DATA STRUCTURES USING "C"

DATA STRUCTURES USING "C"

LECTURE NOTES

Unit-2

Prepared by

DIGICODERS TECHNOLOGIES

Department of Computer Science and Engineering ,

UNIT-2

Lecture-11

Memory Allocation-

Whenever a new node is created, memory is allocated by the system. This memory is taken from list of those memory locations which are free i.e. not allocated. This list is called AVAIL List. Similarly, whenever a node is deleted, the deleted space becomes reusable and is added to the list of unused space i.e. to AVAIL List. This unused space can be used in future for memory allocation.

Memory allocation is of two types-

1. Static Memory Allocation
2. Dynamic Memory Allocation

1. **Static Memory Allocation:**

   When memory is allocated during compilation time, it is called 'Static Memory Allocation'. This memory is fixed and cannot be increased or decreased after allocation. If more memory is allocated than requirement, then memory is wasted. If less memory is allocated than requirement, then program will not run successfully. So exact memory requirements must be known in advance.

2. **Dynamic Memory Allocation:**

   When memory is allocated during run/execution time, it is called 'Dynamic Memory Allocation'. This memory is not fixed and is allocated according to our requirements. Thus in it there is no wastage of memory. So there is no need to know exact memory requirements in advance.

Garbage Collection-

Whenever a node is deleted, some memory space becomes reusable. This memory space should be available for future use. One way to do this is to immediately insert the free space into availability list. But this method may be time consuming for the operating system. So another method is used which is called ‘Garbage Collection’. This method is described below: In this method the OS collects the deleted space time to time onto the availability list. This process happens in two steps. In first step, the OS goes through all the lists and tags all those cells which are currently being used. In the second step, the OS goes through all the lists again and collects untagged space and adds this collected space to availability list. The garbage collection may occur when small amount of free space is left in the system or no free space is left in the system or when CPU is idle and has time to do the garbage collection.

Compaction

One preferable solution to garbage collection is compaction. The process of moving all marked nodes to one end of memory and all available memory to other end is called compaction. Algorithm which performs compaction is called compacting algorithm.

Lecture-12

Infix to Postfix Conversion

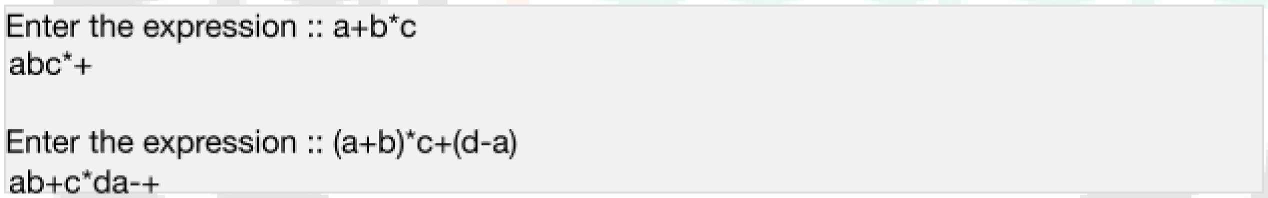
```
#include<stdio.h
> char stack[20];
int top = -1;
void push(char x)
{
    stack[++top] = x;
}

char pop()
{
    if(top == -1)
        return -1;
    else
        return stack[top--];
}

int priority(char x)
{
    if(x == '(')
        return 0;
    if(x == '+' || x == '-')
        return 1;
    if(x == '*' || x == '/')
        return 2;
}

main()
{
    char exp[20];
    char *e, x;
    printf("Enter the expression :: ");
    scanf("%s",exp);
    e = exp;
    while(*e != '\0')
    {
        if(isalnum(*e))
            printf("%c",*e);
        else if(*e == '(')
            push(*e);
        else if(*e == ')')
        {
            while((x = pop()) != '(')
                printf("%c", x);
        }
        else
        {
            while(priority(stack[top]) >= priority(*e))
                printf("%c",pop());
            push(*e);
        }
        e++
        ;
    }
    while(top != -1)
    {
        printf("%c",pop());
    }
}
```

OUTPUT:



```
Enter the expression :: a+b*c
abc*+

Enter the expression :: (a+b)*c+(d-a)
ab+c*da-+
```

Evaluate POSTFIX Expression Using Stack

```
#include<stdio.h
> int stack[20];
int top = -1;
 void push(int
 x)
{
        stack[++top] = x;
}

int pop()
{
        return stack[top--];
}

int main()
{
        char exp[20];
        char *e;
        int n1,n2,n3,num;
        printf("Enter the expression :: ");
        scanf("%s",exp);
        e = exp;
        while(*e != '\0')
        {
```

```
        if(isdigit(*e))
        {
            num = *e - 48;
            push(num);
        }
        else
        {
            n1 = pop();
            n2 = pop();
            switch(*e)
            {
                case '+':
                {
                    n3 = n1 + n2;
          break;
                }
                case '-':
                {
                    n3 = n2 - n1;
                    break;
                }
                case '*':
                {
                    n3 = n1 * n2;

                    break;
                }

                case '/':
                {
                    n3 = n2 / n1;
                    break;
                }
            }
            push(n3);
        }
        e++
        ;
    }
    printf("\nThe result of expression %s = %d\n\n",exp,pop());
    return 0;

}
```

Output:

```
Enter the expression :: 245+*

The result of expression 245+* = 18
```

