Binary trees, and Binary Search Trees

Implementation based on linked list (class)

```
#include <bits/stdc++.h>
using namespace std;

class Node {
  public:
    int data;
    Node* left;
    Node* right;

    // Val is the key or the value that
    // has to be added to the data part
    Node(int val)
    {
        data = val;

        // Left and right child for node
        // will be initialized to null
        left = NULL;
        right = NULL;
    }
};
```

Populating the tree (slopy)

```
#include <bits/stdc++.h>
using namespace std;

class Node {
  public:
    int data;
    Node* left;
    Node* right;

    // Val is the key or the value that
    // has to be added to the data part
    Node(int val)
    {
        data = val;

        // Left and right child for node
        // will be initialized to null
        left = NULL;
        right = NULL;
    }
};
```

```
int main()
{

    /*create root*/
    Node* root = new Node(1);
    /* following is the tree after above statement

              1
             / \
          NULL   NULL
    */

    root->left = new Node(2);
    root->right = new Node(3);
    /* 2 and 3 become left and right children of 1
                  1
                /   \
               2      3
              / \     / \
          NULL NULL NULL NULL
    */

    root->left->left = new Node(4);
    /* 4 becomes left child of 2
              1
            /    \
           2      3
          / \     / \
        4 NULL NULL NULL
       / \
     NULL NULL
    */

    return 0;
}
```

Binary Search Trees (specific type of trees)

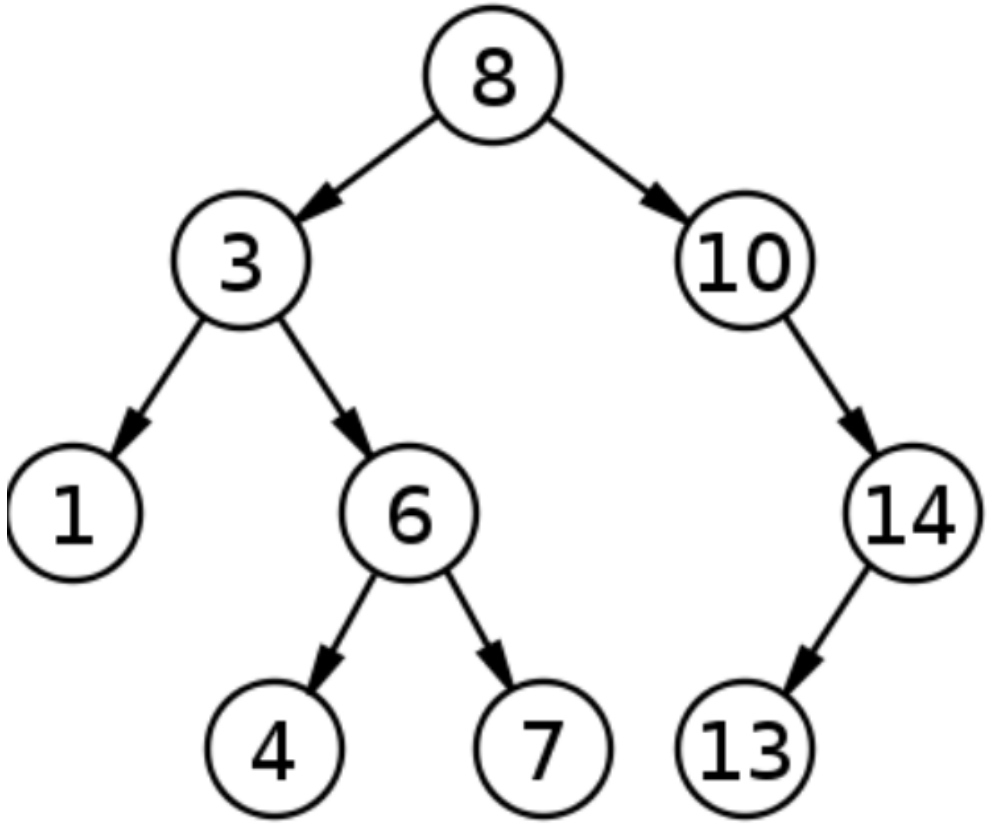
- For each node p, all the values stored in the left subtree of p are LESS than the value stored in p.
- all the values stored in the right subtree of p are GREATER than the value stored in p.
- Searching for a node is super fast!
- Normally, if we search through n nodes, it takes O(n) time

But using binary search tree:

- This ordering property of the tree tells us where to search
- We choose to look to the left OR look to the right of a node
- We are HALVING the search space O(log n) time

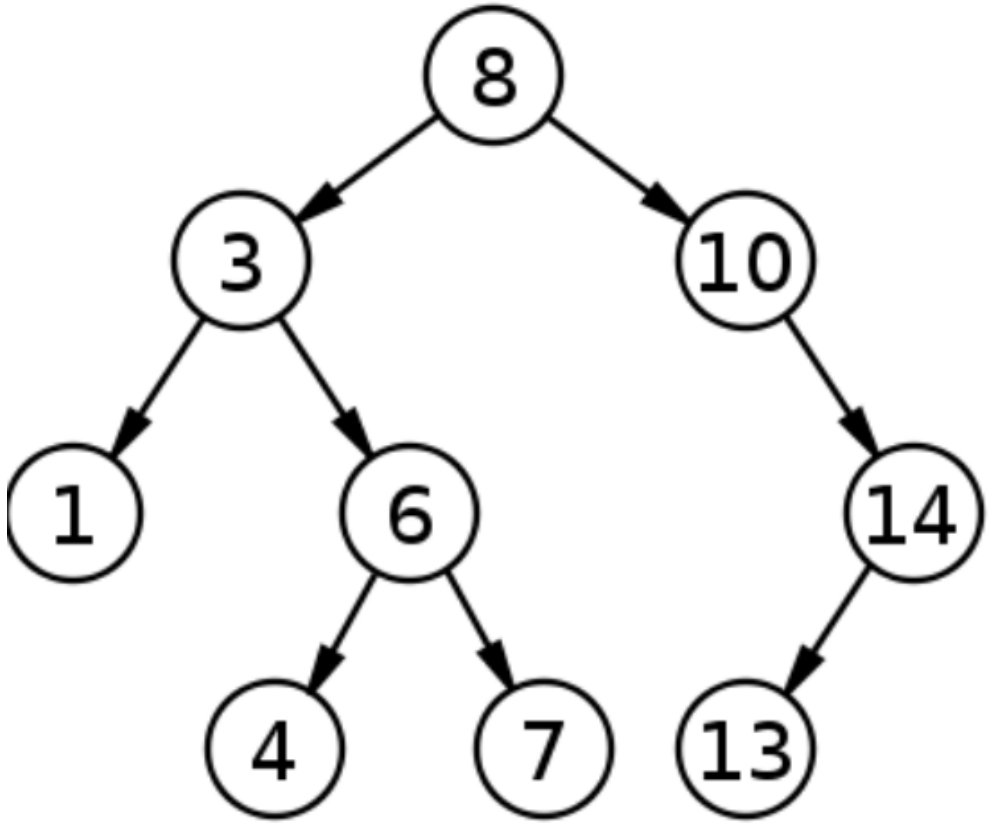
Binary Search Tree (type of binary tree)

