Data Structures

IT 205

Dr. Manish Khare

Lecture – 20
06-Mar-2018

Binary Search Tree

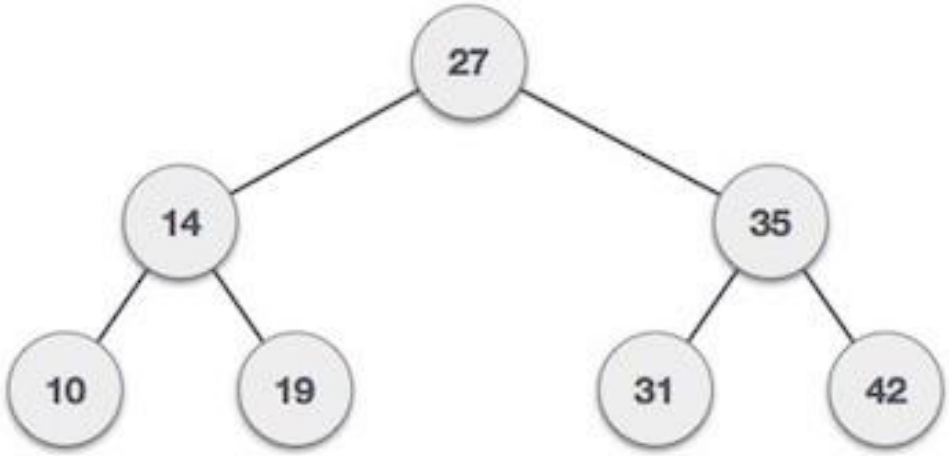
Binary Search tree

- 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 −

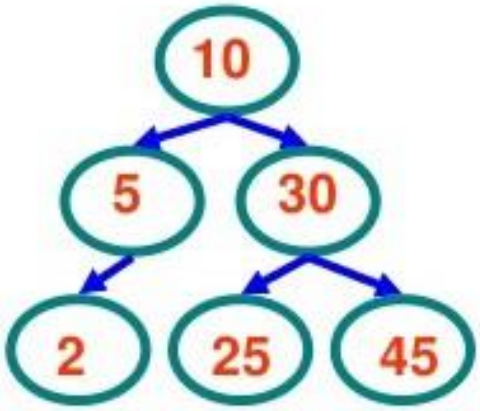
$$\text{left\_subtree (keys)} \leq \text{node (key)} \leq \text{right\_subtree (keys)}$$

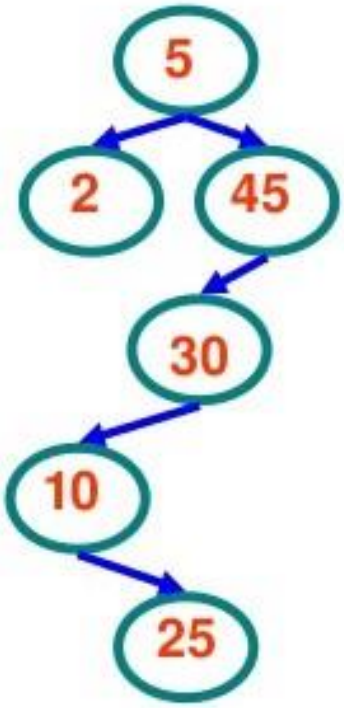
BST Representation

- 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.

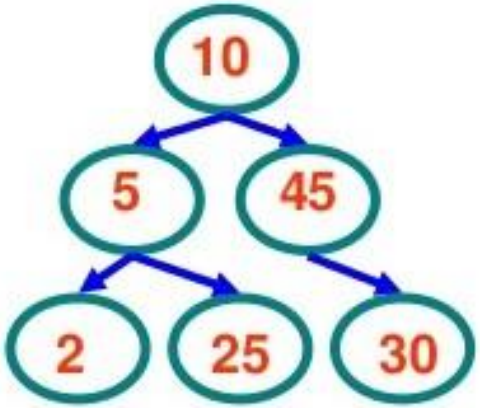


Examples





Binary search trees



Not a binary search tree

Difference between BT and BST

- A binary tree is simply a tree in which each node can have at most two children.
- A binary search tree is a binary tree in which the nodes are assigned values, with the following restrictions.
  - No duplicate values.
  - Left subtree of a node can only have values less than the node.
  - Right subtree of a node can only have values greater than node.
  - The left and right subtree of a node is a binary search tree.

Operations in BST

- 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.

