UNIT-II

Trees

Definition

A tree T is a finite set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following

1. if T is not empty, T has a special tree called the root that has no parent

2. each node v of T different than the root has a unique parent node w; each node with parent w is a child of w

Recursive definition

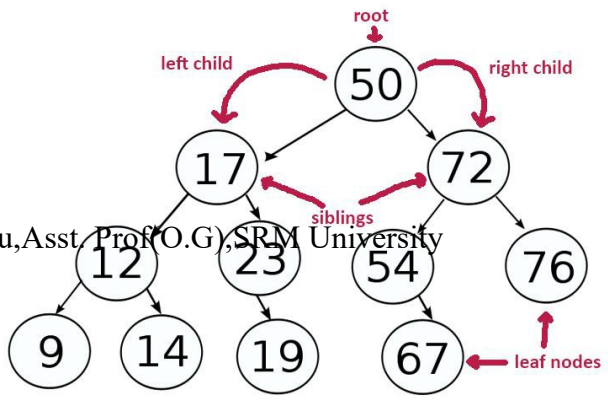
• **T is either** empty or consists of a node r (the root) and a possibly empty set of trees whose roots are the children of r

Terminologies in tree

- **Root –** The top node in a tree.
- **Parent –** The converse notion of a *child*.
- **Siblings –** Nodes with the same parent.
- **Descendant –** a node reachable by repeated proceeding from parent to child. descendant of 50 is 72 and 76
- **Ancestor –** a node reachable by repeated proceeding from child to parent. ancestor

of 76 is 72 and 50

- **Leaf –** a node with no children.
- **Internal node –** a node with at least one child.
- **External node –** a node with no children.
- **Degree –** number of sub trees of a node.
- **Edge –** connection between one node to another.
- **Path –** a sequence of nodes and edges connecting a node with a descendant.
- **Level –** The level of a node is defined by 1 + (the number of connections between the node and the root).
- **Height of tree –**The height of a tree is the number of edges on the longest downward path between the root and a leaf.
- **Height of node –**The height of a node is the number of edges on the longest downward path between that node and a leaf.
- **Depth –**The depth of a node is the number of edges from the node to the tree's root node.



Application of Tree

1. Trees help to store information that naturally to form a hierarchy. For example, the file system on a computer

2. If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of O (Log n) for search.

3. We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists).Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of O(Log n) for insertion/deletion.

4. Manipulate hierarchical data.

5. Make information easy to search.
6. Manipulate sorted lists of data
7. As a work flow for composting digital images for visual effects.
8. Router algorithms

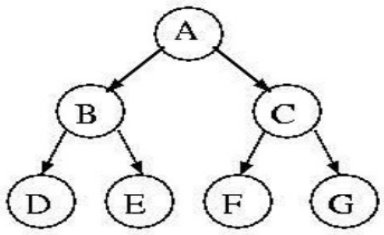
Binary tree

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

Definition: A binary tree is either empty or consists of a node called the root together with two binary trees called the left sub tree and the right sub tree.

If h = height of a binary tree,

1. Maximum number of nodes = $2^{h+1} - 1$
2. A binary tree with height h and $2^{h+1} - 1$ node (or 2^h leaves) is called a full binary tree.
3. Maximum no. of leaves= 2^h



