Trees
Binary Trees
Binary Search Tress

Definition

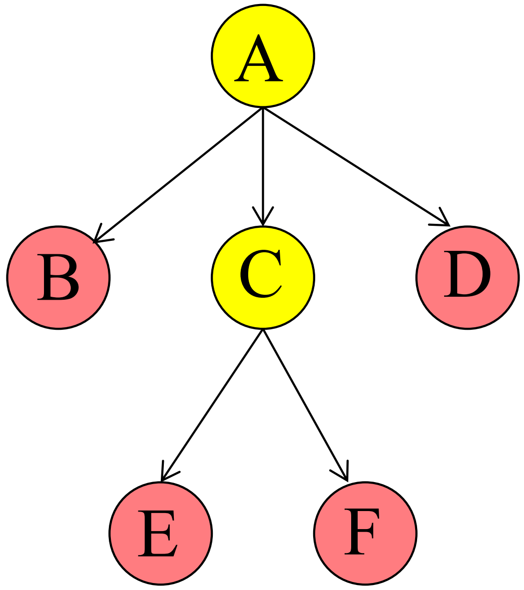
A tree is either:

a. empty, or

b. it has a node called the root, followed by zero or more **trees** called subtrees

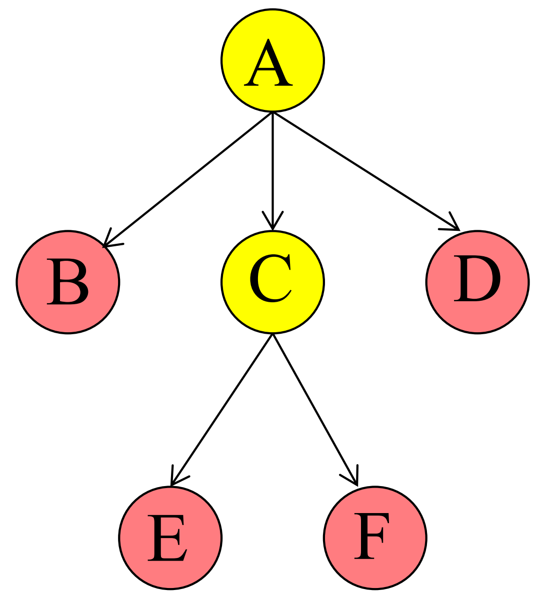
Tree Terminologies

- Nodes: A, B, C, D, E, F
- Root node: A
- Leaf nodes: B, E, F, D
- Arcs: **(A,B), (A,C), (A,D), (C, E), (C, F)**



Tree Terminologies

- Path: a unique sequence of arcs to connect the node from the root.
- Path examples: (A,B), (A,C), (A,C,E), (A,C,F), etc

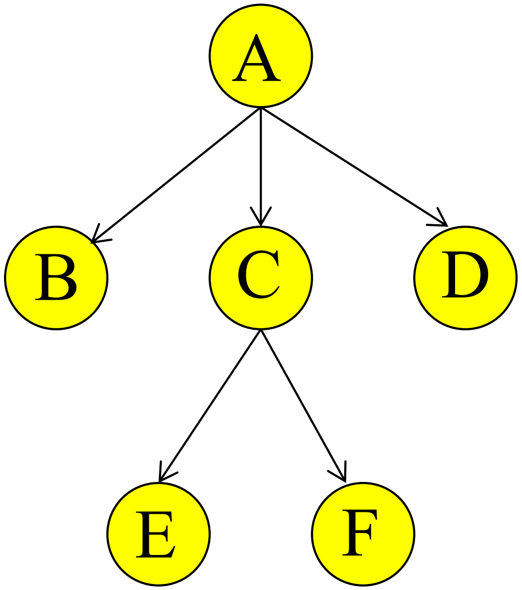


Tree Terminologies

- Length of a path = number of arcs
- Level of a node
  - = the length of the path from the node to the root + 1
  - = the number of nodes in the path
- Height of a non-empty tree = maximum level of a node

