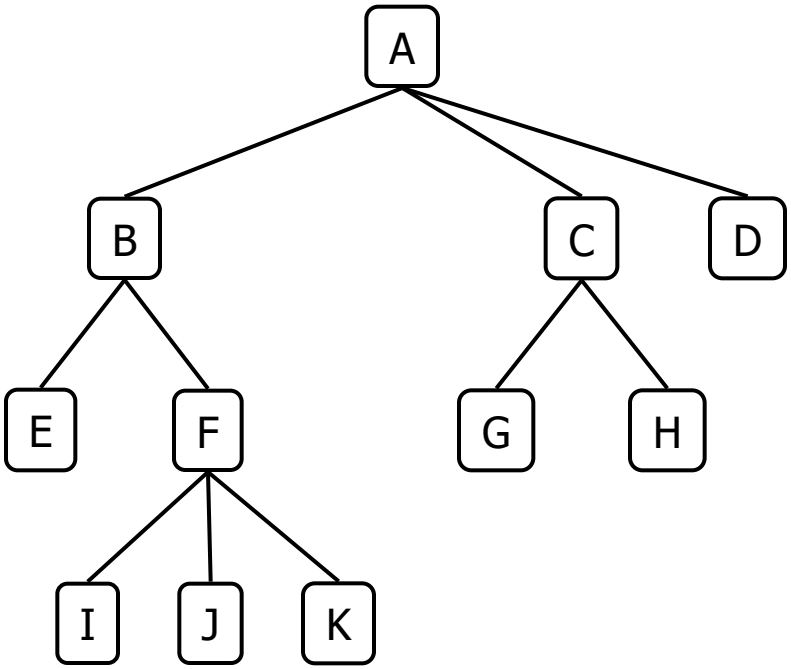
Trees

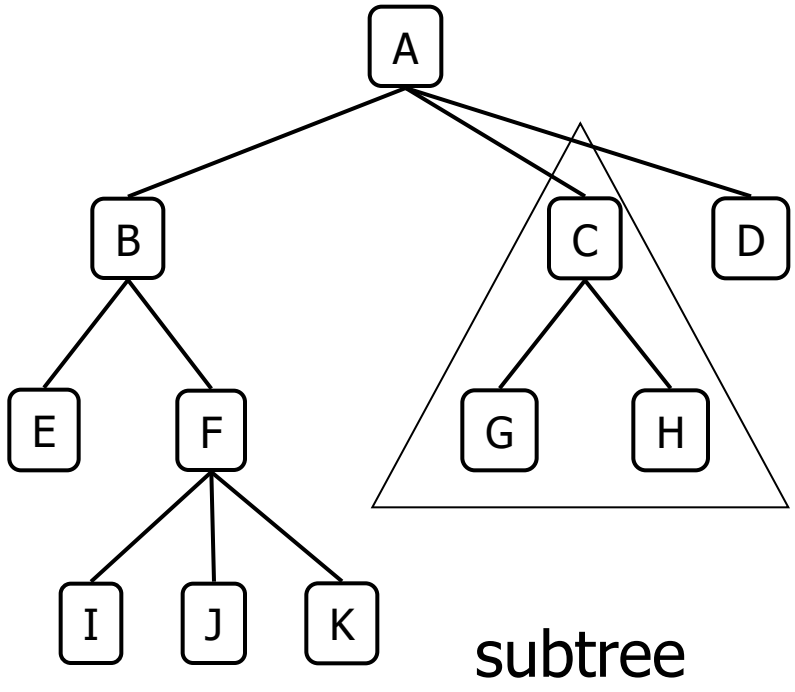
What is a Tree

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- A rooted tree has the following properties:
  - One node is distinguished as the root.
  - Every node x, except the root node, is connected by and edge from exactly one other node y. Node y is parent of node x.