Full Binary Tree

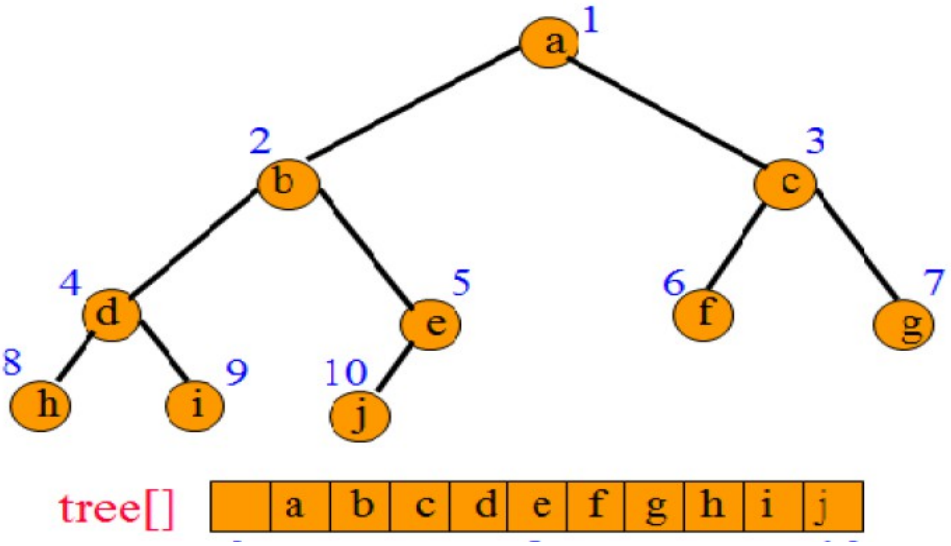
A full binary tree (sometimes proper binary tree or 2-tree) is a tree in which every node other than the leaves has two children.

Complete Binary Tree

A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

Binary treeArray Implementation:

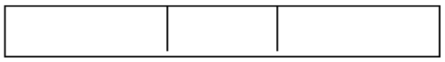
- Use array to store the data.
- Start storing from index 1, not 0.
- For any given node at position i:
- Its **Left Child** is at ***[2*i]*** if available.
- Its **Right Child** is at ***[2*i+1]*** if available.
- **Its Parent Nodes** at***[i/2]***if available.



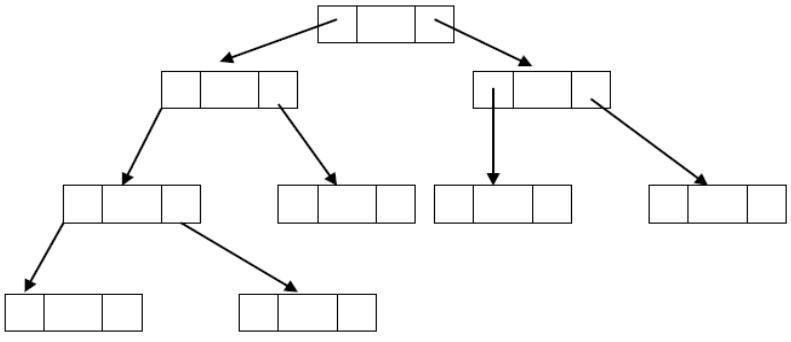
Binary Tree using Link Representation

The problems of sequential representation can be easily overcome through the use of a linked representation.

Each node will have three fields LCHILD, DATA and RCHILD as represented below



LCHILD DATA RCHILD

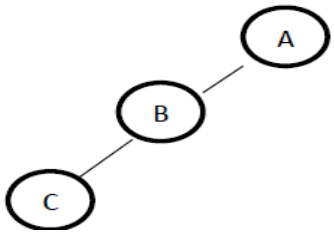


Using the linked implementation we may declare,

```
Struct treenode
{
int data;
structtreenode *leftchild;
structtreenode *rightchild;
}*T;
```

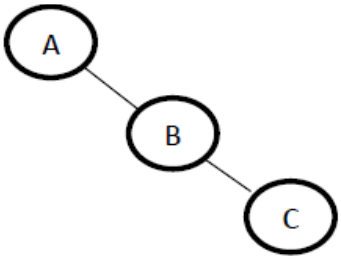
Left Skewed Binary tree :

A binary tree which has only left child is called left skewed binary tree.



Right Skewed Binary tree :

A Binary tree which has only right child is called right skewed binary tree.



TREE TRAVERSALS

There are basically three ways of binary tree traversals.

