Review

- Recursion & Examples
  - Quick Sort
  - Sum of Range
  - The Fibonacci Series
  - Greatest Common Devisor
  - Binary Search
- Operator Overloading
  - As a member function
  - As a Friend function

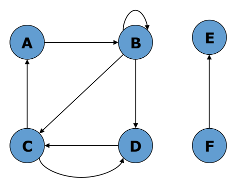
Preview

- Graph
- Tree
  - Binary Tree
  - Binary Search Tree
  - Binary Search Tree Property
  - Binary Search Tree functions
    - In-order walk
    - Pre-order walk
    - Post-order walk
    - Search Tree
    - Insert a element to the Tree
    - Delete a element form the Tree

Graph

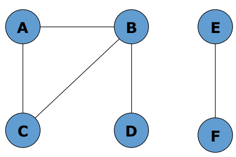
- Directed graph (or digraph)
  G = (V, E)
  V: Set of vertex (node)
  E: Set of edges (ordered vertex pair)
- Undirected graph
  G = (V, E)
  V: Set of vertex
  E: Set of edges (unordered vertex pair)

Graph



Digraph G = (V, E)
V = {A, B, C, D, E, F}
E = {(A, B), (C, A), (B, D), (B, C), (C, D), (D, C), (B, B) (F, E)}

Graph

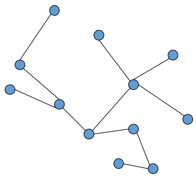


Undirected Graph G = (V, E)
V = {A, B, C, D, E, F}
E = {(A, B), (A, C), (B, D), (B, C), (E, F)}

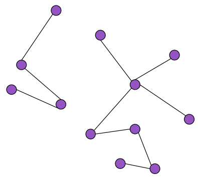
Tree

- A undirected graph is connected if every pair of vertices is connected by a path.
- **Free Tree** – is a connected acyclic, undirected graph.
- **Forest** – a acyclic but possibly disconnected undirected graph
- **Rooted tree** – a free tree where one of the vertices is distinguished from the other. The distinguished vertex is called root.

Tree

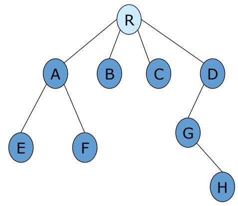


Free Tree



Forest

Tree

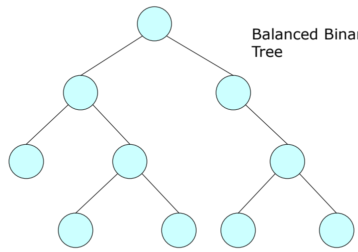


- If the last edge on the path from from the root R of a tree T to a node is (y ,x) then y is **parent** of x and x is **child** of y.
- If two nodes have same parent, they are **siblings**
- A node has no children is **leaf**.
- A non-leaf node is an **internal node**
- The length of the path from the root R to a node x is the **depth** of x in the tree.
- The largest depth of any node in tree T is the **height** of tree T.

Binary Tree

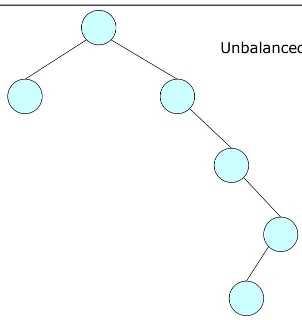
- **A binary tree** is a tree data structure in which <u>each node has at most two children</u>.
- Typically the child nodes are called <u>*left* child</u> and <u>*right child*</u>.
- Binary trees are commonly used to implement **binary search trees** and **heaps**.

Binary Tree



Balanced Binary Tree

Binary Tree

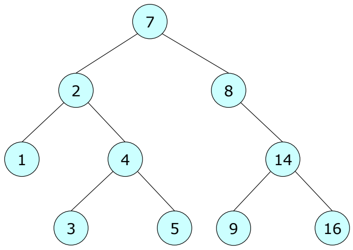


Unbalanced Binary Tree

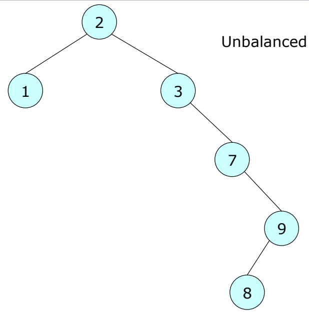
Binary Search Tree

- **Binary search tree** – a binary tree with binary-search-tree property.
- **Binary-Search-Tree Property**
  - Let x be a node in a binary search tree.
  - If y is a node in the left subtree of x then key(y) ≤ key(x).
  - If y is a node in the right subtree of x, then key(x) ≤ key(y).

