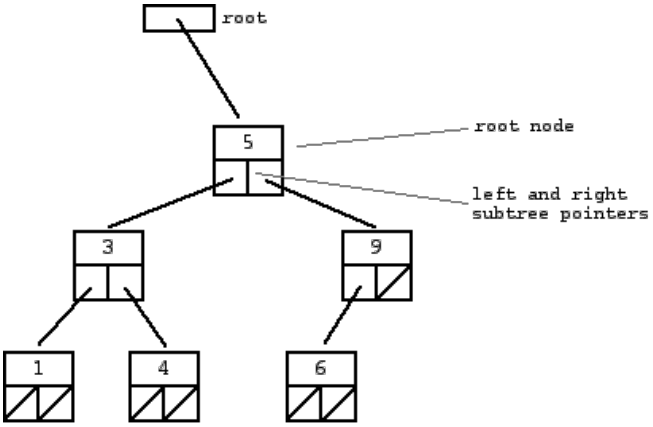
CS 261P | Data Structures
Project 2
[Binary trees and their variants]

Shefali Gupta | 57806943

Contents:

Binary Tree

A binary tree is made of nodes, where each node contains a "left" pointer, a "right" pointer, and a data element. The "root" pointer points to the topmost node in the tree. The left and right pointers recursively point to smaller "subtrees" on either side. A null pointer represents a binary tree with no elements -- the empty tree. The formal recursive definition is: a binary tree is either empty (represented by a null pointer), or is made of a single node, where the left and right pointers each point to a binary tree.



BINARY SEARCH TREES

Binary search trees (BST), sometimes called ordered or sorted binary trees, are a particular type of container: data structures that store "items" (such as numbers, names etc.) in memory. They allow fast lookup, addition and removal of items, and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its *key*

Create operation:

Implementation in Java:

```
class Node {
        int value;
        Node left, right;

        public Node(int new_value) {
            value = new_value;
            left = null;
     right = null;
        }
    }
class BinarySearchTree
{

    Node root;
     // constructor is a create function.
    BinarySearchTree()
  {root = null;}
    void insert(int value ) {…}
            void delete(int value ) {…}
            void search(int value ) {…}
}
```

Search operation:

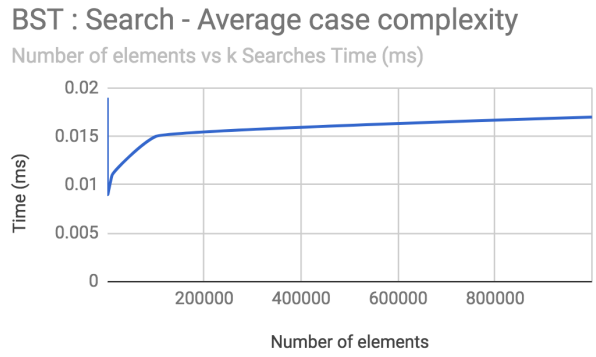
Because in the worst case this algorithm must search from the root of the tree to the leaf farthest from the root, the search operation takes time proportional to the tree's *height* (see tree terminology). On average, binary search trees with *n* nodes have O(log *n*) height. However, in the worst case, binary search trees can have O(*n*) height, when the unbalanced tree resembles a linked list.

```
public Node search(int value)
{
 return search_rec(root, value);
}
public Node search_rec(Node current, int value) // recursive function for tree traversal.
{
```

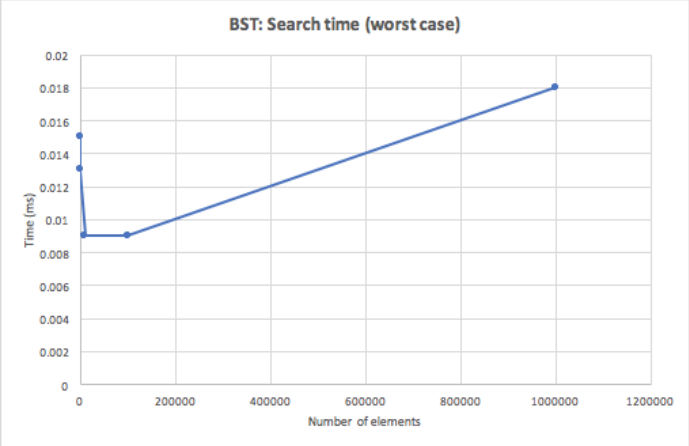
```
    if (current ==null || current.value==value)
        return current;

    if (current.value > value)
        return search_rec(current.left, value);

    return search_rec(current.right, value);
}
```

Amortized analysis:
Created a Binary search tree of N number of elements (N varying from 100 to 1000000) and searched for k (k=10) elements in the tree, and computed time (in ms) of k elements search.

Average case: Inserting elements in order and searching for a random number. O(log N)



Worst case: always searching for element on leaf node.
O(N) in this case: Since, elements are inserted in ascending order, and last element is searched. Thus, making a linked list.



