F & R pointers to the front and rear of a circular queue respectively and vector Q consisting of N element, this fn deletes and returns the last element of the queue. Y is a temporary variable.

1) [Underflow]
If F = 0
then write ('underflow')
Return (0)
2) [Delete element]
Y ← Q[F]
3) [queue empty ?]
If F = R
then F ← R ← 0
Return (Y)

4) [Increment front pointer]
If F = N
then F ← 1
else F ← F + 1
Return (Y)

- procedure CQDISPLAY (F, R, Q)
Given F & R are pointers to the front and rear elements of a queue, a queue consist of N elements. this procedure display queue contents.

1) [check for empty]
If F > = R
then write ('queue is empty')
Return.
2) [Display content]
for (I = front ; I < = REAR ; I++)
write (Q[I])
3) [Return statement]
Return.

12) what is recursion; write a C program for GCD using recursion / write a C program for factorial number using Recursion.

⇒ Recursion is a process of executing certain set of instructions repeatedly by calling the self function repeatedly.

Instead of making use of for, while do while the repetition in code exution is obtained by calling the same function again and again over same condition.

The recursive methods are complex implement and are less efficient memory utilization is more in recursive fuctions recursive methods being compactness in the program.

fuctorial number using recursion:

```
# include <stdio.h>
# include <conio.h>
int fuct (int n)
void main
{
   int a;
   clrscr();
   printf ("enter the number whose fuctorial
            you want to find");
   scanf ("%d", &a);
   printf ("%d! = %d", a, fuct(a));
   getch();
}
int fuct (int n)
{
   if (n == 1)
   {
      return 1;
   }
   else
   {
      return n * fuct (n-1);
   }
}
```

Page No.:
Date: / /

13) compare linked list and array.

⇒ Both array and linked list are non-primitive linear data structures.

Array	linked list
⇒ An array is the data structure that contains a collection of similar type data of elements	⇒ linked list is considered as non-primitive data structure of unordered linked elements known as node
⇒ In the array the elements belong to indexes ie if you want to access a particular element you have to write variable name with its index or location.	⇒ In linked list though you have to start from the head and work your way through until you get the required elements.
⇒ Accessing an element in array is faster.	⇒ It takes linear time so it is quite a bit slower.
⇒ operations like insertion and deletion in an array consume a lots of time.	⇒ performence of insertion and deletion operations in linked list is faster.
⇒ Arrays are of fixed size.	⇒ linked list are dynamic & flexible & can expand & construct in size.
⇒ In array, memory is assigned during complition time.	⇒ In linked list it is allocated during execution or run time.
⇒ elements are stored consecutively in array.	⇒ elements are stored randomly in linked list.
⇒ Memory utilization is inefficient in array.	⇒ Memory utilization is efficient in linked list.

14) write 'c' fuctions / c program 1) insert a node at end 2) delete node from beginning 3) insert node at beginning 4) delete a node at end of single linked list / Doubly & circular linked list.

* for single linked list.

1) insert a node at the end.

```
void insert_end()
{
  newn = (struct node*) malloc (sizeof (struct node));
  newn -> next = NULL;
  printf ("enter the value : ");
  scanf ("%d", &newn -> data);
  if (start == NULL)
  {
    start = newn;
    cur = newn;
  }
  else
  {
    current -> next = newn;
    current = newn;
  }
}
```

2) delete a node from the beginning:

```
void del_beg()
{
  struct node **ptr;
  if (start == NULL)
  {
```

```
        printf (" list is empty ");
    }
    else
    {
        ptr = start;
        start = ptr -> next;
        free (ptr);
        printf ("\n node deleted from the beginning");
    }
}
```

3) Insert a node at the beginning

```
void insert_beg()
{
  newn = (struct node *) malloc (sizeof (struct node));
  newn -> next = NULL;
  printf (" enter data for node : \n ");
  scanf (" %d ", & newn -> data );
  if ( start == NULL)
  {
     start = newn;
     curr = newn;
  }
  else
  {
     newn -> next = start;
     start = newn;
  }
}
```

4) Delete a node at the end

```
void del_end()
{
  struct node *temp = start;
  struct node *hold = start;
  if (temp != NULL)
  {
    temp = temp -> next;
    if (temp == NULL)
    {
      start = NULL;
      free (hold);
      return;
    }
    while (temp != NULL)
    {
      if (temp -> next != NULL)
      {
        temp = temp -> next;
        hold = hold -> next;
      }
      else
      {
        break;
      }
    }
    hold -> next = NULL;
    free (temp);
  }
  else
  {
    printf ("\n linked list is empty");
  }
}
```

