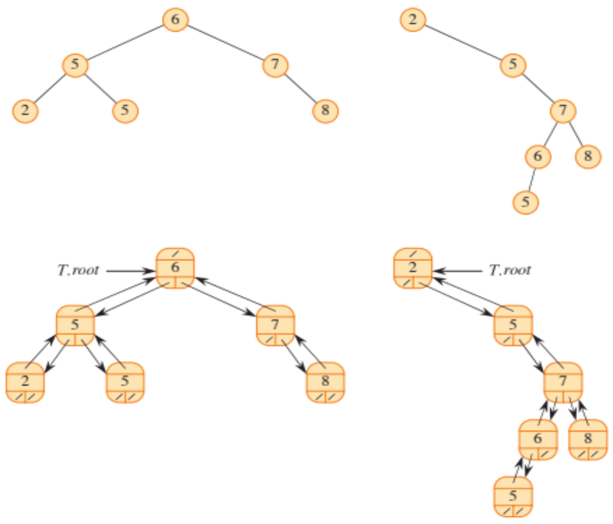
UNIT II - HIERARCHICAL DATA STRUCTURES

Binary Search Trees: Basics – Querying a Binary search tree – Insertion and Deletion- Red Black trees: Properties of Red-Black Trees – Rotations – Insertion – Deletion -B-Trees: Definition of B - trees – Basic operations on B-Trees – Deleting a key from a B-Tree- Heap – Heap Implementation – Disjoint Sets - Fibonacci Heaps: structure – Mergeable-heap operations- Decreasing a key and deleting a node-Bounding the maximum degree.

BINARY SEARCH TREES

A Binary search tree is an organized binary tree. It is represented in a Linked Data structure as nodes. Each node has a Key value, left pointer corresponds to left child, right pointer corresponds to right child and parent pointer. If a child or parent is missing, then the appropriate field contains the value NIL. Each tree has a root node. The root node is the only node with the Parent attribute as NIL.



Properties of Binary Search tree:

The keys in the binary search tree are always stored in a way to satisfy the binary search tree property.

Let x be a node in the binary search tree.

- If y is a node in left subtree, then y.key<x.key
- If y is a node in right subtree, then y.key>x.key

Using a Binary Search tree, the key values can be printed in sorted order using an inorder tree walk recursive algorithm.
It prints the key value of left subtree, the root and then the right subtree

INORDER-TREE-WALK(x)

```
1   if x ≠ NIL
2       INORDER-TREE-WALK(x:left)
3       print x:key
4       INORDER-TREE-WALK(x:right)
```

It takes Θ(n) time to traverse an n-node binary search tree.

QUEUEING A BINARY SEARCH TREE

Binary Search Tree supports the Queries such as Minimum, maximum, Successor, Predecessor and Search.

SEARCHING

Using a procedure Tree-Search, we can search a node with a given value.
Tree-Search(x,k) returns a pointer to a node with key k if it exists or returns NIL.

TREE-SEARCH(x, k)

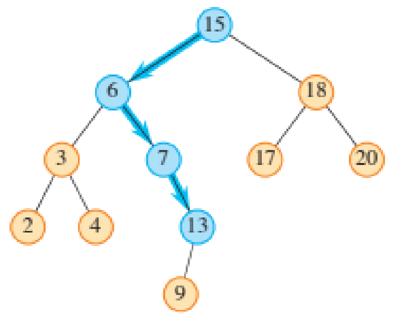
```
1   if x == NIL or k == x.key
2       return x
3   if k < x.key
4       return TREE-SEARCH(x.left, k)
5   else return TREE-SEARCH(x.right, k)
```

ITERATIVE-TREE-SEARCH(x, k)

```
1   while x ≠ NIL and k ≠ x:key
2       if k < x.key
3           x = x.left
4       else x = x.right
5   return x
```

The TREE-SEARCH procedure begins its search at the root. For each node x it compares the key k with x.key. If the two keys are equal, the search terminates. If k is smaller than x.key, the search continues in the left subtree of x, if k is larger than x:key, the search continues in the right subtree of x. The nodes encountered during the recursion form a simple path downward from the root of the tree, and thus the running time of TREE-SEARCH is *O*(h), where h is the height of the tree. The ITERATIVE-TREE-SEARCH procedure is more efficient than the TREE-SEARCH procedure.

