Module 4 : Trees

Definition of Tree:

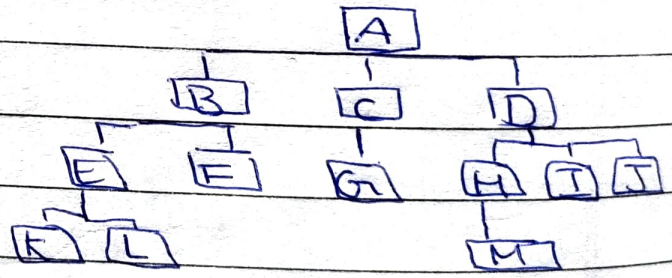
- A tree is a finite set of one or more nodes such that:

1. There is a specially designated node called the root.
2. The remaining nodes are partitioned into $n >= 0$ disjoint sets $T_1, \dots T_n$ where each of these sets is a tree.

Note: We call $T_1, \dots T_n$ the subtrees of the root.

Terminology:

- Nodes - 13.
- Degree of node (No of subtree of node)
- Leaf (terminal) (Nodes with degree 0)
- Non terminal.
- Parent
- children.
- sibling.
- Degree of a tree (3)



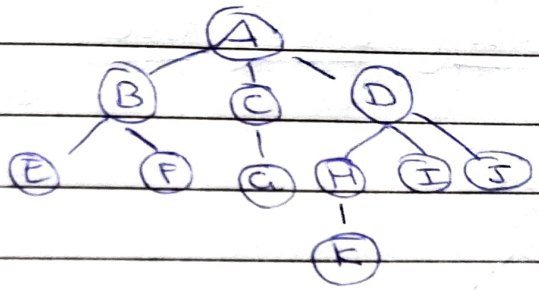
Date / /20

→ Max of degree of nodes in the tree.

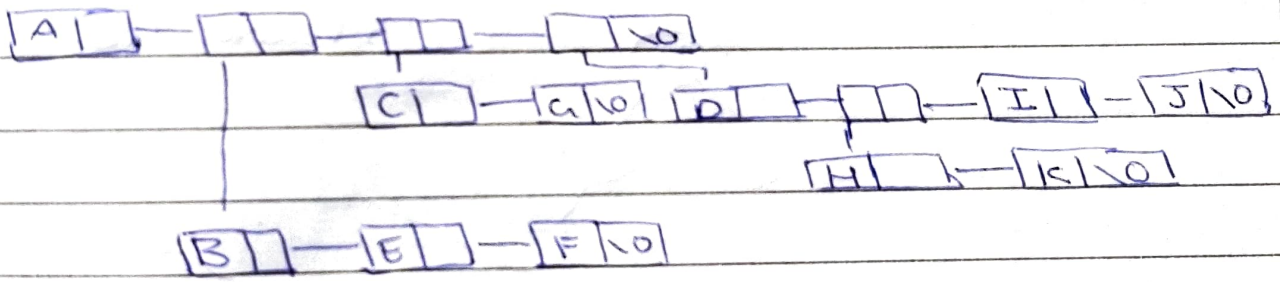
- Ancestor (All nodes along the path from the root to that node).
- Height of a tree (4).

→ Max level of any node in the trees.

⇒

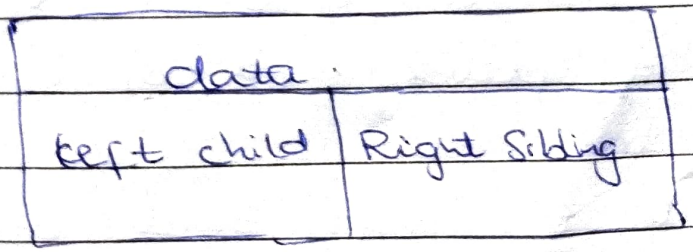


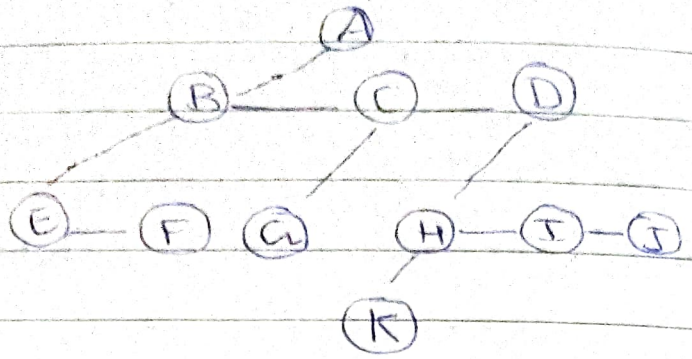
i) List Representation.



Other form : (A(B(E,F),C(G),D(H(K),I,J)))

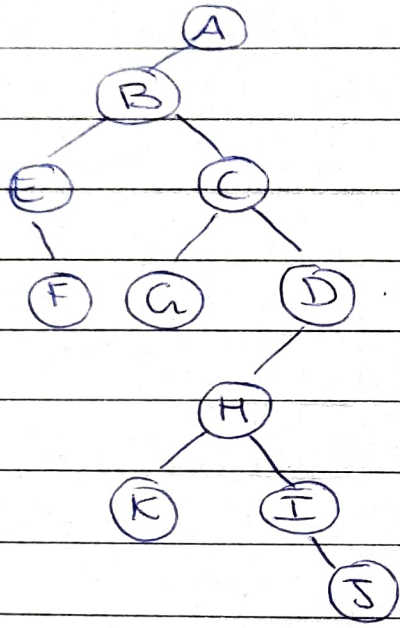
ii) Left child Right Sibling:





iii) Representing using binary tree.

degree-2.



clockwise 45°.

A Binary tree is a finite set of nodes that is either empty or consisting of a root and two disjoint binary trees called as left sub tree and right sub tree.

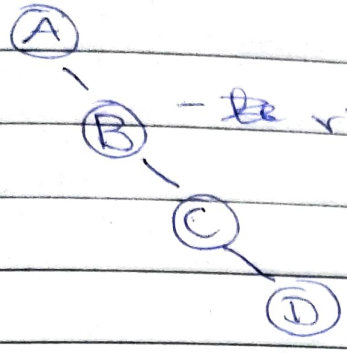
There are special kind of binary tree

1. Skewed tree

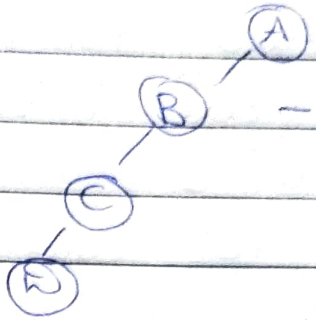
skewed to the left and there is a

Date / /20

corresponding tree and skewes to the right.

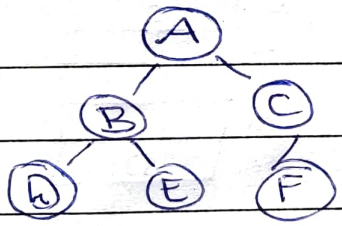


- right

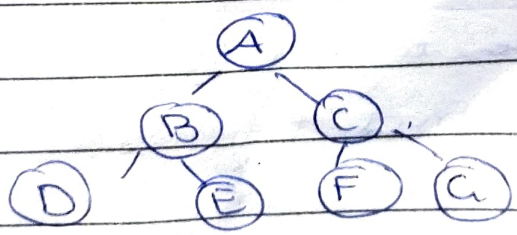


- left.