Tree Terminology

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)
- Descendant of a node: child, grandchild, grand-grandchild, etc.
- Subtree: tree consisting of a node and its descendants

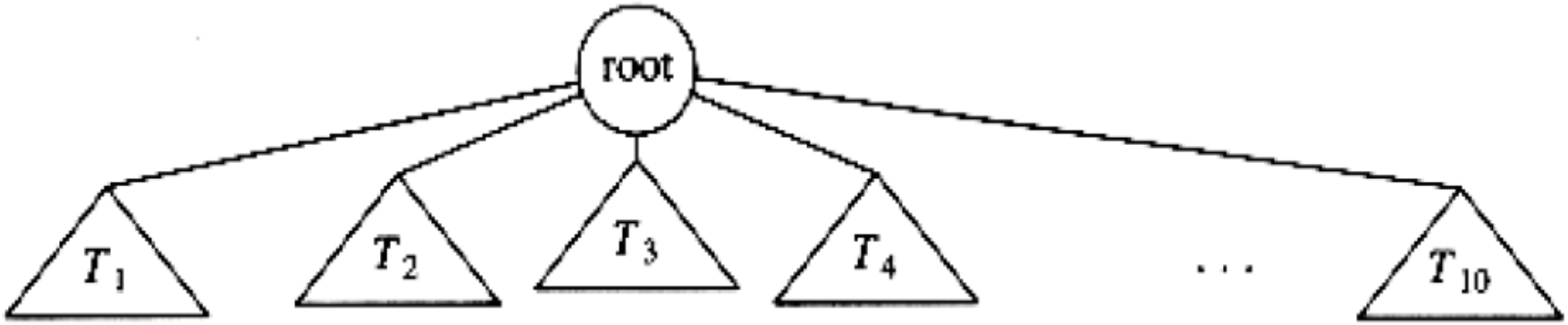


Tree Terminology (cont.)

- Path length: A unique path traverses from root to each node. The number of edges that must be followed is the path length.
- Height of a node: is the length of the path from the node to the deepest leaf.
- Height of a tree = Height of the root node
- Siblings: nodes with the same parents.
- Degree of a node: number of children of the node.
- Degree of the tree = maximum degree of its nodes.
- Balanced tree: difference between the depth of the leaf nodes is at most 1.
- A tree with N nodes must have N-1 edges, since every node except the root has an incoming edge.

Definition of Tree

- Recursive definition of the tree:
  - Either a tree is empty,
  - Or it consists of a root and zero or more nonempty subtrees T1, T2, … , Tk each of whose roots are connected by an edge from the root.
- In certain cases, we may allow some of the subtrees to be empty.



Advantages and Disadvantages of Tree Structure

- – To solve most of the computer science problems, we have to use tree data structure, but it requires more memory space since each node have many number of pointers.
- \+ It provides run time efficiency for solving the problems.
  - \+ Sequential search runs in O(n) time, however binary search tree requires O(lg n) time to search.
- \+ Most of the tree operations are designed as recursive functions and it is easy to implement.
- – Recursive functions require more memory space due to program stack usage.

Types of Trees in Computer Science

- **Binary search tree:** used for searching.
- **Coding tree:** used for text compression.
- **Dictionary tree (trie):** used to store and search words in a dictionary.
- **Heap tree:** used for sorting, implementing priority queues, etc.
- **Arithmetic expression tree:** used to represent precedence of mathematical expressions.

- Applications of trees:
  - Organization charts
  - File systems
  - Programming environments

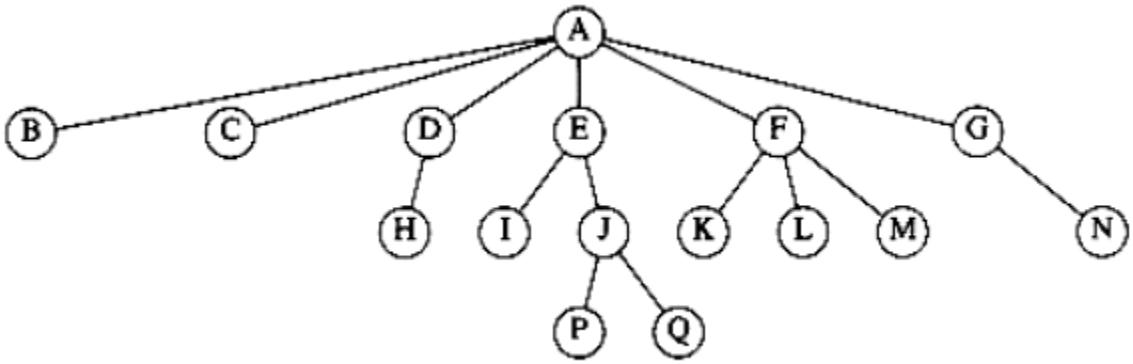
Tree Operations

- Create the tree,
- Traverse the tree,
- Insert new node,
- Search for a node,
- Delete a node,
- Save/load the nodes of the tree.