1. Inorder --- (left child,root,right child)

2. Preorder --- (root,left child,right child)

3. Postorder --- (left child,right child,root)

Inorder Traversal

Steps :

Traverse left subtree in inorder

Process root node

Traverse right subtree in inorder

Pseudo code:

```
void inorder(struct node * node)
{
if(node!=NULL)
{
in order(node->left)
```

```
print(node->data)
inorder(node->right)
}
}
```

Preorder Traversal

Steps :

Process root node

Traverse left subtree in preorder

Traverse right subtree in preorder

```
void  preorder(struct node * node)
{
if(node!=NULL)
{
print(node->data)
preorder(node->left)
preorder(node->right)
}
}
```

Postorder Traversal

Steps :

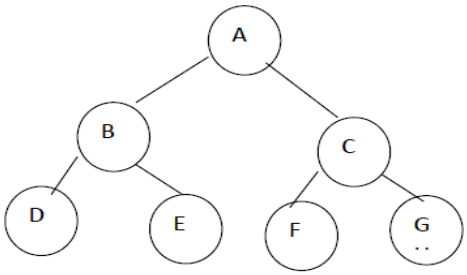
Traverse left subtree in postorder

Traverse right subtree in postorder

process root node

```
void  postorder(struct node * node)
{
if(node!=NULL)
{
preorder(node->left)
preorder(node->right)
print(node->data)
}
}
```

2.FIND THE TRAVERSAL OF THE FOLLOWING TREE



POSTORDER: DEBFGCA INORDER: DBEAFCG

PREORDER:ABDECFG

EXPRESSION TREE

Expression Tree is a binary tree in which the leaf nodes are operands and th interior nodes are operators.

CONSTRUCTING AN EXPRESSION TREE

Let us consider the postfix expression given as the input, for constructing an expression tree by performing the following steps :

1. Read one symbol at a time from the postfix expression.

2. Check whether the symbol is an operand or operator.

i. If the symbol is an operand, create a one node tree and push a pointer on to the stack.

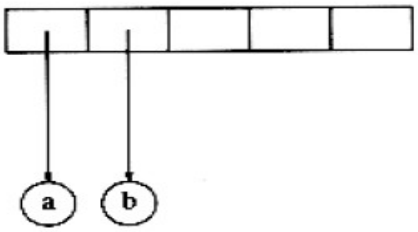
ii. If the symbol is an operator, pop two pointers from the stack namely, T1 and T2 and form a new tree with root as the operator, and T2 as the left child and T1 as the right child.

iii. A pointer to this new tree is then pushed on to the stack.

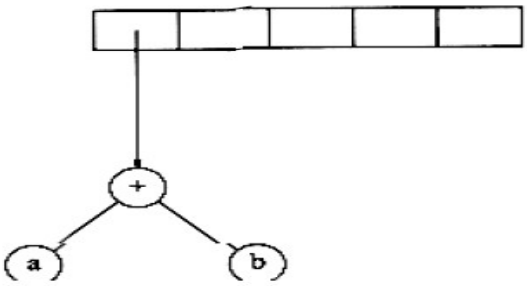
As an example, suppose the input is

a b + c d e + * *

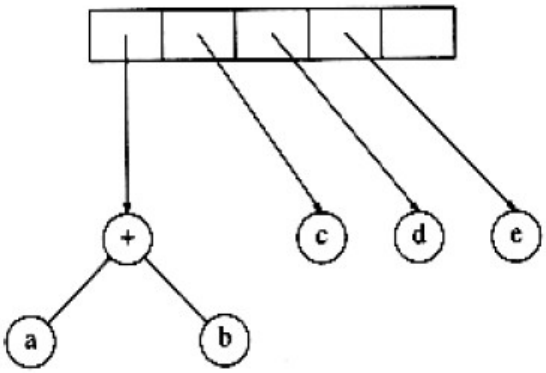
1)The first two symbols are operands, so we create one-node trees and push pointers to them onto a stack



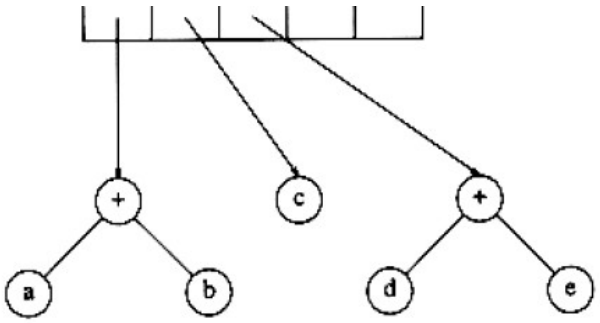
2)Next, a '+' is read, so two pointers to trees are popped, a new tree is formed, and a pointer to it is pushed onto the stack.