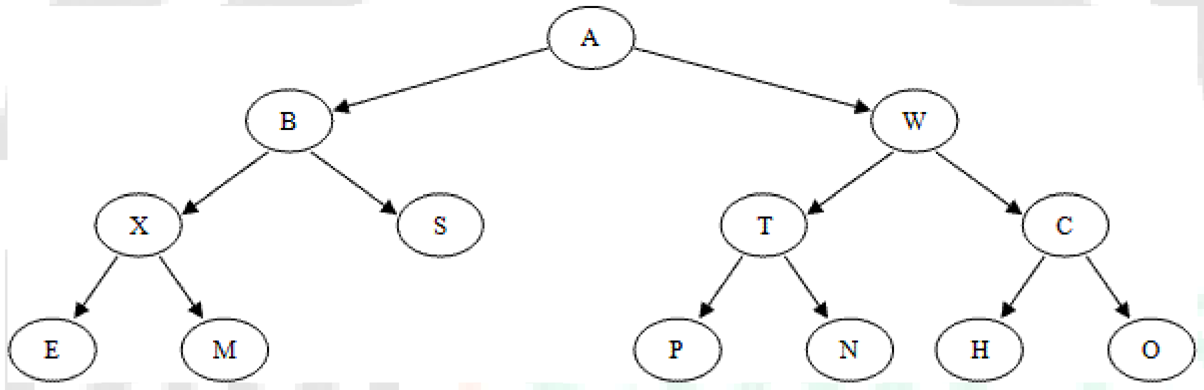
Lecture-13

Binary Tree

A *binary tree* consists of a finite set of nodes that is either empty, or consists of one specially designated node called the *root* of the binary tree, and the elements of two disjoint binary trees called the *left subtree* and *right subtree* of the root.

Note that the definition above is recursive: we have defined a binary tree in terms of binary trees. This is appropriate since recursion is an innate characteristic of tree structures.

Diagram 1: A binary tree



Binary Tree Terminology

Tree terminology is generally derived from the terminology of family trees (specifically, the type of family tree called a *lineal chart*).

- Each root is said to be the *parent* of the roots of its subtrees.
- Two nodes with the same parent are said to be *siblings*; they are the *children* of their parent.
- The root node has no parent.
- A great deal of tree processing takes advantage of the relationship between a parent and its children, and we commonly say a *directed edge* (or simply an *edge*) extends from a parent to its children. Thus edges connect a root with the roots of each subtree. An *undirected edge* extends in both directions between a parent and a child.

- *Grandparent* and *grandchild* relations can be defined in a similar manner; we could also extend this terminology further if we wished (designating nodes as cousins, as an uncle or aunt, etc.).

Other Tree Terms

- The number of subtrees of a node is called the *degree* of the node. In a binary tree, all nodes have degree 0, 1, or 2.
- A node of degree zero is called a *terminal node* or *leaf node*.
- A non-leaf node is often called a *branch node*.
- The *degree of a tree* is the maximum degree of a node in the tree. A binary tree is degree 2.
- A *directed path* from node n_1 to n_k is defined as a sequence of nodes n_1, n_2, ..., n_k such that n_i is the parent of n_i+1 for 1 <= i < k. An *undirected path* is a similar sequence of undirected edges. The length of this path is the number of edges on the path, namely k – 1 (i.e., the number of nodes – 1). There is a path of length zero from every node to itself. Notice that in a binary tree there is exactly one path from the root to each node.
- The *level* or *depth* of a node with respect to a tree is defined recursively: the level of the root is zero; and the level of any other node is one higher than that of its parent. Or to put it another way, the level or depth of a node n_i is the length of the unique path from the root to n_i.
- The *height* of n_i is the length of the longest path from n_i to a leaf. Thus all leaves in the tree are at height 0.
- The *height of a tree* is equal to the height of the root. The *depth of a tree* is equal to the level or depth of the deepest leaf; this is always equal to the height of the tree.
- If there is a directed path from n_1 to n_2, then n_1 is an ancestor of n_2 and n_2 is a descendant of n_1.

<u>**Lecture-14**</u>

Special Forms of Binary Trees

There are a few special forms of binary tree worth mentioning.

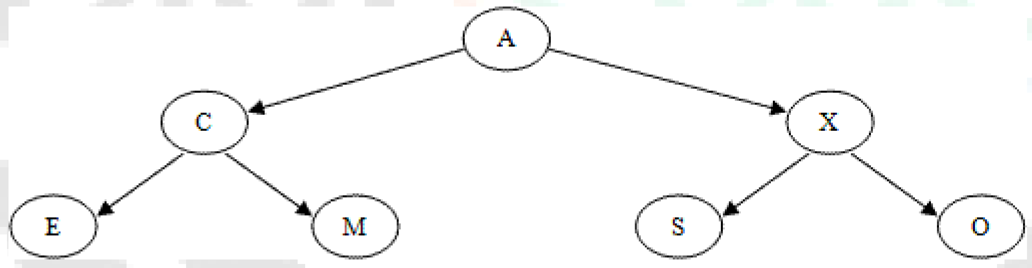
If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is termed a *strictly binary tree*. Or, to put it another way, all of the nodes in a strictly binary tree are of degree zero or two, never degree one. A strictly binary tree with *N* leaves always contains 2*N* – 1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

Some texts call this a "full" binary tree.

A *complete binary tree* of depth *d* is the strictly binary tree all of whose leaves are at level *d*.

The total number of nodes in a complete binary tree of depth *d* equals $2^{d+1} - 1$. Since all leaves in such a tree are at level *d*, the tree contains 2^d leaves and, therefore, $2^d - 1$ internal nodes.

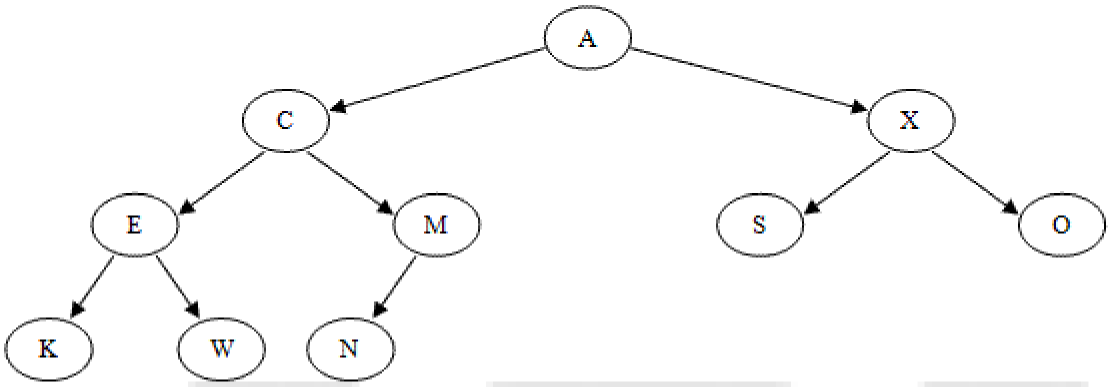
Diagram 2: A complete binary tree



A binary tree of depth *d* is an *almost complete binary tree* if:

- Each leaf in the tree is either at level *d* or at level *d* – 1.
- For any node n_d in the tree with a right descendant at level *d*, all the left descendants of n_d that are leaves are also at level *d*.