Implementation of Tree Data Structure

- Using pointers
  - Using one pointer for each child,
  - Using only two pointers.
- Using arrays
  - Store array indices of child nodes,
  - Use index expression to reach child nodes.

Using One Pointer for Each Child



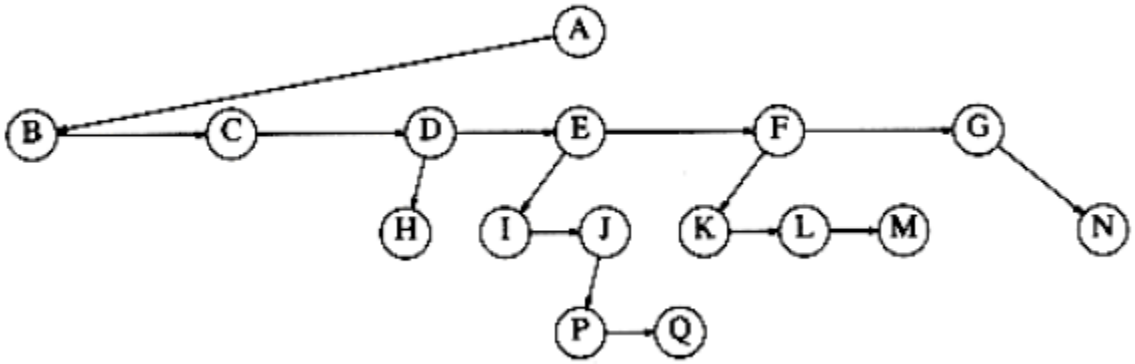
A tree of degree m is defined as follows:

```
#define m 30    // degree of the tree

typedef struct degree_m{
   int info;
   char message[100];
   struct degree_m *P[m];  // for each child a pointer is
}TREE_M;                   // defined

TREE_M *root=NULL;
```

Using Only Two Pointers



A tree of degree m is defined by using two pointers as follows:

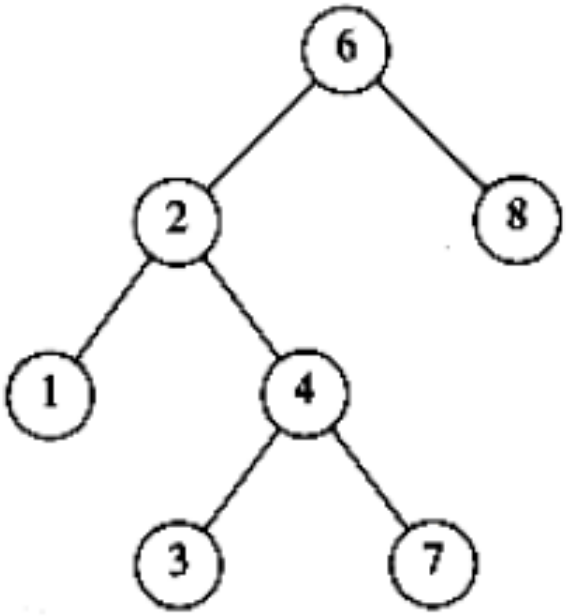
```
typedef struct degree_m2{
   int info;
   char message[100];
   struct degree_m2 *c;  // a pointer to the first child
   struct degree_m2 *s;  // a pointer to the right sibling
}TREE_M2;

TREE_M2 *root=NULL;
```