Height of the tree: 3

level = 1 for A



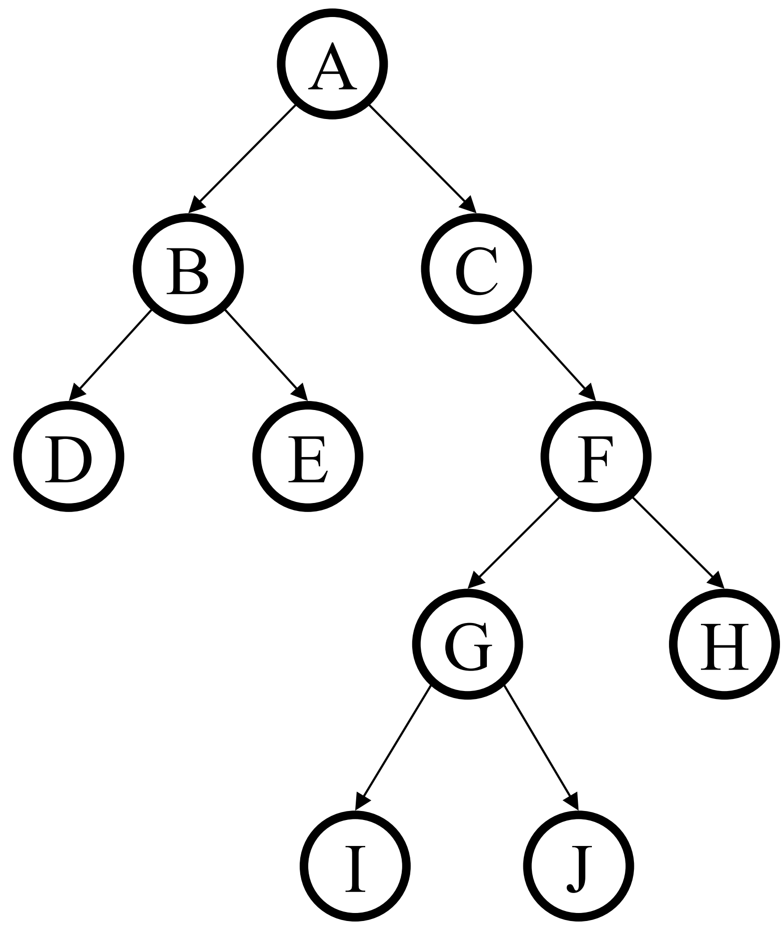
Level = 3 for E

Binary tree

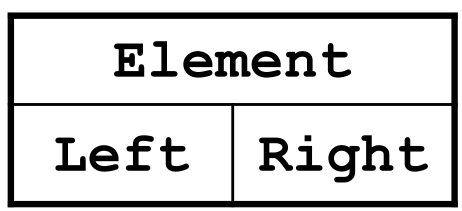
- A tree whose nodes have 2 children (possibly empty), and each child is designated as either a left child or a right child.

Binary Trees

- Representation:



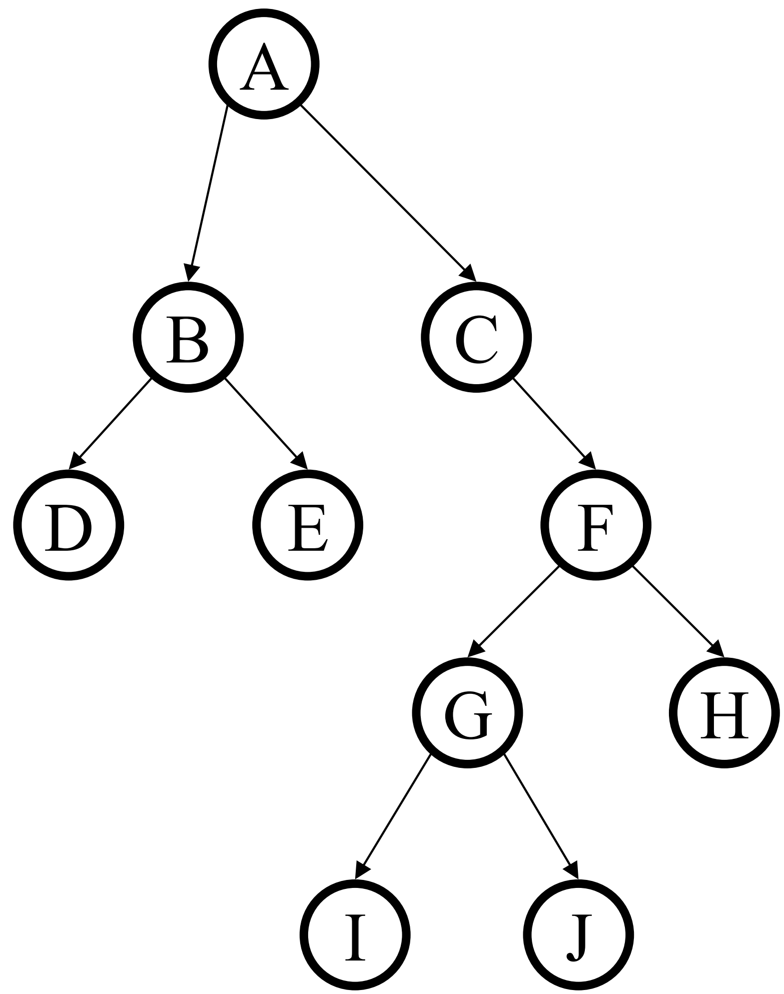
TreeNode:



Binary Trees

- Properties
  – Max # of leaves = $2^{height(tree)-1}$
  – Max # of nodes = $2^{height(tree)} - 1$

- Height of a binary tree with n nodes:
  – Max height: n (same as linked list)
  – Min height: floor(lg(n)) +1



Complete Binary Trees

- If all the nodes at all levels except the last level had 2 children, then it is a complete binary tree.
- Level 1, 1 node
- Level i, 2^{i-1} node
- Leaf nodes m, non-leaf nodes k, then m = k+1.
- Height i, 2^{i} -1 nodes.
- Example: A complete binary tree of height 4, then total 15 nodes with 8 leaves and 7 non-leaves.

Implementation of Binary tree

- Approach 1:
  - Array (of nodes), each node with an information field, and 2 “pointer” *fields* that contain the indices of the array cells in which the left and right children are stored.
  - Root is cell 0. -1 is a null child.
  - Problems: Sequential locations, holes after deletion.
- Approach 2: (more common)
  - Each node is an object of an information field, and 2 pointer members.