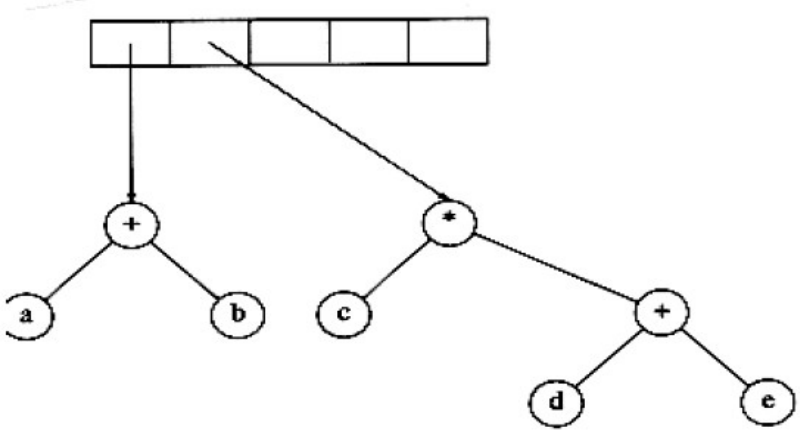
3)Next, *c*, *d*, and *e* are read, and for each a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



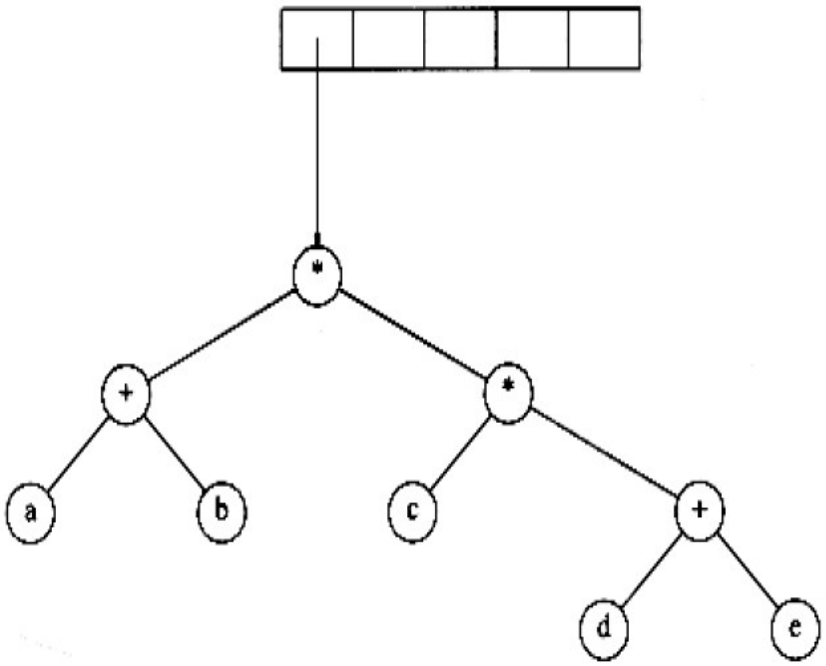
4)Now a '+' is read, so two trees are merged.



5)Continuing, a '*' is read, so we pop two tree pointers and form a new tree with a '*' as root.



6)Finally, the last symbol is read, two trees are merged, and a pointer to the final tree is left on the stack.