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

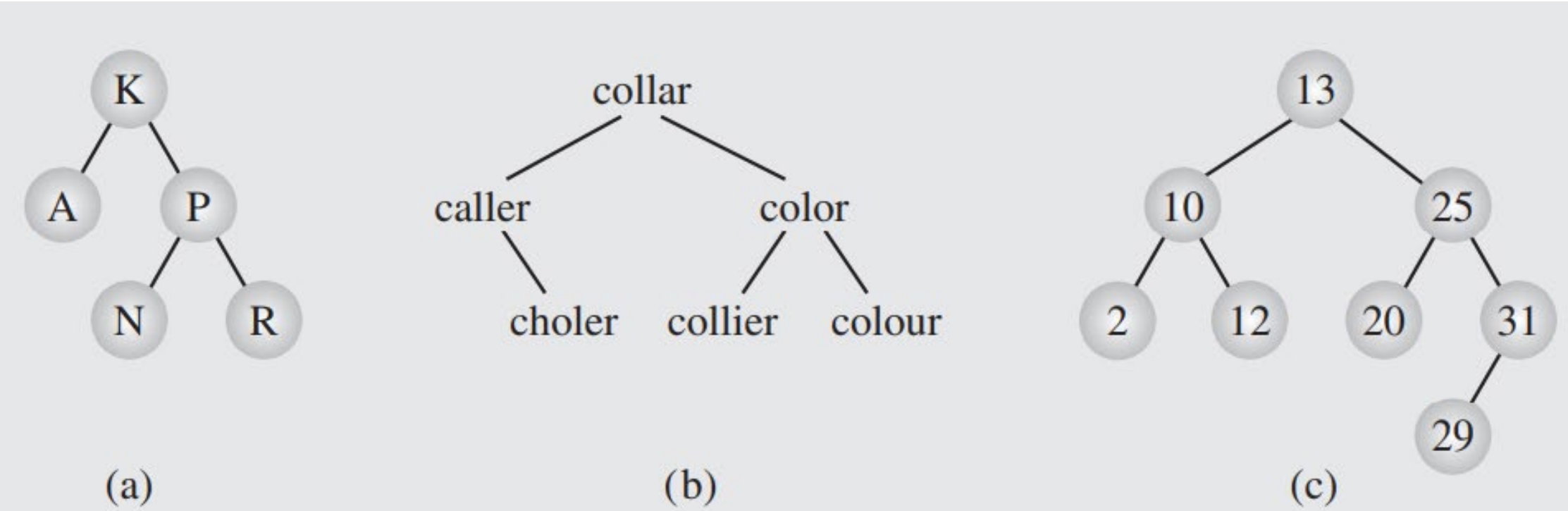


Binary Search Tree

Algorithm	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$

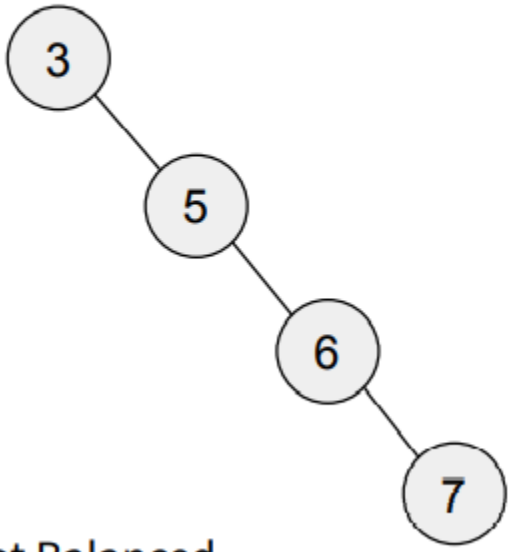


Examples of BST

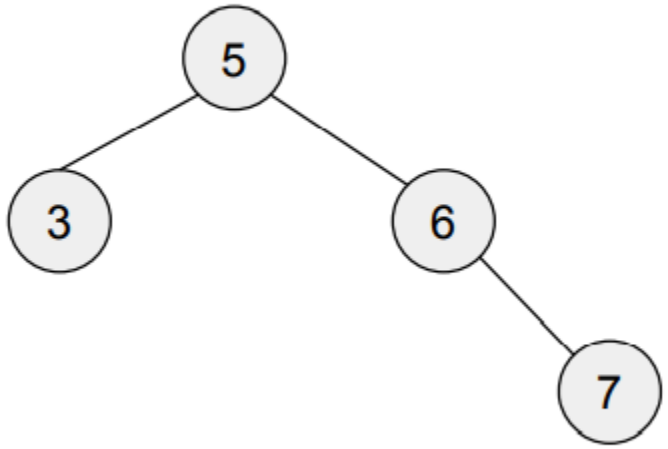


Binary Search trees efficient if they are balanced

A binary tree is balanced if the difference of depth of any two leaves is at most 1.



Not Balanced



Balanced

BST operations and implementation

Basic Operation On Binary Tree:

- Inserting an element.
- Removing an element.
- Searching for an element.
- Traversing an element. There are three types of traversals in a binary tree which will be discussed ahead.