2. Complete tree: A Binary tree T with n levels is complete if all the levels except possibly all the levels are completely full and the last level has all its nodes to the left.



3. Full Binary tree: A full binary tree of depth k is a binary tree having $2^k - 1$ nodes when $k >= 0$.



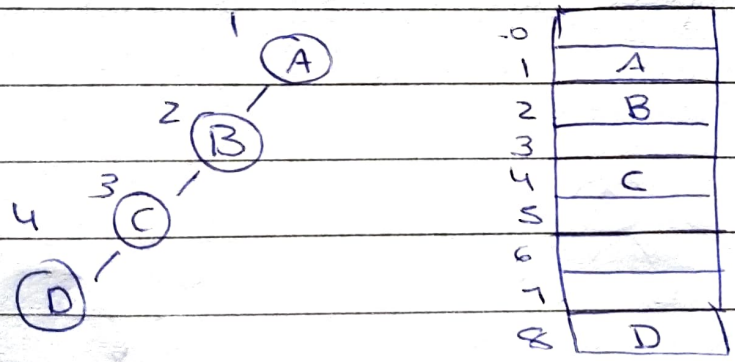
Representation of Binary Trees

1. Array Representation: Nodes are numbered from 1 to N so that we can represent in 1-D array. Position 0 is left empty.

For complete binary array representation there is no wastage of space, but in case of skewed binary tree space will be wasted.

Disadvantages: 1) Wastage of space.

2) Insertion & deletion problem.



2. Linked Representation:

Here each node has 3 fields

* left child * Right child * Data

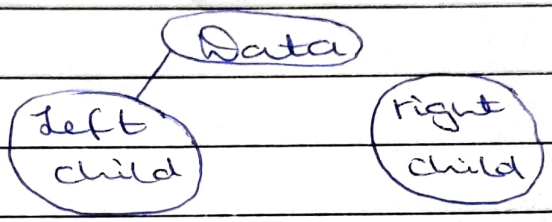
Structure representation of linked representation.

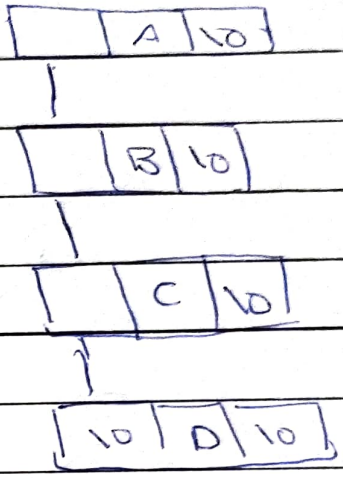
```
typedef struct tree
{
    int data;
```

Date / /20

```
struct tree * left child;
struct tree * right child;
};
```

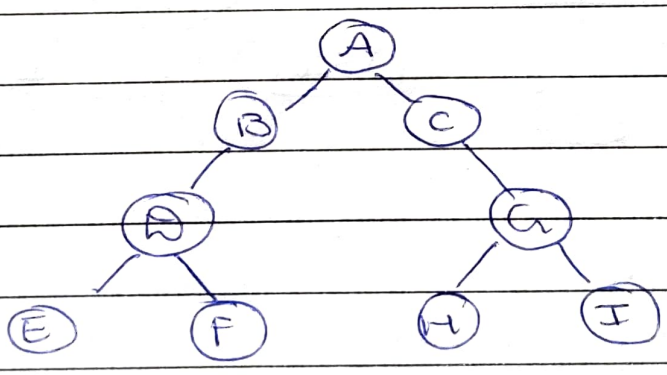
left child	Data	Right child





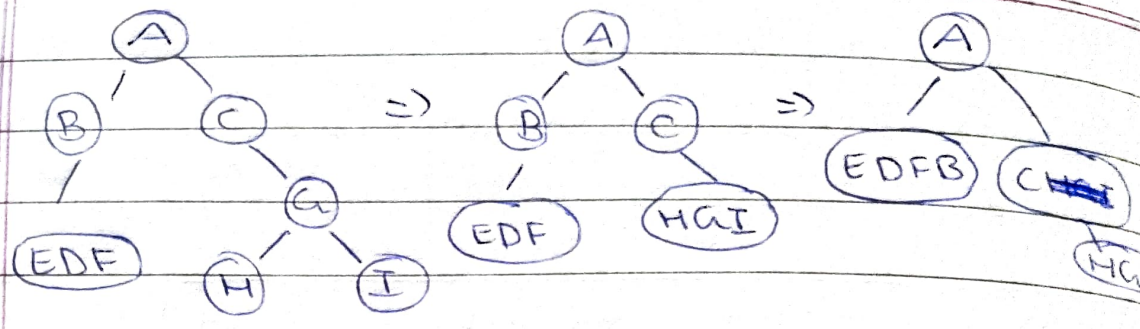
Traversing:

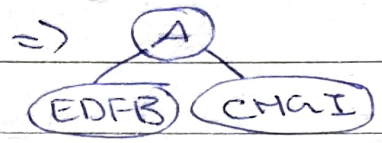
Inorder: LVR.



=> EDFBACHGI.

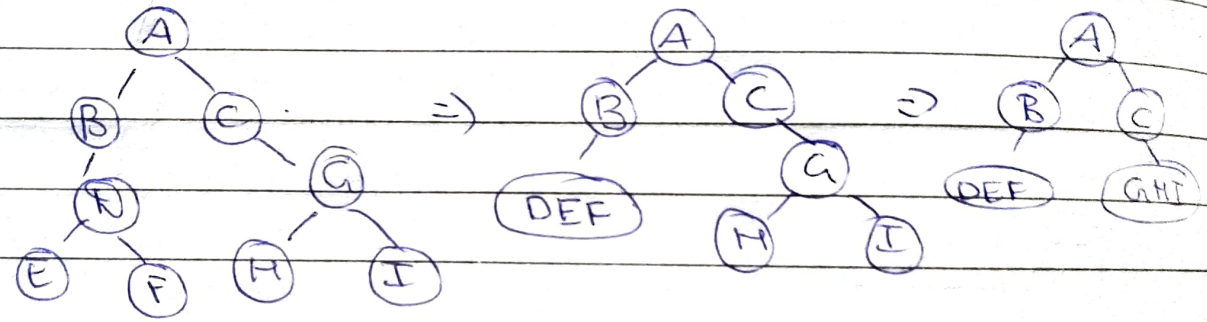
Steps:

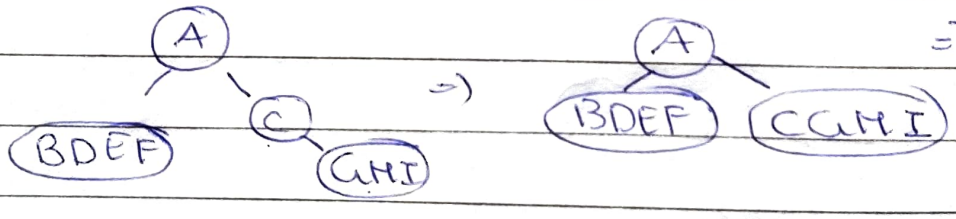




=> EDFBACHGI .