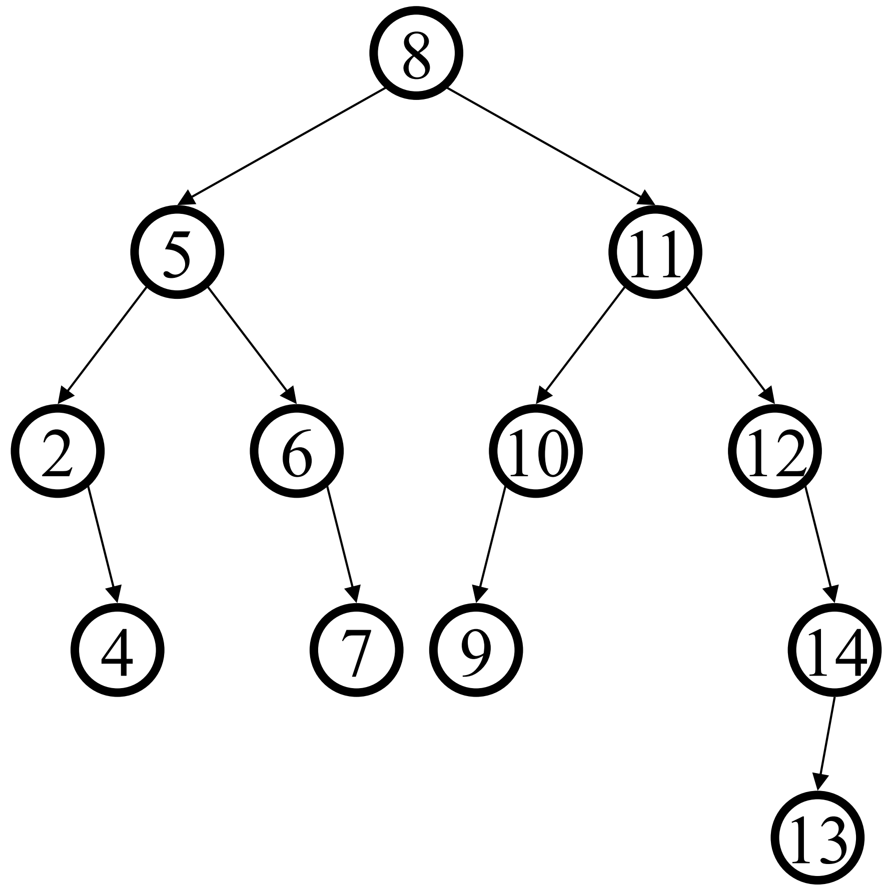
Node of A binary tree

```
template<class T>
Class BSTNode {
 public:
        BSTNode()  { left = right = 0; }
        BSTNode(const T& el, BSTNode *l =0, BSTNode *r = 0)
                { key = el; left = l; right = r; }
        T key;
        BSTNode *left, *right;
}
```

This implementation can also be used for BST. See later.

Binary Search Tree (BST)

- Search tree property
  - all keys in left subtree smaller than root's key
  - all keys in right subtree larger than root's key
  - result:
    - easy to find any given key
    - inserts/deletes by changing links



Search in BST (iterative impl.)

```
template<class T>
T* BST<T>::search(BSTNode<T>* p, const T& el) const {
   while (p !=  0)
        if  (el == p->key)
              return &p ->key;
        else if (el < p->key)
               p = p->left;
        else
               p = p->right;
    return 0;
 }
```

Complexity of Search in BST

- Complexity of search is measured by the number of comparisons performed during the searching process.

Complexity of Search in BST:

- Running time analysis
  - Worst case: O(n) -- when tree is off balance and the shape is like a linked list.
  - Best case: O(logn) – when tree is complete.
  - Average case: O(logn) – close to the best case.

Tree traversal

- Process of visiting each node in the tree exactly one time.
- Concept applicable to any tree. Implementation can be different depending on the nature of the tree. We talk about binary tree here.

Breadth-1st traversal

- Starting from highest level and moving down level by level, visiting nodes on each level from left to right.
- Also called level-order traversal.
- Implement use a queue.

```
template<class T>
void BST<T>::breadthFirst()
{
   queue<BSTNode<T>*> q;
    BSTNode<T> *p = root;
    if (p!=0)
    {   q.push(p);       //enqueue
        while(!q.empty())
         {
             p = q.front();   q.pop();    //dequeue
             visit(p);
             if (p->left != 0)
                  q.push(p->left);           //enqueue
             if  (p->right != 0)
                   q.push(p->right);        //enqueue
         }
     }
}
```

Discussion

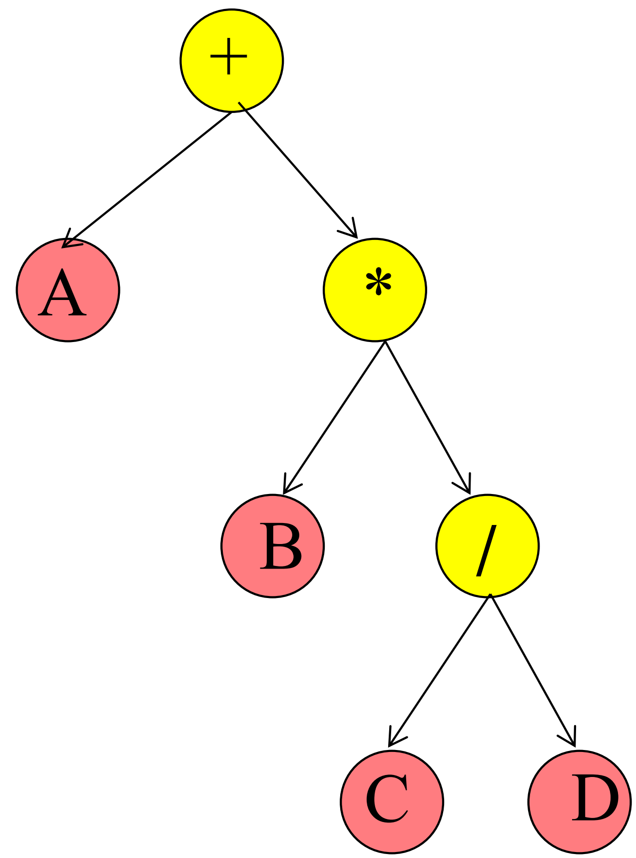
- When a node is visited, put its children in a queue (tail).
- For a node on level n, its children are on level n+1, these children are at the end of the queue, and are visited after all nodes from level n are visited.
  - All nodes on level n must be visited before visiting any nodes on level n+1.
- Enqueue root → dequeue root -> enqueue children of root → dequeue 1st child of root -> enqueue the grandchildren of root -> dequeue 2nd child of root …

Depth-1st traversal

- Preorder -- VLR
- Postorder – LRV
- Inorder – LVR

Traversing Trees (example)

- Preorder: Root, then Children
  – + A * B / C D
- Postorder: Children, then Root
  – A B C D / * +
- Inorder: Left child, Root, Right child
  – A + B * C / D



- Note: This example is in fact a so-called expression tree. The expressions are called prefix, postfix and infix form of an expression, respectively.