Auxiliary Operation On Binary Tree:

- Finding the height of the tree
- Find the level of the tree
- Finding the size of the entire tree.

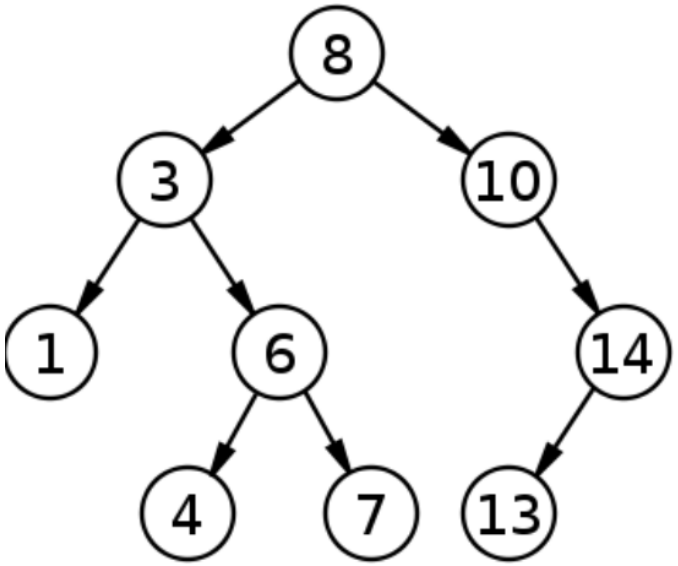
BST searching for an element

start at the root, and then we compare the value to be searched with the value of the root, if it's equal we are done with the search if it's smaller we know that we need to go to the left subtree because in a binary search tree all the elements in the left subtree are smaller and all the elements in the right subtree are larger.

```
struct node* search(struct node* root, int key)
{
    // Base Cases: root is null or key is present at root
    if (root == NULL || root->key == key)
       return root;

    // Key is greater than root's key
    if (root->key < key)
       return search(root->right, key);

    // Key is smaller than root's key
    return search(root->left, key);
}
```



BST Inserting an element

EX: Let's assume we insert the following data values, in their order of appearance into an initially empty BST:

- 10, 14, 6, 2, 5, 15, and 17

What are the steps?

BST Inserting an element

- Create new node for each item.
- Find a parent node.
- Attach new node as a leaf.
- You can find the appropriate position using recursion or loop.

BST Inserting an element

How to build a BST? How to insert nodes properly?

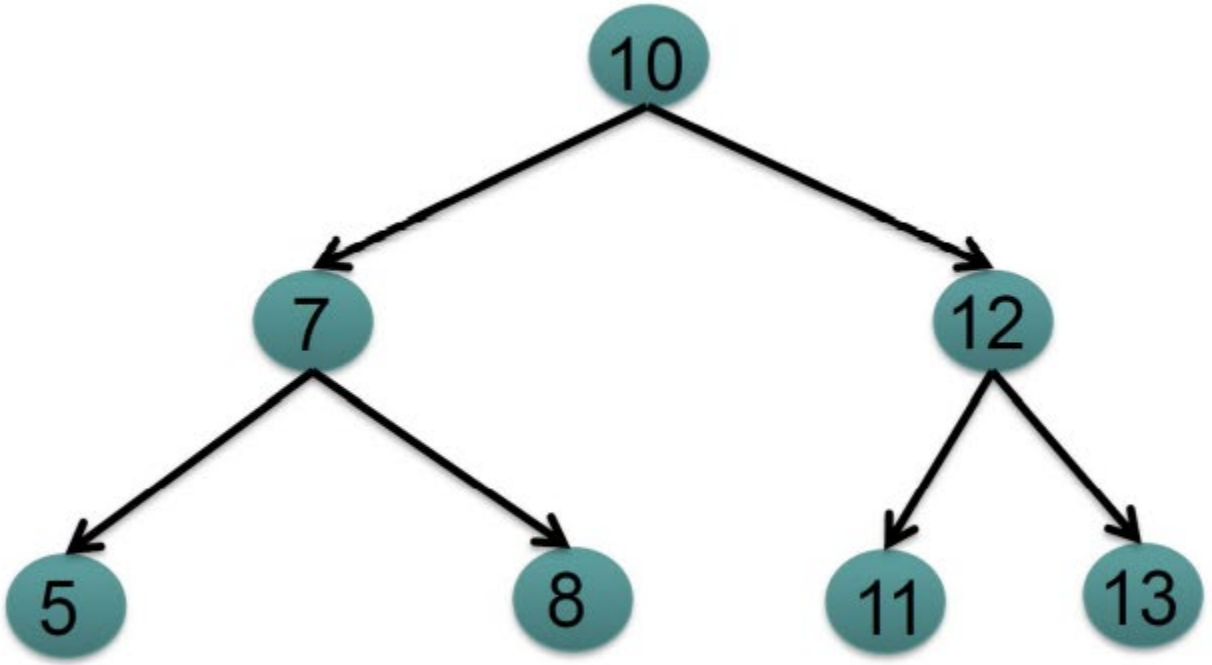
- First: Create the node!

- BSTs must maintain their ordering property
  → Smaller items to the left of any given root : Go left if the new value is less than the root

  → greater items to the right of that root : Go right if the new value is greater than the root