Diagram 3: An almost complete binary tree



An almost complete strictly binary tree with *N* leaves has 2*N* – 1 nodes (as does any other strictly binary tree). An almost complete binary tree with *N* leaves that is not strictly binary has 2*N* nodes. There are two distinct almost complete binary trees with *N* leaves, one of which is strictly binary and one of which is not.

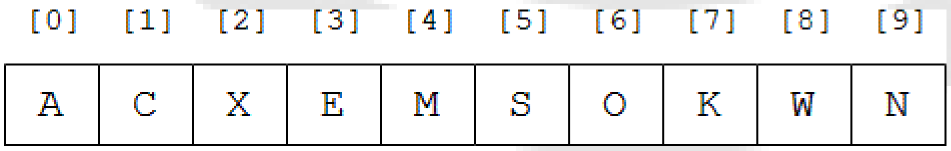
There is only a single almost complete binary tree with *N* nodes. This tree is strictly binary if and only if *N* is odd.

Representing Binary Trees in Memory

Array Representation

For a complete or almost complete binary tree, storing the binary tree as an array may be a good choice.

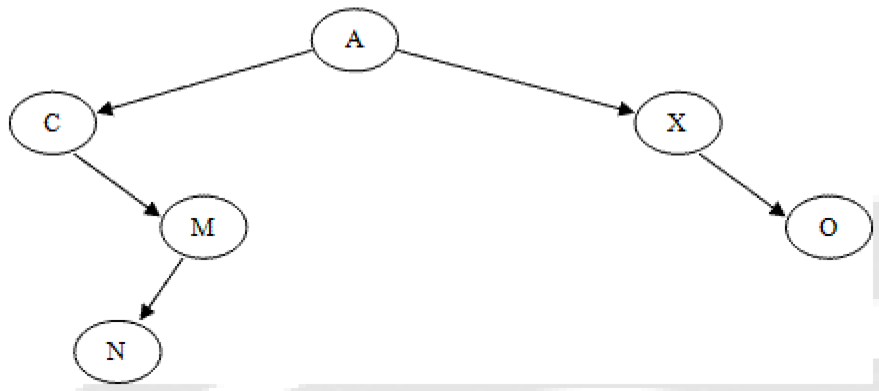
One way to do this is to store the root of the tree in the first element of the array. Then, for each node in the tree that is stored at subscript *k*, the node's left child can be stored at subscript 2*k*+1 and the right child can be stored at subscript 2*k*+2. For example, the almost complete binary tree shown in **Diagram 2** can be stored in an array like so:



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	C	X	E	M	S	O	K	W	N

However, if this scheme is used to store a binary tree that is not complete or almost complete, we can end up with a great deal of wasted space in the array.

For example, the following binary tree

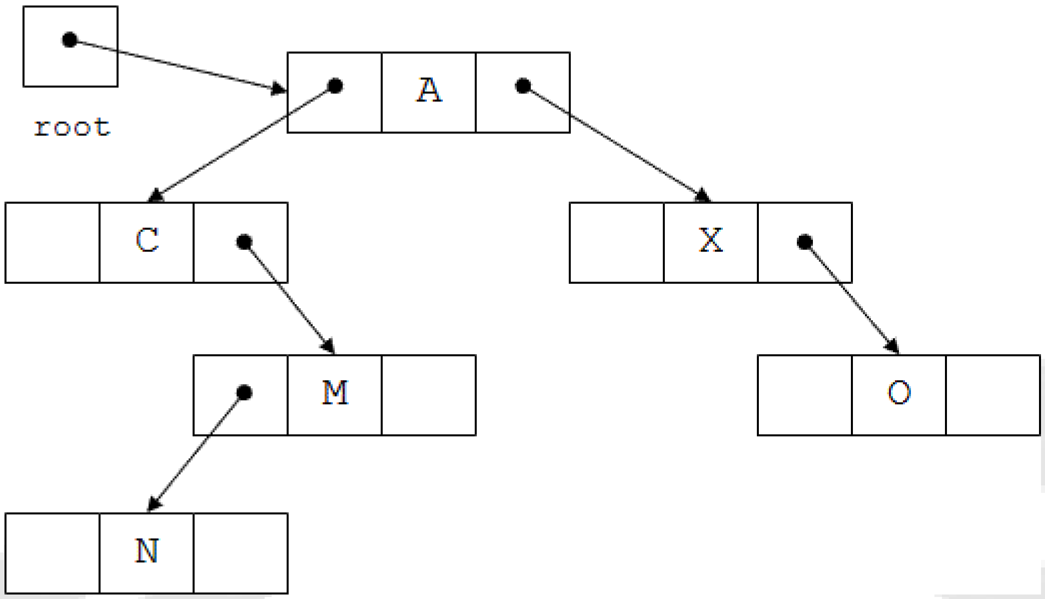


would be stored using this techinque like so:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	C	X		M		O			N

Linked Representation

If a binary tree is not complete or almost complete, a better choice for storing it is to use a linked representation similar to the linked list structures covered earlier in the semester:



Each tree node has two pointers (usually named left and right). The tree class has a pointer to the root node of the tree (labeled root in the diagram above).

Any pointer in the tree structure that does not point to a node will normally contain the value NULL. A linked tree with *N* nodes will always contain *N* + 1 null links.