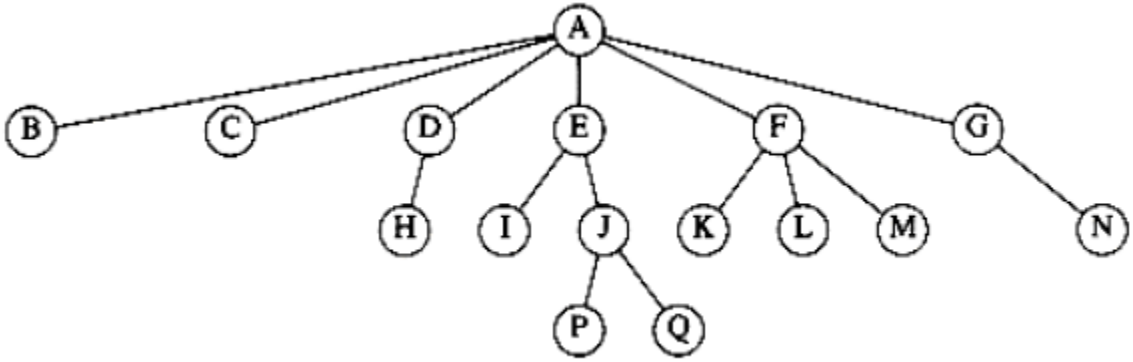
Definition of Binary Trees Using Pointers

```
typedef struct bTree{
    int info;
    char message[100];
    struct bTree *left;
    struct bTree *right;
}BTREE;

BTREE *root=NULL;
```



Using Arrays and Storing Indices of Child Nodes



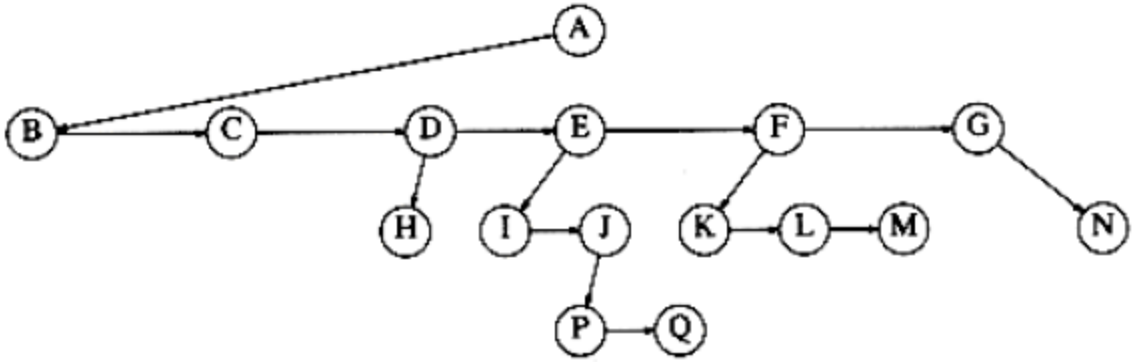
A tree of degree m is defined as follows:

```
#define m 30     // degree of the tree
#define N 1000    // number of nodes in the tree

typedef struct degree_m{
   int info;
   char message[100];
   int P[m];  // store indices of child nodes
}TREE_M;

TREE_M T[N];  // Array which stores the tree
```

Using Arrays and Storing Indices of First Child and Right Sibling Nodes



A tree of degree m is defined by using two indices as follows:

```
#define N 1000   // number of nodes in the tree
typedef struct degree_m2{
   int info;
   char message[100];
   int c;  // array index of the first child
   int s;  // array index of the right sibling
}TREE_M2;

TREE_M2 T[N];
```

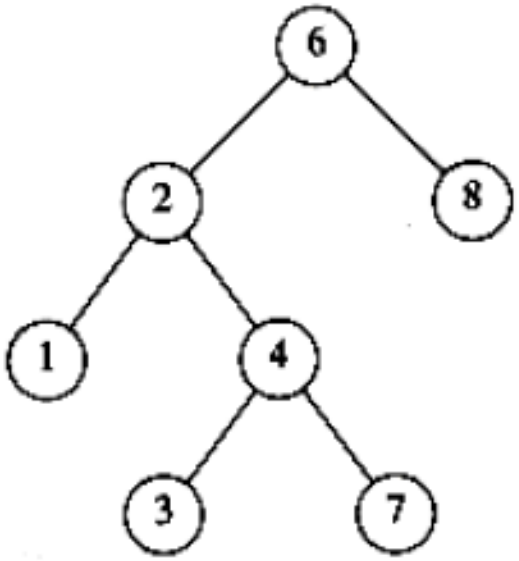
Use Index Expression to Reach Child Nodes

```
#define N 1000

typedef struct bTree{
   int info;
   char message[100];
}BTREE;

BTREE T[N];
```

index of the root node = 0
index of the left child of node i = 2 * i + 1
index of the right child of node i = 2 * i + 2



For trees having degree m

- index of the root node = 0
- index of the 1st child of node i = m * i + 1
- index of the 2nd child of node i = m * i + 2

  …
- index of the mth child of node i = m * i + m

When array implementation should be used?