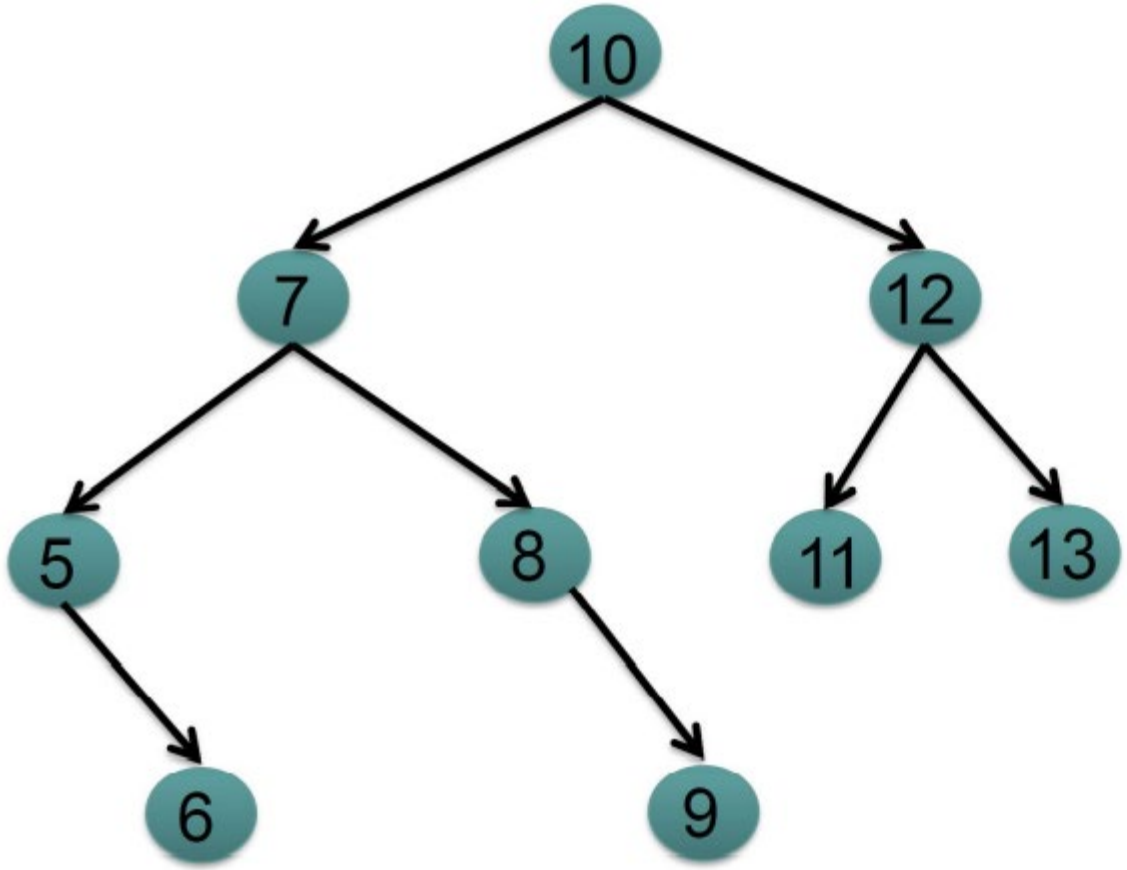
- Keep doing this till you come to an empty position

BST Inserting an element (before)



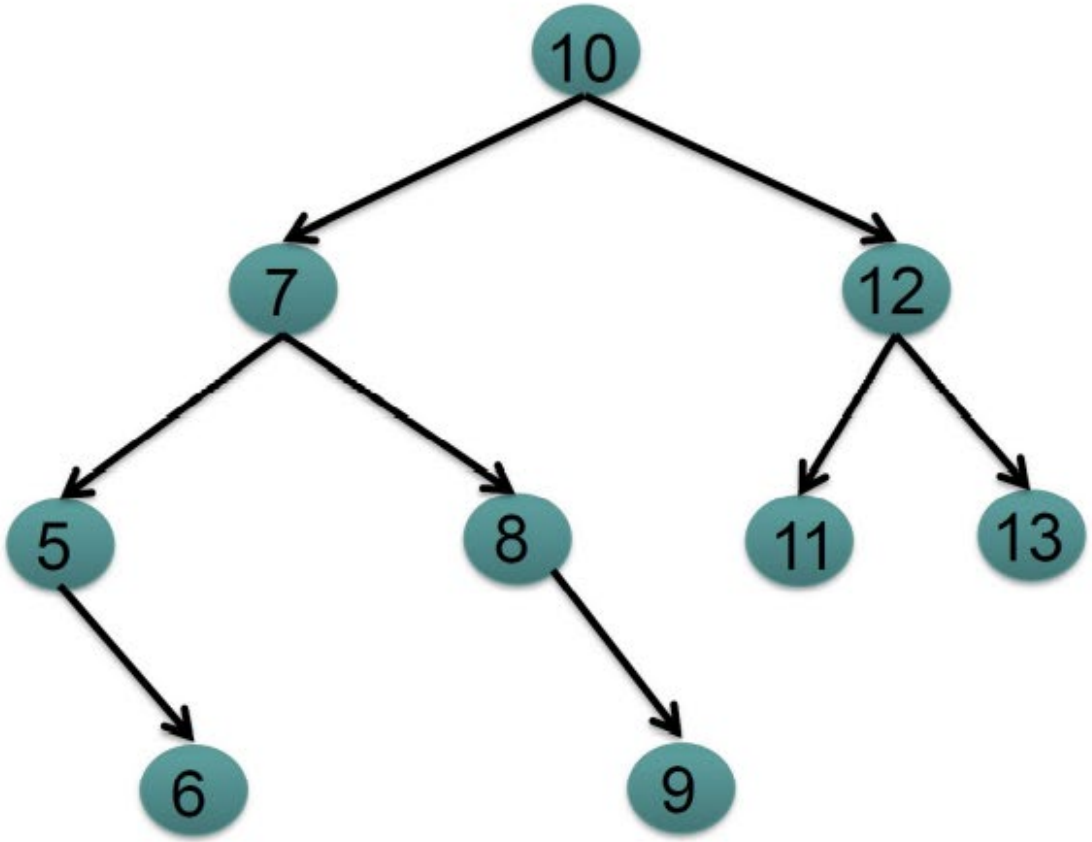
Insert(6);
Insert(9);

BST Inserting an element (after)