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.

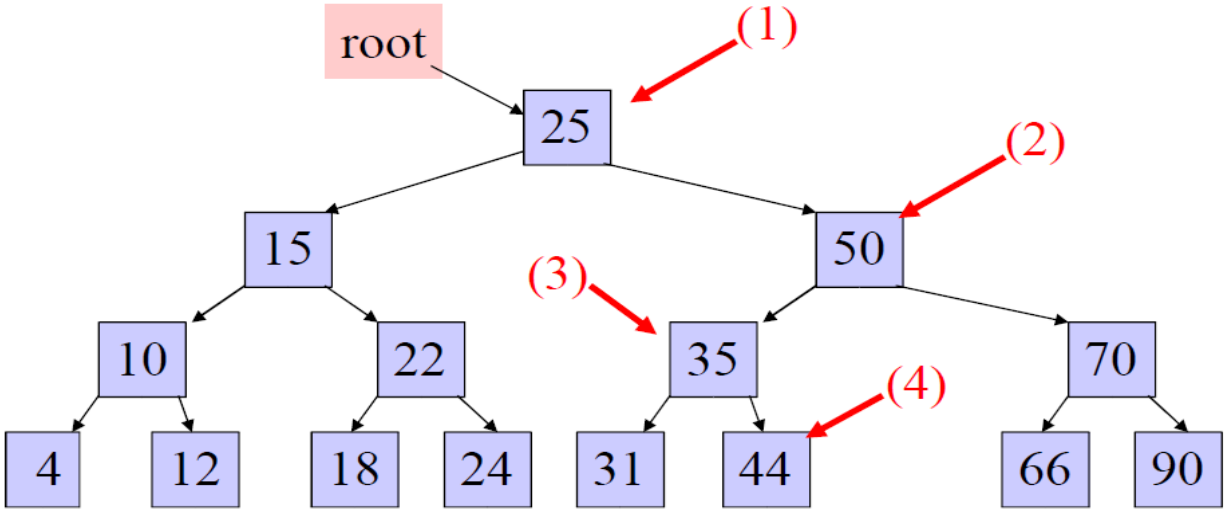
- Search for a matching node
  - 1. Start at the root node as current node
  - 2. If the search key's value matches the current node's key then found a match
  - 3. If search key's value is greater than current node's
    - 1. If the current node has a right child, search right
    - 2. Else, no matching node in the tree
  - 4. If search key is less than the current node's
    - 1. If the current node has a left child, search left
    - 2. Else, no matching node in the tree

```
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;
}
```

