CSE 222/CSE505 SPRING 2022
HOMEWORK 5 REPORT

Burcu Sultan Orhan

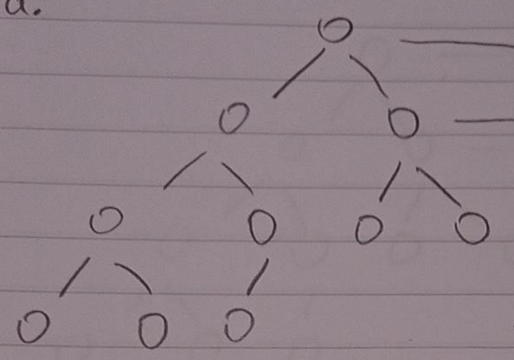
1901042667

1) Answer the following questions, do not give just an answer, show all your work.

a) Calculate the total depth of the nodes in a complete binary tree of height h. Note that total depth is 5 if height is 2 where the depth of the root is one and there are two nodes with depth 2.

1.

a.



Depth	Node
1	1/1
2	2/2
3	4/4
4	3/8

Depth → 1, e → 1 → $2^0 . 1 = 1$

Depth → 2, e → 2 → $2^1 . 2 = 4$

Depth → 3, e → 4 → $2^2 . 3 = 12$

Depth → 4, e → 3 → $3 . 4 = 12$

↳ So, we see a pattern here when all of nodes are occupied. We can get a recursive function such as:

$$D(1) = 1$$

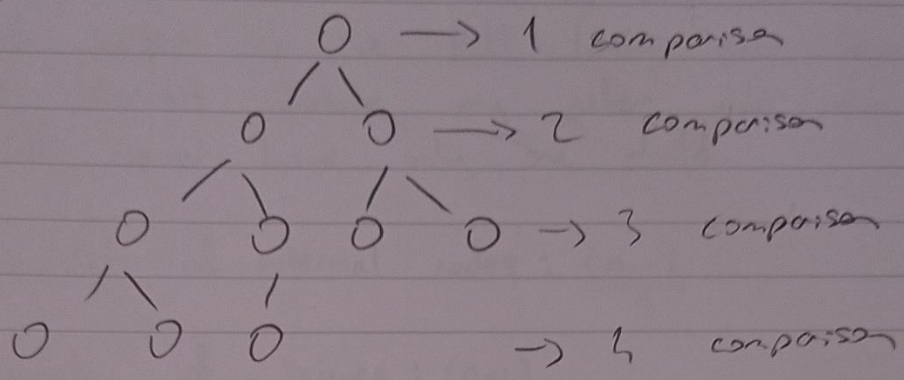
$$D(h) = D(h-1) + 2^{h-1} . h$$

which gives us the maximum total depth of a complete binary tree with height h.

b) Calculate the average number of comparisons for a successful search operation in a binary search tree which has the structural property of being complete binary tree.

1.

b.



Depth = 1, $e = 1$, $c = 1$
Depth = 2, $e = 2$, $c = 2$
Depth = 3, $e = 4$, $c = 3$
Depth = 4, $e = 3/8$, $c = 4$

↳ We see a pattern similar to total depth pattern which can be described as:

$$\left.\begin{aligned} D(1) &= 1 \\ D(d) &= D(d-1) + 2^{d-1} \cdot d \end{aligned}\right\} \text{Total search number}$$

$$\sum_{n=1}^{d} 2^{n-1} = \text{Total node number in a tree with height } d$$

$$\frac{D(d)}{\sum_{n=1}^{d} 2^{n-1}} = \text{Average number of comparisons}$$

c) Is there a restriction on the number of nodes in a full binary tree? What is the number of internal nodes and number of leaves in an n node full binary tree?

1.

c. No, there's not a restriction on the number of nodes in a full binary tree because it doesn't dictate any other rule than a node cannot have only one child.