<u>**Lecture-15**</u>

Tree Traversal:

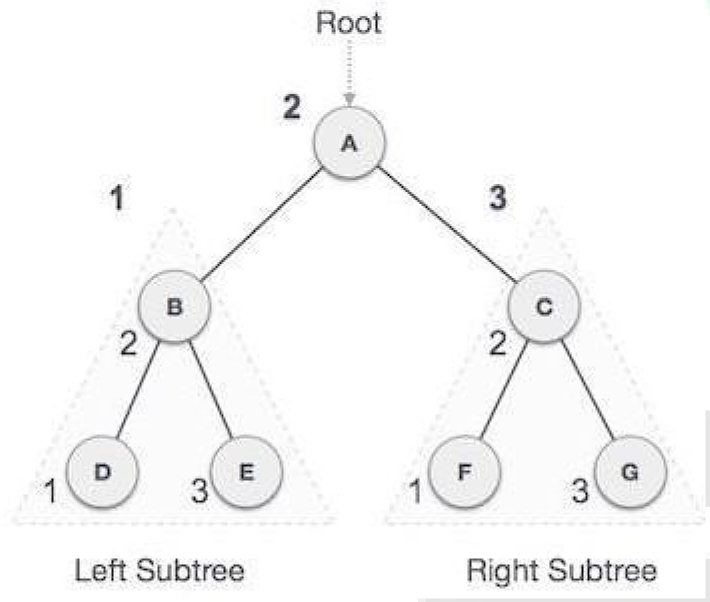
Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree –

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself. If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be –

D → B → E → A → F → C → G

Algorithm

Until all nodes are traversed –

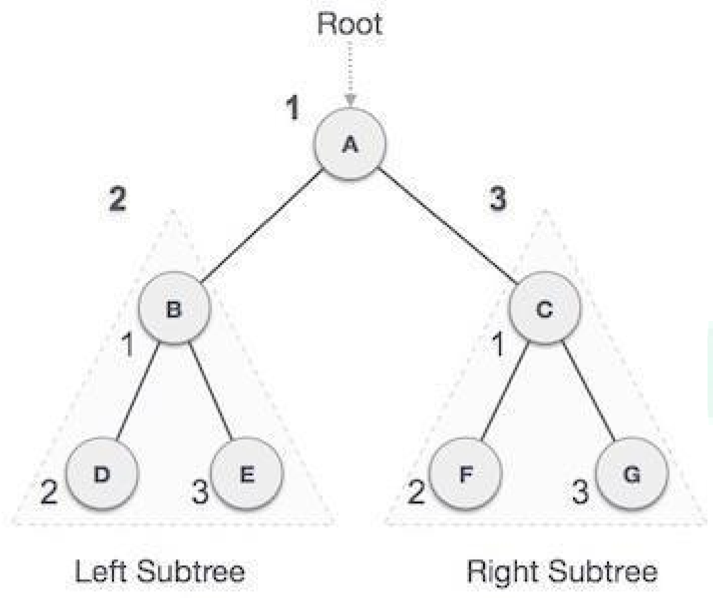
Step 1 – Recursively traverse left subtree.

Step 2 – Visit root node.

Step 3 – Recursively traverse right subtree.

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be –

A → B → D → E → C → F → G

Algorithm

Until all nodes are traversed –

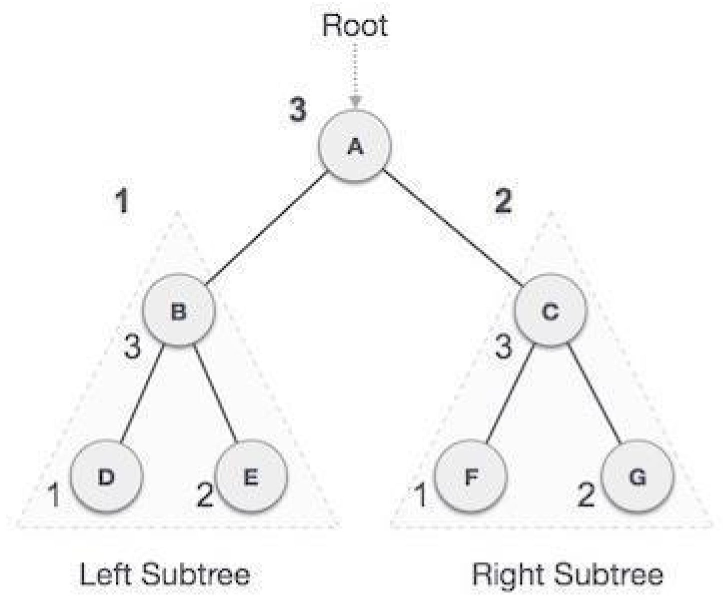
Step 1 – Visit root node.

Step 2 – Recursively traverse left subtree.

Step 3 – Recursively traverse right subtree.

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following Post-order traversal, we first visit the left subtree **B. B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be –

D → E → B → F → G → C → A

Algorithm

```
Until all nodes are traversed –
Step 1 – Recursively traverse left subtree.
 Step 2 – Recursively traverse right subtree.
 Step 3 – Visit root node.
```

Lecture-16

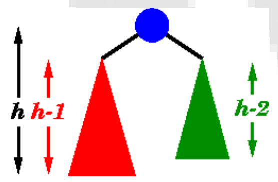
AVL Trees

An **AVL tree** is another balanced binary search tree. Named after their inventors, **A**delson-**V**elskii and **L**andis, they were the first dynamically balanced trees to be proposed. Like red-black trees, they are not perfectly balanced, but pairs of sub- trees differ in height by at most 1, maintaining an ***O*(log*n*)** search time. Addition and deletion operations also take ***O*(log*n*)** time.

Definition of an AVL tree

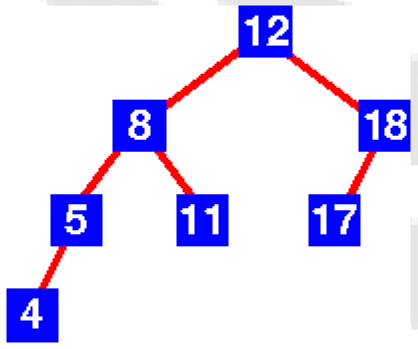
An AVL tree is a binary search tree which has the following properties:

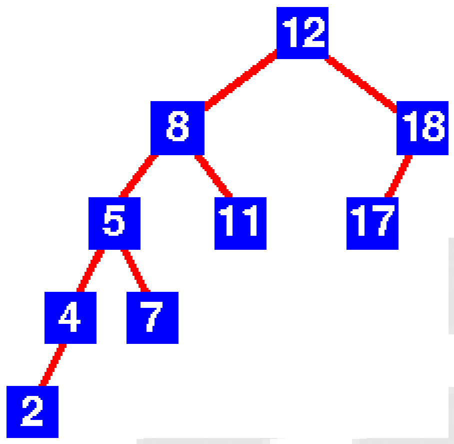
1. The sub-trees of every node differ in height by at most one.
2. Every sub-tree is an AVL tree.