Insert operation:

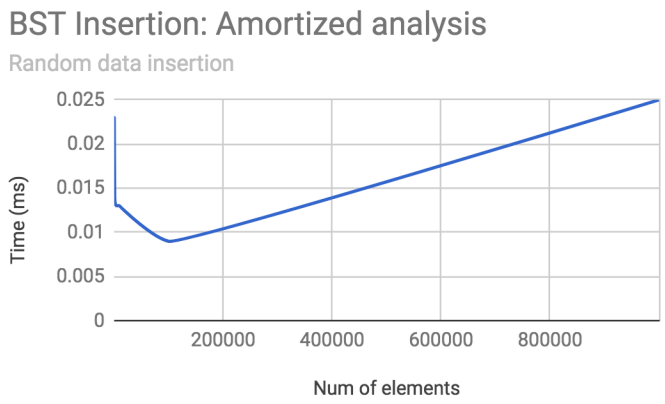
O(log *n*) space in the average case and O(*n*) in the worst case.

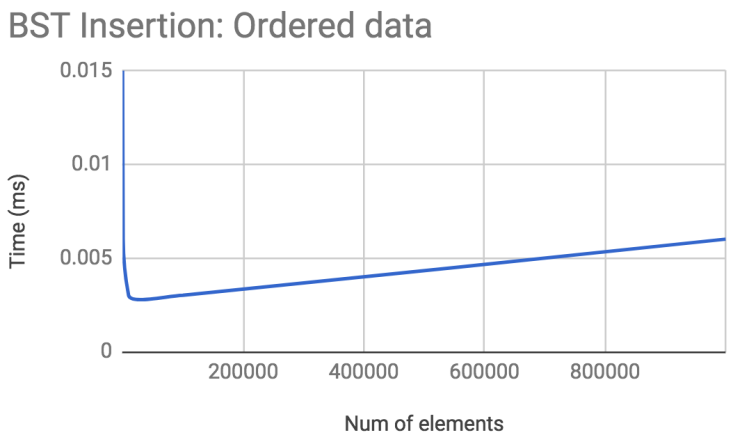
Pseudocode:

```
Node insert(int value)
  {
    return insert_rec(root, value);
  }
    Node insert_rec(Node current, int value) {

     if (current == null) {
          current = new Node(value);
         return current;
     }

     if (value < current.value)
       current.left = insert_rec(current.left, value);
     else if (value > current.value)
       current.right = insert_rec(current.right, value);

     return current;
  }
```

Average case:
N ordered elements inserted to make a tree and k random numbers inserted for running time computation. O(log N)

Worst case:
Insertion of ordered data, at the leaf nodes.



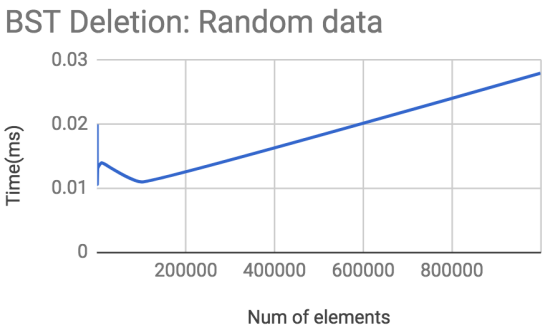


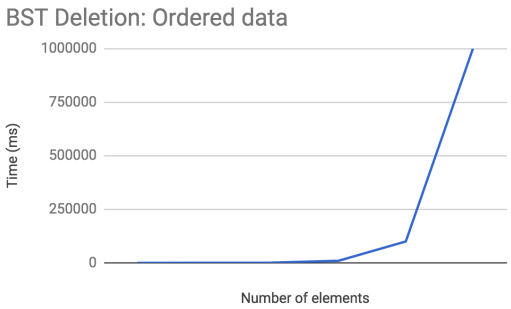
Delete operation:

O (log n) time taken.

```
public void deleteKey(T key)
{
	root=deleteRec(root,key);
}
Node deleteRec(Node<T> root,T key)
{
	if(root==null)
		return root;
	if(key.compareTo(root.key)<0)
		root.left=deleteRec(root.left,key);
	else if(key.compareTo(root.key)>0)
		root.right=deleteRec(root.right,key);
	else
	{
		if(root.left==null)
			return root.right;
		else if(root.right==null)
			return root.left;
		root.key=(T) minValue(root.right);
		root.right=deleteRec(root.right,root.key);
	}
	return root;
}
```

Average and Worst cases:


BST Deletion: Random data
0.03
0.02
0.01
0
Time(ms)
200000
400000
600000
800000
Num of elements


BST Deletion: Ordered data
1000000
750000
500000
250000
0
Time (ms)
Number of elements

AVL trees

In computer science, an AVL tree (named after inventors Adelson-Velsky and Landis) is a self-balancing binary search tree. It was the first such data structure to be invented. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property. Lookup, insertion, and deletion all take $O(\log n)$ time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