* for doubly linked list in
1) Insert a node at the end.

```
void insert_end dll (int item)
{
  struct node *ptr = (struct node *) malloc (sizeof (
  struct node *temp;                          struct node));
  if (ptr == NULL)
  {
    printf ("\n overflow");
  }
  else
  {
    ptr -> data = item;
    if (head == NULL)
    {
      ptr -> next = NULL;
      ptr -> prev = NULL;
      head = ptr;
    }
    else
    {
      temp = head;
      while (temp -> next != NULL)
      {
        temp = temp -> next;
      }
      temp -> next = ptr;
      ptr -> prev = temp;
      ptr -> next = NULL;
    }
    printf ("\n node inserted \n");
  }
}
```

2) Delete node from the beginning

```
void del_beg_dll()
{
  struct node *ptr;
  if (head == NULL)
  {
    printf("\n UNDERFLOW \n");
  }
  else if (head -> next == NULL)
  {
    head == NULL;
    free(head);
    printf("\n node Deleted ");
  }
  else
  {
    ptr = head;
    head = head -> next;
    head -> PREV = NULL;
    free(ptr);
    printf("\n node deleted \n");
  }
}
```

3) Insert a node at the begining

```
void insert_beg_dll()
{
 struct node *ptr = (struct node *) malloc(sizeof(struct node));
 if (ptr == NULL)
 {
   printf("\n Underflow ");
 }
 else
```

```
{
  if (start == NULL)
  {
    ptr -> next = NULL;
    ptr -> prev = NULL;
    ptr -> data = item;
    head = ptr;
  }
  else
  {
    ptr -> data = item;
    ptr -> prev = NULL;
    ptr -> next = head;
    head -> prev = ptr;
    head = ptr;
  }
 }
}
```

4) delete a node at the end

```
void del_end_dll()
{
  struct node *ptr;
  if (head == NULL)
  {
    printf("\n Underflow \n");
  }
  else if (head -> next == NULL)
  {
    head = NULL;
    free(head);
    printf("\n node deleted \n");
  }
  else
```

```
ptr = head;
if (ptr -> next != NULL)
{
    ptr = ptr -> next;
}
    ptr -> prev -> next = NULL;
    free (ptr);
    printf ("\n node deleted \n");
  }
}
```

* for circular linked list :-

1) Insert at end call :-

```
struct node {
    int data;
    struct node * next;
};
struct node * start = NULL;
struct node * curr = NULL;
void add At end (int data) {
struct node * newn = (struct node*) malloc (sizeof
                                    (struct node));
newn -> data = data;
if (start == NULL) {
    start = newn;
    curr = newn;
    newn -> next = start;
} else {
        curr -> next = newn;
        curr = newn;
        curr -> next = start;
    }
}
```

2) Delete a node from begining

```
struct node {
    int num;
    struct node *next;
} start;
struct node *curr, *p, *, *store;
void dlist delete first Node()
{
    p = start;
    while (p -> next != start)
    {
        p = p -> next;
    }
    store = start;
    start = start -> next;
    printf ("\n the deleted node is -> %d",
                                store -> num);
    p -> next = start;
    free (store);
}
```

3) Insert a node at the begining:

```
struct node {
    int data;
    struct node *next;
};
struct node *start = NULL;
struct node *curr = NULL;
* void add At start (int data) {
    struct node *newn = (struct node *) malloc
                        (sizeof (struct node));
    newn -> data = data;
    if (start == NULL) {
```

```
        strct = newn;
        curr = newn;
        newn -> next = strct;
    }
    else {
        struct node *temp = strct;
        newn -> next = temp;
        strct = newn;
        curr -> next = strct;
    }
}
```

Note :
strct → head
curr → tail

4) Delete a node at the end

```
void deleteEND() {
    if (strct == NULL) {
    return;
    }
    else {
    it (strct != curr) {
        struct node *temp = strct;
        while (temp -> next != curr) {
        temp = temp -> next;
    }
        curr = temp;
        curr -> next = strct;
    }
    else
    {
        strct = curr = NULL;
    }
}
}
```

15) write an advantage of link list, doubly link list and circular linklist 2) explain why doubly linked list are more efficient with respect to deletions than singly linked list?

⇒ Advantage of linked list

1) linked list is a Dynamic data structure
2) linked list can grow and shrink during runtime
3) insertion & deletion operation are easier.
4) efficient memory utilization. is no need to pre allocate memory.
5) faste Access time, can be expanded in constant time without memory overhead.
6) linear data structures such as stack, queue can be easily implement using linked list.

Advantage of doubly linked list.

1) A doubly linked list can be traversed in both forward and backward direction.
2) The delete operation in doubly linked list is more efficient if pointer to node to be deleted is given
3) we can quickly insert a new node before a given node.

* Advantages of circular linked list.

1) Any node can be starting point we can traverse the whole list by starting from any point. & we just need to stop when the first visited node is visited again.

2) Useful for implementation of queue. we don't need two pointers for accessing. i.e front & rear. we can maintain a pointer to the last inserted node & front can always be obtained as next of last.

