CSCI 2110 Data Structures and Algorithms

Module 7: Binary Search Trees

DALHOUSIE UNIVERSITY

CSCI 2110: Module 7 Srini Sampalli

3

3

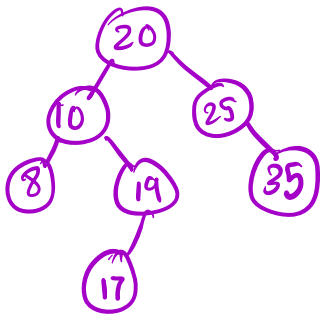
Binary Search Tree

- A binary search tree (BST) is a binary tree that is **sorted or ordered** according to a rule.
- In general, the information contained in each node is a record – one part of the record is called the **key**.
- BST Rule: A BST is a binary tree in which, for every node x
  - **the keys in all nodes in the left subtree of x are smaller than the key in x.**
  - **the keys in all nodes in the right subtree of x are larger than the key in x.**
- Duplicate keys are generally not allowed in a BST.

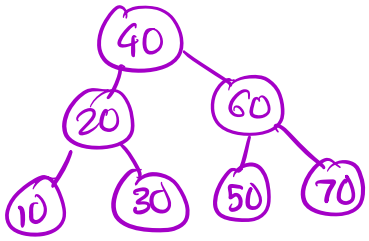
4

BINARY SEARCH TREE EXAMPLES

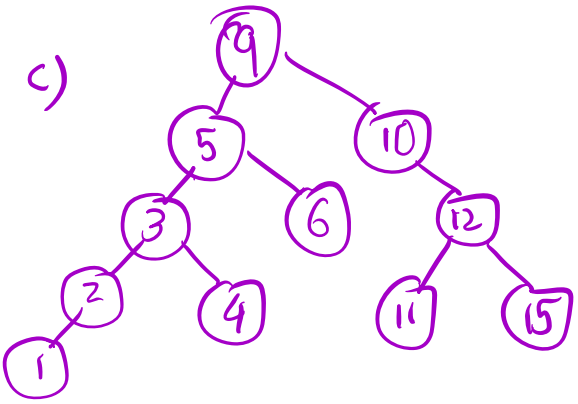
a)



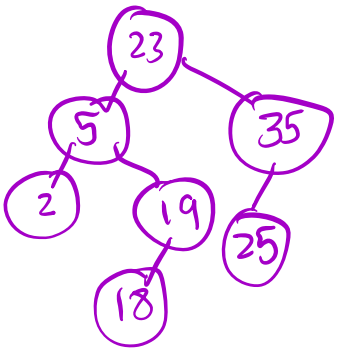
b)



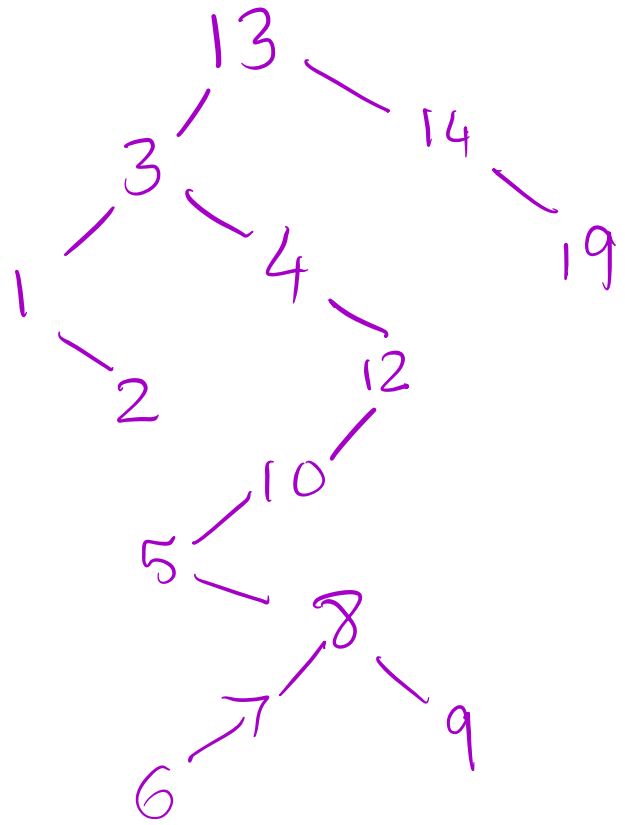
c)