In this Binary Search tree, a key value 13 is searched. So the searching starts from the root 15. Since the search key value 13 is smaller than the root key value 15, the TREE-SEARCH follows the left subtree and encounters a key value 6. The search key value 13 is greater than 6, So the Searching follows the Right Subtree and encounters the value 7 which is not equal to Search key value and Search Key value is greater and hence follows the Right Subtree again. It encounters a value 13 where it found the Search key value and the procedure terminates by returning the node of key 13.



MINIMUM AND MAXIMUM

To find a minimum key element in a binary search tree , follow left child pointers from the root until a NIL pointer reaches. Similarly to find a maximum key element in a binary search tree, follow right child pointers from the root until a NIL pointer reaches.

We use the TREE-MINIMUM and TREE-MAXIMUM procedure to find the minimum and maximum key values of a Binary search tree.

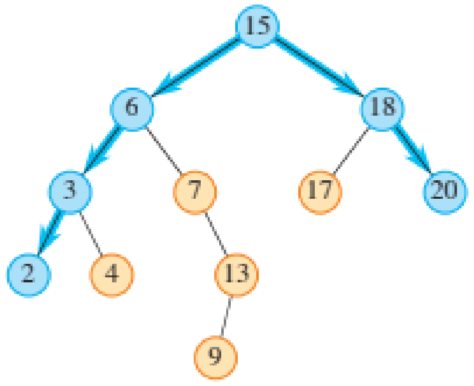
TREE-MINIMUM(x)

```
1   while x.left $\neq$ NIL
2       x = x:left
3   return x
```

TREE-MAXIMUM(x)

```
1   while x.right $\neq$ NIL
2       x = x:right
3   return x
```

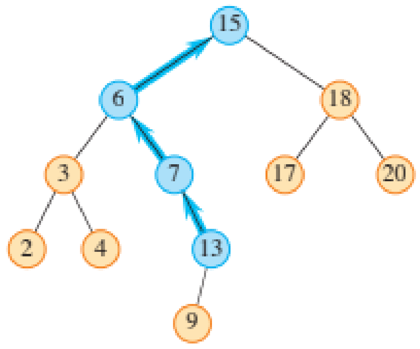
In this Binary Search tree, the minimum value 2 is found by following the left subtree from the root. Since there is no left subtree for the node 2 the procedure stops by returning the minimum value. Similarly, the maximum value 20 is found by following the right subtree. Since there is no more right subtree for the node 20 the procedure stops by returning the maximum value.



SUCCESSOR AND PREDECESSOR

The successor of a node x is the node with the smallest key greater than x.key. The successor of a node is the next node visited in an inorder tree walk. The TREE-SUCCESSOR procedure returns the successor of a node x in a binary search tree if it exists, or NIL if x is the last node that would be visited during an inorder walk. The code for TREE-SUCCESSOR has two cases:

- ➢ If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in x's right subtree. The successor of node 15 is 17, which is the leftmost node of the right subtree.
- ➢ If the right subtree of node x is empty, then the successor of x is found by going up the tree from x until we reach either the root node or a node that is the left child of its parent. The successor of node 13 is 15 which is the root node.



The running time of TREE-SUCCESSOR on a tree of height h is *O*(h), since it either follows a simple path up the tree or follows a simple path down the tree.

The predecessor of the node x is the node with the largest key lesser than x.key. The predecessor of a node is the previous node visited in an inorder tree walk. The TREE-PREDECESSOR procedure returns the predecessor of a node x in a binary search tree if it exists, or NIL if x is the first node in the inorder walk. The code for TREE-PREDECESSOR has two cases:

- ➢ If the left subtree of node x is nonempty, then the predecessor of x is just the rightmost node in x's left subtree. The predecessor of node 6 is 4, which is the rightmost node of the left subtree.

- ➢ If the left subtree of node x is empty, then the predecessor of x is found by going up the tree from x until we reach either the root node or a node that is the right child of its parent.

The running time of procedure TREE-PREDECESSOR is *O*(h) time.