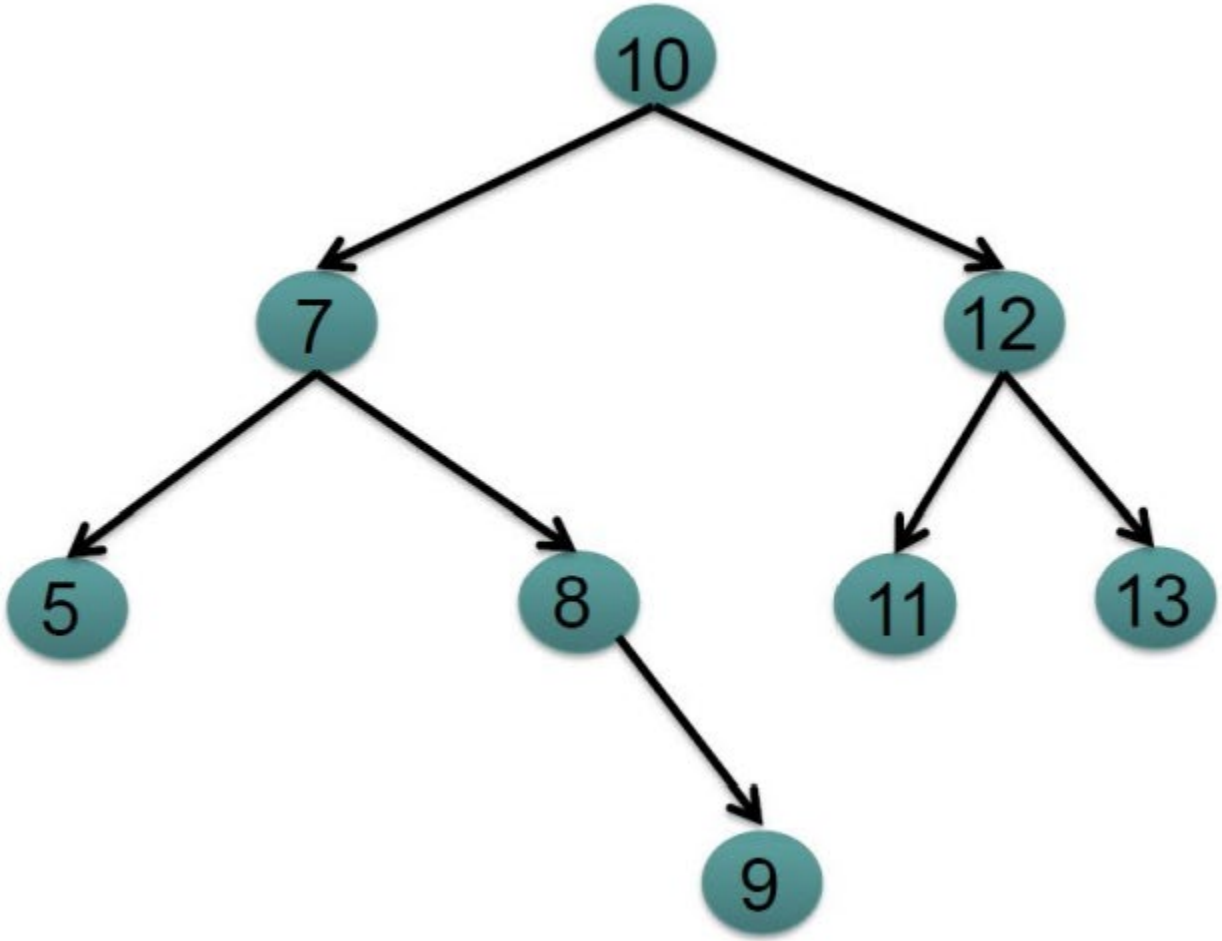
Insert(6);
Insert(9);

BST Delete an element (before)



Case 1 : If it is a leaf (has no child).
delete(6);

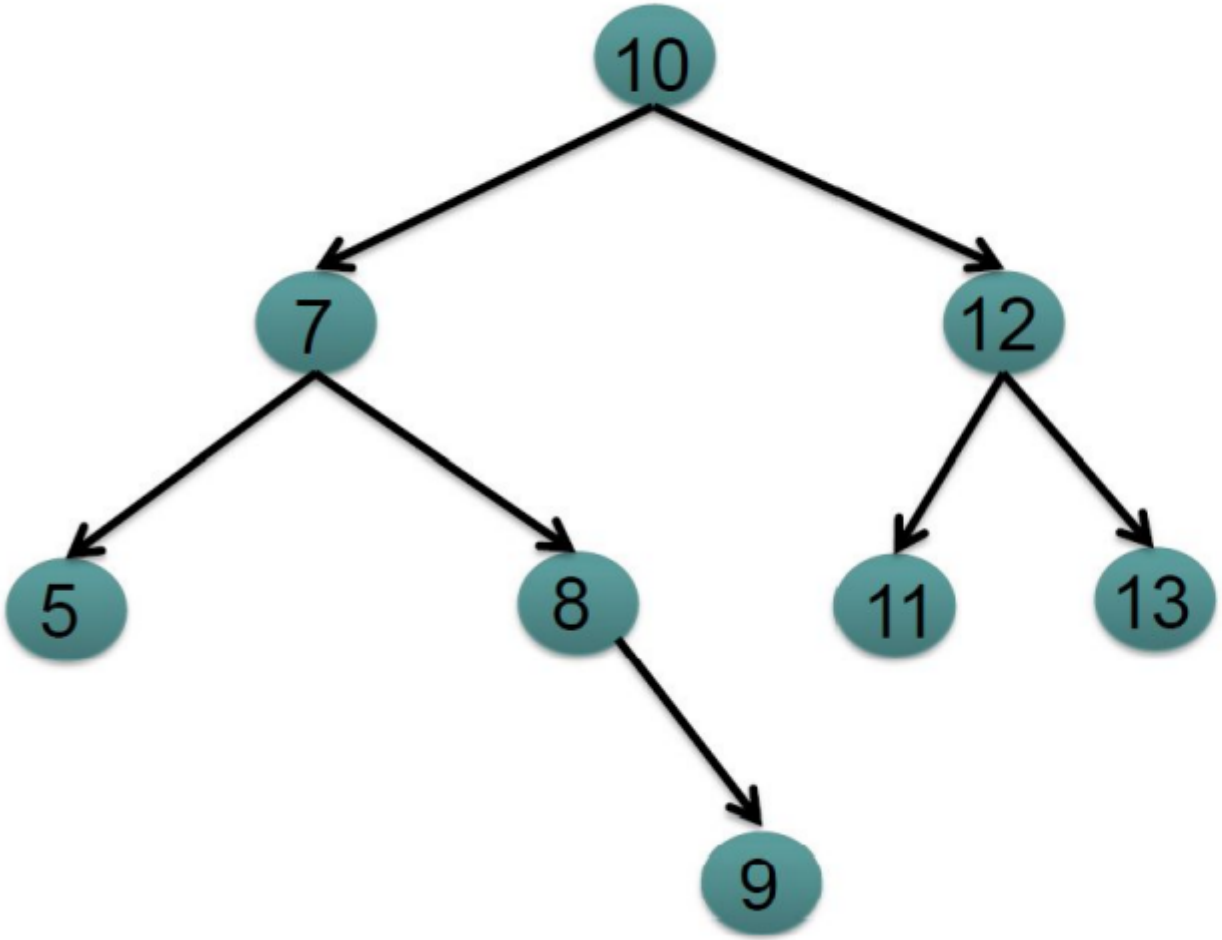
BST Inserting an element (delete)