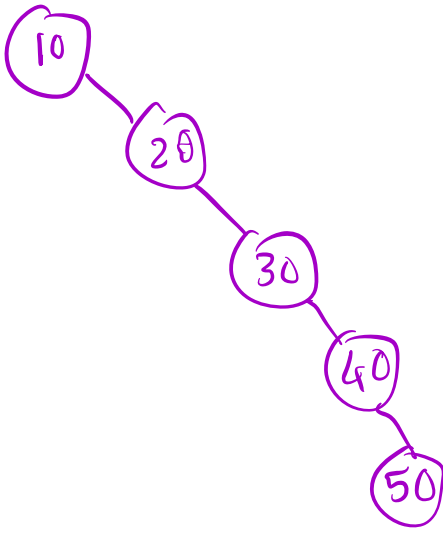
d)



e)



f)



Example: Building a Binary Search Tree with names of students as keys

Devan
Chris
Leon
Wisam
Toni
Theo
Pan
Jonah
Maddie
Nithin
Shreya
Martin
Rvizhe
Fariha

Inorder Traversal: L - Root - R

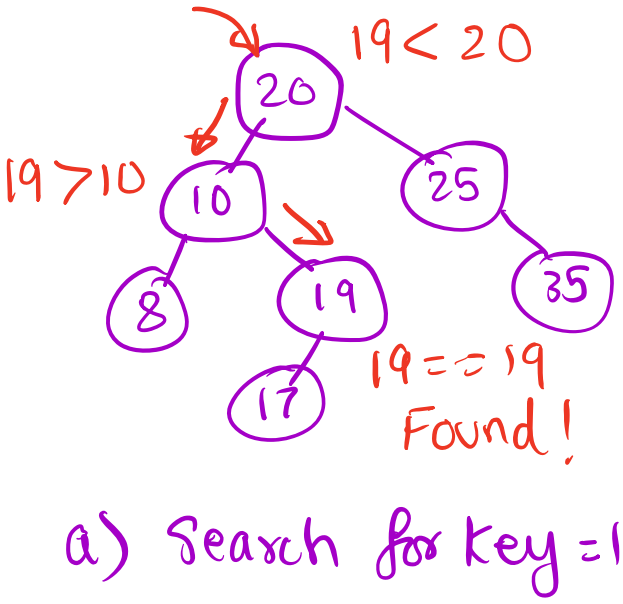
Chris - Devan - Fariha - Jonah - Leon -

Inorder traversal sorts the keys in ascending order!!

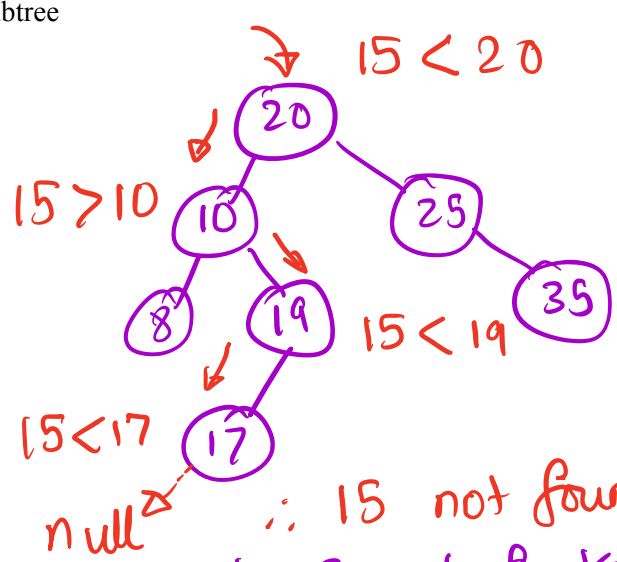
BINARY SEARCH TREE OPERATIONS

SEARCH FOR A GIVEN KEY

Compare target key with the value in the root.
If it is equal, target found. Exit.
If an empty subtree is reached, target not found. Exit.
Otherwise,
    if target is < value in the root, search the left subtree
    if target is > value in the root, search the right subtree