Create operation:

AVL tree node structure:

```
Node(int d)
{
        key=d;
        height=1;
}
```

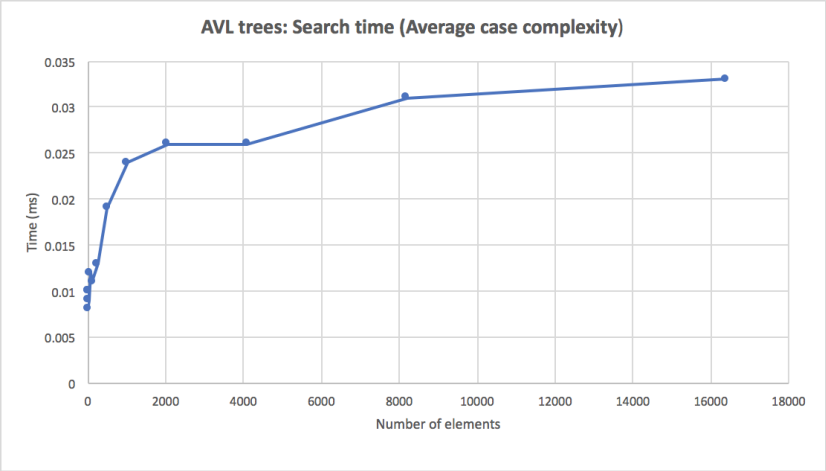
Search operation:

Pseudocode:

```
public int search(Node node, int key) {

        if (node == null) {

            return 0;  // missing from tree

        } else if (key < node.key) {

            return search(node.left, key);

        } else if (key > node.key) {

            return search(node.right, key);

        } else {

            return node.key;  // found it
        }
    }
```

Amortized analysis:

Below graph is obtained, when we create an AVL tree of N elements (N varying from 16 to ~16000) and search for k (k=10) random numbers in the tree, and record time for the same.



Insert operation:

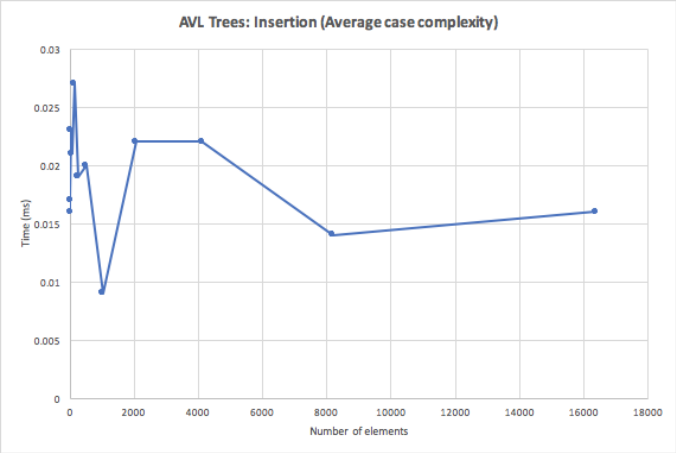
```
public Node insert(Node node,int key)
        {
                if(node==null)
                {
                        return (new Node(key));
                }
                if(key<node.key)
                        node.left=insert(node.left,key);
                else if(key>node.key)
                        node.right=insert(node.right,key);
                else
                        return node;
                node.height=1+max(height(node.left),height(node.right));
                //Check whether this node is unbalanced
                int balance=getBalanceFactor(node);
                //If node becomes unbalanced, 4 cases
                if(balance>1)
                {
                        //left-left
                        if(key<node.left.key)
                                return rotateRight(node);
                        //left-right
                        if(key>node.left.key)
                        {
                                node.left=rotateLeft(node.left);
                                return rotateRight(node);
                        }
                }
                if(balance<-1)
                {
                        if(key>node.right.key)
                                return rotateLeft(node);
                        if(key<node.right.key)
                        {
```

```
                node.right=rotateRight(node.right);
                return rotateLeft(node);
            }

        }
        return node;
    }
```

Amortized analysis:

Created a AVL tree of N number of elements (N varying from 16 to ~18000) and inserted k (k=10) elements in the tree, and computed time (in ms) of k elements insertion.