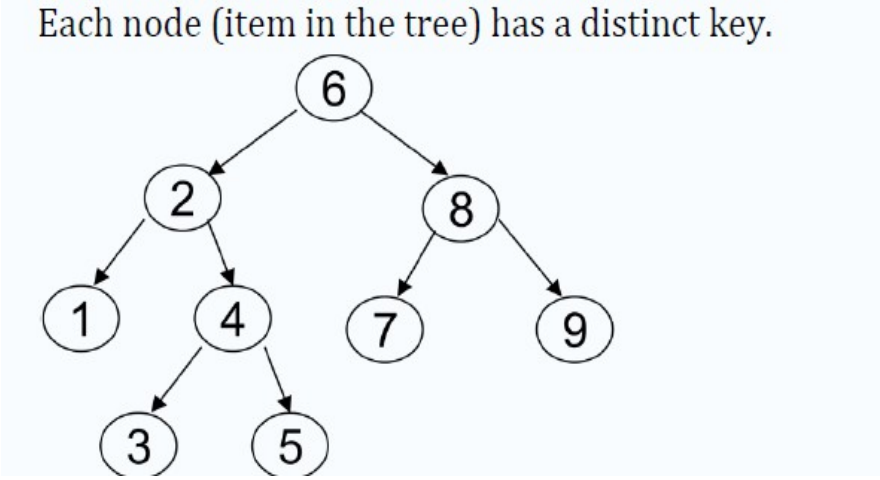


Prepa

BINARY SEARCH TREE

Binary search tree (**BST**) is a node-based binary tree data structure which has the following properties:

- The left sub-tree of a node contains only nodes with keys less than the node's key.
- The right sub-tree of a node contains only nodes with keys greater than the node's key.
- Both the left and right sub-trees must also be binary search trees.



OPERATIONS

- Operations on a binary tree require comparisons between nodes.
- These comparisons are made with calls to a comparator, which is a subroutine that computes the total order (linear order) on any two values.
- This comparator can be explicitly or implicitly defined, depending on the language in which the BST is implemented.
- The following are the operations that are being done in Binary Tree

¬ Searching.

¬ Sorting.

¬ Deletion.

¬ Insertion.

Binary search tree declaration routine

```
Struct treenode;
Typedef struct treenode *position;
Typedef struct treenode *searchtree;
Typedef int elementtype;
Structtreenode
{
Elementtype element;
Searchtree left;
Searchtree right;
};
Struct treenode
{
int element;
struct treenode *left;
struct treenode *right;
};
```

Make_null

This operation is mainly for initialization. Some programmers prefer to initialize the first element as a one-node tree, but our implementation follows the recursive definition of trees more closely. It is also a simple routine.

Routine to make an empty tree

```
SEARCH_TREE  make_null ( void )
{
return NULL;
```

}

Find

This operation generally requires returning a pointer to the node in tree *T* that has key *x*, or *NULL* if there is no such node. The structure of the tree makes this simple. If *T* is *, then we can just return* . Otherwise, if the key stored at *T* is *x*, we can return *T*. Otherwise, we make a recursive call on a sub-tree of *T*, either left or right, depending on the relationship of *x* to the key stored in *T*.

Find operation for binary search trees

```
Position find(structtreenode T, intnum)
{
While(T!=NULL)
{
if(num>T-->data)
{
T=T-->right;
if(num<T-->data)
T=T-->left;
}
else if(num< T-->data)
{
T=T-->left;
if(num>T-->data)
T=T-->right;
}
if(T-->data==num)
break;
}
```

```
return T;
}
```

Find_min and Find_max

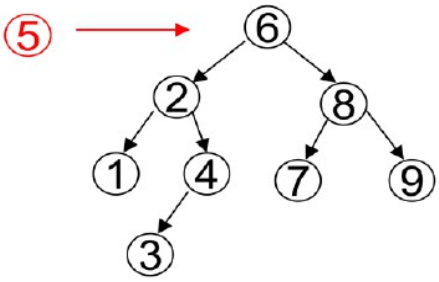
Recursive implementation of find_min & find_max for binary search trees

```
// Finding Minimum
Position findmin(search tree T)
{
if(T==NULL)
return NULL;
else if(T-->left==NULL)
return T;
else return findmin(T-->left);
}
// Finding Maximum
Position findmax(searchtree T)
{
if(T==NULL)
return NULL;
else if(T-->right==NULL)
return T;
else return findmax(T-->right);
}
```