- If we have full and/or balanced trees, array implementation should be preferred.
  - So, it is suitable for heap trees.
- It provides direct access to nodes, so it is faster than pointer based implementation.
- If we have sparse and/or unbalanced trees, we should use pointer based implementation.

Binary Trees

- Degree of the tree is 2.

Some binary tree types:

- Binary search trees,
- Heap trees,
- Expression trees,
- Coding trees.

Binary Search Trees

- Degree of the tree = 2
- For each node i of the tree:

  Left_child(i) < Node(i)

  Right_child(i) >= Node(i)

- Question: Insert the following numbers into a binary search tree:

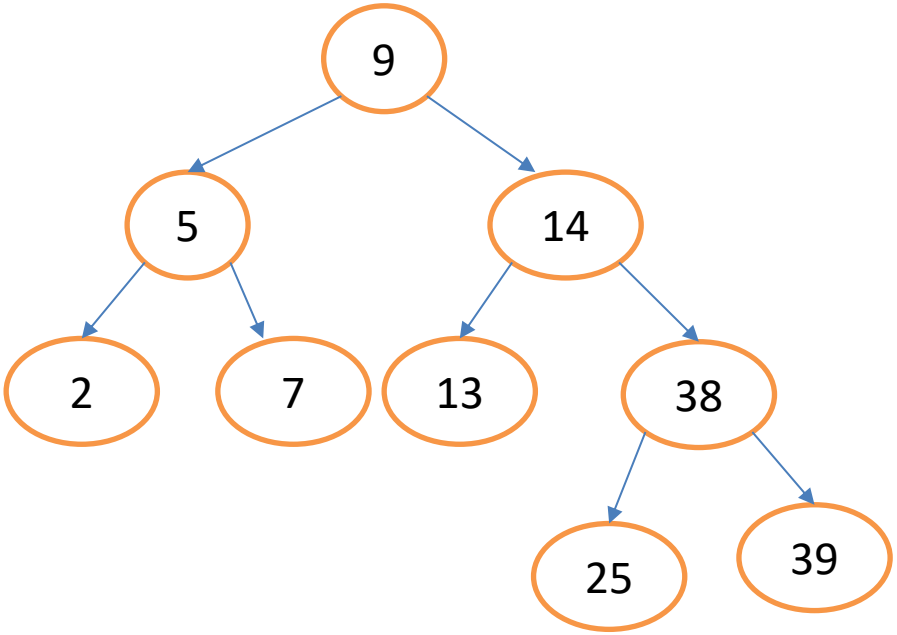
  9, 5, 14, 2, 7, 13, 38, 25, 39 (balanced)

  9, 14, 38, 39 (unbalanced)

  9, 5, 7, 6 (unbalanced)

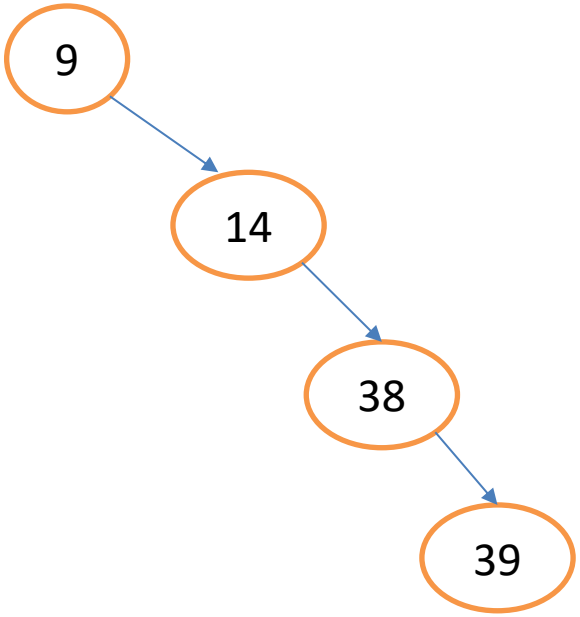
Binary Search Trees

Question: Insert the following numbers into a binary search tree: 9, 5, 14, 2, 7, 13, 38, 25, 39 (balanced)