O(log n)



Delete operation:

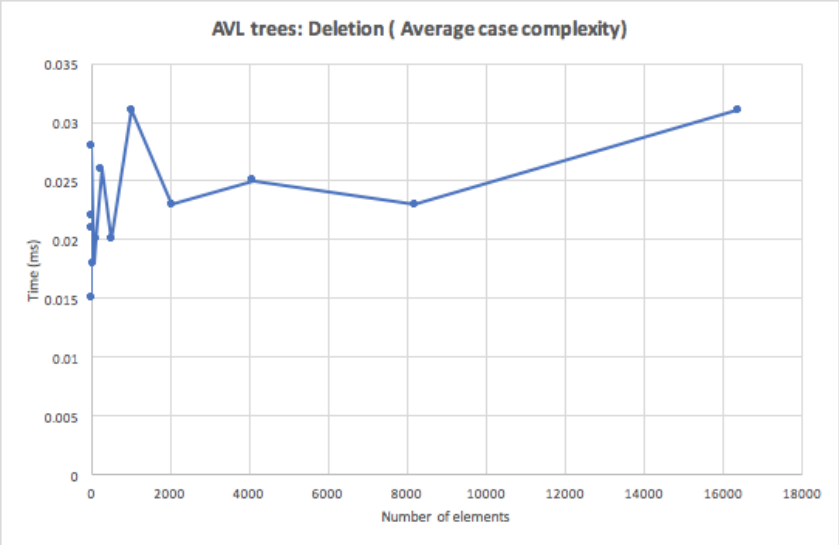
Code:

```
public Node deleteNode(Node root, int key)
  {
      if (root == null)
          return root;

      // If the key to be deleted is smaller than
      // the root's key, then it lies in left subtree
      if (key < root.key)
          root.left = deleteNode(root.left, key);

      // If the key to be deleted is greater than the
      // root's key, then it lies in right subtree
      else if (key > root.key)
          root.right = deleteNode(root.right, key);

      // if key is same as root's key, then this is the node
      // to be deleted
      else
      {
          // node with only one child or no child
```

```
    if ((root.left == null) || (root.right == null))
    {
        Node temp = null;
        if (temp == root.left)
            temp = root.right;
        else
            temp = root.left;

        // No child case
        if (temp == null)
        {
            temp = root;
            root = null;
        }
        else   // One child case
            root = temp; // Copy the contents of
                         // the non-empty child
    }
    else
    {

        // node with two children: Get the inorder
        // successor (smallest in the right subtree)
        Node temp = minValueNode(root.right);

        // Copy the inorder successor's data to this node
        root.key = temp.key;

        // Delete the inorder successor
        root.right = deleteNode(root.right, temp.key);
    }
}

// If the tree had only one node then return
if (root == null)
    return root;

// STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
root.height = max(height(root.left), height(root.right)) + 1;

// STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether
//  this node became unbalanced)
int balance = getBalanceFactor(root);

// If this node becomes unbalanced, then there are 4 cases
// Left Left Case
if (balance > 1 && getBalanceFactor(root.left) >= 0)
    return rotateRight(root);

// Left Right Case
if (balance > 1 && getBalanceFactor(root.left) < 0)
{
    root.left = rotateLeft(root.left);
    return rotateRight(root);
```

```
        }

        // Right Right Case
        if (balance < -1 && getBalanceFactor(root.right) <= 0)
            return rotateLeft(root);
        // Right Left Case
        if (balance < -1 && getBalanceFactor(root.right) > 0)
        {
            root.right = rotateRight(root.right);
            return rotateLeft(root);
        }
        return root;
    }
```

Amortized analysis:
O(log n)



SPLAY TREES

A splay tree is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log n) amortized time. For many sequences of non-random operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown.
All normal operations on a binary search tree are combined with one basic operation, called *splaying*. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. One way to do this with the basic search operation is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top.
The main idea of splay tree is to bring the recently accessed item to root of the tree, this makes the recently searched item to be accessible in O(1) time if accessed again. The idea is to use locality of reference (In a typical application, 80% of the access are to 20% of the items).

Create operation:
Splay node in Java:

```
private class Node {
      private Key key;           // key
      private Value value;       // associated data
      private Node left, right;  // left and right subtrees

      public Node(Key key, Value value) {
         this.key   = key;
         this.value = value;
      }
   }
```

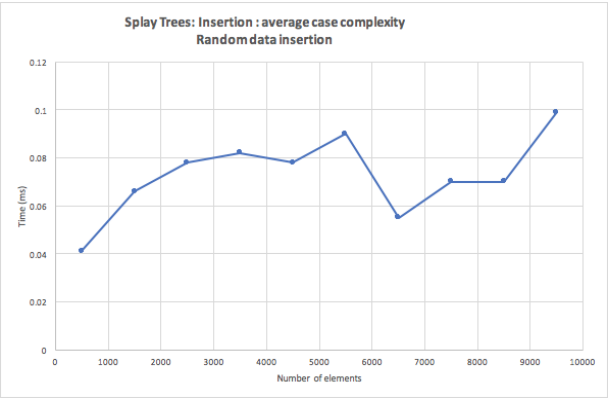
Insert Operation:
Code:

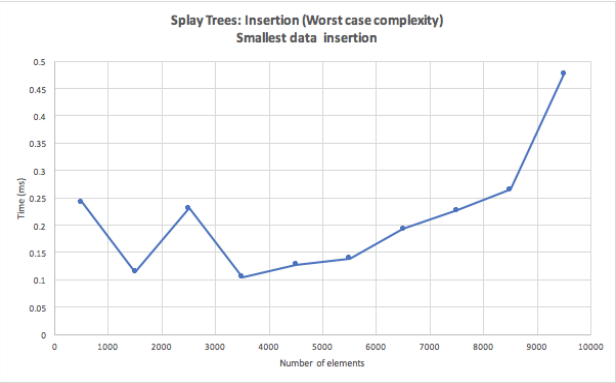
```
public void insert(Key key, Value value) {
      // splay key to root
      if (root == null) {
         root = new Node(key, value);
         return;
      }

      root = splay(root, key);

      int cmp = key.compareTo(root.key);

      // Insert new node at root
      if (cmp < 0) {
         Node n = new Node(key, value);
         n.left = root.left;
```

```
            n.right = root;
            root.left = null;
            root = n;
        }

        // Insert new node at root
        else if (cmp > 0) {
            Node n = new Node(key, value);
            n.right = root.right;
            n.left = root;
            root.right = null;
            root = n;
        }

        // It was a duplicate key. Simply replace the value
        else {
            root.value = value;
        }

    }
```

Amortized analysis:

O(log n) time complexity.





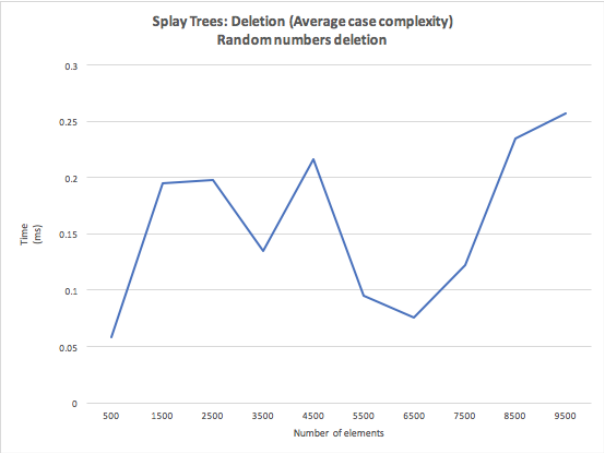
Delete operation:

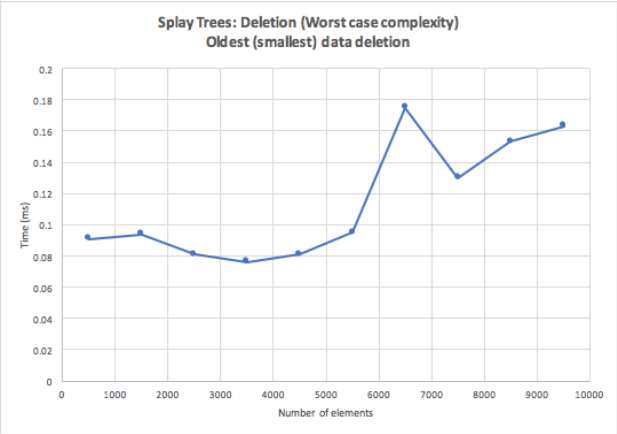
Java code:

```
public void delete(Key key) {
    if (root == null) return; // empty tree

    root = splay(root, key);

    int cmp = key.compareTo(root.key);

    if (cmp == 0) {
        if (root.left == null) {
            root = root.right;
```

```
        }
        else {
            Node x = root.right;
            root = root.left;
            splay(root, key);
            root.right = x;
        }
    }
```

Amortized analysis:
Average case: O(log n) time complexity.
Worst case: O(n) time complexity.





Search operation:

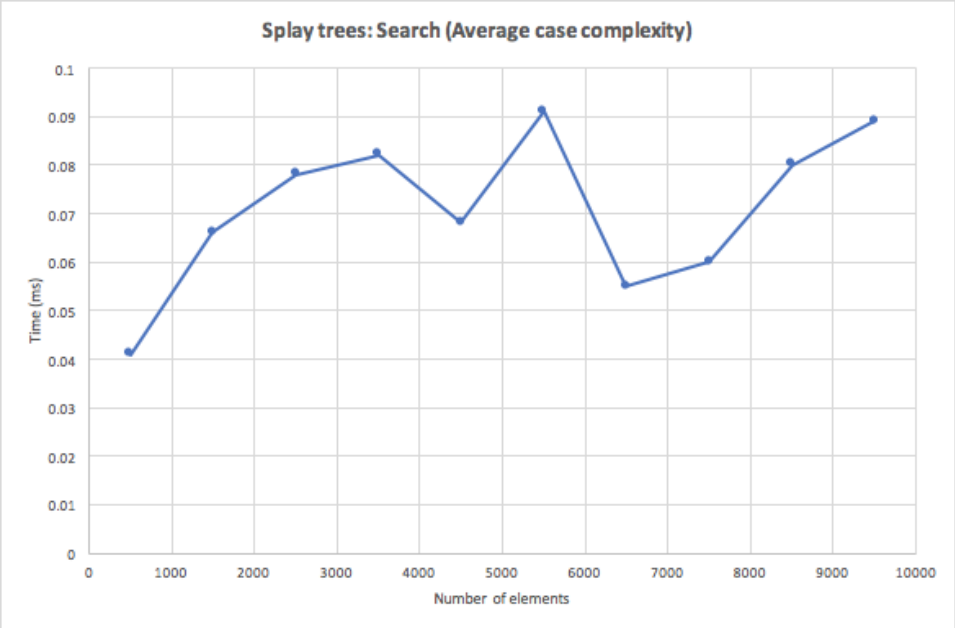
Java Code:

```
public boolean search(int val) {
        return findNode(val) != null;
    }
    private SplayNode findNode(int ele)
    {
        SplayNode z = root;
        while (z != null)
        {
            if (ele < z.element)
                z = z.right;
            else if (ele > z.element)
                z = z.left;
            else
                return z;
        }
        return null;
```