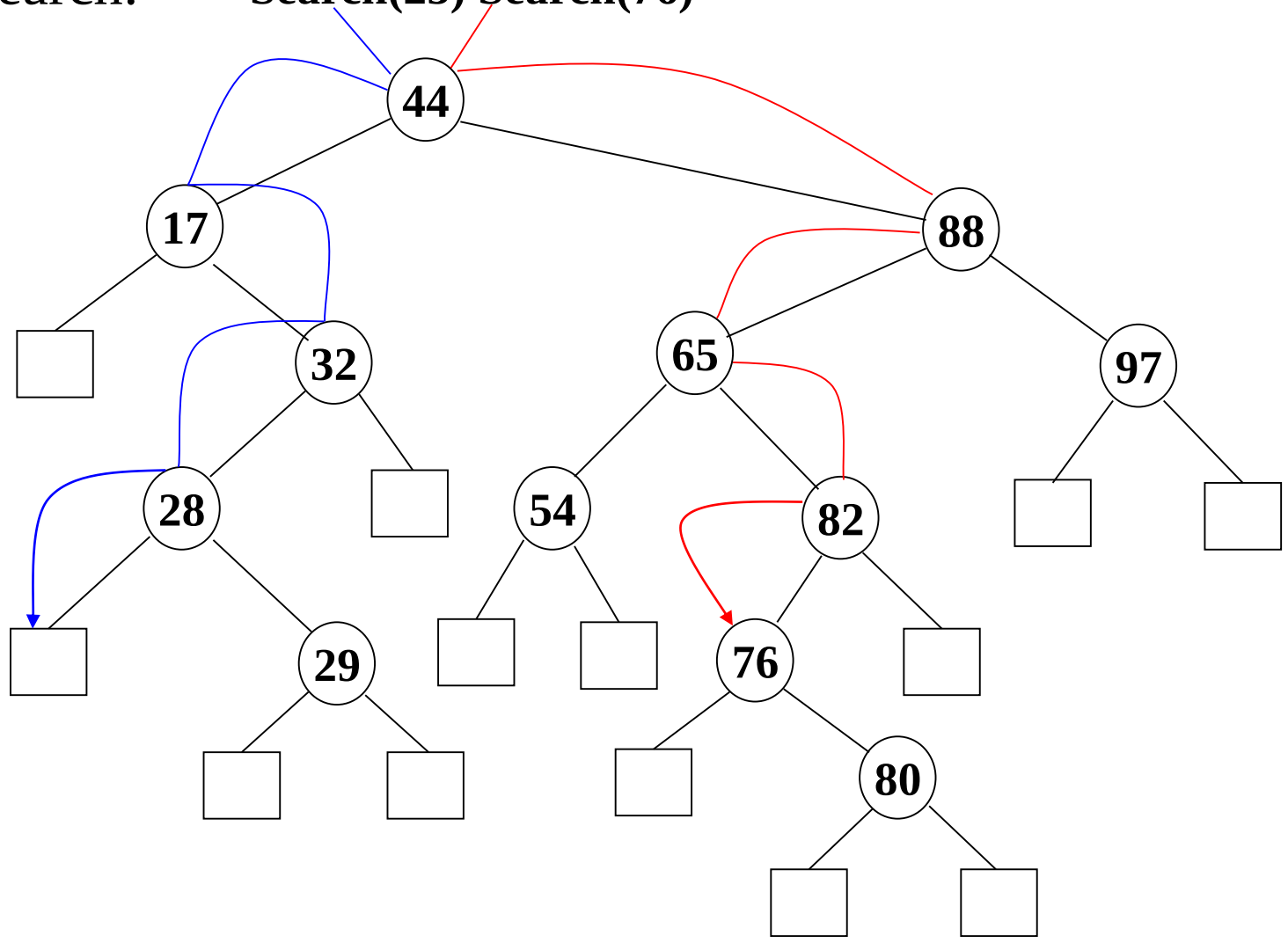
Example: search for 45 in the tree

(key fields are show in node rather than in separate obj ref to by data field):

1. start at the root, 45 is greater than 25, search in right subtree
2. 45 is less than 50, search in 50's left subtree
3. 45 is greater than 35, search in 35's right subtree
4. 45 is greater than 44, but 44 has no right subtree so 45 is not in the BST



Search: **Search(25) Search(76)**



Insert Operation

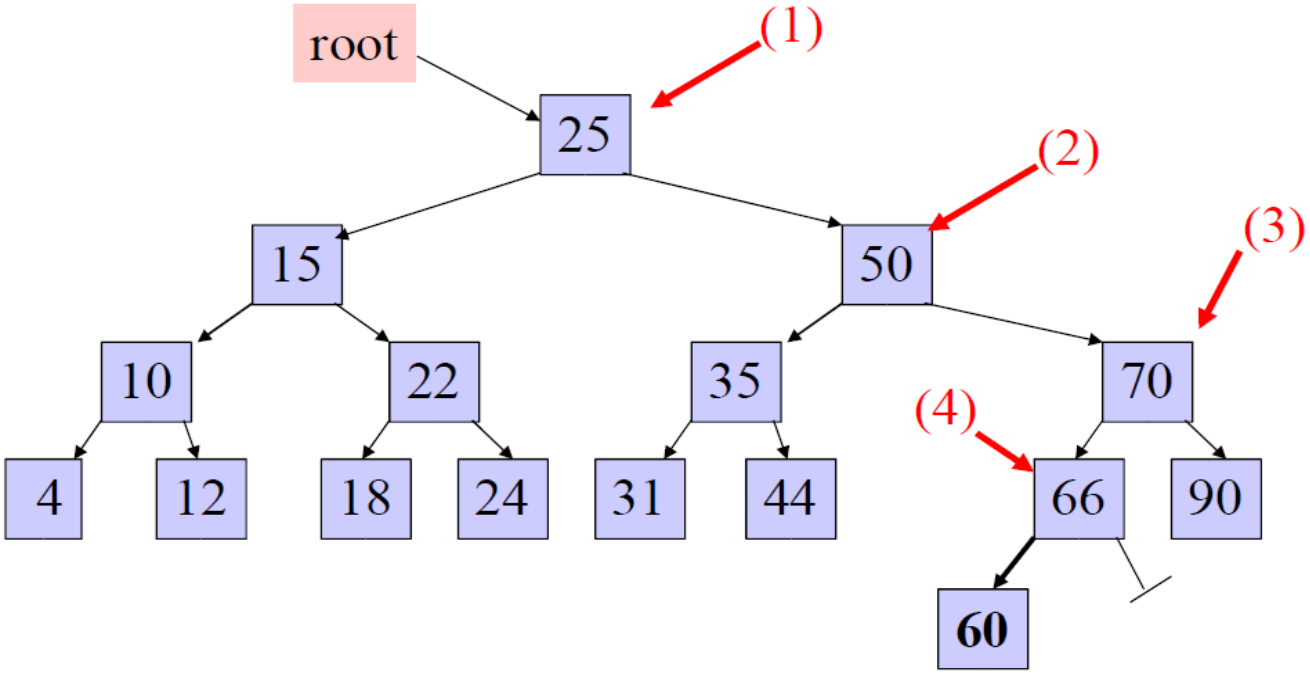
- Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the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