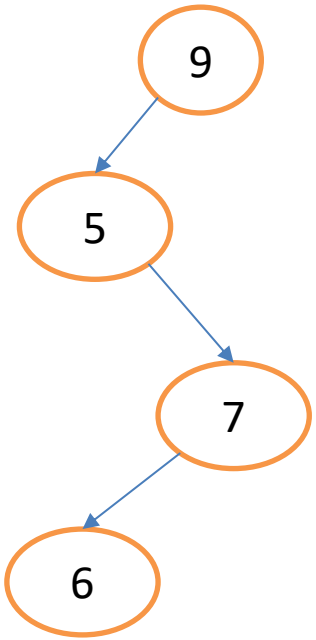


Binary Search Trees

Insert: 9, 14, 38, 39
(unbalanced)



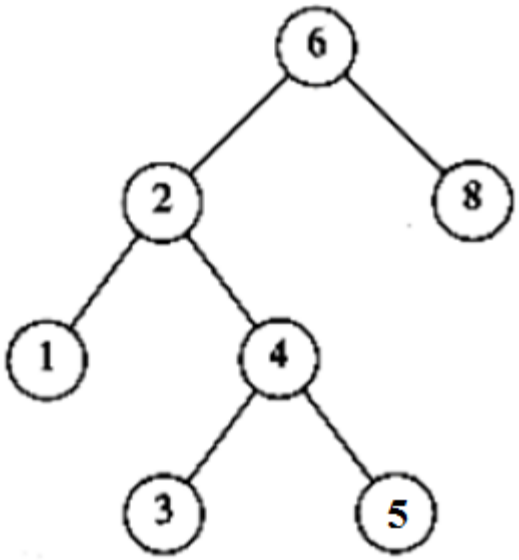
Insert: 9, 5, 7, 6
(unbalanced)



Implementation of a Binary Search Tree

```
typedef struct bTree{
    int info;
    char message[100];
    struct bTree *left;
    struct bTree *right;
}BTREE;

BTREE *root=NULL;
```



Insertion

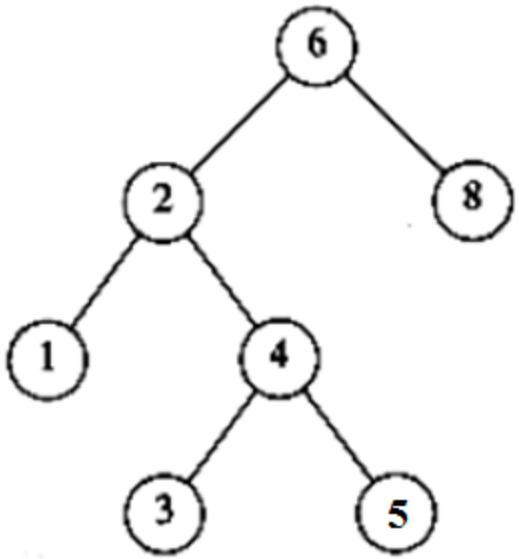
- If the tree is empty,
  - insert into the root node
- Else
  - If value of the new node is less than the root node
    - Continue with the left subtree
  - Else
    - Continue with the right subtree

Insertion function

```
void insert(BTREE *treeRoot, BTREE *newNode){
   if (treeRoot == NULL)
      root = newNode;
   else {
      if (newNode->info < treeRoot->info){
         if (treeRoot->left == NULL)
            treeRoot->left = newNode;
         else
            insert(treeRoot->left,newNode);
      }
      else{
         if (treeRoot->right == NULL)
            treeRoot->right = newNode;
         else
            insert(treeRoot->right,newNode);
      }
   }
}
```

Traversing the Tree

- **Pre-order traversal:**
  root, left, right
    6, 2, 1, 4, 3, 5, 8
- **In-order traversal:**
  left, root, right
    1, 2, 3, 4, 5, 6, 8
- **Post-order traversal:**
  left, right, root
    1, 3, 5, 4, 2, 8, 6