}

Amortized analysis:

O(log n) time complexity.



Worst case: Search for oldest element. O(n) time complexity.

TREAPS

The treap and the randomized binary search tree are two closely related forms of binary search tree data structures that maintain a dynamic set of ordered keys and allow binary searches among the keys. After any sequence of insertions and deletions of keys, the shape of the tree is a random variable with the same probability distribution as a random binary tree; in particular, with high probability its height is proportional to the logarithm of the number of keys, so that each search, insertion, or deletion operation takes logarithmic time to perform.

Create operation:

Treap node in Java implementation:

```
class TreapNode
{
	TreapNode left,right;
	int priority,element;
	public TreapNode()
	{
		this.element=0;
		this.left=null;
		this.right=null;
		this.priority=Integer.MAX_VALUE;
	}
```

Insert operation:

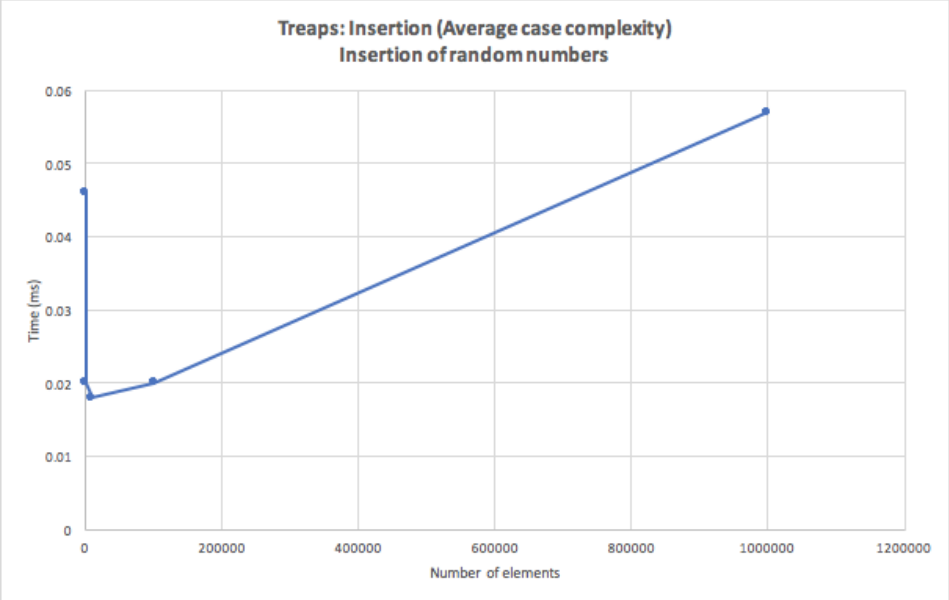
Code:

```
private TreapNode insert(int X,TreapNode T)
	{
		if(T==null)
			return new TreapNode(X,null,null);
		else if(X<T.element)
		{
			T.left=insert(X,T.left);
			if(T.left.priority<T.priority)
			{
				TreapNode L=T.left;
				T.left=L.right;
				L.right=T;
				return L;
			}
		}
		else if(X>T.element)
		{
			T.right=insert(X,T.right);
			if(T.right.priority<T.priority)
			{
				TreapNode R=T.right;
				T.right=R.left;
				R.left=T;
```

```
                    return R;
                }
            }
            return T;
        }
```

Amortized analysis:

O(log n) time complexity.