a) Search for key = 19

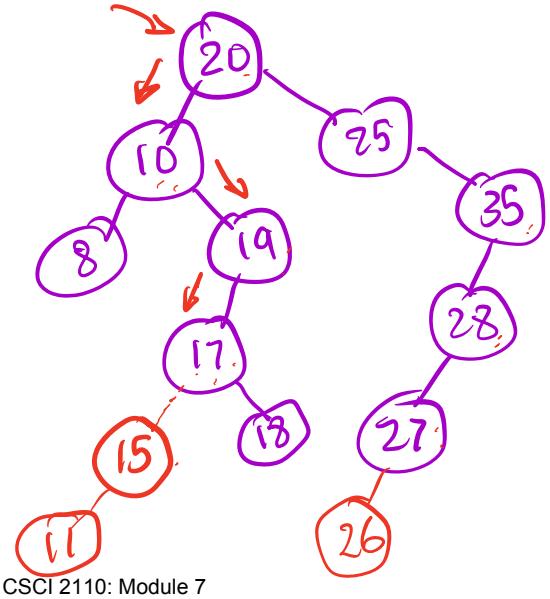


b) Search for key = 15

INSERT A VALUE INTO A BST

In order to insert a value, the search process is employed to force a failure, and the new value is inserted at the place where the search failed.

Note that the new node always becomes the leaf node in the BST.



a) Insert key = 15
b) Insert key = 11
c) Insert key = 26
d) Insert key = 25
↳ can't be inserted.
Duplicates not allowed.

DELETE A NODE WITH A GIVEN VALUE FROM A BST

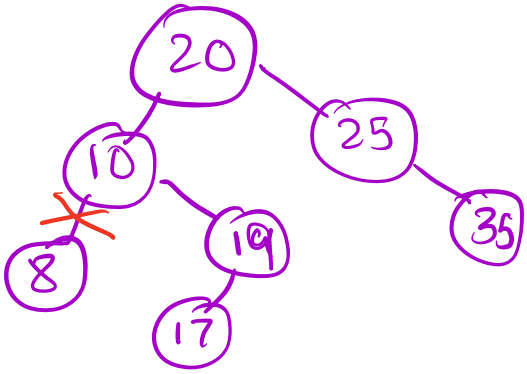
First locate the value in the BST. Let it be found in node X.

Case 1: X is a leaf node (X has no children)

Simply delete X, that is, detach X from its parent.

E.g. Delete 8.

Easy.



Case 2: X has one child

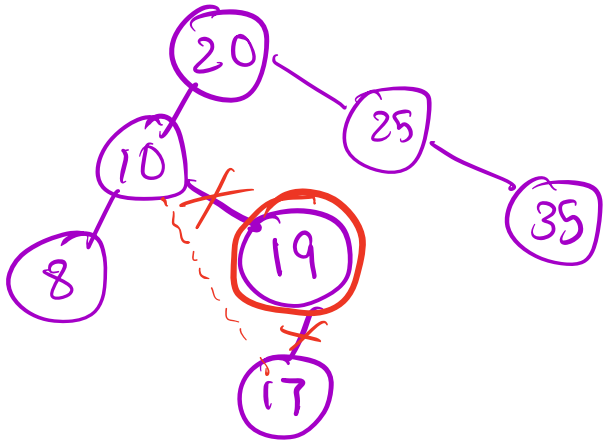
Make the child of X the child of the parent of X.

Then delete X by delinking it from its parent.

E.g. Delete 19.

"Hand over the custody to the grandparent"