Traversal function

```
// pre-order traversal

void traverse(BTREE *treeRoot){
   if (treeRoot != NULL){
      printf("%d %s\n",treeRoot->info,
               treeRoot->message);
      traverse(treeRoot->left);
      traverse(treeRoot->right);
   }
}
```

Searching for a node

While root is not null and item is not in the root

    if item < root

        go to left

    else

        go to right

Return root

Search function

```
BTREE *search(BTREE *treeRoot, int item){
   while ((treeRoot != NULL) &&
            (treeRoot->info != item)){
      if (item < treeRoot->info)
         treeRoot = treeRoot->left;
      else
         treeRoot = treeRoot->right;
   }
   return treeRoot;
}
```

Deleting a node

q: node to be deleted

qa: parent of q

- Find q and qa
- If q is not found
  - Exit from the function
- If q has 2 children
  - Find the node s having the highest value in the left subtree
  - Copy s to q
- Now, delete the node having at most 1 children
- Free space occupied by the deleted node.

Deletion function

```
BTREE *delete(BTREE *treeRoot, int item){
   BTREE *qa, *q, *qc, *sa, *s;
   if (treeRoot == NULL)
      return NULL;
   q = treeRoot;
   qa = NULL;
   // find the item to be deleted
   while ((q != NULL) && (q->info != item)){
      qa = q;
      if (item < q->info)
         q = q->left;
      else
         q = q->right;
   }
   if (q == NULL){
      printf("Item to be deleted is not found\n");
      return NULL;
   }
```

Deletion function (cont.)

```
// if q is found and it has 2 children
if ((q->left != NULL) && (q->right != NULL)) {
   s = q->left;
   sa = q;
   // Find item having the max value in the left
   // subtree. So, take the rightmost node
   // in the left subtree.
   while (s->right != NULL){
      sa = s;
      s = s->right;
   }
   q->info = s->info;  // copy s to q
   strcpy(q->message, s->message);
   q = s;
   qa = sa;
}
```

Deletion function (cont.)

```
   // now q has at most 1 child, and find it

   if ((q->left != NULL)
      qc = q->left;
   else
      qc = q->right;

   // delete q

   if (q == treeRoot)
      root = qc;
   else {
      if (q == qa->left)
         qa->left = qc;
      else
         qa->right = qc;
   }
   free(q);
   return q;
}
```

Create Node function

```
BTREE *readNode(){
    BTREE *newNode;
    newNode = (BTREE *)malloc(sizeof(BTREE));
    if (newNode == NULL)
        return NULL;
    printf("Enter info:");
    scanf("%d",&(newNode->info));
    printf("Enter message:");
    scanf("%s",newNode->message);
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
```

Count Number of Nodes in the Tree

```
int countNodes(BTREE *treeRoot, int sum){
    if (treeRoot == NULL)
       return sum;
    sum++;
    sum = countNodes(treeRoot->left,sum);
    sum = countNodes(treeRoot->right,sum);
    return sum;
}
```

main() function

```
main(){
    BTREE *n , *a;
    int i, amount, choice;
    while (1){
       printf("    1. Insert a node \n
                   2. Delete a node \n
                   3. List nodes \n
                   4. Search for a node \n
                   5. Count number of nodes \n
                   6. Exit \n
                   Enter your choice: ");
       scanf("%d",&choice);
```

main() function (cont.)

```
if (choice == 1){
    n = readNode();
    insert(root,n);
}
else if (choice == 2){
    printf("Enter node to be deleted…");
    scanf("%d",&i);
    a = delete(root,i);
    if (a != NULL)
       printf("Deleted\n");
    else
       printf("Item to be deleted
                          is not found\n");
}
```

main() function (cont.)

```
else if (choice == 3){
    if (root != NULL){
       traverse(root);
    else
       printf("Tree is empty\n");
}
else if (choice == 4){
    printf("Enter node to be searched for…");
    scanf("%d",&i);
    a = search(root,i);
    if (a == NULL)
       printf("Item is not found\n");
    else
       printf("%d %s \n", a->info, a->message);
}
```

main() function (cont.)

```
    else if (choice == 5){
       count = countNodes(root,0);
       printf("Number of nodes
              in the tree = %d \n", count);
    }
    else
       break;
  }
}
```