Preorder

```
template<class T>
Void BST<T>::preorder(BSTNode<T> *p)
{
    if (p != 0)
    {
        visit(p);
        preorder(p->left);
        preorder(p->right);
    }
}
```

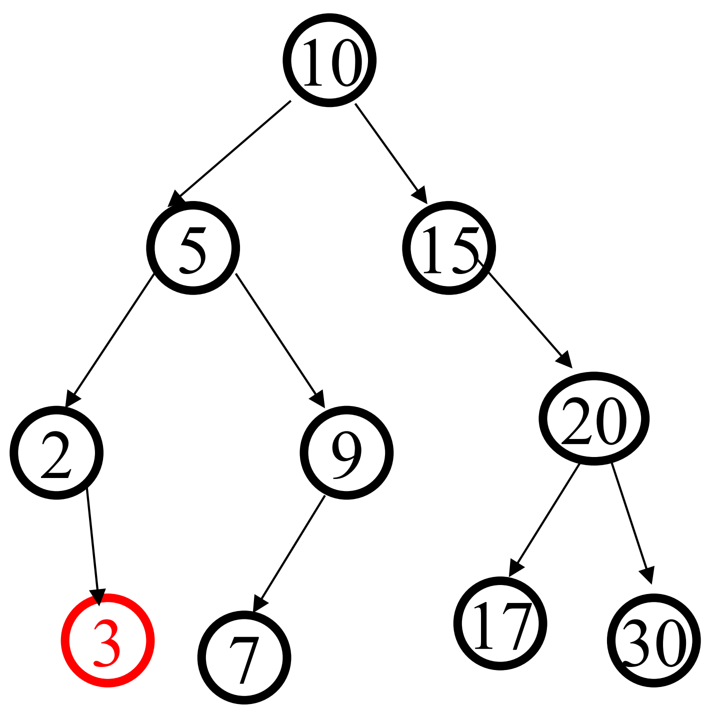
Postorder

```
template<class T>
Void BST<T>::postorder(BSTNode<T> *p)
{
   if (p != 0)
    {
         postorder(p->left);
           postorder(p->right);
           visit(p);
    }
}
```

Inorder

```
template<class T>
void BST<T>::inorder(BSTNode<T> *p)
{
    if (p != 0)
    {
        inorder(p->left);
        visit(p);
        inorder(p->right);
    }
}
```

Binary Search Tree Insertion



Insertion into a BST

```
template<class T>
void BST<T>::insert(const T& el)
{
    BSTNode<T> *p = root, *prev = 0;
    while (p != 0)
    {
        prev = p;
        if(p->key < el)
            p = p->right;
        else
            p = p->left;
    }
    if (root == 0 ) // tree is empty
        root = new BSTNode<T>(el);
    else if (prev->key < el)
        prev->right = new BSTNode<T>(el);
    else
        prev->left = new BSTNode<T>(el);
}
```

Note:

1. If a value is already in the tree, this implementation inserting it as the left child.
2. You can also use recursion to implement it.

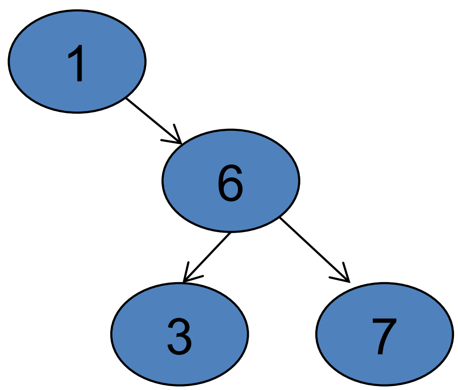
Discussion

- Same technique as the BST search, look for a node with a null end, based on BST ordering property.
- Once found, insert the node as its child. (Use prev, because p is the null pointer already.)
- Running time of BST insertion:

$$\boxed{\text{time} = O(\text{height}(T))}$$

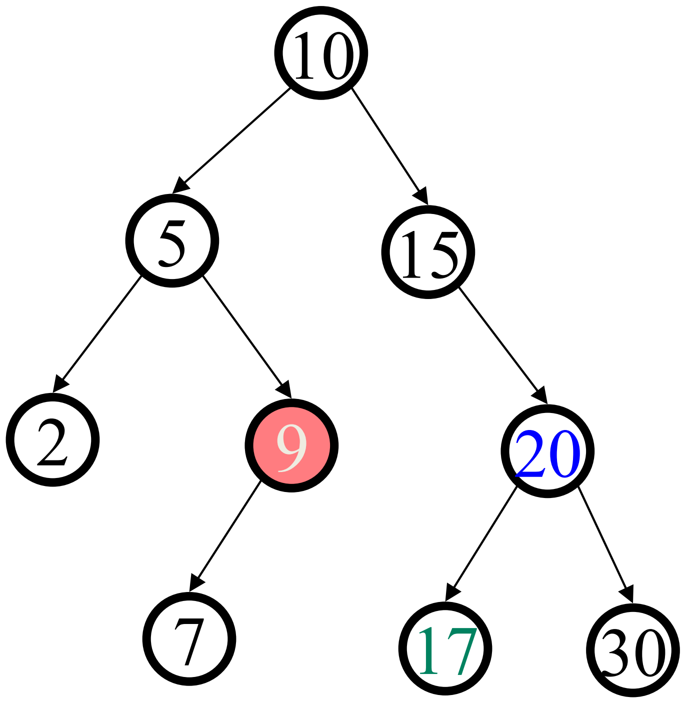
Example of using insertion to construct a binary search tree:

- 1, 6, 3, 7



What if the sequence is 1 3 6 7?