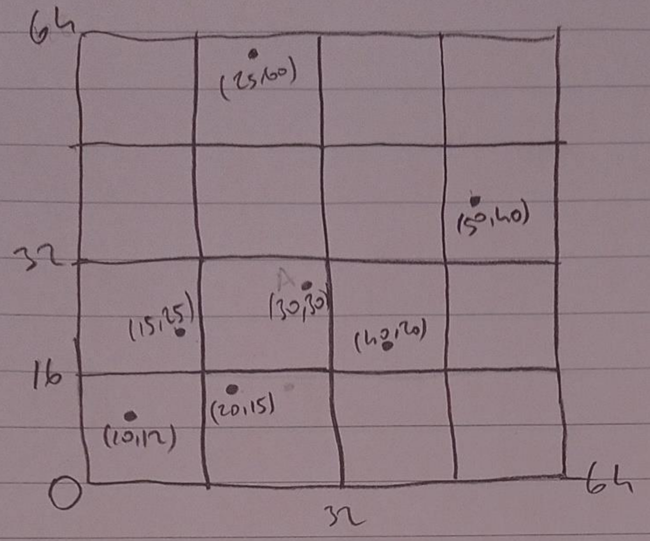
Number of internal nodes : $\frac{(n-1)}{2}$

Number of leaves : $\frac{(n+1)}{2}$

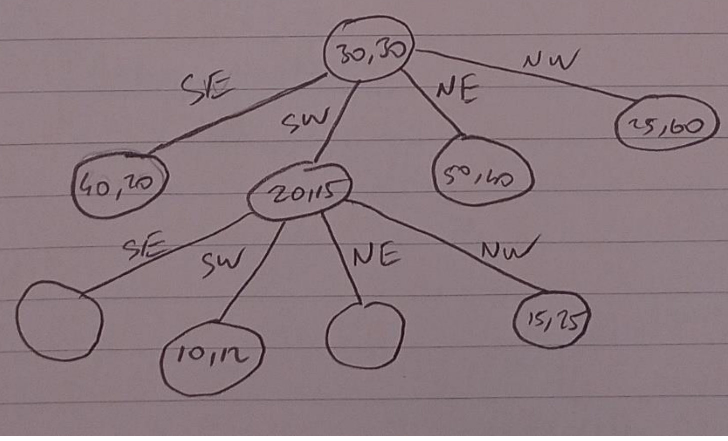
2) Research about the quadtree structure for two-dimensional point data. Consider using the binary tree representation of general trees in our textbook to implement the quadtree structure.

Insert the following elements one by one into an empty quadtree. Show the nodes traversed during each insertion and resulting tree after each insertion. Assume that the range is (0, 100) for both dimensions. Note that you are expected to draw the binary tree representation of the quadtree. (30,30), (20,15), (50,40), (10,12), (40,20), (25,60), (15,25)

2.



NW = (0,32) to (32,64) → (25,60)
NE = (32,64) to (32,64) → (50,40)
SE = (32,64) to (0,32) → (40,20)
SW = (0,32) to (0,32) → (15,25), (30,30), (10,12), (20,15



1- Detailed System Requirements

First, there needs to be a SearchTree interface to be implemented.

```java
public interface SearchTree<E extends Comparable<E>> {
    public boolean add(E item);
    public boolean contains(E target);
    public E find(E target);
    public E delete(E target);
    public boolean remove(E target);
}
```

And then there needs to be a Binary Search Tree class which implements SearchTree interface and its constuctor.

```java
public class MyBST<E extends Comparable<E>> implements SearchTree<E>{

    private E[] data;
    private int capacity=1000;
    private int size=0;

    @SuppressWarnings("unchecked")
    public MyBST(){
        data = (E[])new Comparable[this.capacity];
    }
```

Now come to methods, these two methods are used for inserting. First method with only one parameter which is public can be reached outside class, and this method calls another method which is recursive and private. This recursive method actually does the job.

```java
/**
 * Adds an item to binary search tree
 * @param item item to be added
 * @return true if successful, false if not
 */
public boolean add(E item){
    return addReturn(item, 0);
}

/**
 * Recursive method to find appropiate place to add the item and to add it
 * @param item item to be added
 * @param index index to keep track of which side are we headed
 * @return true if successful, false if not
 */
@SuppressWarnings("unchecked")
private boolean addReturn(E item, int index){
```

First two methods are for checking whether BST consists of a particular item and they are public which means they can be accessed outside the class. These two methods have different return values. One returns whether the target exits (boolean), other returns the target itself if it is present. However, these two functions call the same recursive private method that actually does the job.

```
/**
 * @param target target to be found
 * @return true if found, false if not
 */
public boolean contains(E target){
    return findReturn(target, 0);
}

/**
 * This method finds if a certain target exists in BST
 * @param target target to be found
 * @return target itself if found, else null
 */
public E find(E target){
    if(findReturn(target, 0)){
        return target;
    }
    else{
        return null;
    }
}

/**
 * Recursive method to search for a certain target in BST
 * @param target target to be found
 * @param index index to keep track of which side are we headed
 * @return true if found, false if not
 */
private boolean findReturn(E target, int index){
```