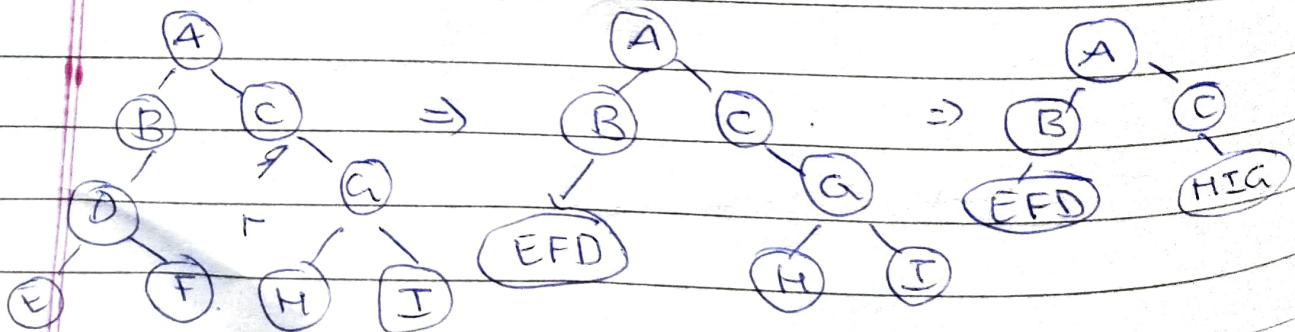
Preorder: VLR.

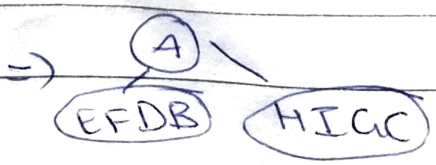




=> ABDEFCGHI

Postfix: LRV

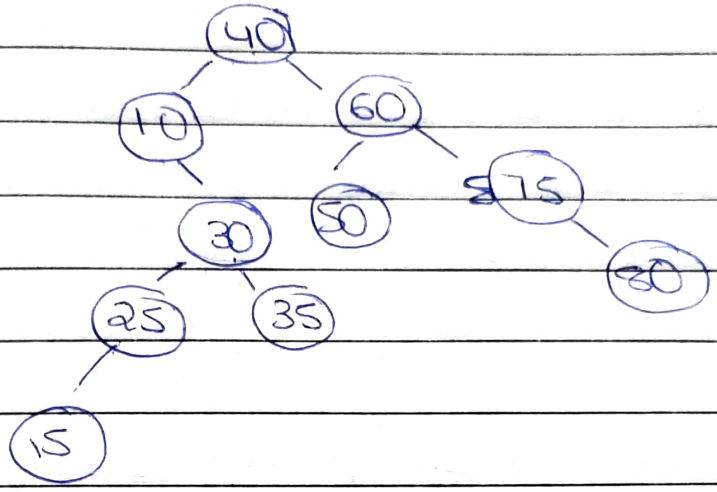




=> EFDBHIGCA .

Date / /20

Create binary search tree for 40 60 10 30 50 35 25 75 80 15.



Module 2 continuation:

Multiple stacks and queues.

- Assume that we have n stacks, we can divide the available memory into n segments.
- Assume i refers to stack no. of one of the n stacks.
- We must create indices for both bottom & top position of this stack.

Note:

→ Boundry[i], $0 \leq i < MAX\_STACKS$ points to the position immediately to the left of the bottom element of stack i.

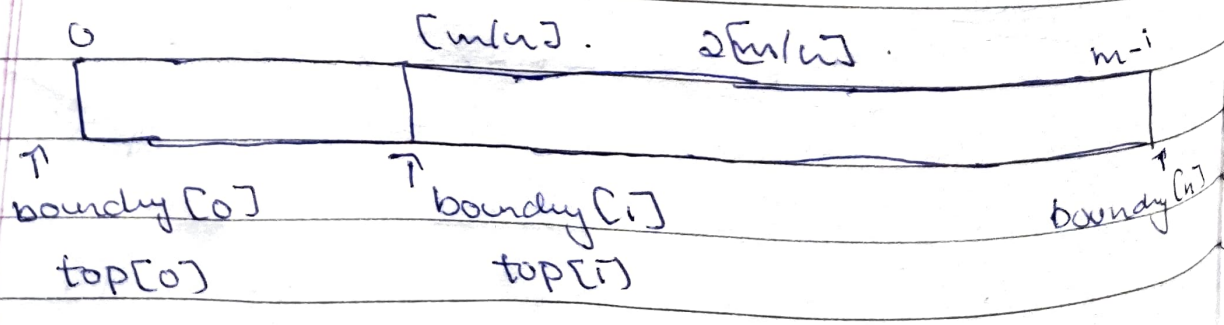
→ top[i], $0 \leq i < MAX\_STACKS$ points to the top element.

```
#define MEMORY_SIZE 100 /* size of memory */
# define MAX-STACKS 10 /* MAX no. of stacks */
/* global memory declaration */
element memory [MEMORY_SIZE];
int top [MAX_STACKS];
int boundary[MAX_STACKS];
int n; /* number of stacks entered by user */
```

To divide the array into roughly equal segments we use the following code.

```
top[0] = boundary[0] = -1;
for(j=1; j<n; j++)
top[j] = boundary[j] = ((MEMORY_SIZE/n)*j);
boundry[n] = MEMORY_SIZE;
```

```
* void push(int i, element item)
{/* add an item to the ith stack */.
if (top[i] == boundry[i+1])
  StackFull (i);
memory[++top[i]] = item;
}.
```



Date / /20

All stacks are empty and divided into roughly equal segments.

* -> element pop(int i).
{/* remove top element from the ith stack */
if(top[i] == boundy[i])
return stackEmpty[i];
return memory[top[i]--];
}.

• To guarantee stack Full adds elements as long as there is free space in array memory if we.

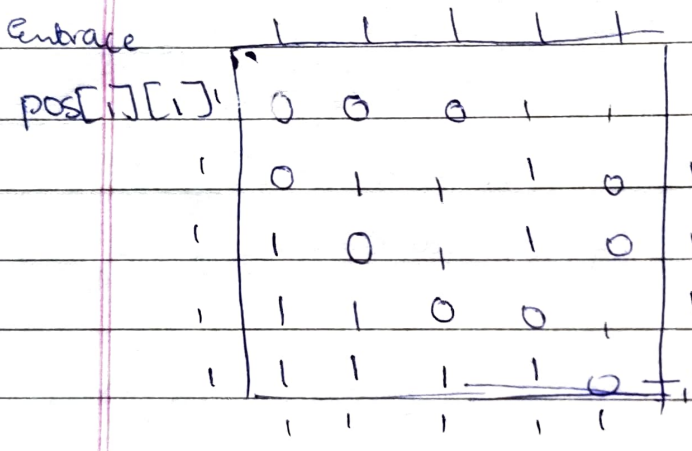
1. Determine the least j, i<j<n, such that there is free space b/w stacks] & j+1. That is, top[j]< boundary[j+1]. If there is such a j, then move i+1, i+2... j one position to the right. This creates a space b/w stack i & i+1.