Case 1 : If it is a leaf (has no child).
delete(6);

Simply remove
it from the tree.

BST delete after

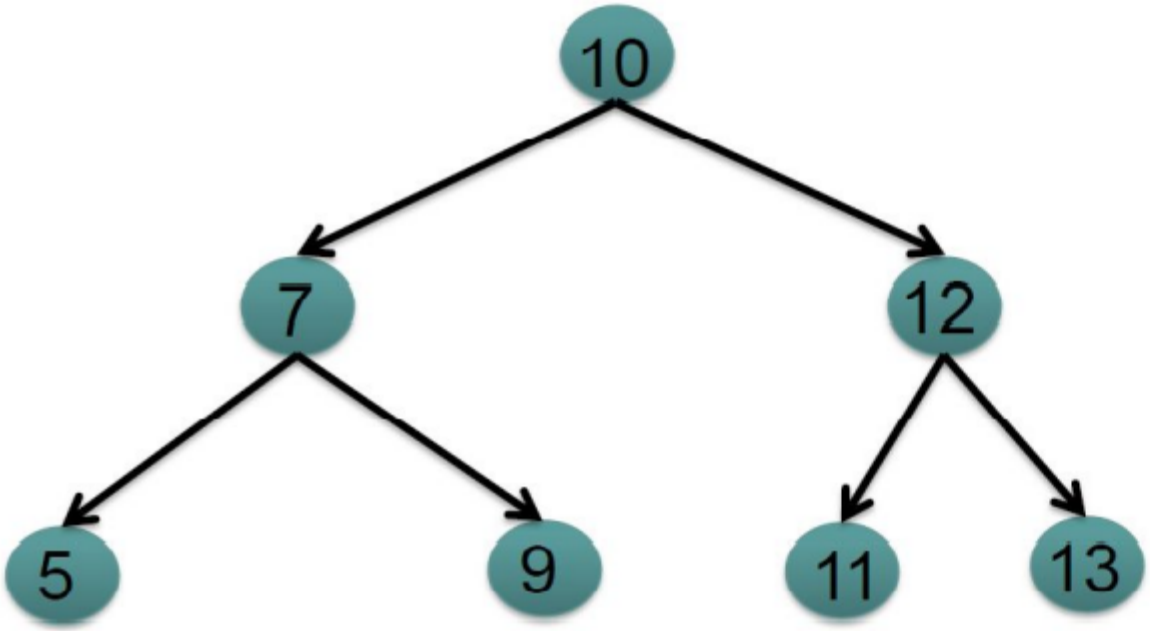


Case 2 : If it has only one child.

Delete(8);

Copy the data of child to the node and remove the child node.

BST delete after

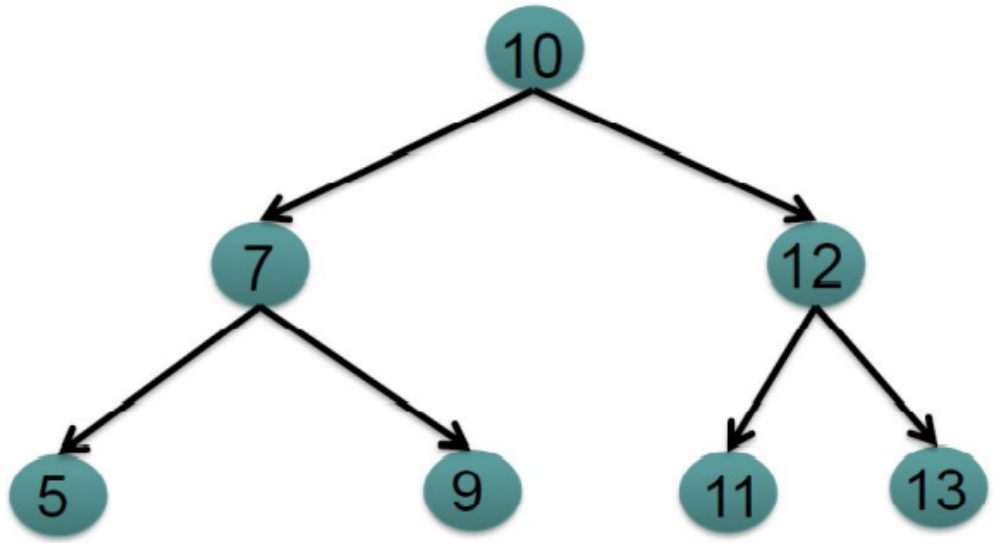


Case 2 : : If it has only one child.

Delete(8);

Copy the data of child to the node and remove the child node.

BST delete before



Case 3 : : If it has two children.