First two methods are for removing a particular item from BST and they are public which means they can be accessed outside the class. These two methods have different return values. One returns whether the removal is succesfull (boolean), other returns the target itself if it is succesfully removed. However, these two functions call the same recursive private method that actually does the job.

```java
/**
 * This method deletes certain target from BST
 * @param target target to be deleted
 * @return target itself if it's found and deleted, null otherwise
 */
public E delete(E target){
    if(deleteReturn(target, 0)){
        return target;
    }
    else{
        return null;
    }
}

/**
 * This method removes certain target from BST
 * @param target target to be removed
 * @return true if successful, false if not
 */
public boolean remove(E target){
    return deleteReturn(target, 0);
}

/**
 * Recursive method to find target and remove it if found
 * @param target target to be removed
 * @param index index to keep track of which side are we headed
 * @return true if found and removed, false if not
 */
private boolean deleteReturn(E target, int index){
```

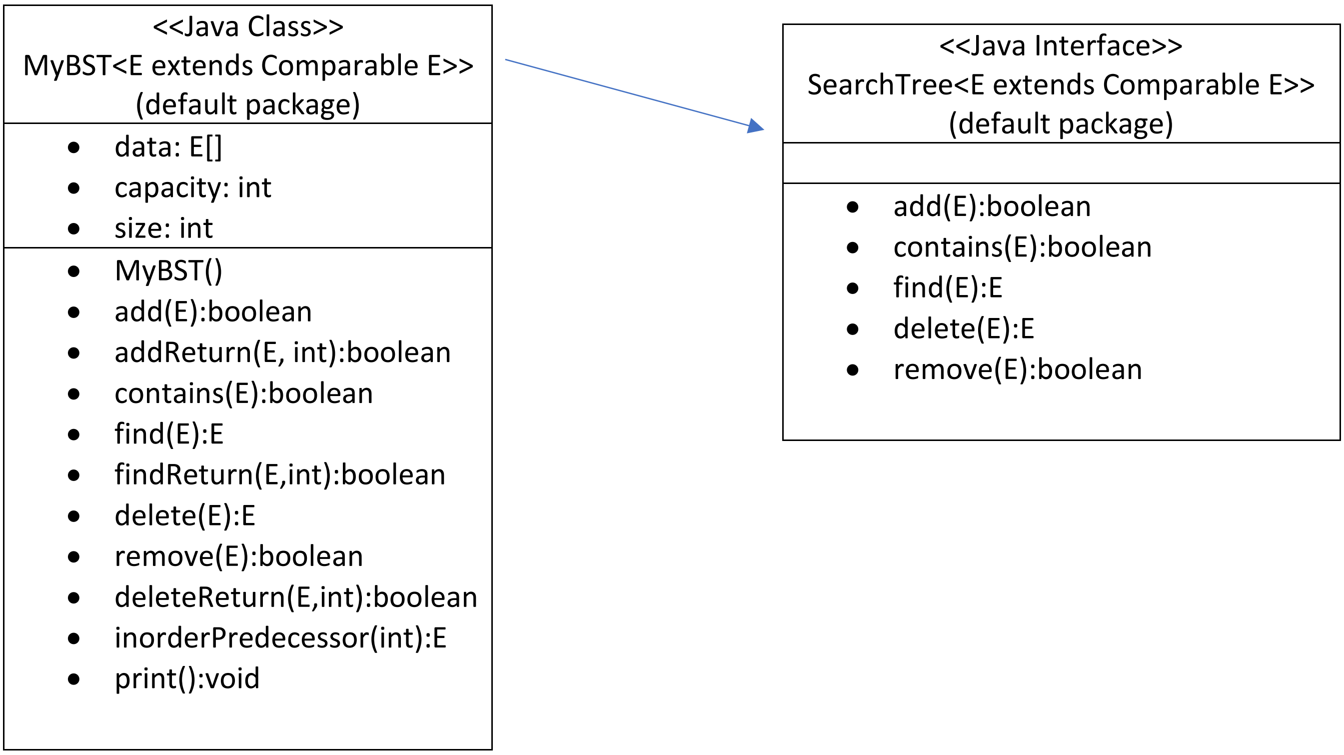
When removing an element from BST, there's a significant situation which is rather more complex. That situation is when the target element has two children. In that situation we need to find the conveniant element to replace this target element. To do that, we use this method and it returns the element which will replace target element

```java
/**
 * This method finds inorder predecessor for remove method
 * @param index index that searching will start at
 * @return inorder predecessor
 */
private E inorderPredecessor(int index){
```