Deletion from BST



How do you delete:

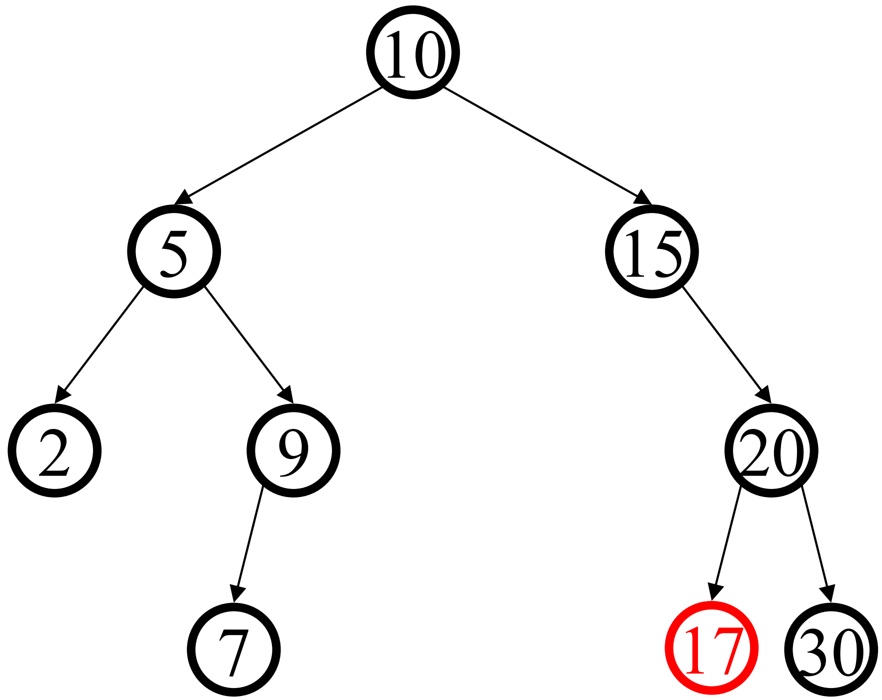
17 ?

9 ??

20 ???

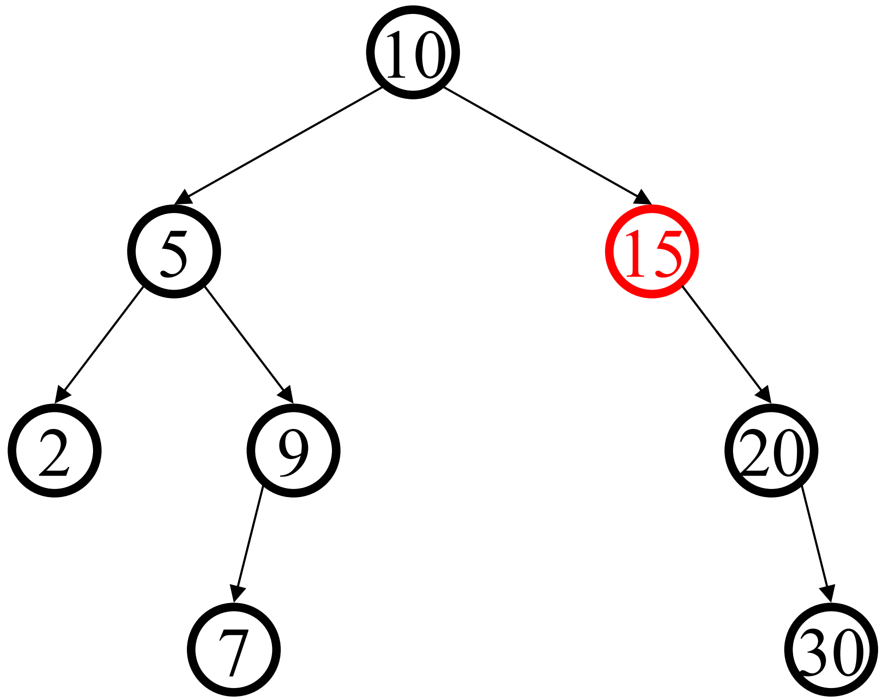
Deletion - Leaf Case

Delete(17)



Deletion - One Child Case

Delete(15)



Just pull up its child. 10 ‘s right child is now 20.

Deletion - Two Children Case

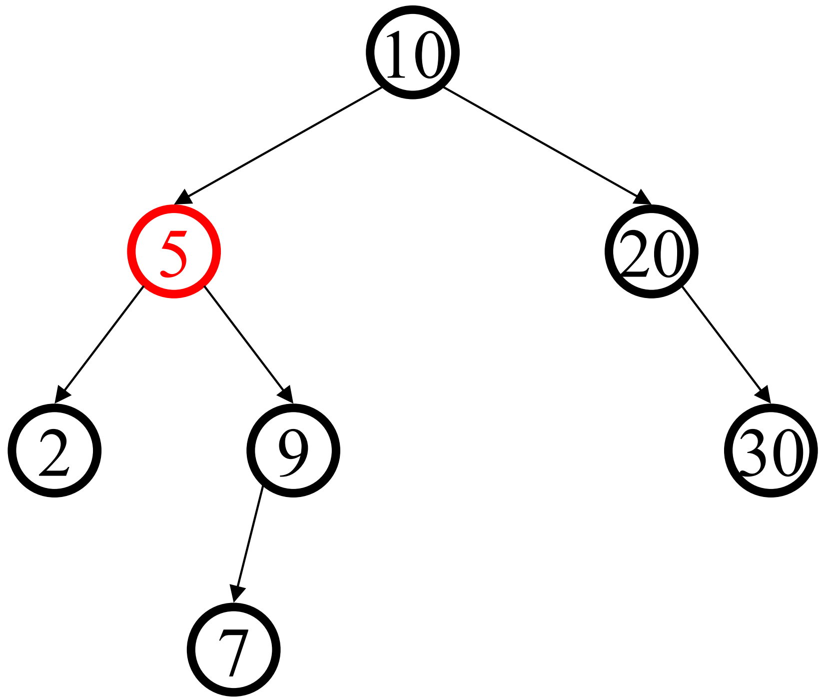
- Most complicated
  - Deletion by copying
  - Deletion by merging