Insertion routine

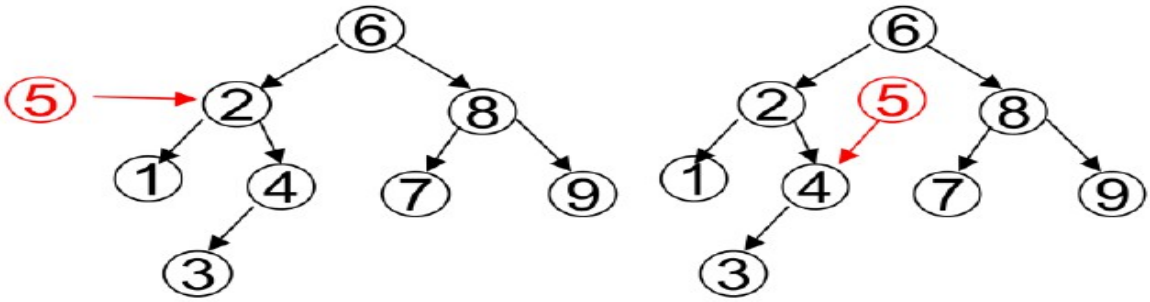
The insertion routine is conceptually simple. To insert *x* into tree *T*, proceed down the tree as you would with a *find*. If *x* is found, do nothing (or "update" something). Otherwise, insert *x* at the last spot on the path traversed.

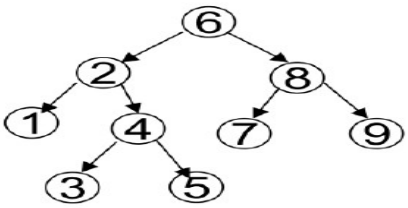
```
Searchtree insert(elementtype X, Searchtree T)
{
If(T== NULL)
{
/* create and return a one node tree*/
T=malloc(sizeof(struct treenode));
If(T==NULL)
Fatal error("Out of Space");
Else
{
T-->element=X;
T-->left=T-->right=NULL;
}
}
Else if(x<T-->element)
T-->left=insert(X,T-->left);
Else if(X>=T-->left)
T-->right=insert(X,T-->right);
Return T;
}
```



EXAMPLE Insert node 5 in given tree

STEP 1: Now 5<6 and 5>2 and 5<4 so





Thus 5 is inserted.

Delete

- As is common with many data structures, the hardest operation is deletion. Once we have found the node to be deleted, we need to consider several possibilities.
- If the node is a leaf, it can be deleted immediately. If the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node (we will draw the pointer directions explicitly for clarity).
- Notice that the deleted node is now unreferenced and can be disposed of only if a pointer to it has been saved. The complicated case deals with a node with two

children.

- The general strategy is to replace the key of this node with the smallest key of the right subtree (which is easily found) and recursively delete that node (which is now empty). Because the smallest node in the right sub tree cannot have a left child, the second *delete* is an easy one.

- To delete an element, consider the following three possibilities :

Case 1: Node to be deleted is a leaf node.

Case 2: Node with only one child.

Case 3: Node with two children.

Case 1: Node with no children | Leaf node :

1. Search the parent of the leaf node and make the link to the leaf node as NULL.
2. Release the memory of the deleted node.

Case 2: Node with only one child :

1. Search the parent of the node to be deleted.
2. Assign the link of the parent node to the child of the node to be deleted.
3. Release the memory for the deleted node.

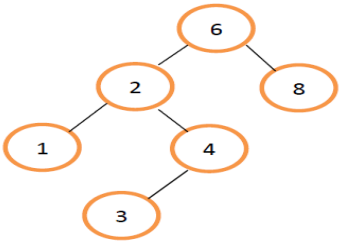
Case 3: Node with two children :

It is difficult to delete a node which has two children.

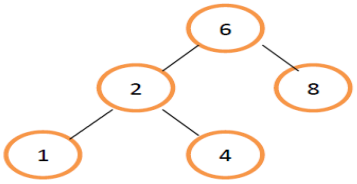
So, a general strategy has to be followed.

1. Replace the data of the node to be deleted with either the largest element from the left sub tree or the smallest element from the right sub tree.