This method prints the tree

```java
    /**
     * This method prints the tree
     */
    public void print(){
```

2- Class Diagrams



3- Problem Solving Approach

I used very similar algorithms when working with different methods because assignment was to implement a binary search tree and BSTs have very specific implementation. I created an array to store datas and datas' indexes got set based on their values. I used recursive methods to check each element, if data is smaller than current element, method gets called again this time with (x*2+1)th element which is basically left node. If it is bigger, method gets called again this time with (x*2+2)th element which is right node. If they are equal, that means method found its target.

4- Test Cases

```
public class Driver {
    public static void main(String[] args) throws Exception {

        MyBST<String> mybinst = new MyBST<>();

        mybinst.add("burcu");
        mybinst.add("gizem");
        mybinst.add("alper");
        System.out.println("'burcu', 'gizem', 'alper' are added");
        mybinst.print();
        mybinst.add("ibrahim");
        mybinst.add("tutku");
        System.out.println("'ibrahim', 'tutku' are added");
        mybinst.print();
        mybinst.add("elif");
        mybinst.add("melissa");
        mybinst.add("enes");
        System.out.println("'elif', 'melissa', 'enes' are added");
        mybinst.print();
        mybinst.delete("alper");
        mybinst.add("sude");
        mybinst.add("bugra");
        System.out.println("'alper' is removed, 'sude', 'bugra' are added");
        mybinst.print();
        mybinst.add("emre");
        mybinst.remove("gizem");
        System.out.println("'gizem' is removed, 'emre' is added");
        mybinst.print();

    }
}
```

5- Running Command and Results

```
'burcu', 'gizem', 'alper' are added
Parent: burcu       Left child: alper       Right child: gizem
Parent: alper       Left child: null       Right child: null
Parent: gizem       Left child: null       Right child: null
'ibrahim', 'tutku' are added
Parent: burcu       Left child: alper       Right child: gizem
Parent: alper       Left child: null       Right child: null
Parent: gizem       Left child: null       Right child: ibrahim
Parent: ibrahim       Left child: null       Right child: tutku
Parent: tutku       Left child: null       Right child: null
'elif', 'melissa', 'enes' are added
Parent: burcu       Left child: alper       Right child: gizem
Parent: alper       Left child: null       Right child: null
Parent: gizem       Left child: elif       Right child: ibrahim
Parent: elif       Left child: null       Right child: enes
Parent: ibrahim       Left child: null       Right child: tutku
Parent: enes       Left child: null       Right child: null
Parent: tutku       Left child: melissa       Right child: null
Parent: melissa       Left child: null       Right child: null
'alper' is removed, 'sude', 'bugra' are added
Parent: burcu       Left child: bugra       Right child: gizem
Parent: bugra       Left child: null       Right child: null
Parent: gizem       Left child: elif       Right child: ibrahim
Parent: elif       Left child: null       Right child: enes
Parent: ibrahim       Left child: null       Right child: tutku
Parent: enes       Left child: null       Right child: null
Parent: tutku       Left child: melissa       Right child: null
Parent: melissa       Left child: null       Right child: sude
Parent: sude       Left child: null       Right child: null
'gizem' is removed, 'emre' is added
Parent: burcu       Left child: bugra       Right child: enes
Parent: bugra       Left child: null       Right child: null
Parent: enes       Left child: elif       Right child: ibrahim
Parent: elif       Left child: null       Right child: emre
Parent: ibrahim       Left child: null       Right child: tutku
Parent: emre       Left child: null       Right child: null
Parent: tutku       Left child: melissa       Right child: null
Parent: melissa       Left child: null       Right child: sude
Parent: sude       Left child: null       Right child: null
```