Delete operation

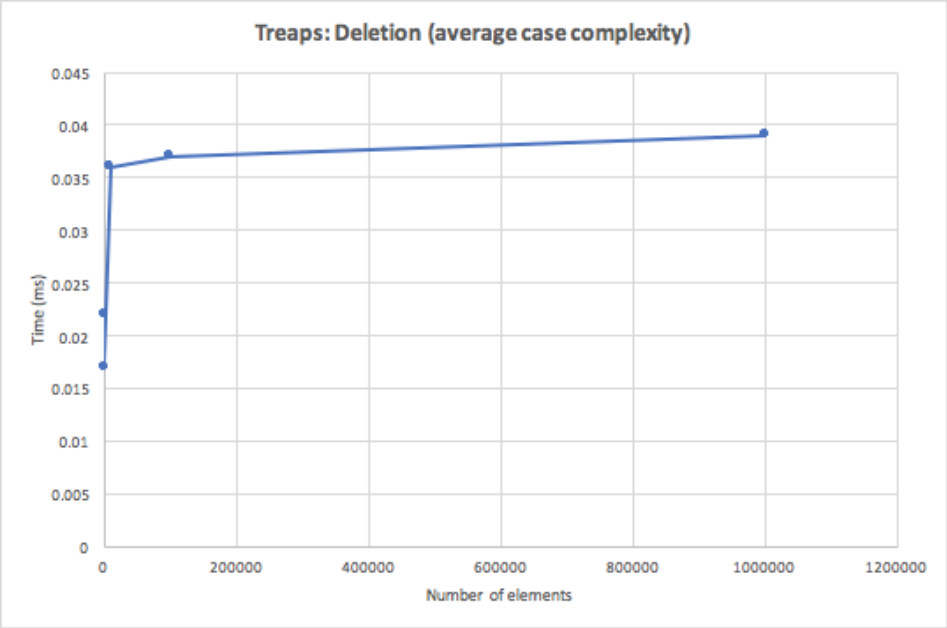
Code:

```
public void delete(int key)
  {
            delete(root,key);
  }
  private TreapNode delete(TreapNode root,int key)
  {
      if(root==null)
            return root;
      if(key<root.element)
            root.left=delete(root.left,key);
      else if(key>root.element)
            root.right=delete(root.right,key);
      else if(root.left==null)
      {
            TreapNode temp=root.right;
            root=temp;
      }
      else if(root.right==null)
```

```
        {
                TreapNode temp=root.left;
                root=temp;
        }
        else if(root.left.priority<root.right.priority)
        {
                root=leftRotate(root);
                root.left=delete(root.left,key);
        }
        else
        {
                root=rightRotate(root);
                root.right=delete(root.right,key);
        }
        return root;
}
```

Amortized analysis:

O(log n) time complexity.



Search operation:

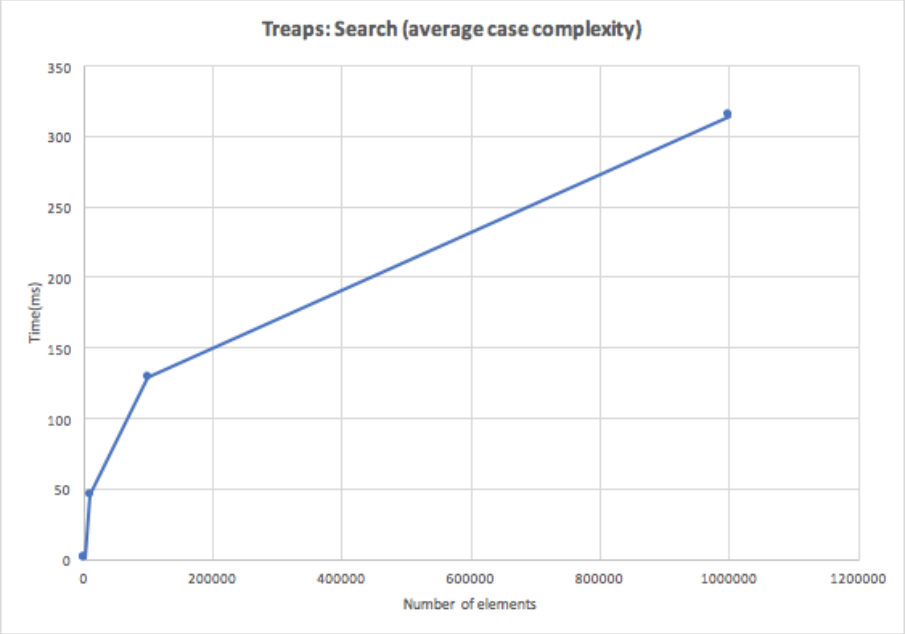
Code:

```
public boolean search(int val)
        {
```

```
        return search(root,val);
    }
    private boolean search(TreapNode r,int val)
    {
        boolean found=false;
        while((r!=null)&&!found)
        {
            int rval=r.element;
            if(val<rval)
                r=r.left;
            else if(val>rval)
                r=r.right;
            else
            {
                found=true;
                break;
            }
            found=search(r,val);
        }
        return found;
    }
```

Amortized analysis:

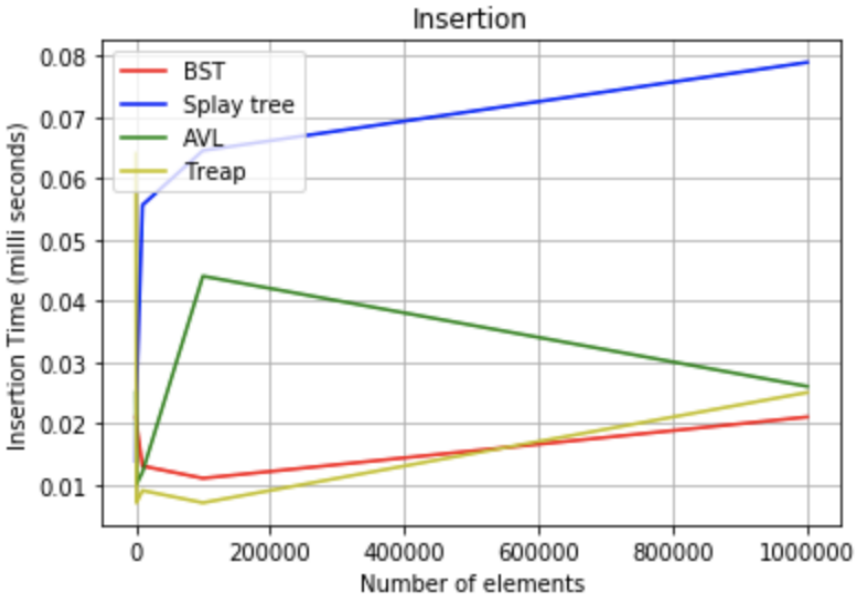
O(log n) time complexity.



Combined analysis (average case) of all four binary trees:

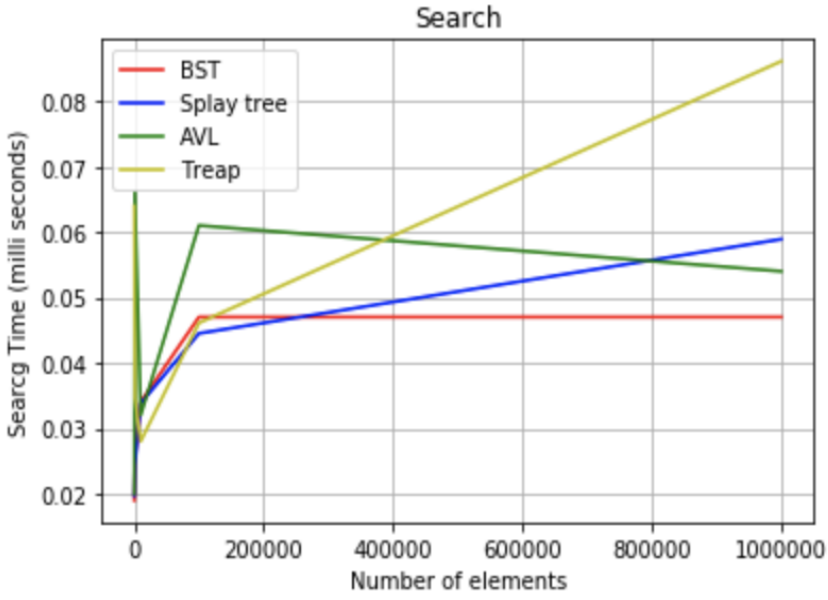
Insertion:

Analysis method: Created trees of size N (N varying from 100 to 1000000) and inserting k (k=10) random numbers to the tree, and noting time for k insertions.

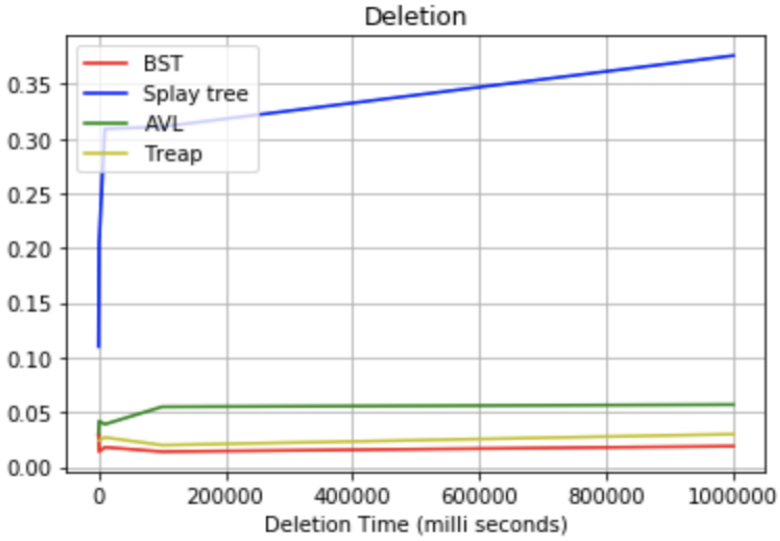


Search:

Same method as above.



Deletion:
Same method of analysis as above.



References:
https://en.wikipedia.org/wiki/AVL_tree
https://en.wikipedia.org/wiki/Splay_tree
https://www.geeksforgeeks.org/