Binary Search Tree



Binary Search Tree



Operations on Binary Search Tree

- Binary Search Tree Operations
  - Inorder walk
  - Preorder walk
  - Postorder walk
  - Search Tree
  - Insert a element to the Tree
  - Delete a element form the Tree

Operations on Binary Search Tree
(Inorder, Preorder, Postorder)

- The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an **inorder tree** walk.

```
Inorder_Tree_Walk(x)
{
        If x ≠ NIL
        {
                Inorder_Tree_Walk(x→leftchild);
                print x → key
                Inorder_Tree_Walk(x → rightchild);
        }
}
```

Running Time T(n) = 2T(n/2) + 1 = O(n)

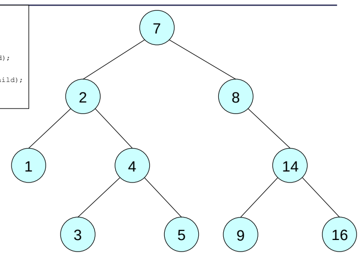
Operations on Binary Search Tree
(Inorder, Preorder, Postorder)

```
Preorder_Tree_Walk(x)
{
        If x ≠ NIL
        {
                Print x → key;
                preorder_Tree_Walk(x→leftchild);
                preorder_Tree_Walk(x → rightchild);
        }
}
```

```
Postorder_Tree_Walk(x)
{
        If x ≠ NIL
        {
                postorder_Tree_Walk(x→leftchild);
                postorder_Tree_Walk(x → rightchild);
                Print x → key;
        }
}
```


Operations on Binary Search Tree
(Inorder, Preorder, Postorder)

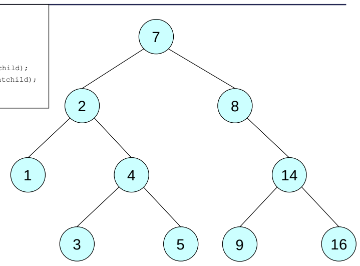
```
Inorder_Tree_Walk(x)
{
        If x ≠ NIL
        {
                Inorder_Tree_Walk(x→leftchild);
                print x → key
                Inorder_Tree_Walk(x → rightchild);
        }
}
```



1 2 3 4 5 7 8 9 14 16

Operations on Binary Search Tree
(Inorder, Preorder, Postorder)

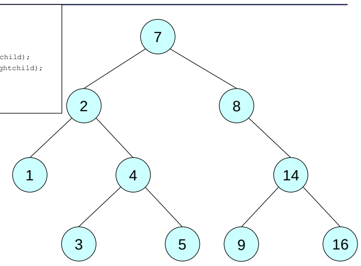
```
Preorder_Tree_Walk(x)
{
        If x ≠ NIL
        {
              print x → key;
              prenorder_Tree_Walk(x→leftchild);
              preorder_Tree_Walk(x → rightchild);
        }
}
```



7 2 1 4 3 5 8 14 9 16

Operations on Binary Search Tree
(Inorder, Preorder, Postorder)

```
Postorder_Tree_Walk(x)
{
      If x ≠ NIL
      {
              Postorder_Tree_Walk(x→leftchild);
              Postorder_Tree_Walk(x → rightchild);
              Print x → key;
      }
}
```



1 3 5 4 2 9 16 14 8 7

Operations on Binary Search Tree
(Search an Element)

- Searching a binary search tree for a specific value can be a recursive or iterative process.
- We begin by examining the root node. If the tree is null, the value we are searching for does not exist in the tree. Otherwise, if the value equals the root, the search is successful. If the value is less than the root, search the left sub-tree.
- If it is greater than the root, search the right subtree. This process is repeated until the value is found or the indicated sub-tree is null. If the searched value is not found before a null sub-tree is reached, then the item must not be present in the tree.