- Always insert new node as leaf node

  - 1. Start at root node as current node
  - 2. If new node's key < current's key
    - 1. If current node has a left child, search left
    - 2. Else add new node as current's left child
  - 3. If new node's key > current's key
    - 1. If current node has a right child, search right
    - 2. Else add new node as current's right child

Example: insert 60 in the tree:

1. start at the root, 60 is greater than 25, search in right subtree
2. 60 is greater than 50, search in 50's right subtree
3. 60 is less than 70, search in 70's left subtree
4. 60 is less than 66, add 60 as 66's left child

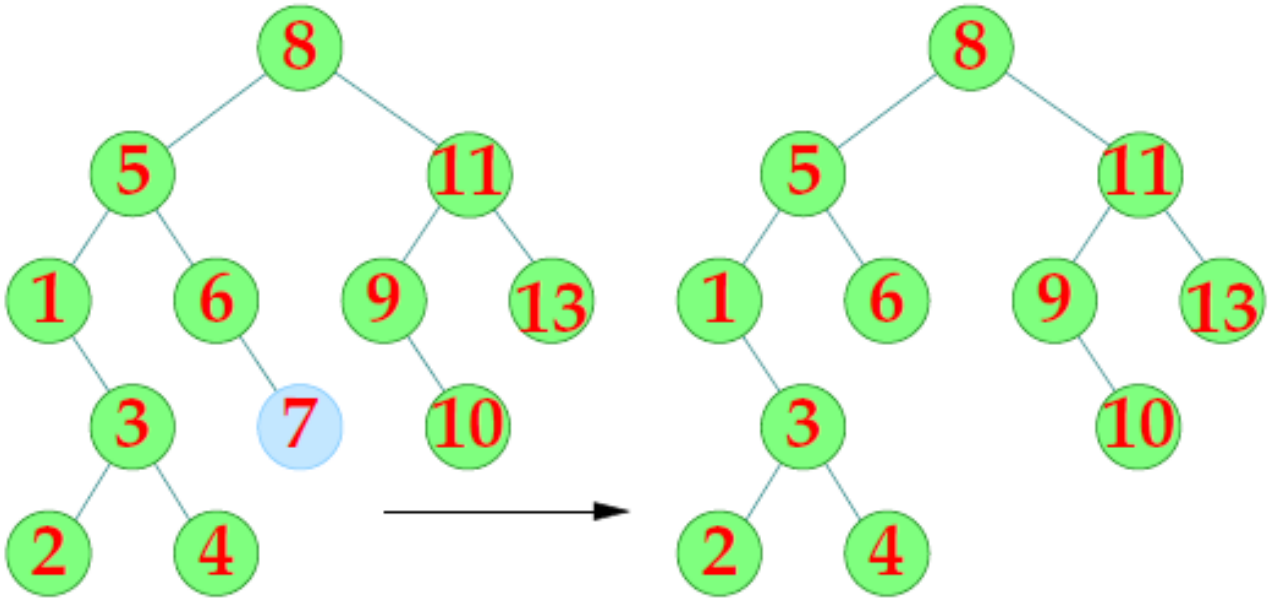


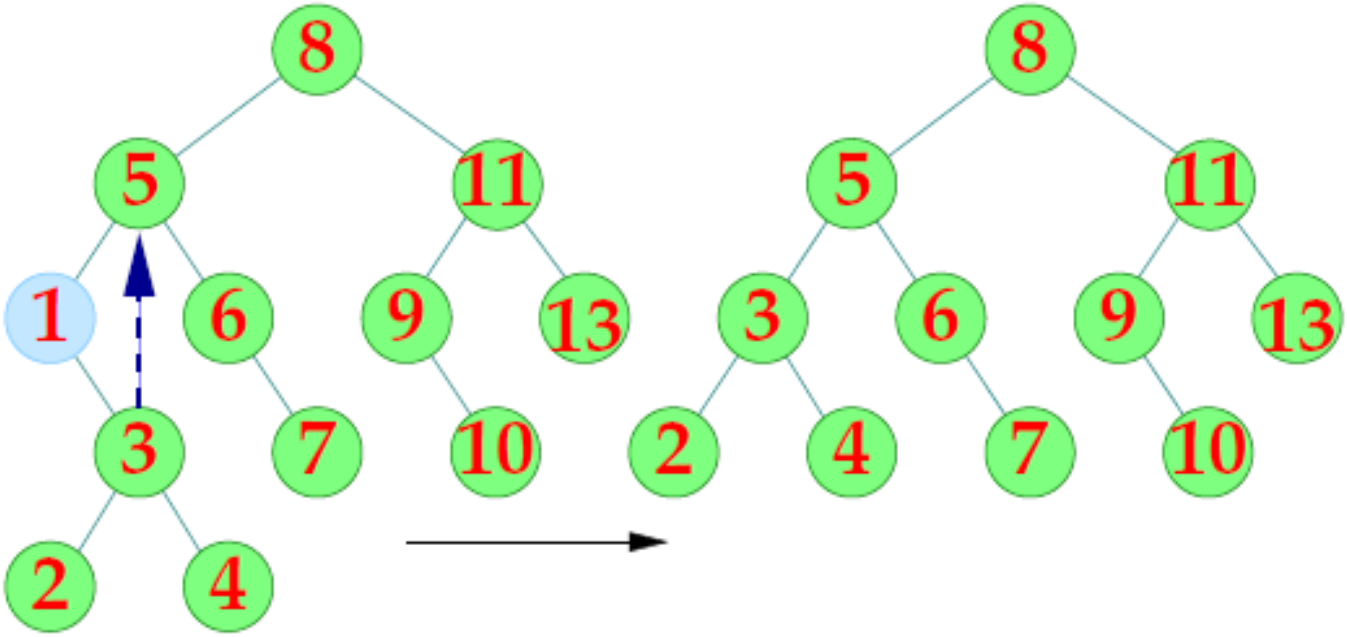
Traversal Operation

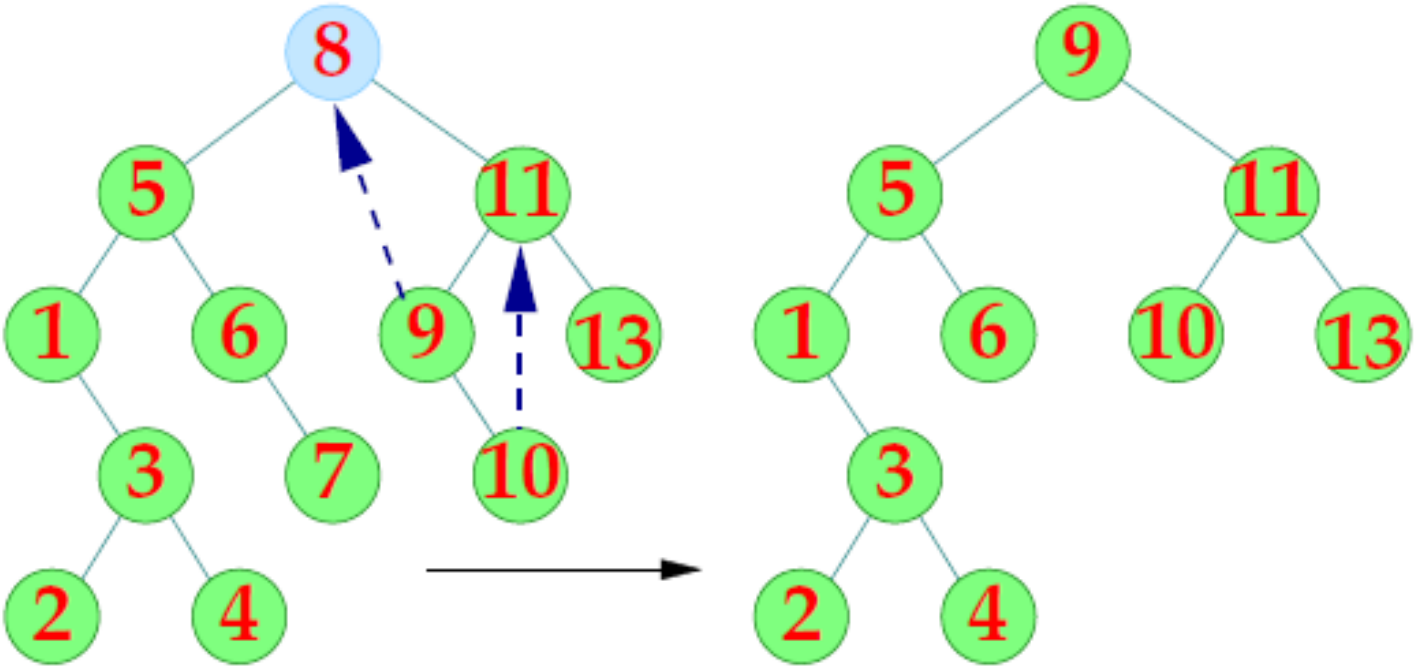
- Same as normal Tree Traversal
  - Pre order
  - In-order
  - Post order