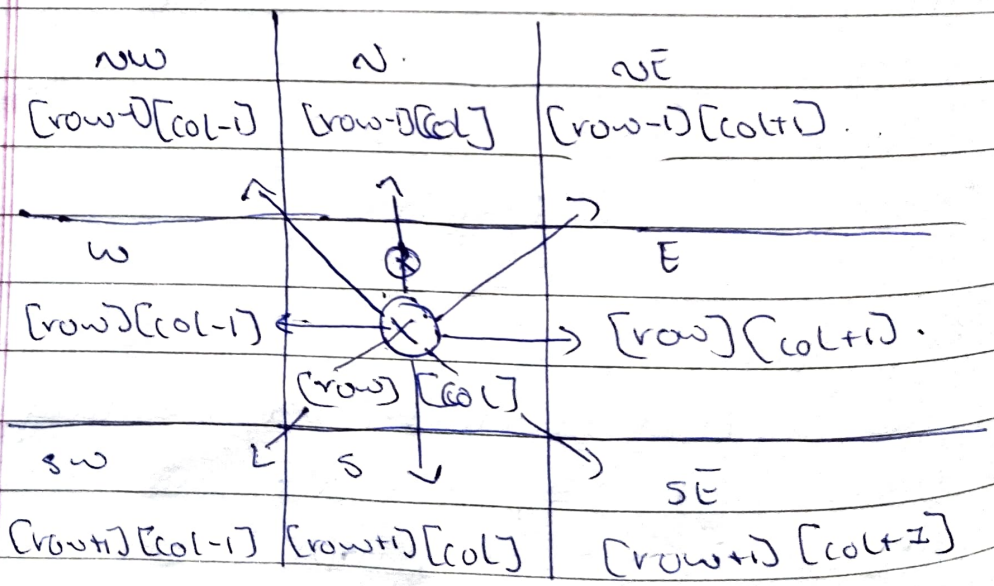
2. If there is no j as in (1), then look to the left of stack i. Find the largest j such that 0≤j<i & there is space b/w stacks j & j+1, top[j]< boundy[j+1]. If there is such a j, then move stacks

j+1, j+2 ... i one space to the left.

3. If there no j satisfying either conditions, then all MEMORY_SIZE spaces of memory are utilized and there is no free space.

A Mazing Problem:





Date / /20

Note: Not every position has 8 neighbors, on a border, possibly 3. We can surround maze by a border of ones. Thus an mxp maze requires an (m+2)x(p+2) array.

Possible Implementation:

```
typedef struct {
  short int vert;
  short int horiz;
  } offsets;
offsets move[8];
```

Name	Dir	move[dir].vert	move[dir].horiz
N	0	-1	0
NE	1	-1	1
E	2	0	1
SE	3	1	1
S	4	1	0
SW	5	1	-1
W	6	0	-1
NW	7	-1	-1

```
next_row = row + move[dir].vert;
next_col = col + move[dir].horiz;
```

Important Points:

- We need to save our current position, we can return to it and try another path if we take a hopeless path.
- To avoid returning to previously tried path mark[row][col] to record maze positions already checked.
- EXIT_ROW & EXIT_COL gives the coordinates of the maze exit.

Representation of stack.

```
typedef struct {
    short int row;
    short int col;
    short int dir;
} element.
```

- when searching this maze for an entrance to exit path, all positions with value zero will be on stack when the exit is reached.
- Since an mxp maze, can have at most mxp zeroes, it is sufficient for stack

Date / /20

to have this capacity.

```
void path(void).
{ int i, row, col, next_row, next_col, dir,
  found = FALSE;
  element position;
  mark[1][1] = 1; top=0;
stack[0].row=1; stack[0].col= 1; stack[0].dir=0;
while(top>-1 && !found)
  {
position = pop();
row=position.row; col=position.col;
dir = position.dir;
while(dir < 8 && !found)
  {
    /* move in direction dir*/
next_row = row + move[dir].vert;
next_col = col + move[dir].horiz;
if (next_row == EXIT_ROW && next_col == EXIT_COL)
    found=true;
else if (!maze[next_row][next_col] &&
 !mark[next_row][next_col])
  {
mark[next_row][next_col] = 1;
  position.row = row;
```

```
position.col=col;
position.dir = ++dir;
push(position);
row = next_row;
col = next_col;
dir = 0;
}
else ++dir;
}
if (found) {
printf("The path is:\n");
printf("row col\n");
for (i=0; i<=top; i++)
printf("%2d %5d", stack[i].row, stack[i].col);
printf("%2d %5d\n", row, col);
printf("%2d %5d\n", EXIT_ROW, EXIT_COL);
}
else printf("The maze does not have a path\n");
}
```

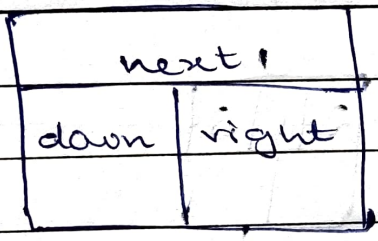
Date / /20

Module 3: Linked Lists.

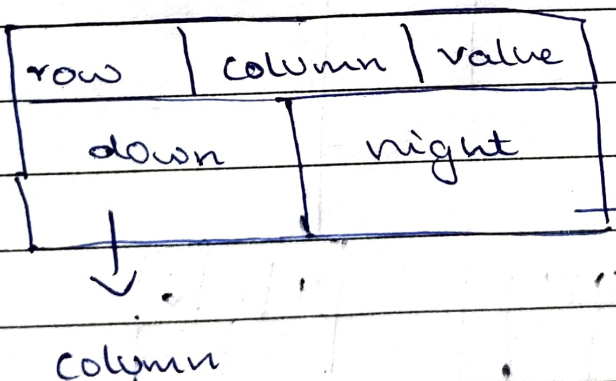
Sparse matrix Representation.

	0	1	2	3	4
0	0	0	3	0	0
1	3	0	2	0	0
2	0	5	0	0	2
3	0	0	0	0	0
4	4	6	0	0	1

- question.