TIME COMPLEXITY ANALYSIS

```java
public boolean add(E item){
    return addReturn(item, 0);
}
```

$\rightarrow T(\text{addReturn})$

```java
@SuppressWarnings("unchecked")
private boolean addReturn(E item, int index){
    if(index >= data.length){
        this.capacity = this.capacity*2;

        E[] tempArr = (E[])new Object[this.capacity];
        for (int i = 0; i < size; i++) {
            tempArr[i]=data[i];
        }
        data=tempArr;
    }

    if(data[index] == null){
        data[index] = item;
        size++;
        return true;
    }

    int bools = item.compareTo(data[index]);


    if(bools == 0){

        return false;
    }
    else if(bools>0){

        return addReturn(item, index*2 + 2);
    }
    else{

        return addReturn(item, index*2 + 1);
    }
}
```

$O(1)$

$O(n)$

$O(n)$

$O(1)$

$O(1)$

$O(1)$

T_n

$T_B = O(\log n)$

$T_W = O(n)$

$T(n) = O(n)$

```java
 * @param target target to be found
 * @return true if found, false if not
 */
public boolean contains(E target){
    return findReturn(target, 0);
}

/**
 * This method finds if a certain target exists in BST
 * @param target target to be found
 * @return target itself if found, else null
 */
public E find(E target){
    if(findReturn(target, 0)){
        return target;
    }
    else{
        return null;
    }
}
```

$\rightarrow O(\log n)$

$O(\log n)$

$O(1)$

$T(n) = O(\log n)$

```java
private boolean findReturn(E target, int index){

    if(data[index] == null || index >= data.length){
        return false;
    }

    int bools = target.compareTo(data[index]);

    if(bools == 0){
        return true;
    }
    else if(bools>0){
        return findReturn(target, index*2 + 2);
    }
    else{
        return findReturn(target, index*2 + 1);
    }
}

/**
 * This method deletes certain target from BST
 * @param target target to be deleted
 * @return target itself if it's found and deleted, null otherwise
 */
public E delete(E target){
    if(deleteReturn(target, 0)){
        return target;
    }
    else{
        return null;
    }
}
```

$\Theta(1)$

$T(n)$

$T_B = \Theta(1)$

$T_W = \Theta(\log n)$

$T(n) = O(\log n)$

$O(n)$

$\Theta(1)$

```java
*/
public boolean remove(E target){
    return deleteReturn(target, 0);
}
```

$O(n)$

```java
private boolean deleteReturn(E target, int index){
    if(data[index] == null || index >= data.length){

        return false;
    }
    int bools = target.compareTo(data[index]);

    if(bools == 0){
        if(data[index*2+1] == null && data[index*2+2] == null){
            data[index] = null;
            size--;
            return true;
        }
        if(data[index*2+1] == null && data[index*2+2] != null){
            data[index] = data[index*2+2];
            data[index*2+2] = null;
            size--;
            return true;
        }
        else if(data[index*2+1] != null && data[index*2+2] == null){
            data[index] = data[index*2+1];
            data[index*2+1] = null;
            size--;
            return true;
        }
        else{
            data[index] = inorderPredecessor(index*2+1);
            size--;
            return true;
        }
    }
    else if(bools>0){
        return deleteReturn(target, index*2 + 2);
    }
    else{
        return deleteReturn(target, index*2 + 1);
```

$\Theta(1)$

$\Theta(1)$

$\Theta(1)$

$O(n)$

T_n

$T_B = O(\log n)$

$T_W = O(n)$

$T_n = O(n)$

```java
    private E inorderPredecessor(int index){
        E tempE;

        while(data[index*2+2] != null){
            index = index*2 + 2;
        }
        tempE = data[index];
        remove(data[index]);
        return tempE;
    }

    /**
     * This method prints the tree
     */
    public void print(){
        int i=0;
        try { while(i<=data.length){
            if(data[i]!=null){
                System.out.print("Parent: " + data[i] + "        ");
                System.out.print("Left child: " + data[i*2+1] + "        ");
                System.out.println("Right child: " + data[i*2+2]);
            }
            i++;
        }} catch(ArrayIndexOutOfBoundsException e){};
    }
}
```

$O(n)$

$\theta(1)$

$T_n = O(n)$

$\Theta(n)$

$\theta(1)$

$T_n = \Theta(n)$