TREE-SUCCESSOR(x)

```
1   if x:right ≠ NIL
2       return TREE-MINIMUM(x.right)/ // leftmost node in right subtree
3   else // find the lowest ancestor of x whose left child is an ancestor of x
4       y = x.p
5   while y ≠ NIL and x == y.right
6       x = y
7       y = y:p
8   return y
```

TREE-PREDECESSOR(x)

```
1   if x:right ≠ NIL
2       return TREE-MAXIMUM(x.left)  // rightmost node in left subtree
3   else // find the lowest ancestor of x whose right child is an ancestor of x
4       y = x.p
5   while y ≠ NIL and x == y.right
6       x = y
7       y = y.p
8   return y
```

INSERTION AND DELETION

INSERTION

The TREE-INSERT procedure inserts a new node into a binary search tree. The procedure takes a binary search tree T and a node *z* for which *z*.key the z.left = NIL and z.right = NIL. It modifies T and some of the attributes of *z* so as to insert z into an appropriate position in the tree.

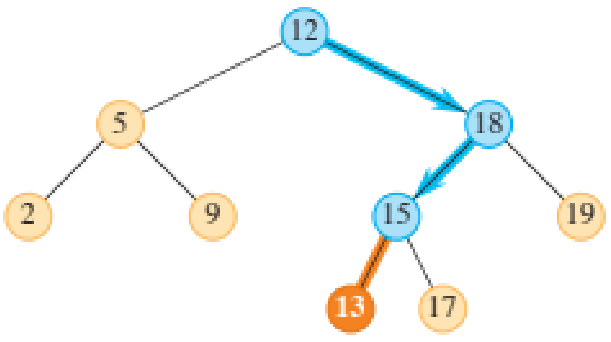
TREE-INSERT (T,z)

```
1   x = T.root // node being compared with z
2   y = NIL // y will be parent of  z
3   while x ≠ NIL // descend until reaching a leaf
```

```
4               y = x
5               if z.key < x.key
6                       x = x.left
7               else x = x.right
8       z.p = y // found the location insert  z with parent y
9       if y == NIL
10              T.root = z // tree T was empty
11      elseif z.key < y.key
12              y.left = z
13      else y.right = z
```

The procedure maintains the trailing pointer y as the parent of x. After initialization, the while loop in lines 3 to 7 causes these two pointers to move down the tree, going left or right depending on the comparison of z.key with x.key, until x becomes NIL. This NIL occupies the position where node z will go. Lines 8 to 13 set the pointers that cause z to be inserted.

For example insert a node with key 13 into a binary search tree. First the root node will be x and y is NIL. If x is not NIL, then the trailing pointer y points to x i.e. 12. The key values of node x 12 and newly inserting node 13 are compared. Since key 13 is greater the traversal is done with the right node by changing x as x.right. The looping continues by changing the trailing pointer y as the node 18. Since 13 is less than 18, the traversal moves to the left node by changing the x pointer and y pointer. Comparison is done between key 15 and 13. Since 13 is less again the pointer x and y changed to the node 15. Now the comparison is done between the key 13 and 15 and x is changed to pointer x.left. As x.left is empty, the position to insert the node with 13 is located as y. Now the parent of z is assigned as y.