Case 1: (EXAMPLE)



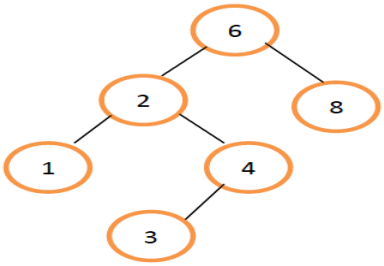
Before Deletion



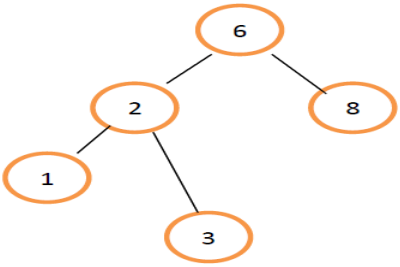
After Deletion of 3

CASE 2:(EXAMPLE)

Case 2 :



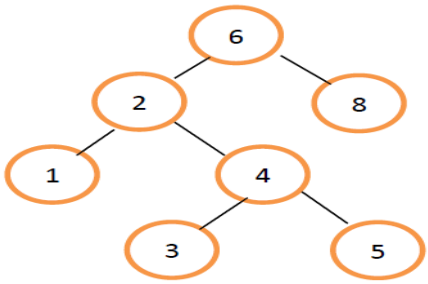
Before deletion of 4



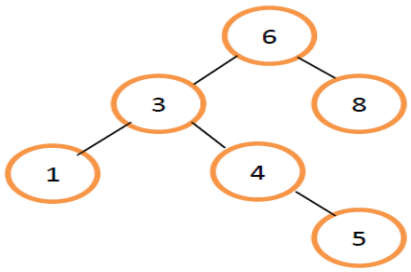
After deletion of 4

CASE 3:(EXAMPLE)

Case 3 :



Before deletion of 2



After deletion

Deletion routine for binary search trees

```
Searchtree delete(elementtype X, searchtree T)
{
position tmpcell;
if(T==NULL)
error("element not found");
else if(X<T-->element)
T-->left=delete(X,T-->left);
Else if(X>T-->element)
T-->right=delete(X,T-->right);
Else if(T-->left != NULL && T-->right!=NULL)
{
/* Replace with smallest in right subtree*/
Tmpcell=findmin(T-->right);
T-->element=tmpcell-->element;
T-->right=delete(T-->element,T-->right);
}
Else
```

```
{
/* One or Zero children*/
tmpcell=T;
if(T-->left==NULL)
T=T-->right;
Else if(T-->right==NULL)
T=T-->left;
Free(tmpcell);
}
Return T;
}
```

Applications of Binary Search Trees

One of the applications of a binary search tree is the implementation of a dynamic dictionary. This application is appropriate because a dictionary is an ordered list that is required to be searched frequently, and is also required to be updated (insertion and deletion mode) frequently. So it can be implemented by making the entries in a dictionary into the nodes of a binary search tree.

SPLAY TREES

Introduction

• Splay trees support all the operations of binary trees.

• But they do not guaranteeO(log *N*)worst-case performance.

• Instead, its bounds are amortized, meaning that although individual operations can be expensive, any sequence of operations is guaranteed to behave as if each operation in the sequence exhibited logarithmic behavior.

Self-adjusting and Amortized Analysis

• Limitations of balanced search trees:

o Balanced search trees require storing an extra piece of information per node.

o They are complicated to implement, and as a result insertions and deletions are expensive and potentially error-prone.

o We do not win when easy inputs occur.

• Balanced search trees have a second feature that seems improvable:

o Their worst-case, average-case, and best-case performance are essentially identical.

o It would be nice if the second access to the same piece of data was cheaper than the first.

o *The 90-10 rule* – empirical studies suggest that in practice 90% of the accesses are to 10% of the data items.

o It would be nice to get easy wins for the 90% case.

Amortized Time Bounds

• Given the above information, what can we do?

• Since the time to access in a binary tree is proportional to the depth of the accessed node,

we can attempt to restructure the tree by moving frequently accessed items toward the root.