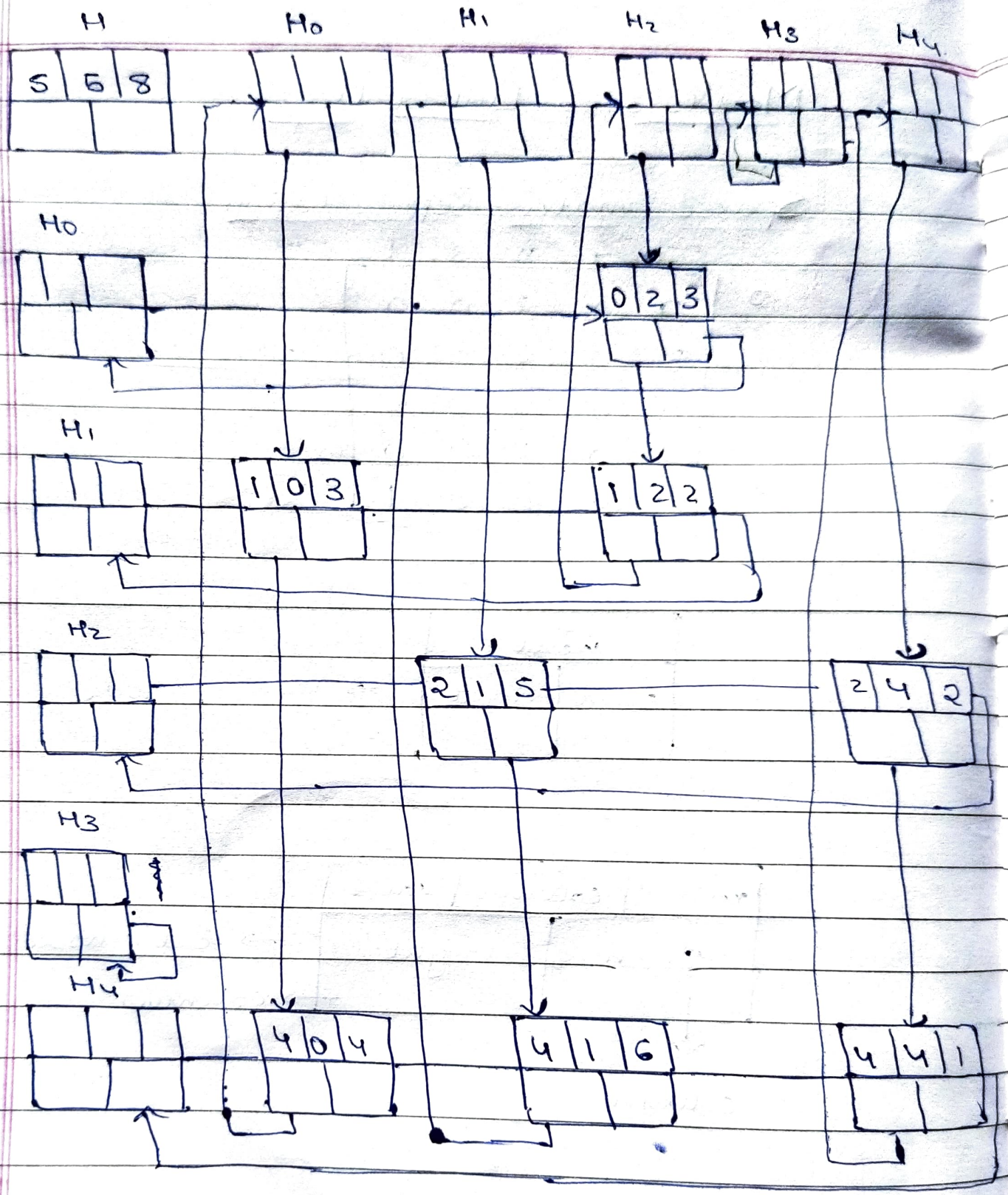


→ Header node.



→ Entry node.

Ans:



Date / /20

C declarations are as follows:

```
#define MAX_SIZE 50
typedef enum {head, entry} tagfield;
typedef struct matrixnode * matrix pointer;
typedef struct {
        int row;
        int col;
        int value;
      } entyNode;

typedef struct {
        matrix pointer down;
        matrix pointer right;
        tagfield tag;

union {
      matrixpointer next;
      entyNode entry;
      } u;
      } matrixNode;
matrixpointer hdNode [MAX_size];
```

Threaded Binary Tree:

Linked representation of any binary tree has more null links than actual pointers. There are n+1 null links out of 2n total links. They are replaced by the pointer called threads. To construct the threads we use the following rules.

1) If ptr → left child is null then replace ptr → left child with pointer to the node that would be visited before ptr in a inorder traversal.
2) If ptr → right child is null the replace ptr → right child with pointer to the node that would be visited after ptr in an inorder traversal.

- To distinguish between threads and normal pointers, we add 2 additional fields to the node structure.
  → left thread & right thread.
- If ptr → leftthread = TRUE then ptr → leftchild contains a thread otherwise it contains a pointer to the left child.

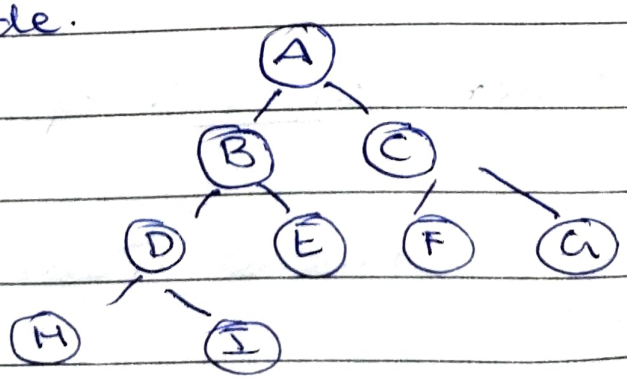
child contains a thread otherwise it contains a pointer to the right child.

Node structure:

- typedef struct threadedTree * threadedPoint
- typedef struct {
  int leftthread;
  threadedPointer leftchild;
  char data;
  int rightthread;
  threaded Pointer rightchild;
  } threaded Tree;

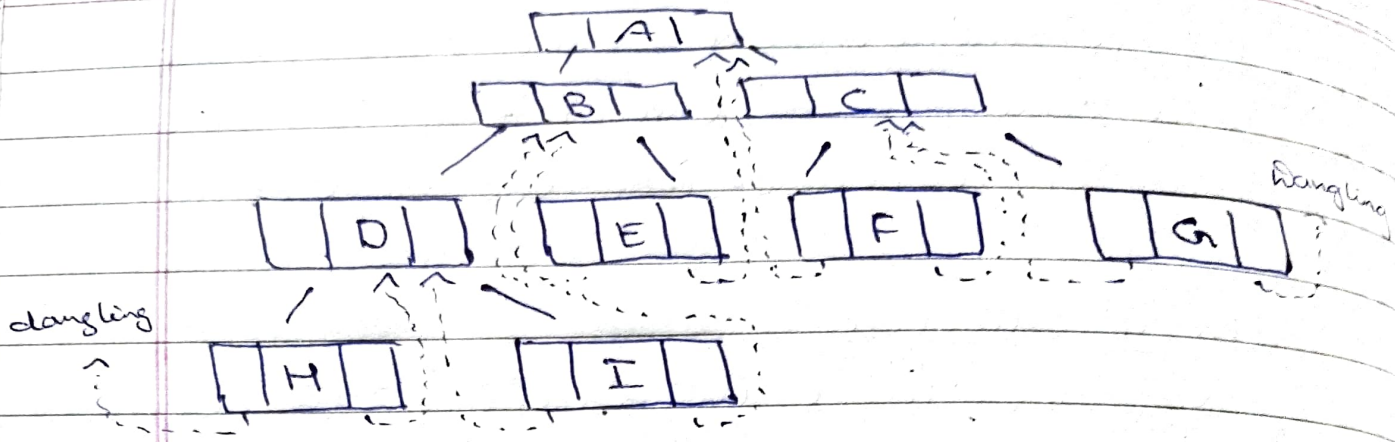
- For given example, we have 2 threads left dangling; one in the leftchild of H and the other in the right child of
- Solution: We assume a header node fo all threaded binary trees. The original tree is the left subtree of the header node.