Predecessor and Successor

- Predecessor: A key's predecessor is the key in the rightmost node in the left subtree.
- successor: A key's successor is the key in the leftmost node in the right subtree.

Deletion - Two Children Case

Delete(5)

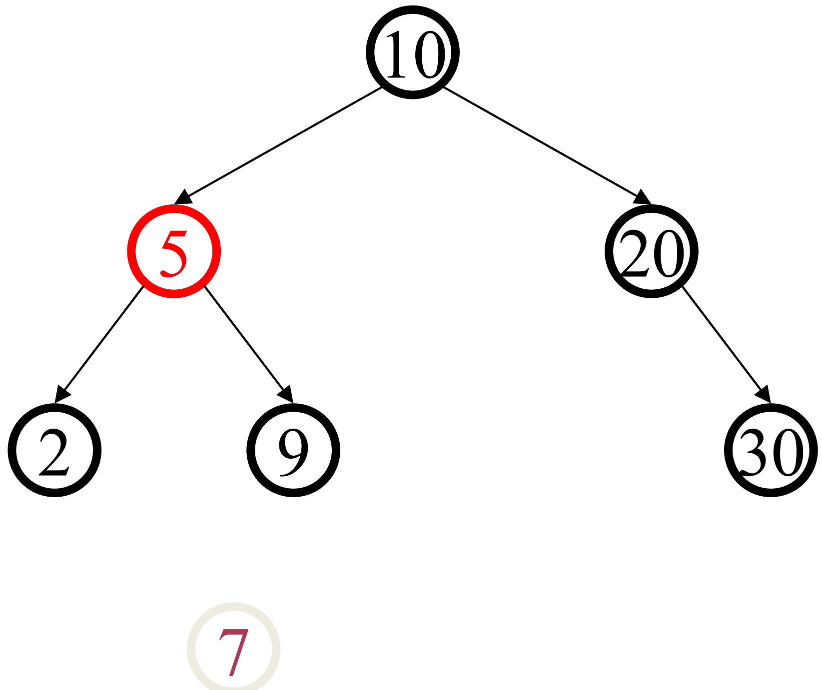


Replace the node with the value guaranteed to be between the left and right subtrees: the successor.

WHAT IF USE PREDECESSOR?

Deletion - Two Children Case

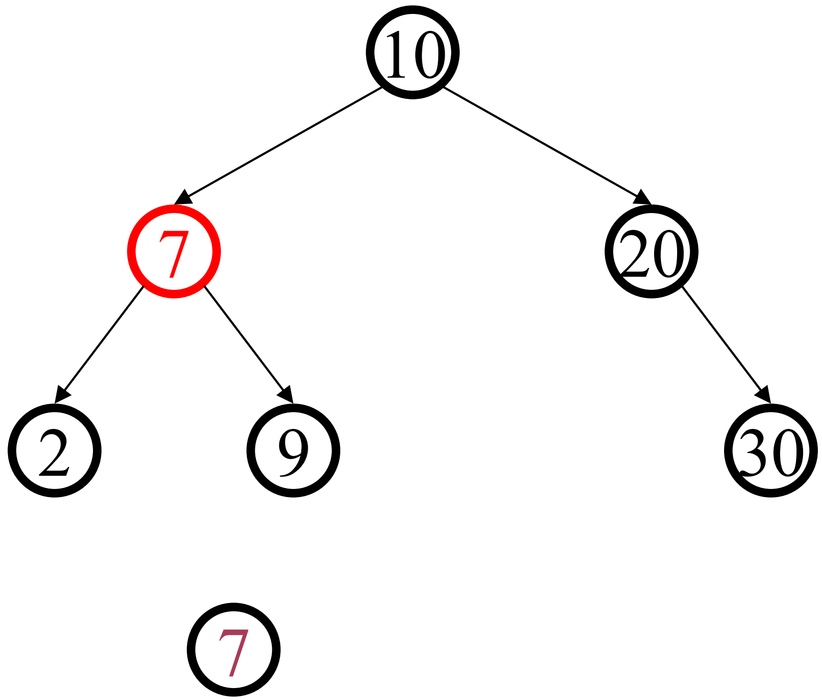
Delete(5)



Always easy to delete the successor – always has either 0 or 1 child!

Deletion - Two Child Case

Delete(5)



Finally copy data value from deleted successor into original node

- What is the cost of a delete operation ?
- Can we use the *predecessor* instead of successor ?