Balance requirement for an AVL tree: the left and right sub-trees differ by at most 1 in height.

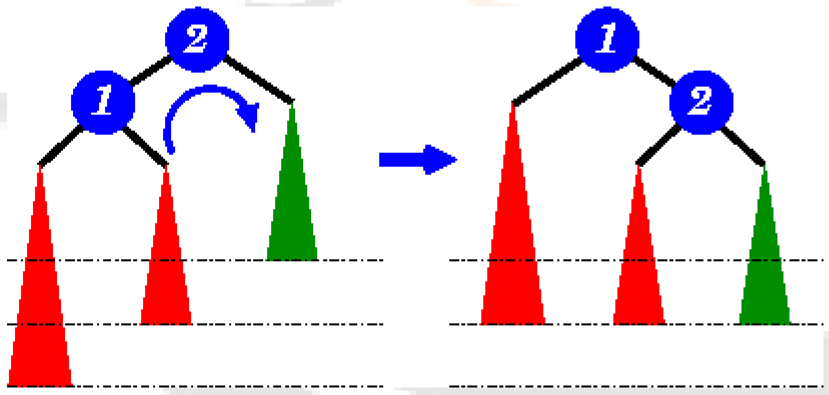
You need to be careful with this definition: it permits some apparently unbalanced trees! For example, here are some trees:

Tree	**AVL tree?**
	Yes Examination shows that *each* left sub-tree has a height 1 greater than each right sub- tree.

	No Sub-tree with root 8 has height 4 and sub-tree with root 18 has height 2

Insertion

As with the red-black tree, insertion is somewhat complex and involves a number of cases. Implementations of AVL tree insertion may be found in many textbooks: they rely on adding an extra attribute, the **balance factor** to each node. This factor indicates whether the tree is *left-heavy* (the height of the left sub-tree is 1 greater than the right sub-tree), *balanced* (both sub-trees are the same height) or *right-heavy*(the height of the right sub-tree is 1 greater than the left sub-tree). If the balance would be destroyed by an insertion, a rotation is performed to correct the balance.



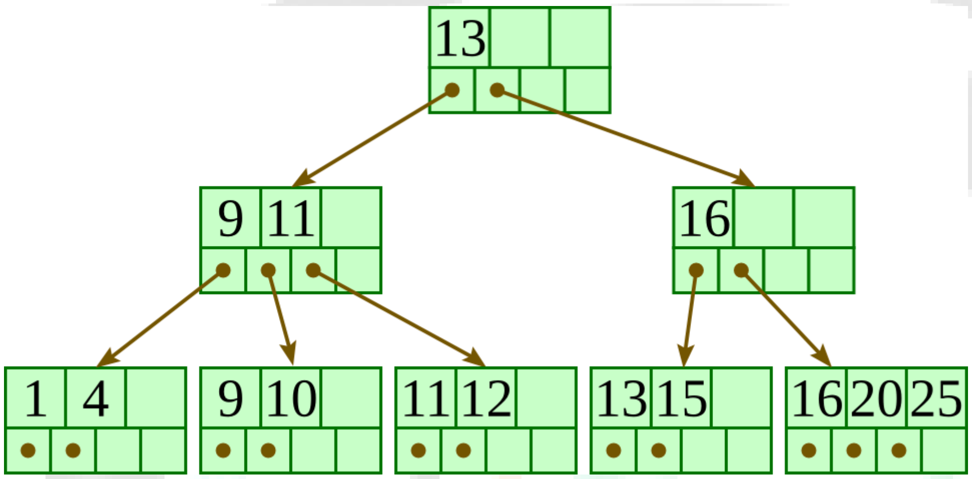
A new item has been added to the left subtree of node 1, causing its height to become 2 greater than 2's right sub-tree (shown in green). A right-rotation is performed to correct the imbalance.

Lecture-17

B+-tree

In B+-tree, each node stores up to *d* references to children and up to *d* – 1 keys. Each reference is considered "between" two of the node's keys; it references the root of a subtree for which all values are between these two keys.

Here is a fairly small tree using 4 as our value for *d*.



A B+-tree requires that each leaf be the same distance from the root, as in this picture, where searching for any of the 11 values (all listed on the bottom level) will involve loading three nodes from the disk (the root block, a second-level block, and a leaf).

In practice, *d* will be larger — as large, in fact, as it takes to fill a disk block. Suppose a block is 4KB, our keys are 4-byte integers, and each reference is a 6-byte file offset. Then we'd choose *d* to be the largest value so that $4\ (d - 1) + 6\ d \le 4096$; solving this inequality for *d*, we end up with $d \le 410$, so we'd use 410 for *d*. As you can see, *d* can be large.

A B+-tree maintains the following invariants:

- Every node has one more references than it has keys.
- All leaves are at the same distance from the root.
- For every non-leaf node *N* with *k* being the number of keys in *N*: all keys in the first child's subtree are less than *N*'s first key; and all keys in the *i*th child's subtree ($2 \le i \le k$) are between the (i – 1)th key of *n* and the *i*th key of *n*.
- The root has at least two children.
- Every non-leaf, non-root node has at least *floor*(*d* / 2) children.

- Each leaf contains at least *floor*(*d* / 2) keys.
- Every key from the table appears in a leaf, in left-to-right sorted order.

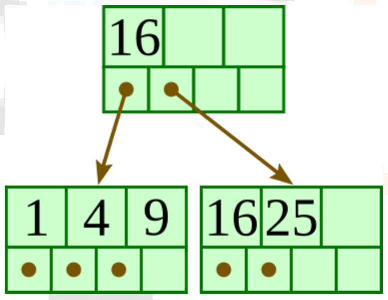
In our examples, we'll continue to use 4 for *d*. Looking at our invariants, this requires that each leaf have at least two keys, and each internal node to have at least two children (and thus at least one key).