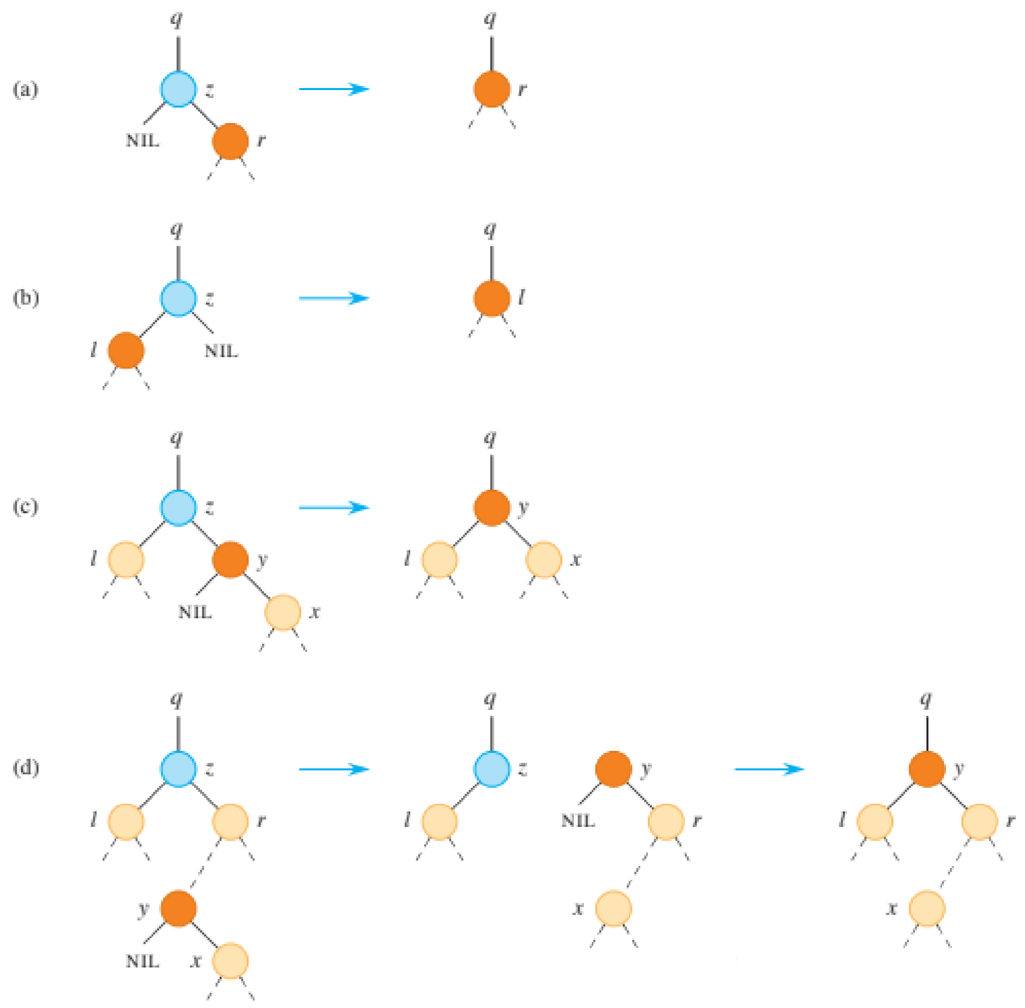


DELETION

There are three cases for deleting a node in a Binary Search Tree.

1. If z has no children, then remove it by changing the parent as NIL.

2. If z has one child, then replace z.child in z's location by changing the z's parent for z's child
3. If z has two children, then find y, the successor of z.
   a. If y has no left subtree l and the successor y = r, the right child of z, then replace z by y and the original right subtree of z becomes y's right subtree and the original left subtree of z becomes the left subtree of y.
   b. If y has no left subtree and the successor y ≠ r, then replace y by its own right child x, and set y to be r's parent. Then set y to be q's child and the parent of l.



The subroutine TRANSPLANT replaces one subtree as a child of its parent with another subtree. When TRANSPLANT replaces the subtree rooted at node u with the subtree rooted at node v, node u's parent becomes node v's parent, and u's parent ends up having v as its appropriate child. TRANSPLANT allows v to be NIL instead of a pointer to a node.

TRANSPLANT (T, u, v)

```
1 if u.p = = NIL
2       T.root = v
3 elseif u = = u.p.left
4       u.p.left = v
5 else u.p.right = v
6       if v ≠ NIL
7             v.p = u.p
```

TREE-DELETE (T, z)

```
1 if z.left == NIL
2       TRANSPLANT (T, z, z.right)          // replace  z by its right child
3 elseif z.right == NIL
4       TRANSPLANT (T,z,z.left)             // replace z by its left child
5 else y = TREE-MINIMUM(z.right)            // y is z's successor
6       if y ≠ z.right                      // is y farther down the tree?
7             TRANSPLANT (T, y, y.right)    // replace y by its right child
8             y.right = z.right             // z's right child becomes
9             y.right .p = y                // y's right child
10      TRANSPLANT (T, z, y)                // replace z by its successor y
11      y.left = z.left                     // and give z's left child to y,
12      y.left.p = y                        // which had no left child
```

Each line of TREE-DELETE, including the calls to TRANSPLANT, takes constant time, except for the call to TREE-MINIMUM in line 5. Thus, TREE-DELETE runs in $O(h)$ time on a tree of height h.