Deletion Operation

- Suppose we want to delete a node z.
  - 1. If z has no children, then we will just replace z by nil.
  - 2. If z has only one child, then we will promote the unique child to z's place.
  - 3. If z has two children, then we will identify z's successor. Call it y. The successor y either is a leaf or has only the right child. Promote y to z's place. Treat the loss of y using one of the above two solutions.


8
5
11
1
6
9
13
3
7
10
2
4
8
5
11
1
6
9
13
3
10
2
4


8
5
11
1
6
9
13
3
7
10
2
4
8
5
11
3
6
9
13
2
4
7
10


8
5
11
1
6
9
13
3
7
10
2
4
9
5
11
1
6
10
13
3
2
4

- BST-Delete(T, z)
  - 1: if left[z] = nil or right[z] = nil
  - 2: then y ← z
  - 3: else y ← BST-Successor(z)
  - 4: ✂ y is the node that's actually removed.
  - 5: ✂ Here y does not have two children.
  - 6: if left[y] ≠ nil
  - 7: then x ← left[y]
  - 8: else x ← right[y]
  - 9: ✂ x is the node that's moving to y's position.
  - 10: if x ≠ nil then p[x] ← p[y]
  - 11: ✂ p[x] is reset If x isn't NIL.
  - 12: ✂ Resetting is unnecessary if x is NIL.

- 13: if p[y] = nil then root[T] ← x
- 14: ✂ If y is the root, then x becomes the root.
- 15: ✂ Otherwise, do the following.
- 16: else if y = left[p[y]]
- 17: then left[p[y]] ← x
- 18: ✂ If y is the left child of its parent, then
- 19: ✂ Set the parent's left child to x.
- 20: else right[p[y]] ← x
- 21: ✂ If y is the right child of its parent, then
- 22: ✂ Set the parent's right child to x.
- 23: if y ≠ z then
- 24: { key[z] ← key[y]
- 25: Move other data from y to z }
- 26: return (y)

Height of a binary search tree

- The height of a binary search tree with n elements can become as large as n.
- It can be shown that when insertions and deletions are made at random, the height of the binary search tree is $O(\log_2 n)$ on the average.
- Search trees with a worst-case height of $O(\log_2 n)$ are called *balance search trees*

Ranked Binary Search Tree

- In ranked binary search tree, each node of the tree has an additional field name “leftsize”, which is one plus number of elements in the left subtree of the node.

- The algorithm searches for the K-th smallest element in ranked tree in O(h) time where h means height of the tree.

```
Algorithm Search K(k)
  {
    Found=false
    t=tree
    while((t ≠ 0) and not found) do
     {
       if (k=(t → leftsize)) then
           found=true
       else if (k<(t → leftsize)) then
           t = t → l_child
      else
{
   k=k – (t → leftsize)
    t = t → r_child
}
}
if (not found) then
     return 0
else
     return 1
}
```

Threaded Binary Tree

Binary Tree

Do you find any drawback of the previous tree?

Too many null pointers in current representation of binary trees

n: number of nodes

number of non-null links: n-1

total links: 2n

null links: 2n-(n-1) = n+1