Medium

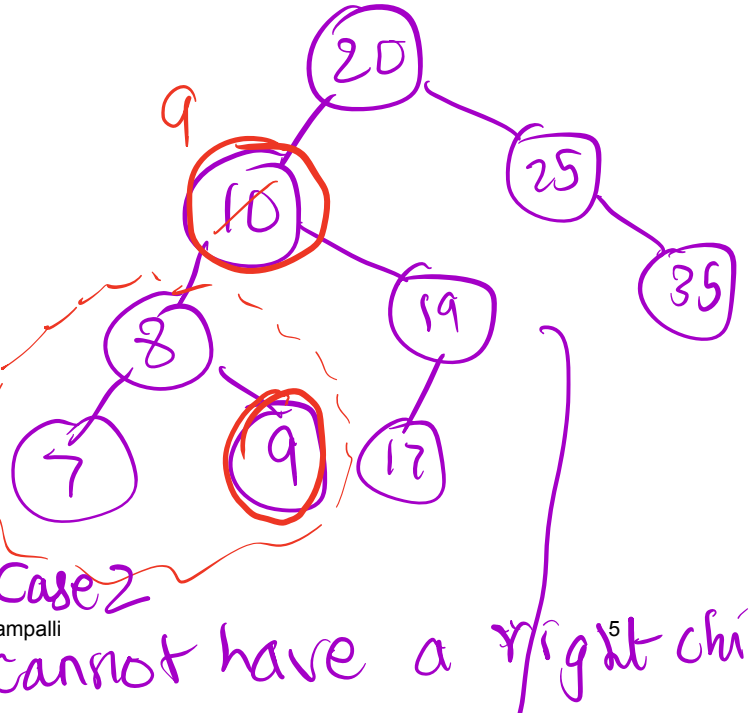


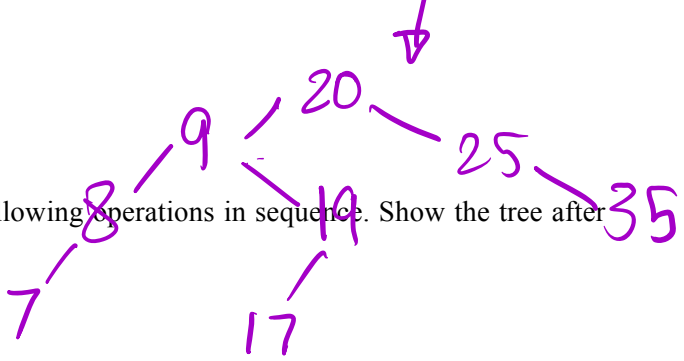
Case 3: X has two children

Replace the value in X by the largest value in the left subtree of X.
Let that value be found in node Y.
Delete Y. This step will be either Case 1 or Case 2.
Why?

Hard

Delete 10.
Go to the left subtree of 10.
Find the largest key = 9
Replace node 10 with node 9.
Then go & delete node 9.
This will be Case 1 or Case 2 because 9 cannot have a right child.





BST Exercise: Starting from an empty BST, perform the following operations in sequence. Show the tree after each operation:

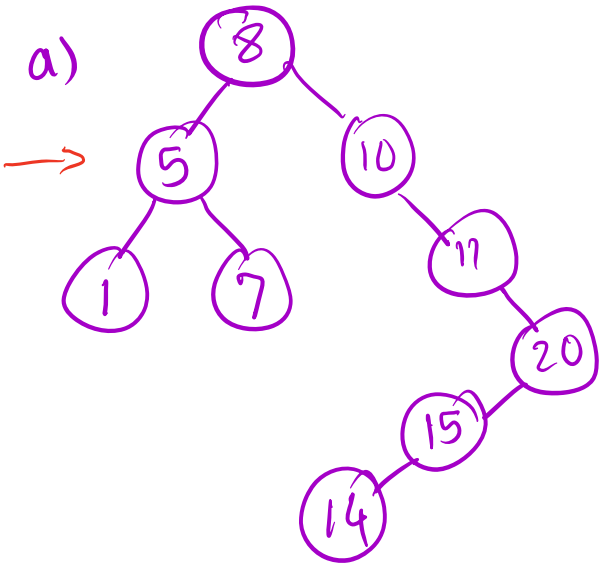
a) Insert 8, 10, 5, 1, 7, 11, 20, 15, 14