Operations on Binary Search Tree
(Search an Element)

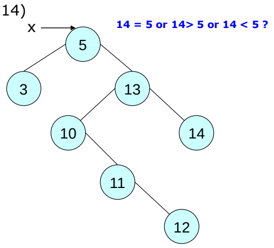
Searching
Input: pointer to the root of the tree and a key k,
Output: returns a pointer to a node with key k if one exists; otherwise return NIL.

```
Tree_Search (x, k)
{
         if x == NIL or k == x → key
                  return x;
         if k < x → key
                  return Tree_Search (x → leftchild, k);
         else
                  return Tree_Search(x → rightchild, k);
}
```

T(n) = T(n/2) + 1 = O(log n) if a binary tree is balanced
T(n)= T(n-1) + 1 = O(n) if a binary tree is not balanced

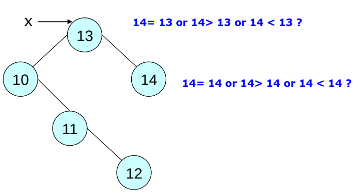
Operations on Binary Search Tree
(Search an Element)

Search (x, 14)



Operations on Binary Search Tree
(Search an Element)

Search (x, 14)



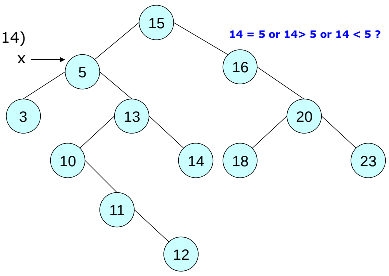
Operations on Binary Search Tree
(Search an Element)

- Searching a binary search tree for a specific value can be a recursive or iterative process. Following shows the iterative version of tree search

```
Tree_Search_II(x, k)
{
        y = x; //better not modify pointer x
        while y ≠ NIL and k ≠ y → key
        {
                if k < y → key
                        y = y → leftchild;
                else
                        y = y → rightchild;
        }
        return y;
}
```

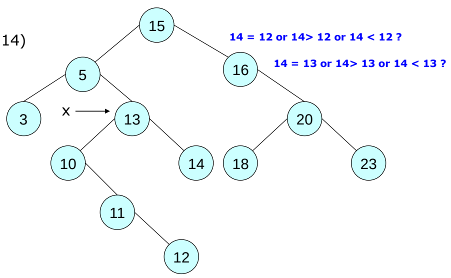

Operations on Binary Search Tree
(Search an Element)

Search (x, 14)



Operations on Binary Search Tree
(Search an Element)

Search (x, 14)



Operations on Binary Search Tree
(Minimum Element)

Minimum

- An element in a binary search tree whose key is a minimum can always be found by leftmost child.

```
Tree_Minimum(x)
{
        y = x;
        while y → leftchild ≠ NIL
                y = y → leftchild;
        return y;
}
```

$T(n) = O(\log_2 n)$ or possible worst case $O(n)$

Operations on Binary Search Tree
(Maximum Element)

Maximum

- An element in a binary search tree whose key is a maximum can always be founded by rightmost child.

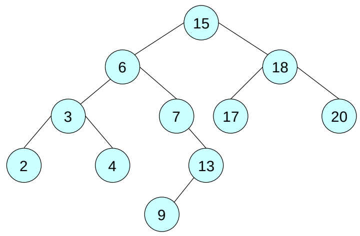
```
Tree_Maximum(x)
{
        y = x;
        while y → rightchild ≠ NIL
                y = y → rightchild;
        return y;
}
```

$T(n) = O(\log_2 n)$

Operations on Binary Search Tree
(Successor of a Node)

- The successor of a node x is the node with the smallest key greater than x->key
- We need to concern two cases
  1. If the right subtree of x is not empty, then the successor of x is the minimum of right subtree.
  2. If the right subtree of x is empty and x has a successor y, then **y is the lowest ancestor of x** whose left child is also an ancestor of x.

Operations on Binary Search Tree
(Successor of a Node)



Ex) successor of node 15 is 17 (min of right subtree)
Ex) successor of node 13 is 15.

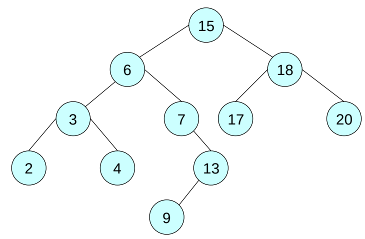
Operations on Binary Search Tree
(Successor of a Node)

```
Tree_Successor (x)
{
        if x → rightchild ≠ NIL
        return Tree_Minimum(x → rightchild);
        else
        {
                y = x → parent;
                // if x is y's leftchild then y is successor of x
                while y ≠ NIL and x ==y → rightchild
                {
                        x = y;
                        y = y → parent;
                }
                return y;
        }
}
```


Operations on Binary Search Tree
(Predecessor of a Node)

- The predecessor of a node x is the node with the largest key smaller than key(x)
- We need to concern two cases
  1. <u>If the left subtree of x is not empty</u>, then the predecessor of x is the <u>maximum of right subtree</u>.
  2. <u>If the left subtree of x is empty and x has a predecessor y</u>, then y is the lowest ancestor of x whose right child is also an ancestor of x.