3) circular list are useful in applications to repeatedly go around the list. when multiple applications are running on a PC. it is convinent for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

4) circular doubly linked list are used for implementation of advanced data structures like fibonacci heap.

2) Ans :-
Doubly linked lists are more efficient with respe to deletions than a singly linked list because In singly linked list, to delete a node, pointer to the pointer previous node, sometimes the list is traversed. In doubly linked list we can get the previous node using previous pointer.

16) write a program to perform all (create, inser delete, display) the operations in singly linke list / doubly & circular linked list.

2) for singly linked list
1) create.

```
void create ()
{
  char ch;
```

```
struct node {
  int data;
  struct node *next
}; struct = NULL;
```

```
do
{ struct node *newn , *curr;
  newn = (struct node *) malloc (sizeof (struct node));
  printf ("\n enter the data ");
  scanf ("%d", &newn -> data);
  newn -> next = NULL;
  if (start == NULL)
  {
    start = newn;
    curr = newn;
  }
  else
  {
    curr -> next = newn;
    curr -> newn; }
  printf ("Do you want to create another");
  ch = getch()
 }
  while (ch != 'n');
 }
```

2) Display :

```
void display ()
{
 struct node *newn;
 printf ("the linked list \n");
 newn = start;
 while (newn != NULL)
{
 printf ("%d ", newn -> data);
 newn = newn -> next;
 }
  printf ("NULL");
 }
```

* for doubly linked list

1) create:

```
struct node
{
    struct node *prev;
    int num;
    struct node *next;
};
void create (struct node *list)
{
    char conf = 'y';
    int i;
    printf ("\n enter a number");
    scanf ("%d", &list -> numm);
    list -> next -> prev = list;
    conf = getchel);
    printf ("\n");
    if (conf == 'n' || conf == 'N')
    {
        list -> next == NULL;
    }
    else
    {
        list -> next = (struct node *) malloc (
                        sizeof (struct node));
        create (list -> next);
    }
}
```

2) Display:

```
void display (struct node *disp)
{
    Printf (" THE ADDRESS - PREVIOUS ADDRESS - VALUE - NE
            ADDRESS of THE LINK LISTARE: \n");
```

```
while (disp != NULL)
{
printf ("%u - %u - %u - %u \n", disp, disp->
        prev, disp -> num, disp -> next);
disp = disp -> next;
}
}
```

3) for circular linked list.

```
struct node {
int data;
struct node *next;
}; *start;
void create list (int n)
{
  int i, data;
  struct node *prev, *newn;
  if (n >= 1)
  {
    start = (struct node *) malloc (sizeof (struct
                                    node));
    printf (" Data of node 1 : ");
    scanf ("%d", &data);
    start -> data = data;
    start -> next = NULL;
    prev = start;
    for (i=2; i<=n; i++)
  {
    newn = (struct node *) malloc (sizeof (struct node));
    printf (" data of node %d ", i);
    scanf ("%d", &data);
    newn -> data = data;
    newn -> next = NULL;
```

```
        prev -> next = newn;
        prev = newn;
    }
    else
    {
        prev -> next = start;
    }
}
```

ii) display :

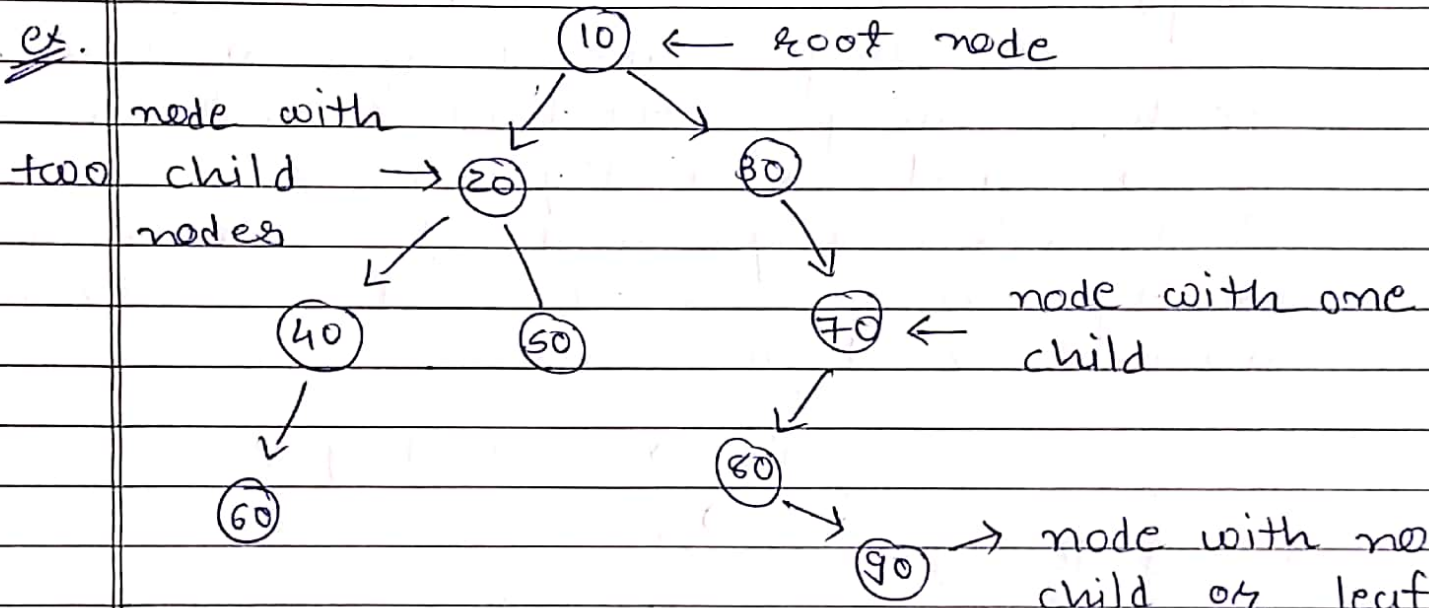
```
void displaylist()
{
    struct node *curr;
    int n=1;
    if (start == NULL)
    {
        printf("list is empty \n");
    }
    else
    {
        curr = start;
        printf("the circular linked list is\n");
        do {
            printf("%d \t", curr -> data);
            curr = curr -> next;
            n++;
        }
        while (curr != start);
    }
}
```