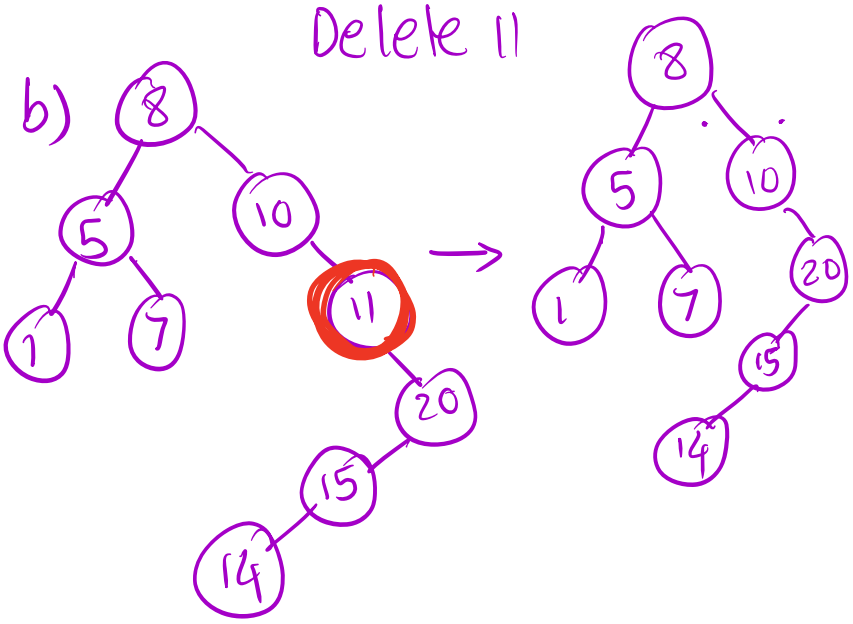
b) Delete 11

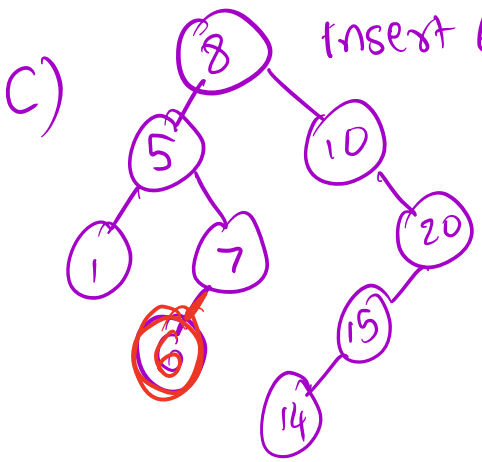
c) Insert 6

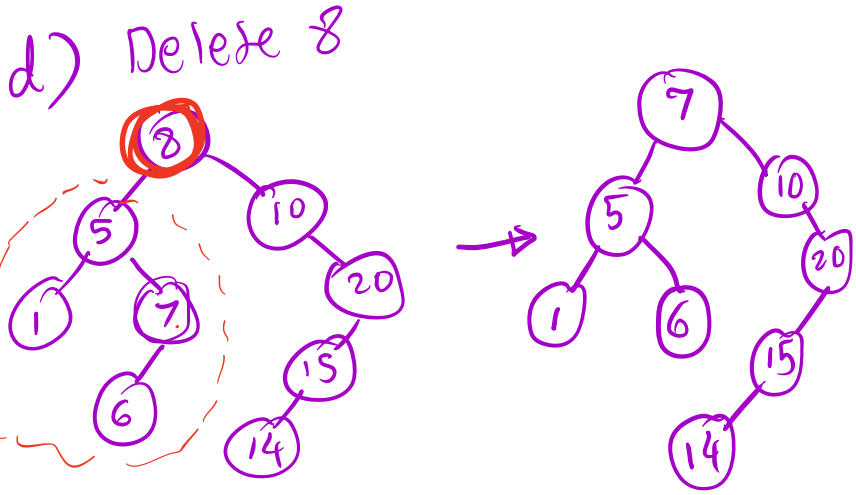
d) Delete 8

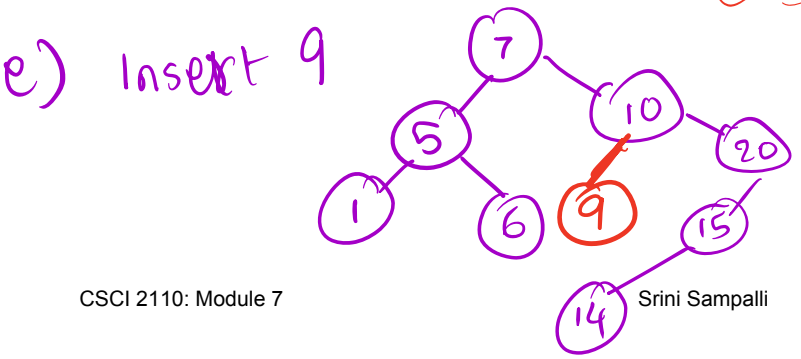
e) Insert 9











BINARY SEARCH TREE CLASS

Attributes

BinaryTree<T> tree;
int size;