Operations on Binary Search Tree
(Predecessor of a Node)



Ex) Predecessor of node 15 is 13 (Maximum of left subtree)
Ex) Predecessor of node 9 is 7.

Operations on Binary Search Tree
(Predecessor of a Node)

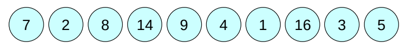
```
Tree_Predecessor (x)
{
        if x → leftchild ≠ NIL
        return Tree_Maximum(x → leftchild)
        else
        {
                y = x → parent

                // if x is y's rightchild then y is predecessor of x
                while y ≠ NIL and x ==y → leftchild
                {
                        x = y
                        y = y → parent
                }
                return y
        }
}
```


Operations on Binary Search Tree
(Insert a New Node)

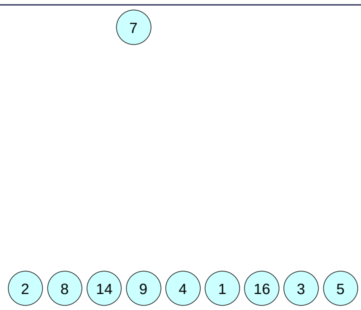
- Insertion begins as a search would begin; if the root is not NIL, we search the left or right sub-trees as before.
- Eventually, we will reach an external node and add the value as its right or left child, depending on the node's value.

```
// T is point to root of binary tree z is point to new node
Tree_Insert (T, z)
{
        y = NIL;
        x = T;
        //while loop find out the location for new node
        while x ≠ NIL
        {
                y = x;
                if z → key < x → key
                        x = x → leftchild;
                else
                        x = x → rightchild;
        }
        z → parent = y;
        if y = NIL; // means there is any node in the BST
                T = z; //new node become root
        else if z → key < y → key
                y → leftchild = z; // insert new node as a left child o y
        else
                y → rightchild = z; //insert new node as a right child of y
}
```

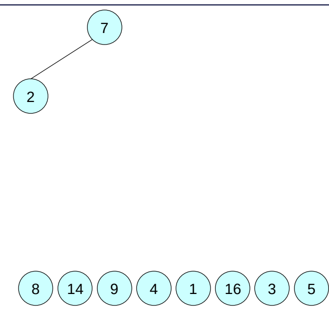

Operations on Binary Search Tree
(Insert a New Node)



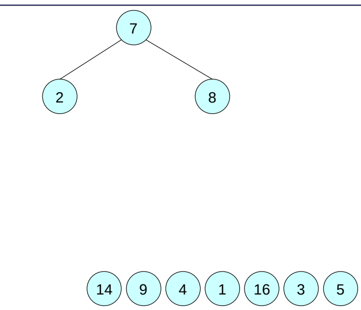
Operations on Binary Search Tree
(Insert a New Node)



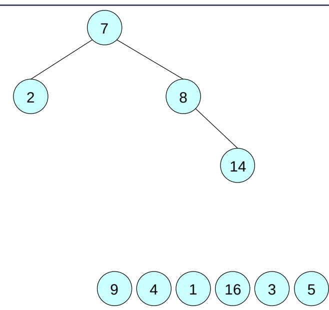
Operations on Binary Search Tree
(Insert a New Node)



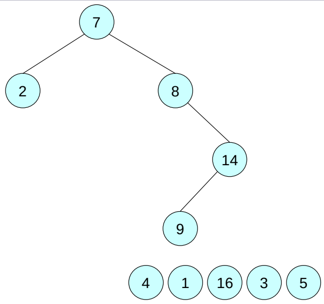
Operations on Binary Search Tree
(Insert a New Node)



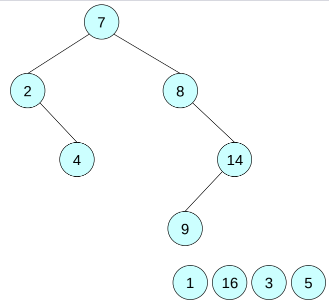
Operations on Binary Search Tree
(Insert a New Node)



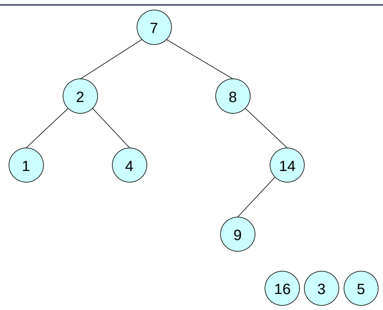
Operations on Binary Search Tree
(Insert a New Node)