17) Define following terms related to tree with example :

1) Binary Tree :-

A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint binary trees called the left subtree and right subtree.

ex.



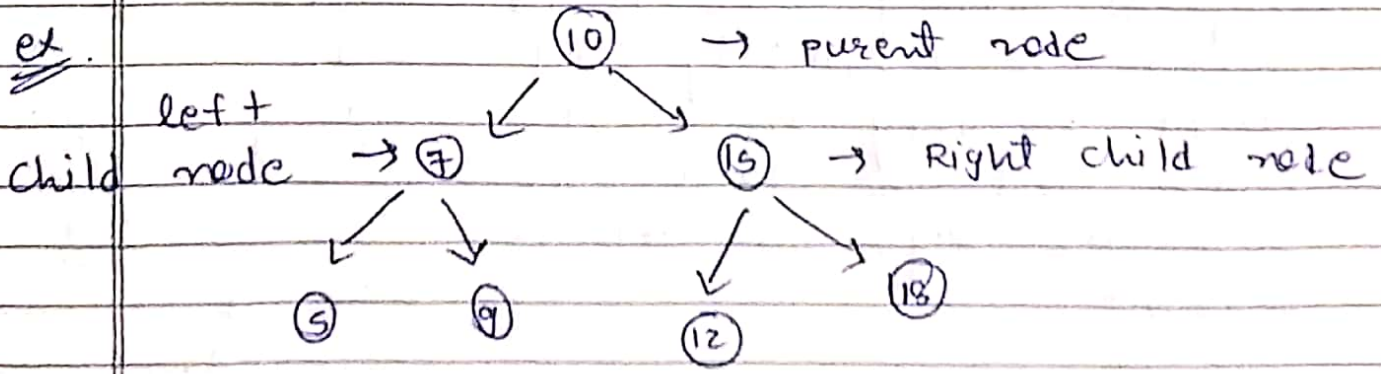
a) binary tree

2) Binary search tree :-

Binary search tree is a binary tree in which left node is less than parent node is less than right node

⇒ left node < parent node < right node

ex.



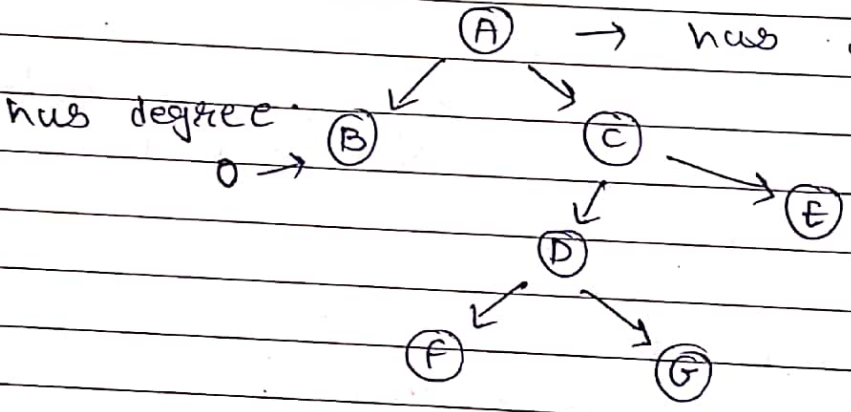
3) strictly binary tree :

If every nonleaf node in a binary tree has nonempty left and right subtrees the tree is called a strictly binary tree.