2. Insertion algorithm

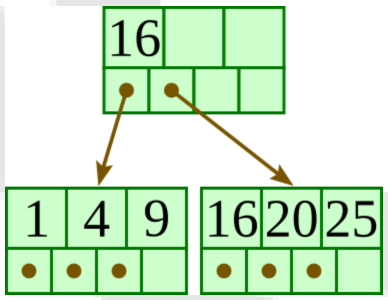
Descend to the leaf where the key fits.

1. If the node has an empty space, insert the key/reference pair into the node.
2. If the node is already full, split it into two nodes, distributing the keys evenly between the two nodes. If the node is a leaf, take a copy of the minimum value in the second of these two nodes and repeat this insertion algorithm to insert it into the parent node. If the node is a non-leaf, exclude the middle value during the split and repeat this insertion algorithm to insert this excluded value into the parent node.

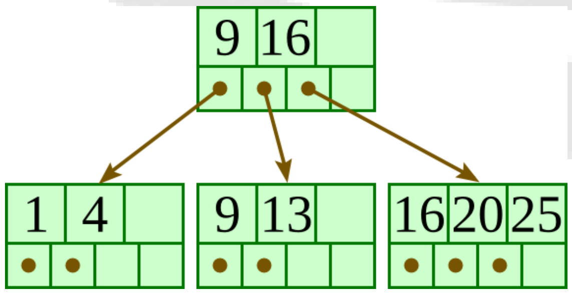
Initial:



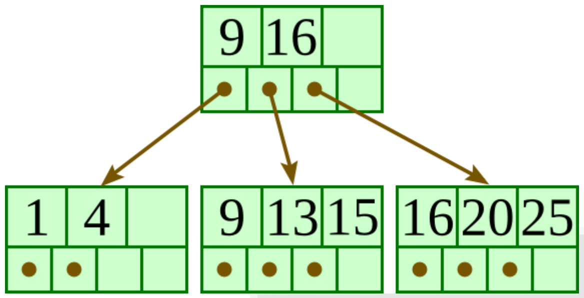
Insert 20:



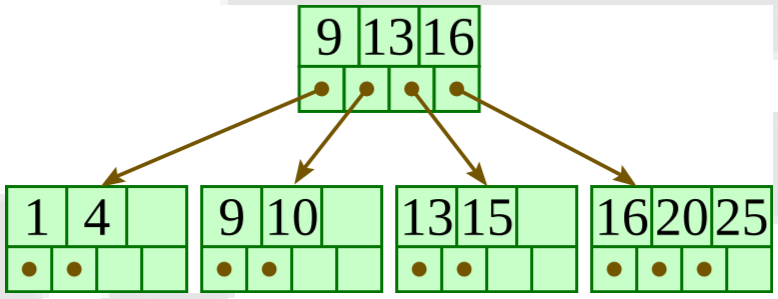
Insert 13:



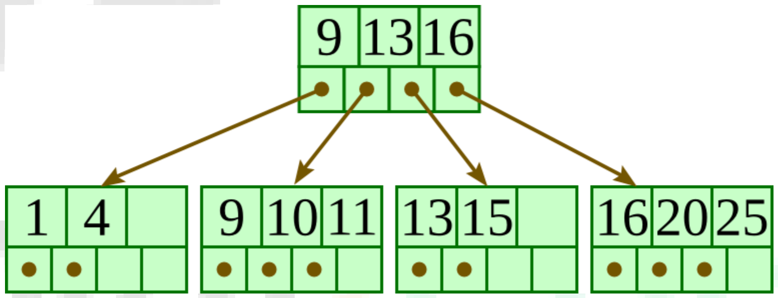
Insert 15:



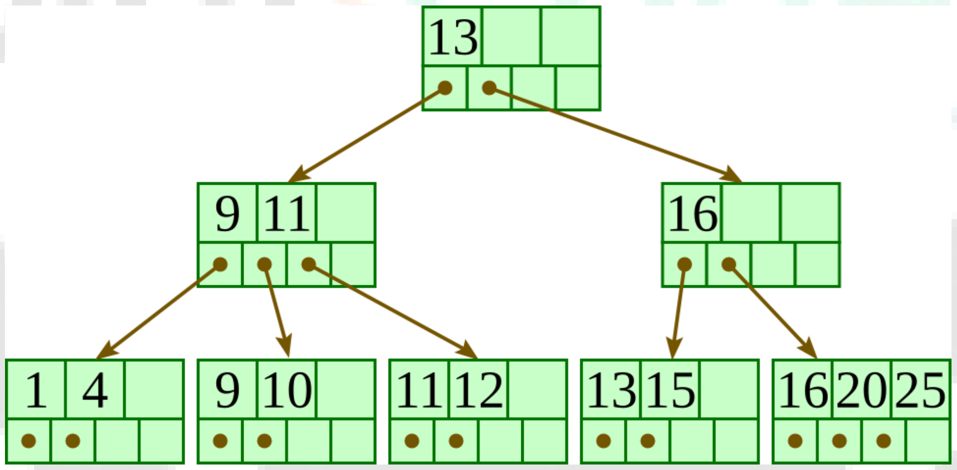
Insert 10:



Insert 11:



Insert 12:



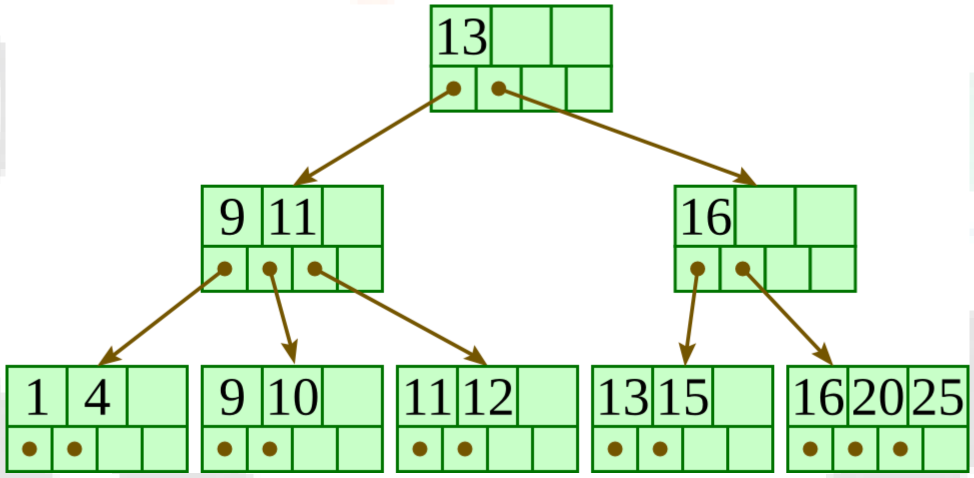
3. Deletion algorithm

Descend to the leaf where the key exists.