Eg:



Inorder - LVR.

HDIBEAFCG.



Memory Representation of threaded tree

- Variable root points to the header node of the tree, while root → leftchild points to the start of first node of the actual tree.
- This is tree for all threaded trees.

Inorder traversal of a threaded binary tree

1) If ptr → nightthread = TRUE the inorder successor of ptr is ptr → nightchild.
2) Otherwise we follow a path of leftchild link from the nightchild of ptr we reach a node with a leftthred = TRUE.

Finding the inorder successor of a node

```
threadedPointer insucc(threadedPointer tree)
{
```

Date / /20

```
threadedPointer temp;
temp = tree -> rightchild;
if (tree -> rightthread)
  while (!temp -> leftthread)
  temp = temp -> leftchild;
  return temp;
}
```

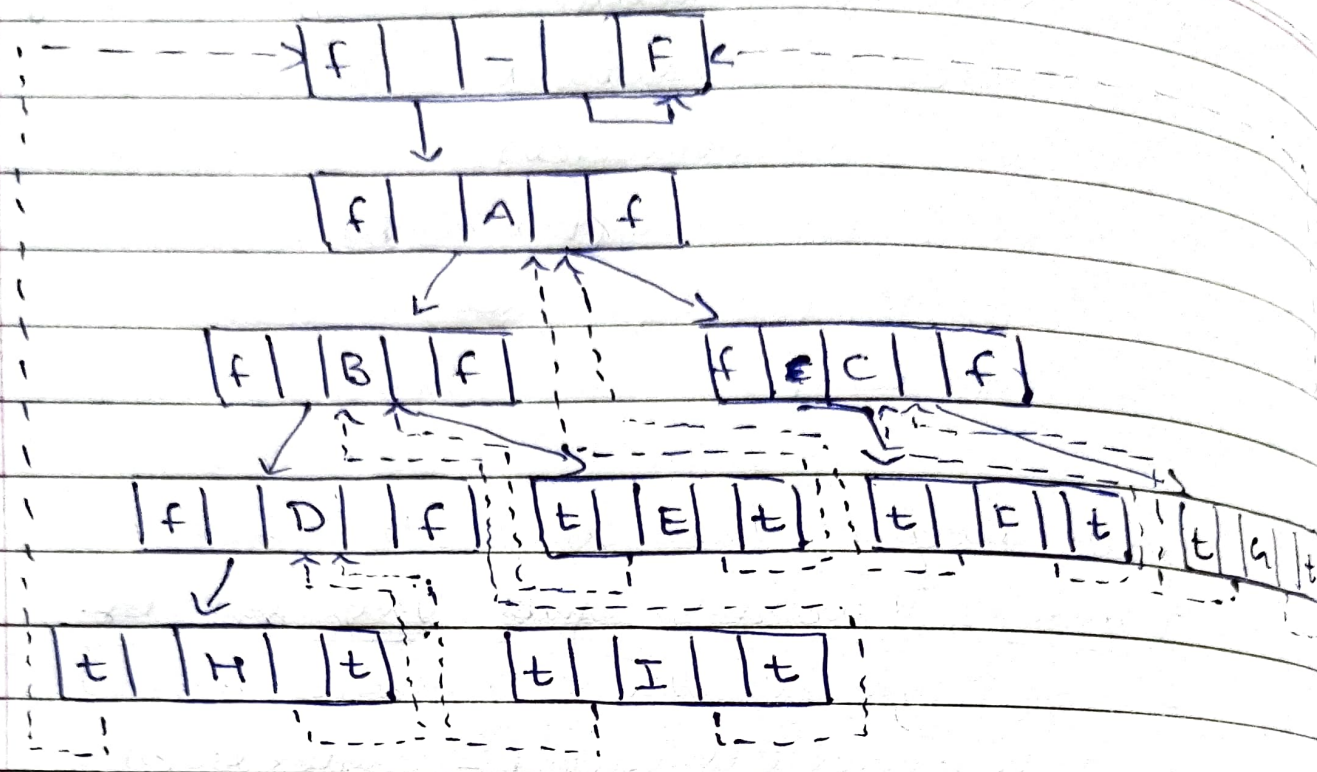
Inorder Traversal of a threaded BT:

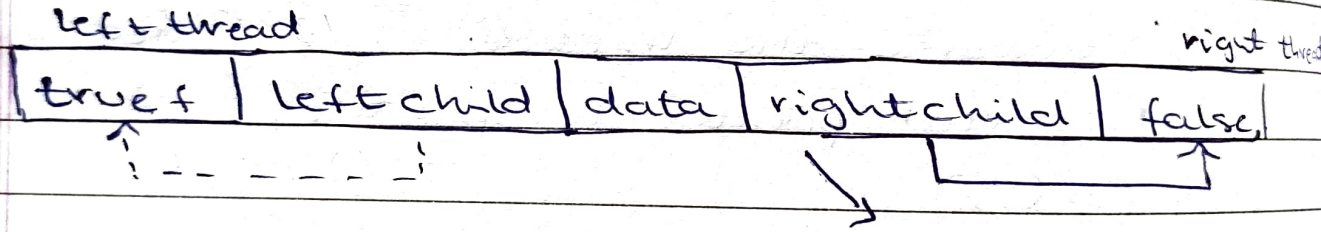
```
void tinorder (threaded - pointer tree)
{
/* traverse the threaded binary tree inorder */
 threaded_pointer temp = tree;
 for (;;)
 {
  temp = insucc(temp);
  if (temp == tree) break;
 printf ("%3c", temp -> data);
   }
}
```

Memory Representation of the threaded tree

HDIBEAFCG.

f = false
t = true





An empty threaded binary tree.

(Finding the inorder successor of a node threaded).

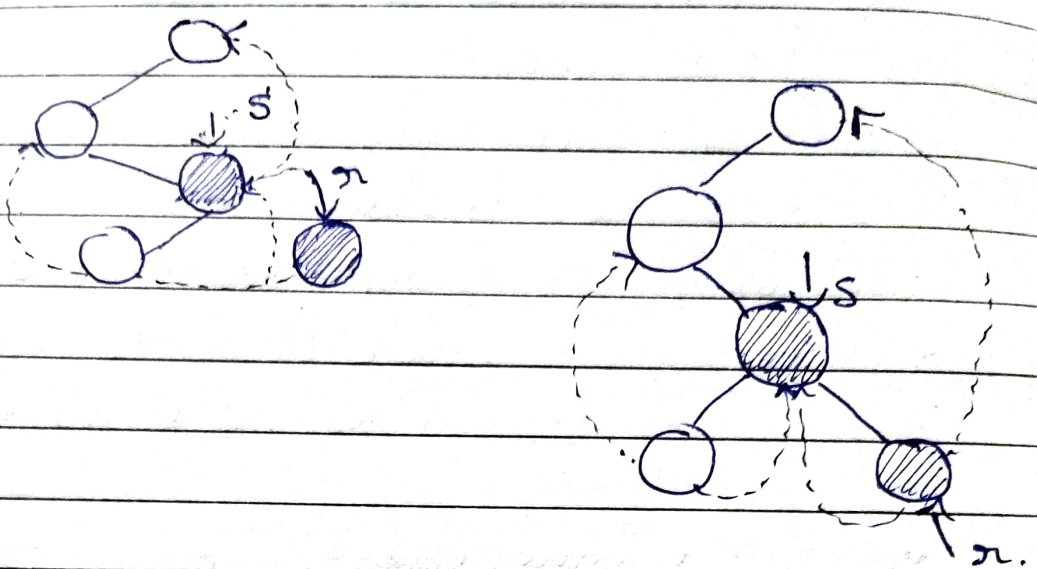
Inserting a node into a threaded binary tree:

1) If s has an empty right subtree then the insertion is simple.