• Even if there are intervening operations, we would expect the node to remain close to the root and thus be quickly found.

• We could call this strategy the *rotate-to-root strategy*.

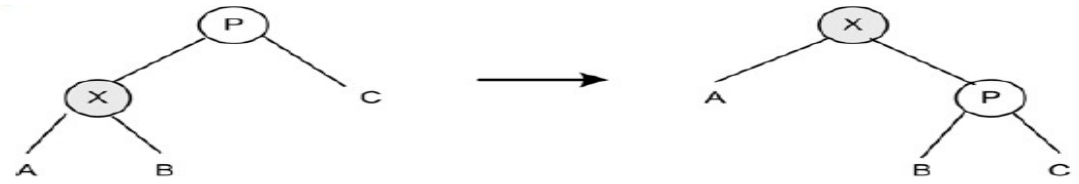
The Basic Bottom-Up Splay Tree

• A technique called ***splaying*** can be used so a logarithmic amortized bound can be achieved.

The *zig* case.

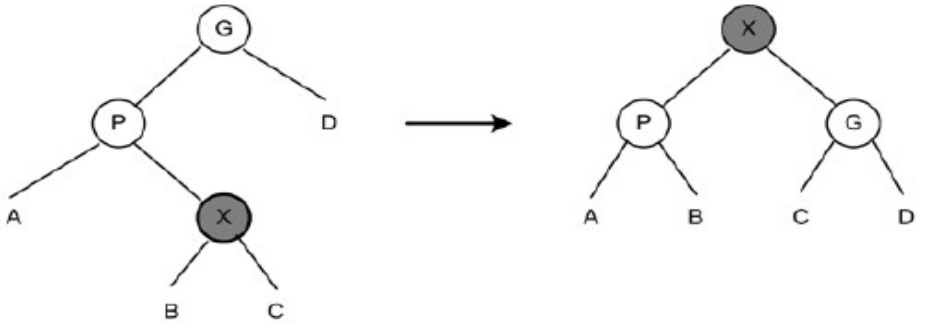
o Let *X* be a non-root node on the access path on which we are rotating.

o If the parent of *X* is the root of the tree, we merely rotate *X* and the root as shown in Figure.

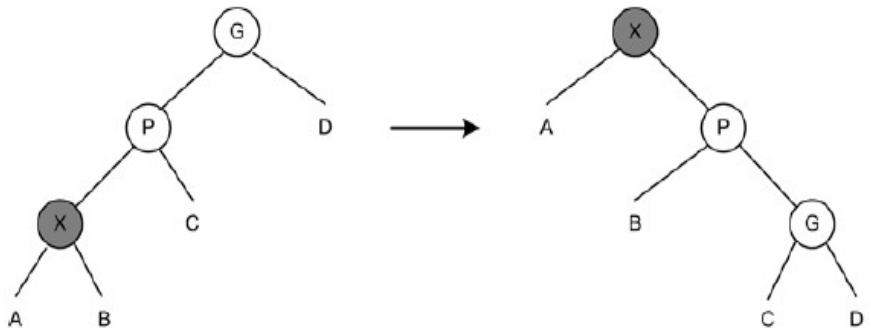


The *zig-zag* case.

o In this case, *X* and both a parent *P* and a grandparent *G*. *X* is a right child and *P* is a left child (or vice versa).
o This is the same as a double rotation.



The *zig-zig* case.
o This is a different rotation from those we have previously seen.
o Here *X* and *P* are either both left children or both right children.
o The transformation is shown in Figure.
o This is different from the rotate-to-root. Rotate-to-root rotates between *X* and *P* and then between *X* and *G*. The zig-zig splay rotates between *P* and *G* and *X* and *P*.



Basic Splay Tree Operations

• Insertion – when an item is inserted, a splay is performed.

o As a result, the newly inserted item becomes the root of the tree.

• Find – the last node accessed during the search is splayed

o If the search is successful, then the node that is found is splayed and becomes the new root.

o If the search is unsuccessful, the last node accessed prior to reaching the NULL pointer is splayed and becomes the new root.

NOTE:Refer EXAMPLE from class notes