← instance vars

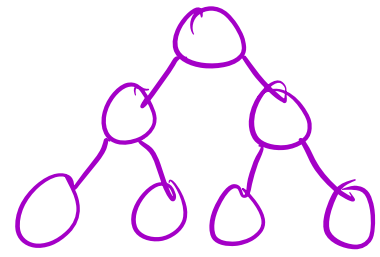
Constructors

BinarySearchTree()	Creates an empty binary search tree

Methods

Name	What it does	Header	Price tag (complexity)
getTree	Returns the BinaryTree	BinaryTree<T> getTree()	O(1)
isEmpty	Checks if the tree is empty	boolean isEmpty()	O(1)
size	Returns the size of the binary tree	int size()	O(1)
search	Searches for a given key and returns the node with the key	BinaryTree<T> search (T key)	$O(\log_2 n)$ GOOD O(n) BAD
insert	Inserts a given item into the binary search tree	void insert (T key)	$O(\log_2 n)$ GOOD O(n) BAD
delete	Delete the node with the given key	void delete (T key)	$O(\log_2 n)$ GOOD O(n) BAD

We will also add two more methods findPredecessor and deleteHere. These are helper methods for the delete method.



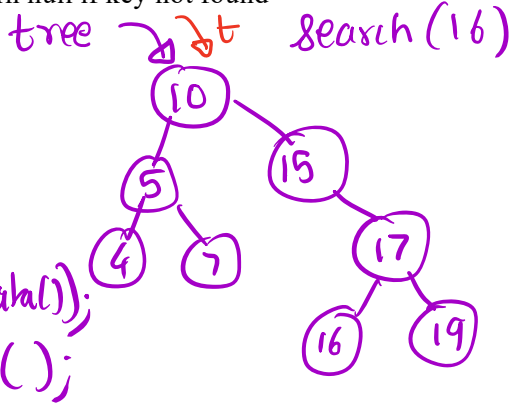
n = 7 #searches = 3
GOOD TREE

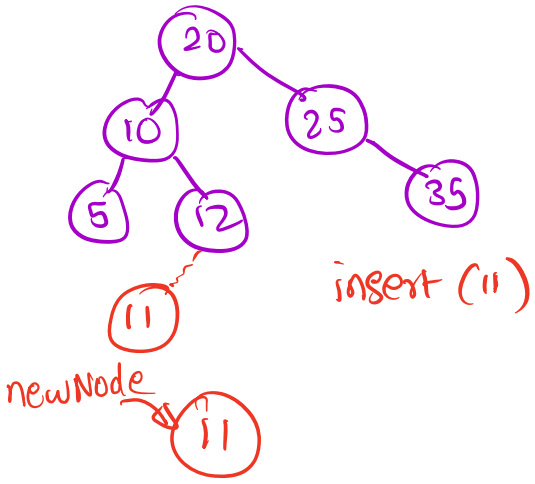
n = 7 #searches = 7
BAD TREE

IMPLEMENTATION

```
public class BinarySearchTree<T extends Comparable<T>>
{
        //attributes
        private BinaryTree<T> tree;
        private int size;

        //constructor
        public BinarySearchTree()
        {
                tree = new BinaryTree<T>();
                size = 0;
        }

        //other methods
        public BinaryTree<T> getTree()
        {
                return tree;
        }


        public boolean isEmpty()
        {
                return tree.isEmpty();
        }

        public int size()
        {
                return size;
        }

        //Search for a given key and return the reference to that node; return null if key not found
        public BinaryTree<T> search(T key)
        {
```

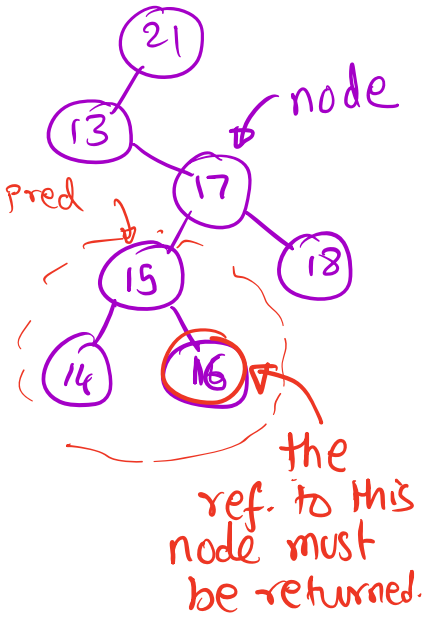
```
BinaryTree<T> t = tree;
boolean found = false;
while (t != null && !found)
{   int c = key.compareTo(t.getData());
    if (c < 0) t = t.getLeft();
    if (c > 0) t = t.getRight();
    if (c == 0) found = true;
}
if (found) return t;
else return null;
}
```






```
//insert an item into a binary search tree
public void insert(T item)
{

    //first create a new single node Binary Tree with the item
    BinaryTree<T> newNode = new BinaryTree<T>();
    newNode.setData(item);

    //if this is the first node in the binary search tree
    if (size===0){
            tree = newNode;
            size++;
            return;
    }


    //Otherwise, start at the root of the binary search tree and find the place to insert
    BinaryTree<T> t = tree;
    boolean done   = false;
    while (! done)
    {
      int c = item.compareTo(t.getData());
      if (c==0)
      {
        System.out.println("Duplicate key. Can't insert");
        return;
      }
      else if (c<0)          // need to go left
      {
        if (t.getLeft()==null)  // found the place to insert
        {
          t.setLeft(newNode);
          newNode.setParent(t);
          done = true;
          size++;
        }
        else                   // keep going
          t = t.getLeft();
      }
      else if (c>0)
      {                        (*) replace left with right
      }
    }
```