Delete(10);

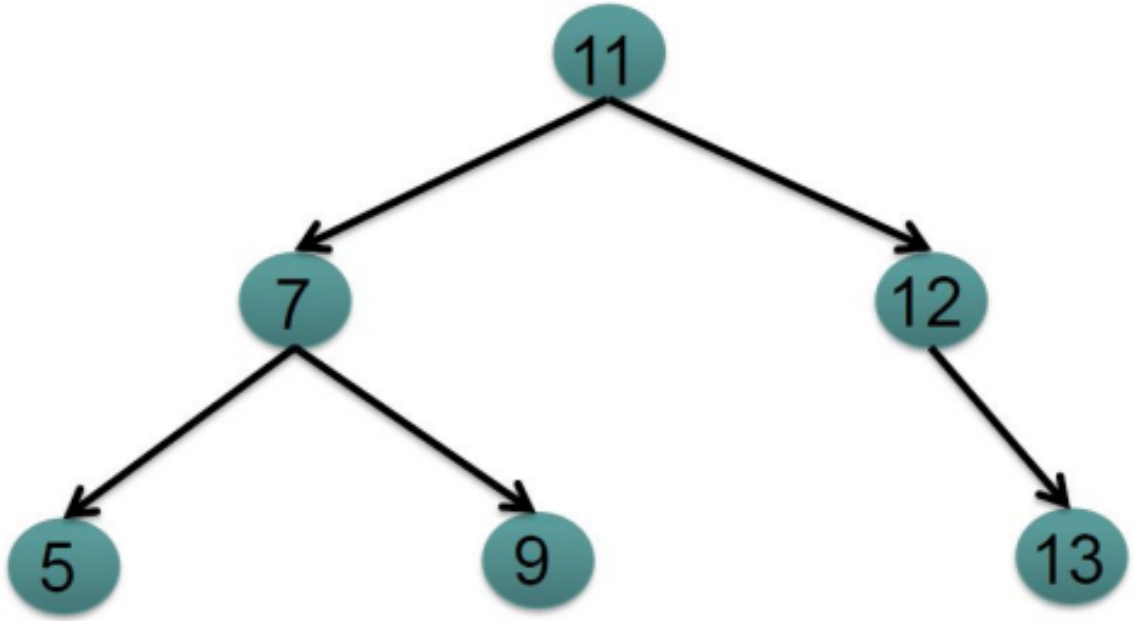
Find the minimum element of its right subtree which is 11 here.

Copy the data of found node to the node and remove the found node.

More general, find the inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

In Binary Search Tree, Inorder Successor of an input node can also be defined as the node with the smallest key greater than the key of the input node. So, it is sometimes important to find next node in sorted order.

BST delete after



Case 3 : : If it has two children.

Delete(10);

Find the minimum element of its right subtree which is 11 here.

Copy the data of found node to the node and remove the found node.

More general, find the inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

In Binary Search Tree, Inorder Successor of an input node can also be defined as the node with the smallest key greater than the key of the input node. So, it is sometimes important to find next node in sorted order.

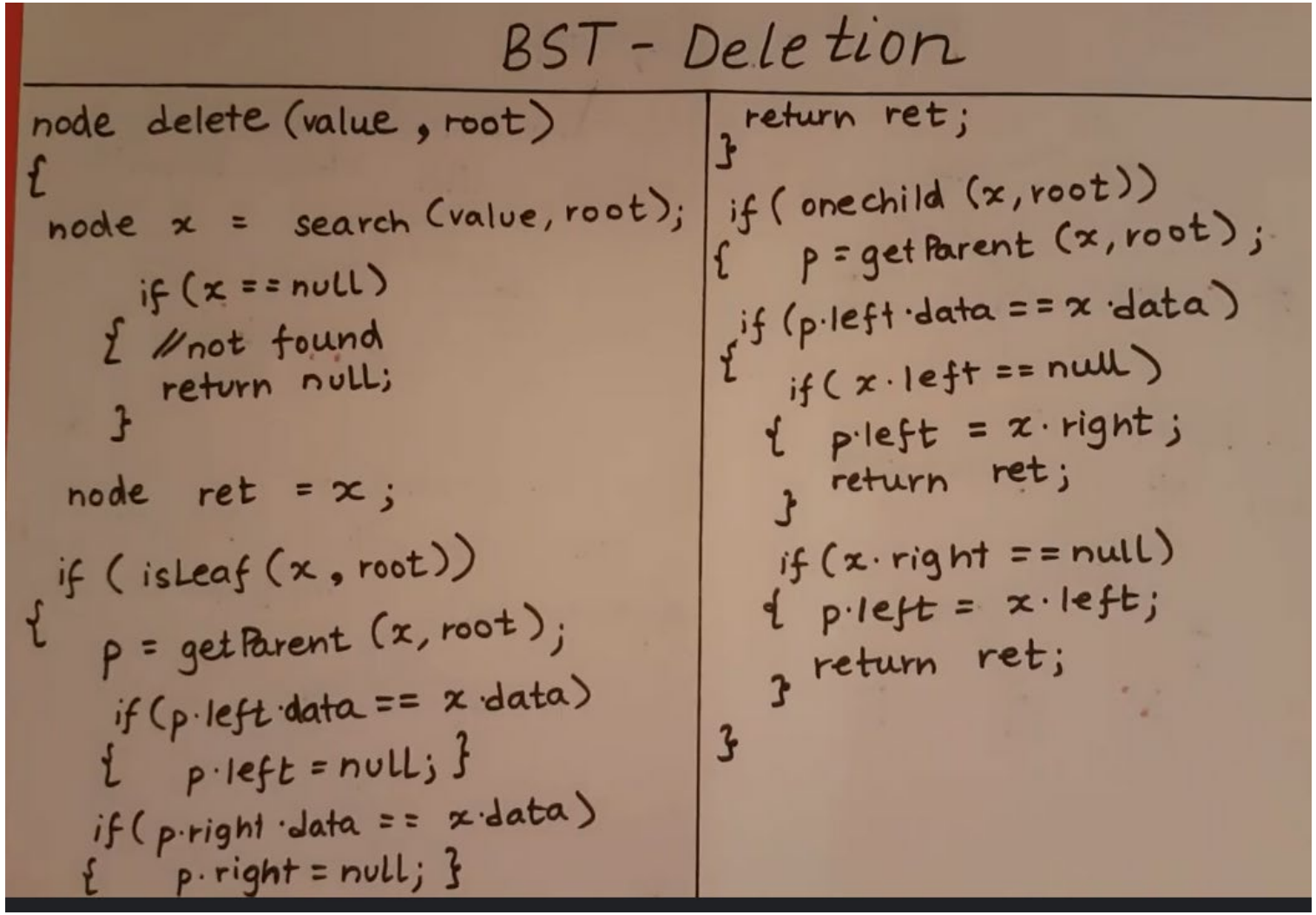
Time complexity of BST operations

Operation	Array	Linked List	Sorted Array	Balanced BST
Search(value)	Linear O(n)	Linear O(n)	O(log n)	O(log n)
Insert(value)	Constant O(1)	Linear O(n) (end of the list) Constant O(1) (head of the list)	Linear O(n)	O(log n)
Delete(value)	Linear O(n)	Linear O(n)	Linear O(n)	O(log n)