A strictly binary tree with n leaves always contains 2n-1 nodes.

If every non-leaf node in a binary tree has nonempty left & right subtrees, the tree is termed as strictly binary tree, or to put it another way all of the nodes in a strictly binary tree are of degree zero or two, never degree one.

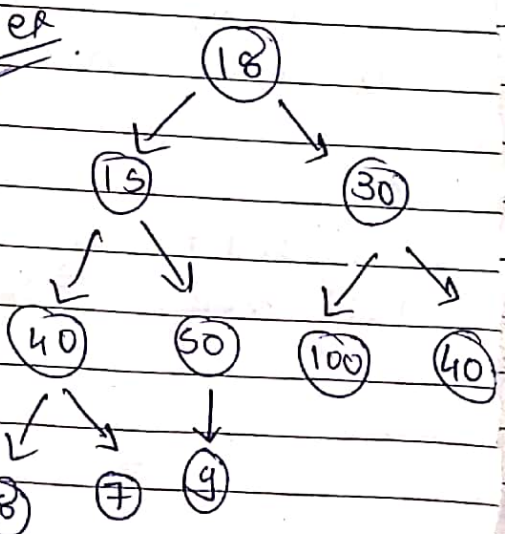
ex



4) complete binary tree :

A binary tree is complete binary tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible.

ex

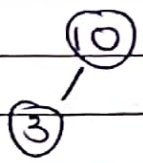


⇒ practical example of complete binary Tree is Binary leaf

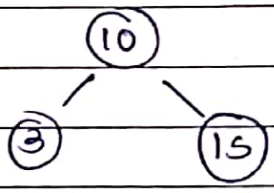
18) Construct binary search tree for the following data: 10, 3, 15, 22, 6, 45, 65, 23, 78, 34, 5
find its inorder, preorder and postorder traversal