Deletion by Copying

- The previous method demonstrated is called "deletion by copying".

```
template<class T>
void BST<T>::deleteByCopy(BSTNode<T>*& node)
{
    BST<Node<T> *previous, *temp = node;
     if (node->right ==0)        //node has no right child
        node = node->left;
     else if (node->left == 0)  //node has no left child
        node = node->right;
     else  {                         //node has both children
         tmp = node->left;
         previous = node;                    //1. set previous
         while (tmp->right != 0) {                //2. search for  predecessor
             previous = tmp;
             tmp = tmp->right;
          }
          node->key = tmp->key;                 //3.  copy
          if (previous == node)                  //4. break the link for predecessor
                  previous->left = tmp->left;
            else previous->right = tmp->left;
        }
        delete tmp;                                   //5. delete
}
```

Discussion on Deletion by Copying

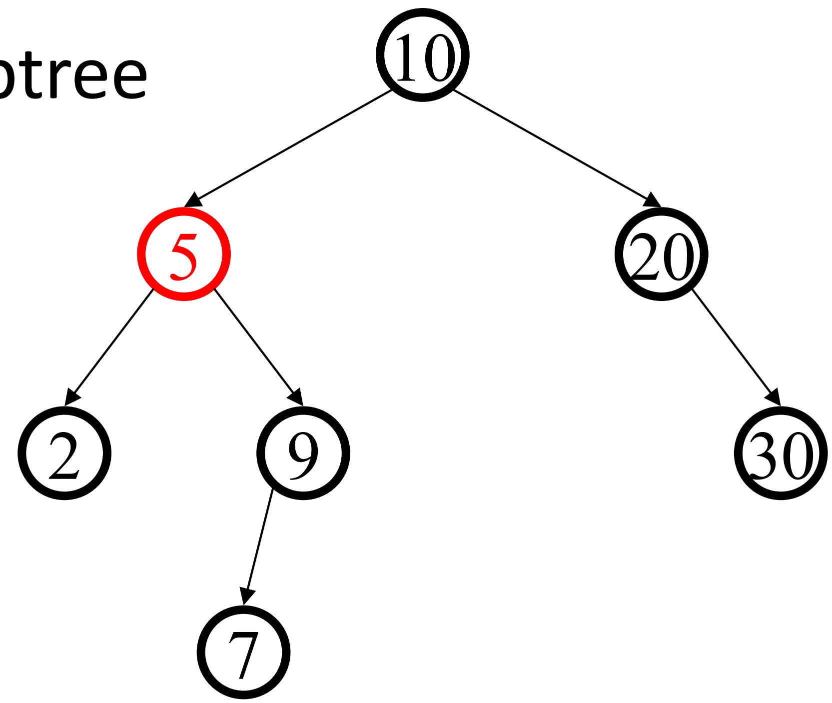
- Advantages
  - Does not increase height
- Problems
  - May become unbalanced if always use the successor -- Alternate the use of successor and predecessor.

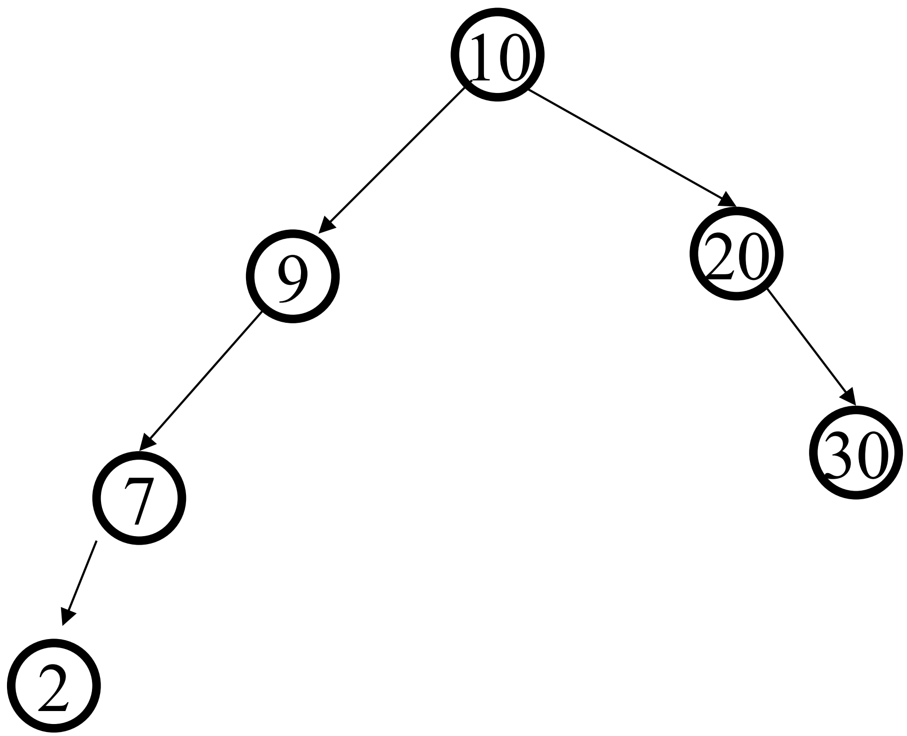
Deletion by merging

- Still need to find successor (or predecessor)
- Make the successor the parent of left subtree. (or make predecessor the parent of right subtree)
- make 7 the parent of left subtree

Delete(5)

Parent points to the right subtree.