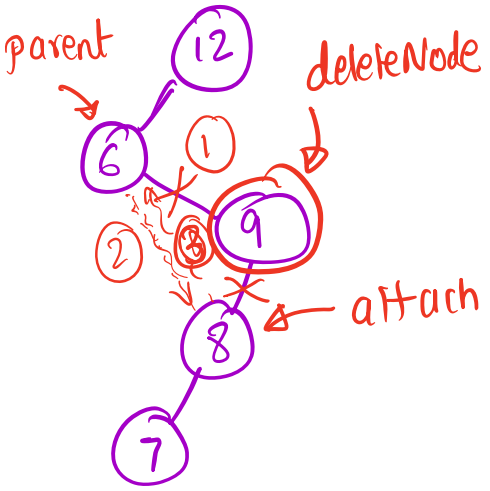
```
//findPredecessor – helper method for the delete method
//returns the largest node in the left subtree of the given node
public BinaryTree<T> findPredecessor(BinaryTree<T> node)
{
        if (node == null)
                return null;
        BinaryTree<T> pred = node.getLeft();
        if (pred <= null) return null;
        while (pred.getRight()!=null)
                pred = pred.getRight();
        return pred;
}

//deleteHere – helper method for the delete method
//deletes a given node and attaches its subtree(s) to its parent node
public void deleteHere(BinaryTree<T> deleteNode, BinaryTree<T> attach)
{
        if (deleteNode == null)
                return;
        BinaryTree<T> parent = deleteNode.getParent();

        if (parent == null)
                return;
        if (attach == null)
        {
                if (parent.getLeft()== deleteNode)
                        parent.setLeft(null);
                else
                        parent.setRight(null);
                return;
        }

        if (deleteNode==parent.getRight()){
                parent.detachRight();
                deleteNode.setParent(null);
                parent.attachRight(attach);
                attach.setParent(parent);
        }
        else
        {
                parent.detachLeft();
                deleteNode.setParent(null);
                parent.attachLeft(attach);
                attach.setParent(parent);
        }
        deleteNode.clear();
}
```



Case 1 or Case 2

Case 1

Case 2



```
//delete method: deletes a node with a given key
public void delete (T key)
{
        if (size == 0) {
                System.out.println("Can't delete. Empty tree");
                return;
        }

        BinaryTree<T> deleteNode = search(key);

        if (deleteNode == null) {
                System.out.println("Can't delete. Key not found");
                return;
        }

        BinaryTree<T> hold = null;

        //Case 1: deleteNode has no children
        if (deleteNode.getLeft() == null && deleteNode.getRight() == null)
                deleteHere (deleteNode, null);

        //Case 2: deleteNode has one child (left child)
        else if (deleteNode.getRight() == null)
        {
                hold = deleteNode.getLeft();
                deleteHere (deleteNode, hold);
        }
        //Case 2: deleteNode has one child (right child)
        else if (deleteNode.getLeft() == null)
        {
                hold = deleteNode.getRight();
                deleteHere (deleteNode, hold);
        }
        //Case 3: deleteNode has two children
        else
        {
                hold = findPredecessor (deleteNode);
                deleteNode.setData (hold.getData());
                deleteHere ( hold, hold.getLeft() );
        }

}
```

→ Get to the node to be deleted.