⇒ step 1 : Insert 10

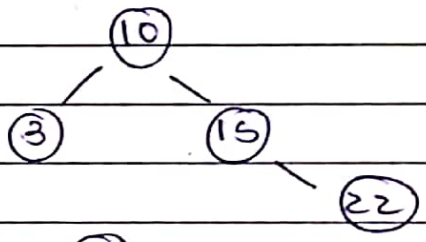
⇒ step 2 : Insert 3



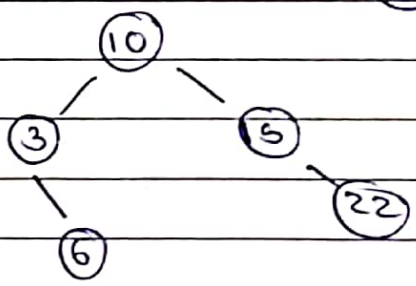
⇒ step 3 : Insert 15



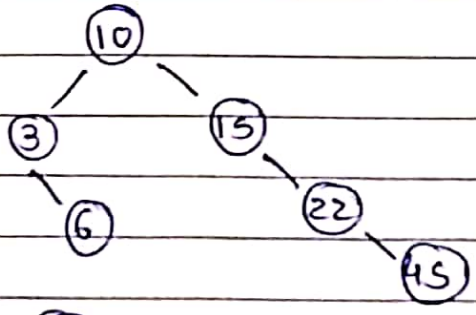
⇒ step 4 : Insert 22



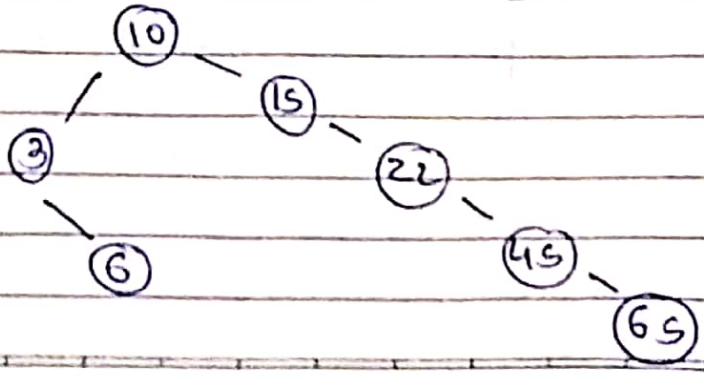
⇒ step 5 : Insert 6



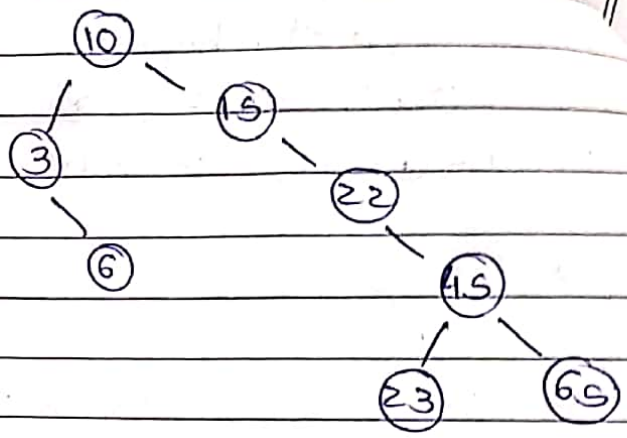
⇒ step 6 : Insert 45



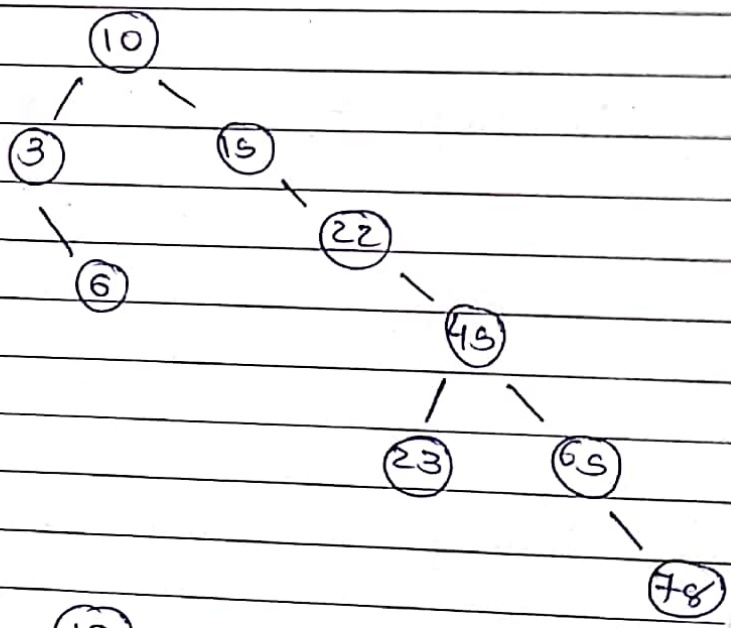
⇒ step 7: Insert 65



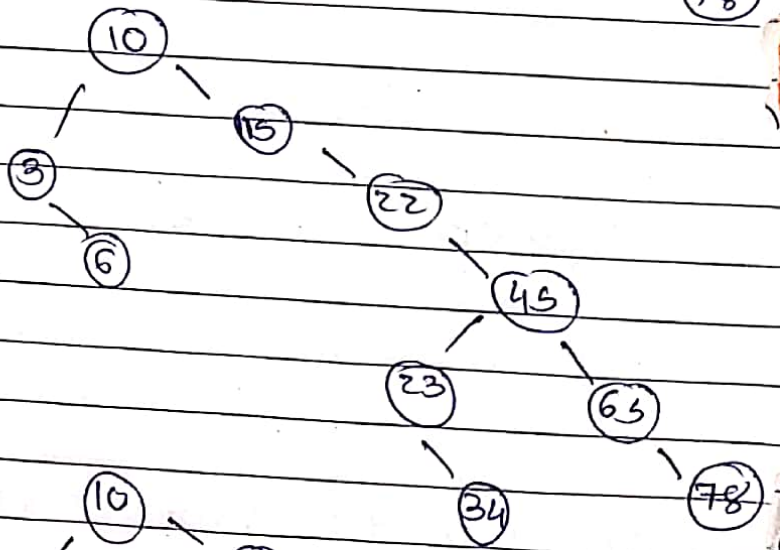
=> step 8 : Insert 23



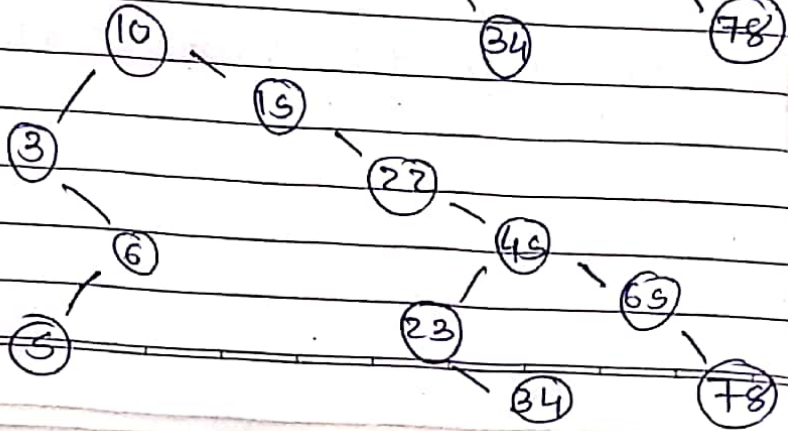
=> step 9 : Insert 78



=> step 10 : Insert 34



=> step 11 : Insert 5



Traversal

1) Inorder (LDR) : 5 → 6 → 3 → 10 → 15 → 22 → 34 → 23 → 45 → 65 → 78

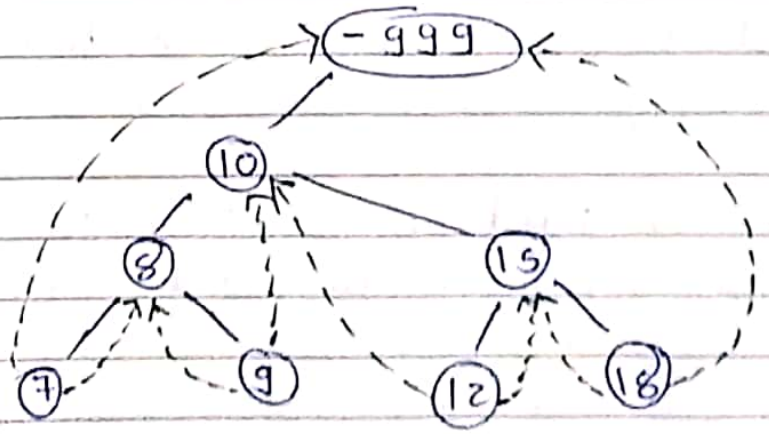
2) postorder (LRD) : 5 → 6 → 3 → 34 → 23 → 78 → 65 → 45 → 22 → 15 → 10

3) preorder (DLR) : 10 → 3 → 6 → 5 → 15 → 22 → 45 → 23 → 34 → 65 → 78

19) write a short note on treaded binary tree.

=) As we know that during binary tree creation for the leaf nodes there is no sub-tree further so we are just setting the left & right fields of leaf nodes NULL. since NULL value is