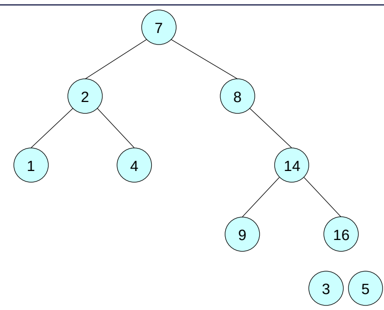
Operations on Binary Search Tree
(Insert a New Node)



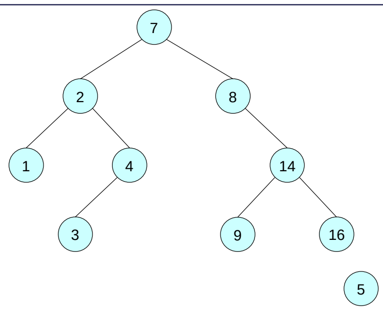
Operations on Binary Search Tree
(Insert a New Node)



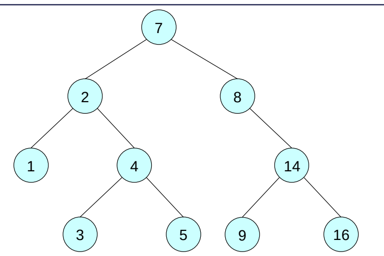
Operations on Binary Search Tree
(Insert a New Node)



Operations on Binary Search Tree
(Insert a New Node)



Operations on Binary Search Tree
(Insert a New Node)



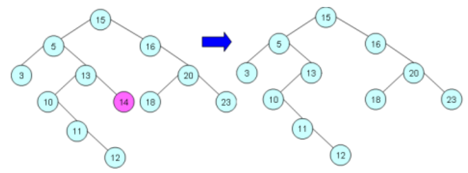
Operations on Binary Search Tree
(Delete a Node)

- There are three cases needed to be concern in binary tree deletion.

1. A deleting node has no children (deleting node is a leaf)
2. A deleting node has one children.
3. A deleting node has two children.
   a) If successor of deleting node is a leaf
   b) If successor of deleting node is not a leaf

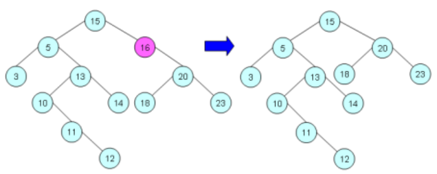
Operations on Binary Search Tree
(Delete a Node)

Delete Leaf Node

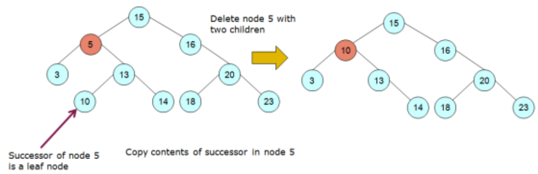


Operations on Binary Search Tree
(Delete a Node)

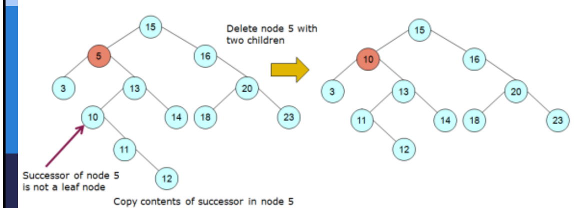
Delete a Node with one child



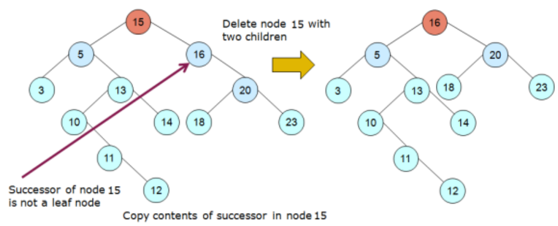
Operations on Binary Search Tree
(Delete a Node)



Operations on Binary Search Tree
(Delete a Node)



Operations on Binary Search Tree
(Delete a Node)



```
Tree_Delete (T, z)
{
   if z →leftchild == NIL or z  → rightchild == NIL
        y = z
   else
        y = Tree_Successor(z)
   if y  → leftchild ≠  NIL
        x = y  → leftchild
   else
        x = y  → rightchild
   if x ≠ NIL
        x  → parent = y  → parent
   if y  → parent ==NIL
        T → root = x
   else if y == y  → parent → left
        y  → parent → leftchild = x
   else
        y  → parent → rightchild = x

   if y ≠ z
        z  → key =y  → key
   delete y
}
```