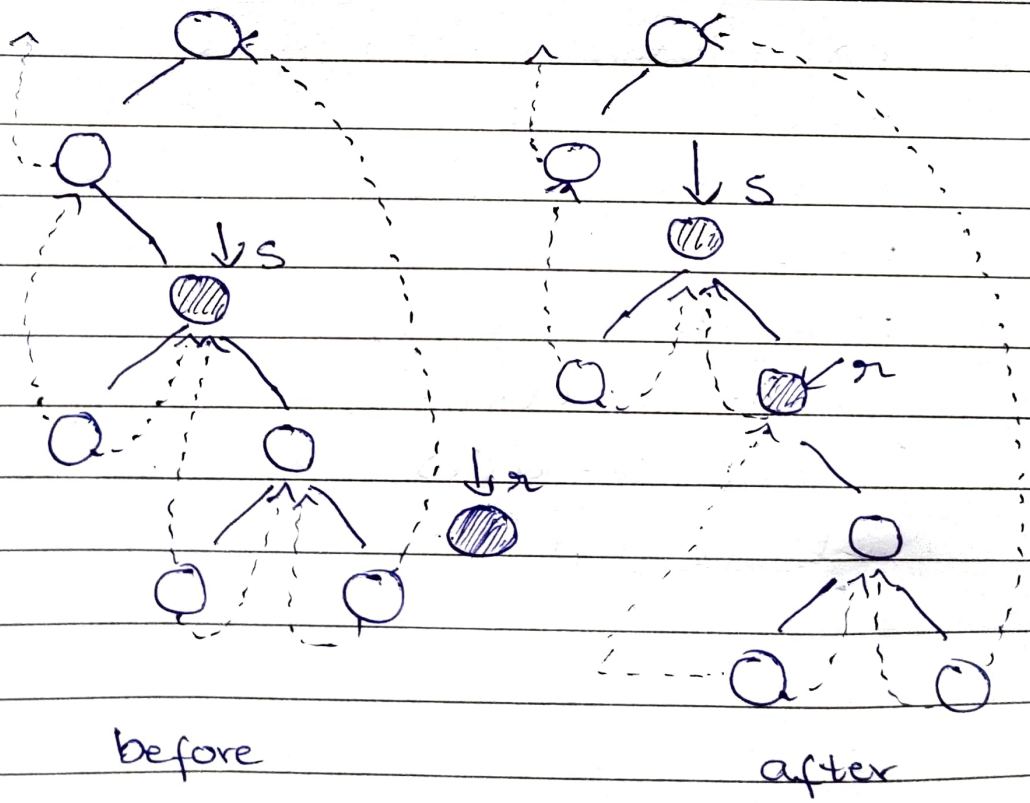
Date / /20

2) If right subtree of s is not empty then this right subtree is made as the right subtree of r after insertion. When this is done r becomes the inorder predecessor of a node that has a left thread = TRUE field and consequently there is a thread which has to be updated to point to r.

```
void insert_right (threaded_pointer s, threaded_pointer r)
{
  threaded_pointer temp;
  r->rightchild = s->rightchild;
  r->rightthread = s->rightthread;
  r->leftchild = s;
  r->leftthread = TRUE;
  s->rightchild = r;
  s->rightthread = FALSE;
  if (!r->rightthread)
  { temp = insucc(r);
    temp->leftchild = r;
  }
}
```



(a).



before

after

(b).

Date / /20

Binary Search Tree:

- It is a binary tree, which can be empty, if it is not empty then it satisfies following properties.

1) Each node has exactly one key and the keys in the trees are distinct.
2) The key (if any) in the left subtree are smaller than the key in the root.
3) The key (if any) in the right subtree should be greater than the key in the root.
4) The left & right subtree are also binary tree.

Common operations:

1. Insert - (lab program create())
2. Searching.
3. deletion.
4. Traversal - (lab program)

Iterative function to search.

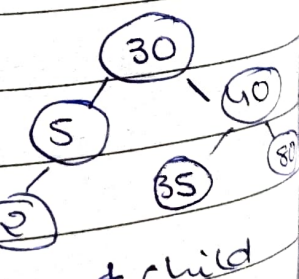
```
element * itersearch(treePointer tree, int k)
{
   while(tree)
   {
```

```
if(k==tree->data.key) return &(tree->data);
if(k<tree->data.key) tree=tree->leftchild;
else
 tree =tree->nightchild;
 }
   return NULL;
  }
```

Recursive function to search.

```
element * recursivesearch(treePointer root,int k)
{
  if(!root)return NULL;
 if(k==root->data.key) return &(root->data);
if(k<root->data.key) return search(root->
  leftchild, k);
  else
return search(root->nightchild, k);
   }
 }.
```

Deletion



case 1: Deletion of a leaf node.

→ set parent to that child to NULL & child node is freed.

Date / /20

case 2: deletion of a non leaf node that has only one child.

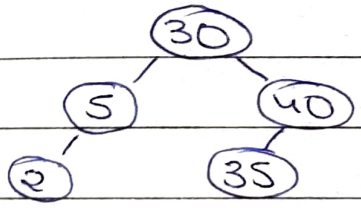
→ change the pointer from the parent node to the single child node.

case 3: Deletion of a non leaf node that has two children.

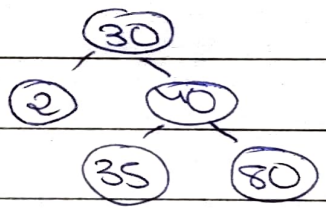
→ Replaced by largest pair in the left subtree or

→ smallest one in its right subtree

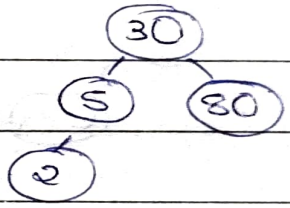
case 1:



case 2:



case 3:

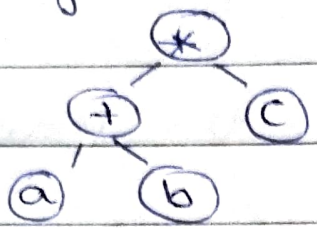


A Binary Expression Tree is

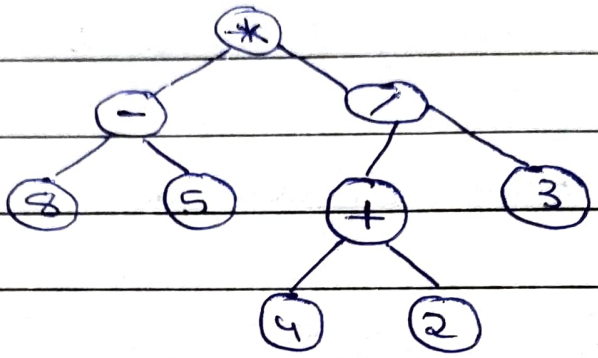
A special kind of binary tree in which:

1. Each leaf node contains a single operand.
2. Each nonleaf node contains a single binary operator.
3. The left & right subtrees of an operator node represent subexpressions that must be evaluated before applying the operator at the root of the subtree.