put in the left and right field of the node it just wastage of memory. so to avoid NULL values in the node we just set the threads which are actually the links to the predecessor & successor nodes.

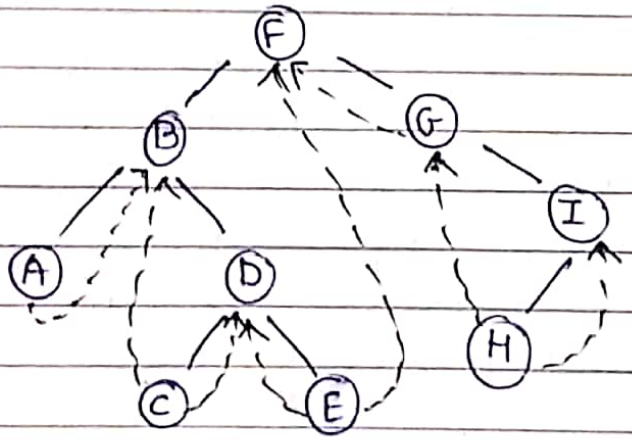
There types of threading possible. Inorder preorder and postorder threading technique. the typical treaded binary tree looks as below.



The basic idea in inorder threading is that the left & the right thread points to the predecessor & successor. there we are assuming the head node as the starting node and the root node of the tree is attached to left of head node which is shown in figure.

A binary tree is threaded by making all right child pointers that would normally be null point to the order successor of the node (if it exists), and all left chid pointer that would normally be null point to the inorder predecessor of the node.

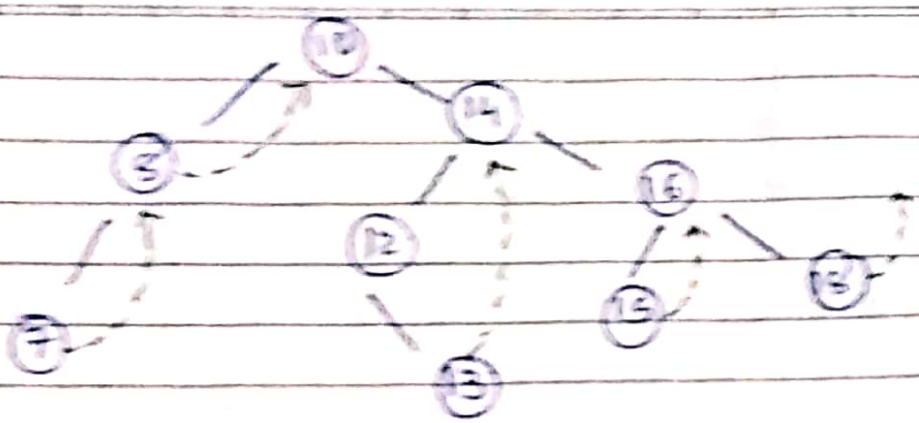
Threaded binary tree is a binary tree variant that facilitates traversal in particular order



- threaded tree, with special threading links shown by dashed arrows.