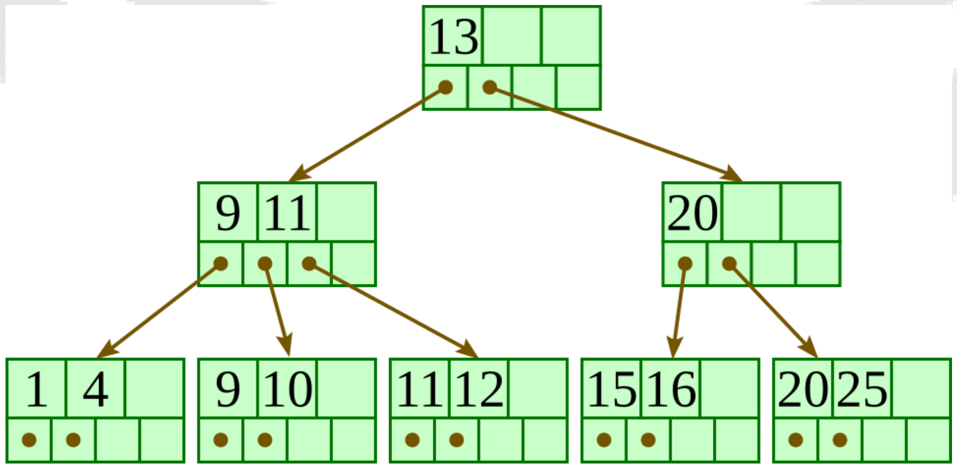
1. Remove the required key and associated reference from the node.
2. If the node still has enough keys and references to satisfy the invariants, stop.

3. If the node has too few keys to satisfy the invariants, but its next oldest or next youngest sibling at the same level has more than necessary, distribute the keys between this node and the neighbor. Repair the keys in the level above to represent that these nodes now have a different “split point” between them; this involves simply changing a key in the levels above, without deletion or insertion.
4. If the node has too few keys to satisfy the invariant, and the next oldest or next youngest sibling is at the minimum for the invariant, then merge the node with its sibling; if the node is a non-leaf, we will need to incorporate the “split key” from the parent into our merging. In either case, we will need to repeat the removal algorithm on the parent node to remove the “split key” that previously separated these merged nodes — unless the parent is the root and we are removing the final key from the root, in which case the merged node becomes the new root (and the tree has become one level shorter than before).

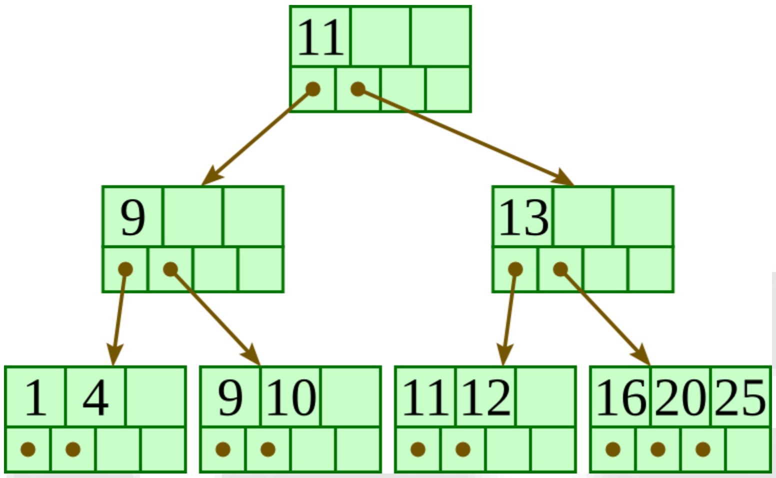
Initial:



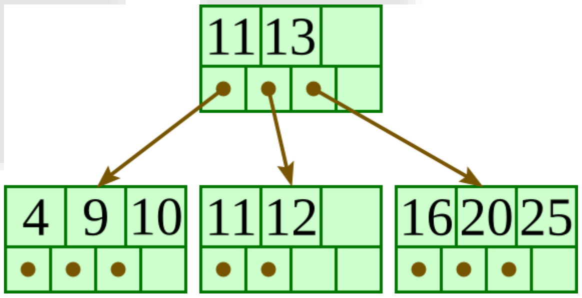
Delete 13:



Delete 15:

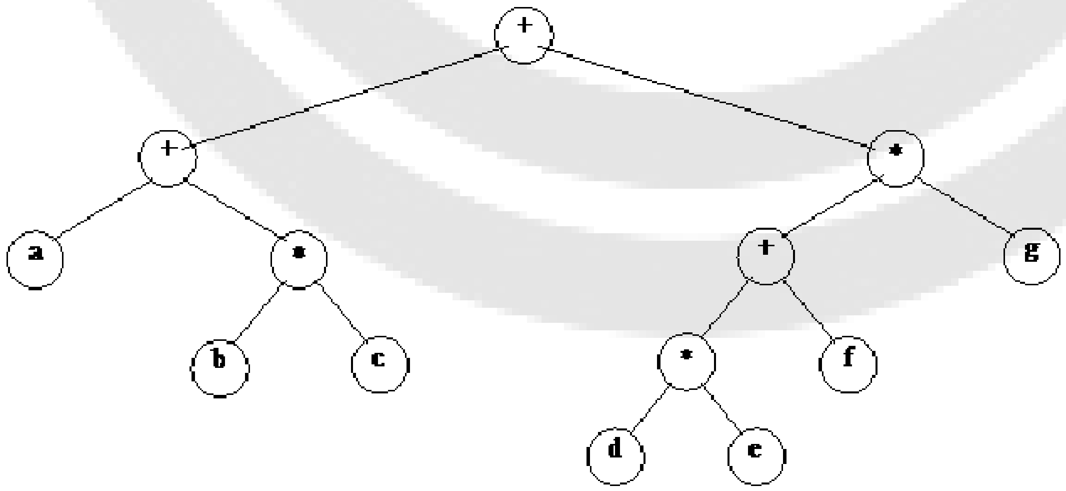


Delete 1:



Expression Trees:

Trees are used in many other ways in the computer science. Compilers and database are two major examples in this regard. In case of compilers, when the languages are translated into machine language, tree-like structures are used. We have also seen an example of expression tree comprising the mathematical expression. Let's have more discussion on the expression trees. We will see what are the benefits of expression trees and how can we build an expression tree. Following is the figure of an expression tree.



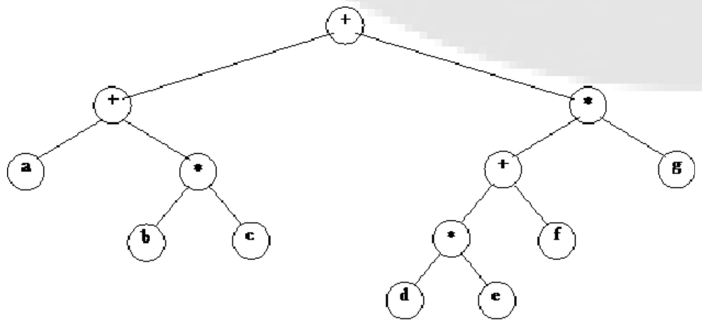
In the above tree, the expression on the left side is a + b * c while on the right side, we have d * e + f * g. If you look at the figure, it becomes evident that the inner nodes contain operators while leaf nodes have operands. We know that there are two types of nodes in the tree i.e. inner nodes and leaf nodes. The leaf nodes are such nodes which have left and right subtrees as null. You will find these at the bottom level of the tree. The leaf nodes are connected with the inner nodes. So in trees, we have some inner nodes and some leaf nodes.

In the above diagram, all the inner nodes (the nodes which have either left or right child or both) have operators. In this case, we have + or * as operators. Whereas leaf nodes contain operands only i.e. a, b, c, d, e, f, g. This tree is binary as the operators are binary. We have discussed the evaluation of postfix and infix expressions and have seen that the binary operators need two operands. In the infix expressions, one operand is on the left side of the operator and the other is on the right side. Suppose, if we have

+ operator, it will be written as 2 + 4. However, in case of multiplication, we will write as 5*6. We may have unary operators like negation (-) or in Boolean expression we have NOT. In this example, there are all the binary operators. Therefore, this tree is a binary tree. This is not the Binary Search Tree. In BST, the values on the left side of the nodes are smaller and the values on the right side are greater than the node. Therefore, this is not a BST. Here we have an expression tree with no sorting process involved.

This is not necessary that expression tree is always binary tree. Suppose we have a unary operator like negation. In this case, we have a node which has (-) in it and there is only one leaf node under it. It means just negate that operand.

Let's talk about the traversal of the expression tree. The inorder traversal may be executed here.



Inorder traversal yields: a+b*c+d*e+f*g

Lecture-18

Binary Search Tree (BST)

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties –

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than to its parent node's key.

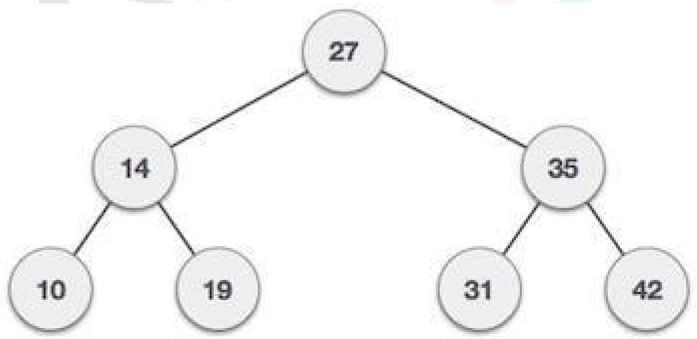
Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as –

```
left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)
```

Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST –



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Basic Operations

Following are the basic operations of a tree –

- **Search** – Searches an element in a tree.
- **Insert** – Inserts an element in a tree.
- **Pre-order Traversal** – Traverses a tree in a pre-order manner.
- **In-order Traversal** – Traverses a tree in an in-order manner.
- **Post-order Traversal** – Traverses a tree in a post-order

manner. Node

Define a node having some data, references to its left and right child nodes.

```
struct node
   { int data;
struct node *leftChild; struct
   node *rightChild;
};
```

Search Operation

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```
struct node* search(int data){
   struct node *current = root;
   printf("Visiting elements: ");

while(current->data != data){

      if(current != NULL) {
         printf("%d
         ",current->data);

         //go to left tree
         if(current->data > data){
current = current->leftChild;
} //else go to right tree else {
current = current->rightChild;
         }


//not found
```

```
            if(current == NULL){
               return NULL;
            }
         }
      }

return current;
 }
```

Insert Operation

Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm

```
void insert(int data) {
struct node *tempNode = (struct node*) malloc(sizeof(struct node)); struct
    node *current;
struct node *parent;

tempNode->data = data;
    tempNode->leftChild = NULL;
    tempNode->rightChild =
    NULL;

//if tree is empty if(root
    == NULL) {
root = tempNode;
} else {
current = root; parent
      = NULL;
while(1) {
parent = current;

          //go to left of the tree
          if(data < parent->data) {
current = current->leftChild;
//insert to the left

if(current == NULL) {
parent->leftChild = tempNode; return;
             }
} //go to right of the tree else {
current = current->rightChild;

             //insert to the right
             if(current == NULL) {
parent->rightChild = tempNode; return;
             }
          }
       }
    }
 }
```