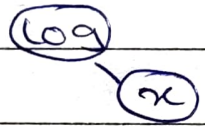
Eg: (a+b)*c.



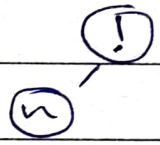
Infix: ((8-5)*((4+2)/3))



log x.



n!



Additional Binary Tree Operations:

1) Copying Binary Trees.

```
treePointer copy(treePointer original)
{
  treePointer temp;
  if (original)
  {
    MALLOC(temp, sizeof(**temp));
temp->leftchild = copy(original->leftchild);
temp->rightchild = copy(original->rightchild);
```

Date / /20

```
temp -> data = original -> data;
return temp;
}
return NULL;
}.
```

Testing for equality of binary trees:-

```
int equal (tree_pointer first, tree_pointer second)
{
/* function returns FALSE if the binary trees
first and second are not equal, otherwise
it returns TRUE */.
return ((!first && !second) || (first && second
&& (first -> data == second -> data) && equal
(first -> left_child, second -> left_child) &&
(first -> right_child, second -> right_child))).
}.
```

The satisfability Problem:

- Variables $x_1, x_2, x_3 \cdots x_n$, & operators $\wedge$(and) $\vee$(or), $\neg$(not).
- Rules for set of expression that we can form using these variables and operators.

→ A variable is an expression.

→ If x & y are expressions then $\neg x$, $x \wedge y$, $x \vee y$ are expressions.

→ Parentheses can be used to alter the normal order of evaluation ($\neg > \wedge > \vee$)

- Example. $x_1 \vee (x_2 \wedge \neg x_3)$.

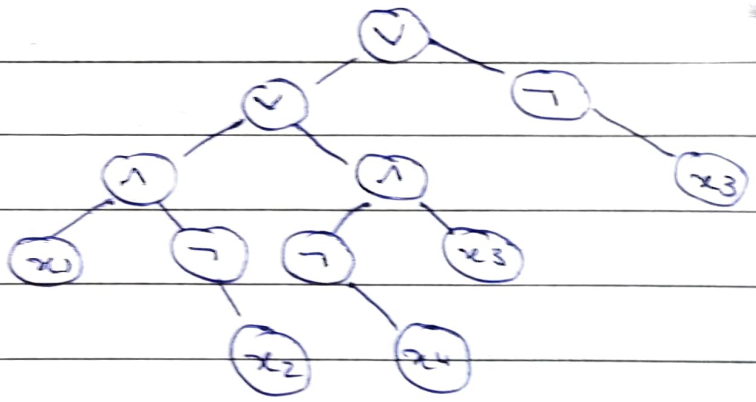
• satisfiability problem: Is there an assignment to make an expression true?

• $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$.

· Inorder traversal of this tree is

$x_1 \wedge \neg x_2 \vee \neg x_1 \wedge x_3 \vee \neg x_3$.

(t, t, t)
(t, t, f)
(t, f, t)
(t, f, f)
(f, t, t)
(f, t, f)
(f, f, t)
(f, f, f)



postorder traversal.

2nd possible combinations for n variables.

Node structure.

| left child | data | value | right-child |

typedef enum {not, and, or, true, false} logical;

Date / /20

```
typedef struct node *tree_pointer;
typedef struct node {
    tree_pointer left_child;
    logical data;
    short int value;
    tree_pointer right_child;
};
```

First version of satisfiability algorithm:

- for (all 2^n possible combinations) {
  generate the next combination;
  replace the variables by their values;
  evaluate root by traversing it in postorder;
  if (root → value) {
  printf ("<combination>");
  return;
  }
}
printf ("no satisfiable combination \n");

- void post_order_eval (tree_pointer node)
{
/* modified postorder traversal to evaluate a propositional calculus tree */

```
if (node) {
post_order_eval (node->left_child);
post_order_eval (node->right_child);
switch (node->data) {
case not: node->value = !node->right_child->value;
          break;
case and : node->value =
          node->right_child value || node->left_child->value;
case or: node->value =
  node->right_child || node->left child->value;
          break;
  case true : node->value = TRUE;
              break;
  case false : node->value = FALSE;
        }
    }
  }
```

Theorem:

- If T is a tree with degree k with n nodes, each having a fixed size, then $n(k-1)+1$ of nk child fields are zero, where $n \geq 1$.

Proof:

since each non zero child field points to a

node and there is exactly one pointer to each node other than the root, the no. of non-zero child fields in a n nodes tree is exactly n-1.

The total no. of child fields in a k array tree with n nodes is nk. Hence no. of zero fields ie $nk-(n-1) = n(k-1)+1$.

Properties of Binary trees.

Theorem 1:

1. The maximum no. of nodes on level i of a binary tree is 2^{i-1}, $i >= 1$.
2. The maximum no. of nodes in a binary tree of depth k is $2^k - 1$, $k >= 1$.

1. Proof:

The proof is by induction on i.

→ Induction base: The root is the only node on level $i = 1$. Hence the max. no. of nodes on level $i = 1$, is $2^{i-1} = 2^0 = 1$.

→ Induction hypothesis.

Let i be an arbitary +ve integer greater than 1. Assume that the max no. of nodes on level

$i-1$ is 2^{i-2}.

→ Induction step:

The max no. of nodes on level $i-1$ is 2^{i-2} by the induction hypothesis.

Since each node in binary tree has a maximum degree of 2, the max no. of nodes on level i is 2 times the maximum no. of nodes on level $i-1$ or $2 * 2^{i-1}$.

2) The max^A no. of nodes of depth k is (max no. of nodes on each level i)

$$\sum_{i=1}^{k} 2^{i-1} = 2^{k}-1$$

Theorem 2.

Relation b/n no. of leaf nodes to degree 2 nodes

For any non empty binary tree, T if n_0 is the no. of leaf nodes and n_2 the no. of nodes of degree 2, then $n_0 = n_2 + 1$.

Proof:

Let n_1 be the no. of nodes of degree 1 & n be the total no. of nodes. Since all nodes in T are atmost of degree 2 we have

$$n = n_0 + n_1 + n_2 \quad -(1),$$

Date / /20

If we count the no. of branches in a binary tree we see that every node except the root has a branch leading into it.

If B is the no. of branches than $n = B+1$.

All branch stem from a node of degree one or two thus $B = n_1 + 2n_2$.

Hence we obtain $n = B+1 = n_1 + 2n_2 + 1$ - (2).

Subtracting equation (2) from (1) & rearranging the terms we get,

$n_0 = n_2 + 1$.