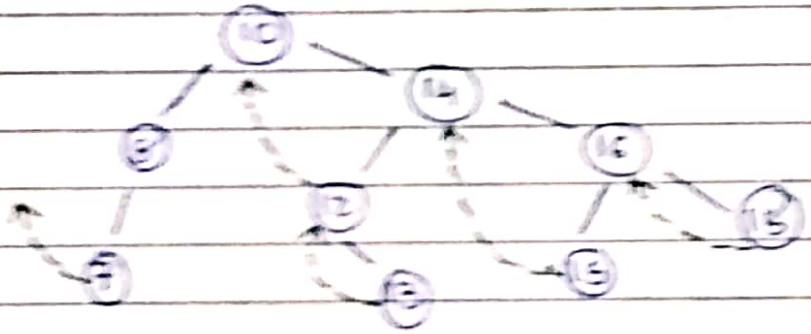
1) Right-in-threaded binary tree :-

The right in threaded binary tree is a binary tree in which the threads are only on right side of the node and are pointing to the successor node.



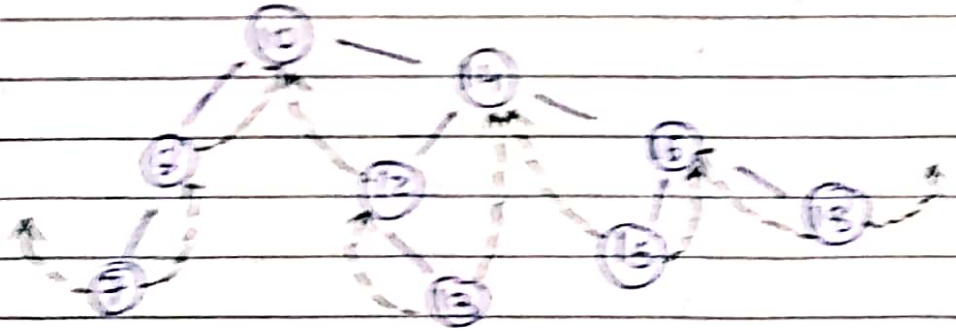
2) left - in - threaded binary tree :-

The left in threaded binary tree is a binary tree in which the threads are only on the left side of the node and are pointing to the predecessor node.



3) full - in - threaded binary tree :-

a threaded binary tree in which the left thread points to predecessor node and right thread points to successor node.



2a) write an algorithm for binary search method and discuss its efficiency.

precondition :

The prerequisite for this searching technique is that the list should be sorted.

ex.

so let us take an array of sorted elements.

0	1	2	3	4	5	6	7	
-40	11	33	37	42	45	99	100	

step 1 : Now the key element which is to be searched is 99, key = 99

step 2 : find the middle element of the array compare it with key.

Array

0	1	2	3	4	5	6	7
-40	11	33	37	42	45	99	100

sublist 1 — Middle element — sublist 2

if middle ! key

if 42 ! 99

if 42 < 99 search sublist 2

Now handle only sublist 2, Again divide it find mid of sublist 2

45	99	100

sublist 1 — Middle element — sublist 2

if Middle ! key

ex. if 99 ! 99

so, match is found at 7th position of array

ex at array [6]

4	5	6	7
45	99	100	

key = 99

3) Thus by binary search method we can find the element 99 present in given list at array [6]th location

Page No.:
Date: / /

* Algorithem for binary search using recursive definition

```
1) if (low > high)
2) return;
3) mid = (low + high) / 2;
4) if (x == a[mid])
5) else
6) search for x in a[mid + 1] to a[high];
7) return (mid);
8) if (x < a[mid])
9) search for x in a[low] to a[mid-1];
```

* Advantage of binary searching.

1) It is an efficient technique.

explanation.

To evaluate binary search, count the number of comparisions in the best case and worst case omitting average case.

=> The best case occurs if the middle item happens to be the target. then only one comparision is needed.

=> The worst case occurs if the target is not in the list then process dividing list in half continues until there is only one item left to check. suppose we have 1024 total items then it take 10 operations for worst case

=> In general if N is the size of the list to be searched and c is the number of comparisons to do so in the worst case, $c = \log_2 N$. Thus the efficiency of binary search can be expressed as logarithmic functions, when no of comparisions increases logarithmically with size of the list

efficiency comparision.

Note that, worst case number of comparisions is just 16 for a list with 1,00,000 items whereas for linear search it is 1,00,000 comparisions. similarly, if data items is doubled ie. increased from 1,00,000 to 2,00,000 number of comparisions in binary search increase from 16 to 17 whereas in linear search no of comparisions also gets doubled, ie. from 1,00,000 to 2,00,000. so,

we can say binary search is more efficient than linear search.

* Disadvantage of binary searching.

1) It requires specific ordering before the applying method.

2) It is complex to implement.