Implementations

- Delete
- Find the code in Canvas ->Modules ->Week8

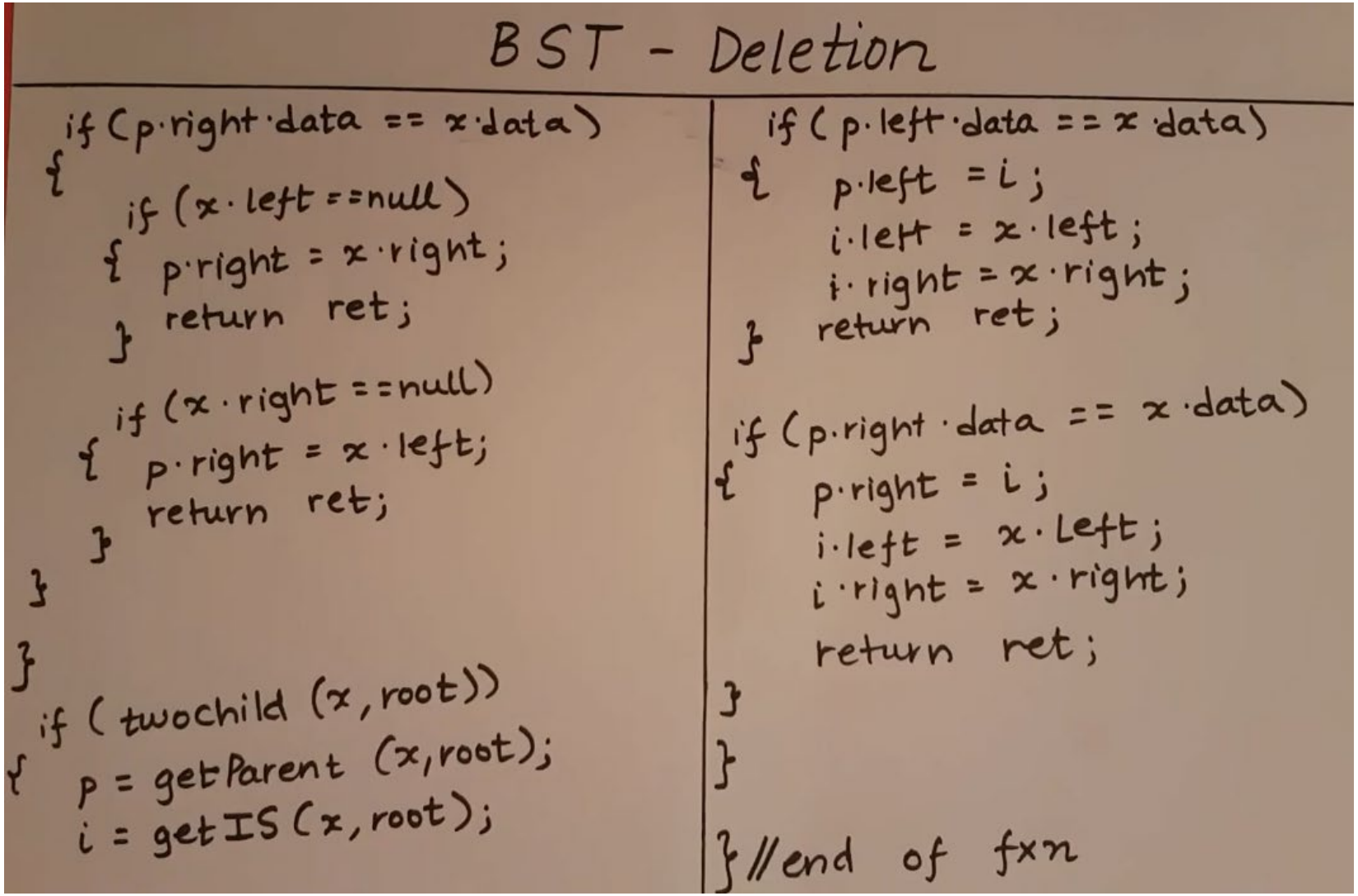
Delete



BST - Deletion

```
node delete (value , root)
{
 node x = search (value, root);
      if (x == null)
    { //not found
      return null;
    }
  node ret = x ;
  if ( isLeaf (x , root))
{
    p = getParent (x, root);
     if (p.left.data == x.data)
    {   p.left = null; }
    if ( p.right.data == x.data)
   {    p.right = null; }
  return ret;
}
 if ( onechild (x, root))
{   p = getParent (x, root);
 if (p.left.data == x.data)
{
   if ( x.left == null)
  {  p.left = x.right;
     return ret;
  }
   if (x.right == null)
  {  p.left = x.left;
     return ret;
  }
}
```

Delete


BST - Deletion
if (p.right.data == x.data)
{
if (x.left ==null)
{ p.right = x.right;
return ret;
}
if (x.right ==null)
{ p.right = x.left;
return ret;
}
}
}
if (twochild (x, root))
{ p = getParent (x,root);
i = getIS (x, root);
if (p.left.data == x.data)
{ p.left = i;
i.left = x.left;
i.right = x.right;
return ret;
}
if (p.right.data == x.data)
{ p.right = i;
i.left = x.Left;
i.right = x.right;
return ret;
}
}
}//end of fxn