10
9
20
7
30
2

Deletion by merging

- Problem
  - May increase or decrease the height after deletion

```
template<class T>
void BST<T>::findAndDeleteByMerging(const T& el)
{
   BSTNode<T> *node = root, *prev = 0;
   while (node!=0)
       if (node->key == el)
             break;
       prev  = node;
       if (node -> key < el)
              node = node-> right;
       else
             node = node->left;
  if (node!=0 && node->key == el)
        if (node == root)
            deleteByMerging(root);
       else  if (prev->left == node)
           deleteByMerging(prev->left);
       else
            deleteByMerging(prev->right);
  else if (root !=0)
        cout << "key" << el << "not found";
   else   cout << "the tree is empty":
 }
```

```
template<class T>
void BST<T>::deleteByMerging(BSTNode<T>*& node)
{
   BSTNode<T> *tmp = node;
   if (node !=0)
        //node has no right child: its left child (if any) is attached to its parent
        if (node->right == 0)
               node = node->left;
        //node has no left child: its right child is attached to its parent.
        else if (node->left == 0)
                node = node->right;
        else            //merge subtree
        {    tmp  =  node -> left;          //1. get predecessor
             while (tmp->right != 0)   tmp = tmp->right;         // 2. tmp is pointing to the predecessor
             // 3. establish link between predecessor and the right subtree.
             tmp->right = node-> right;
             tmp = node;            // 4. tmp is now pointing to the original node (to be removed)
             node = node->left;   // 5. node is its left child (which connects with the parent of original node)
         }
         delete tmp;   // 6. remove the original node
  }
}
```

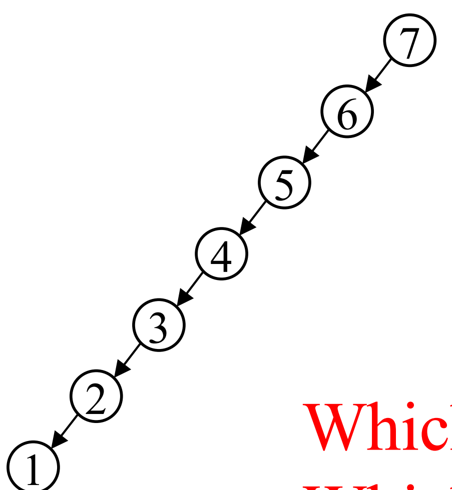
Cost of the Operations

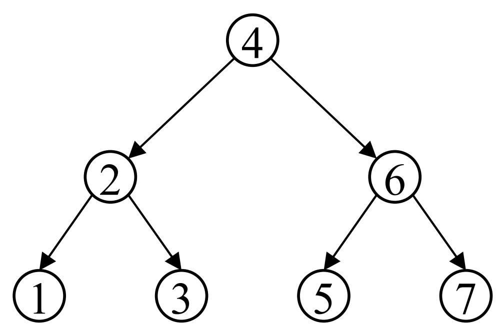
- find, insert, delete : $\boxed{\text{time} = O(\text{height}(T))}$

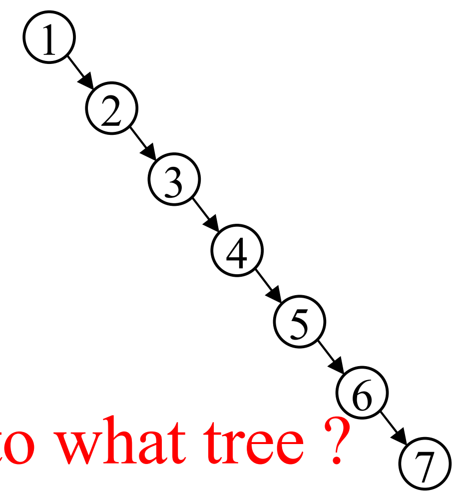
- For a tree T with n nodes:
  - height(T) $\leq$ n and
  - height(T) $\geq$ log(n) or more precisely, floor(log(n)) +1

Height of the BST

- Height depends critically on the order in which we insert the data:
  - E.g. 1,2,3,4,5,6,7 or 7,6,5,4,3,2,1, or 4,2,6,1,3,5,7







Which insertion order corresponds to what tree ?
Which tree do we prefer and why ?

Balancing the tree (globally)

- Definition:
  – A balanced tree: If the difference in height of both subtrees of any node in the tree is either 0 or 1.
- Globally balancing a tree (AVL tree is locally balanced, later)
  – Sort them in an array, the middle element is used as root.
  – Then recursively build the tree.

Code for global balancing

```
template<class T>
void BST<T>::balance(T data[], int first, int last)  {
If (first <= last) {
   int middle = (first + last)/2;
   insert(data[middle]);
   balance(data,first, middle-1);
   balance(data,middle+1, last);
  }
}
```

Example Operations on Binary Trees

- Height
- Size
- # of Leaves
- Delete the tree

Height of a Binary Tree

- The height can be found by performing a *postorder traversal*.
  - The height of the left subtree is determined;
  - then the height of the right subtree is determined.
  - During the "visit" step, the height of the tree is determined as:
    - height = maximum{leftHeight, rightHeight} + 1

Pseudo Code for Getting Height

```
if(ptr is null)
    return 0
endif
leftHeight = height(ptr->leftChild)
rightHeight = height(ptr->rightChild)
if (leftHeight > rightHeight)
return
        leftHeight + 1
else return
        rightHeight + 1
endif
```

Size of a Binary Tree

- *Any of the four traversal methods* can be used, since each method visits each node exactly one time.
- Pseudo code using width-1st (or level-order) traversal:

while (there are elements in the tree)
    count++
    if (there is a left child)
      push onto a queue
    endif
    if (there is a right child)
      push onto a queue
    endif
    pop the queue
endwhile

Find # of leaves

- Recursive or iterative -- *Any traversal methods* can be used.
- Example of using iterative level-order traversal:

```
while (there are elements in the tree)
    if (leftChild is null and rightChild is null)
          leavesCount ++;
    if (there is a left child)
          push onto a queue
    endif
    if (there is a right child)
          push onto a queue
    endif
    pop the queue
endwhile
```

Deleting a Binary Tree

- Perform a *post-order traversal*. The node is deleted during "visit" step.
- Pseudo codes:
  – Pubic method:

delete()

delete(root)

set root to null

- The private recursive method:

delete (pointer to a tree node)

    If (pointer is not null)
        delete (pointer to left child)
        delete (pointer to right child)
        delete pointer
    End if