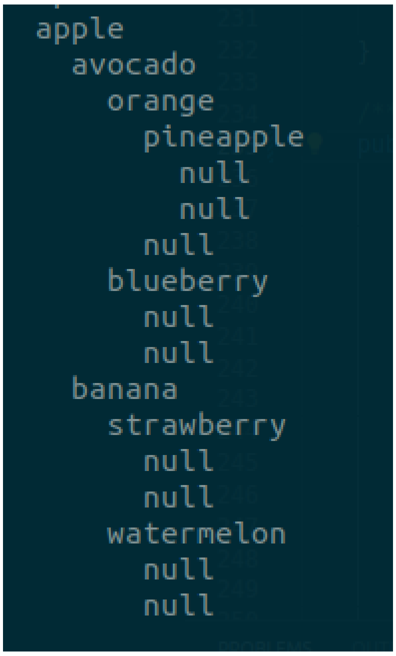
GTU Department of Computer Engineering
CSE 222/505 - Spring 2022
Homework 5 Report

Emirkan Burak Yılmaz
1901042659



1. SYSTEM REQUIREMENTS

Q1.

- **a)** Calculate the total depth of the nodes in a complete binary tree of node n.
- **b)** Calculate the average number of comparisons for a successful search in a binary search tree which is structured as complete binary tree.
- **c)** What is the number relationship between number of internal nodes and leave nodes in full binary tree?

Q2.

Implement a quadtree structure for 2D point data. Observe the required steps for insertion. Display the quadtree as representation of general trees in our textbook.

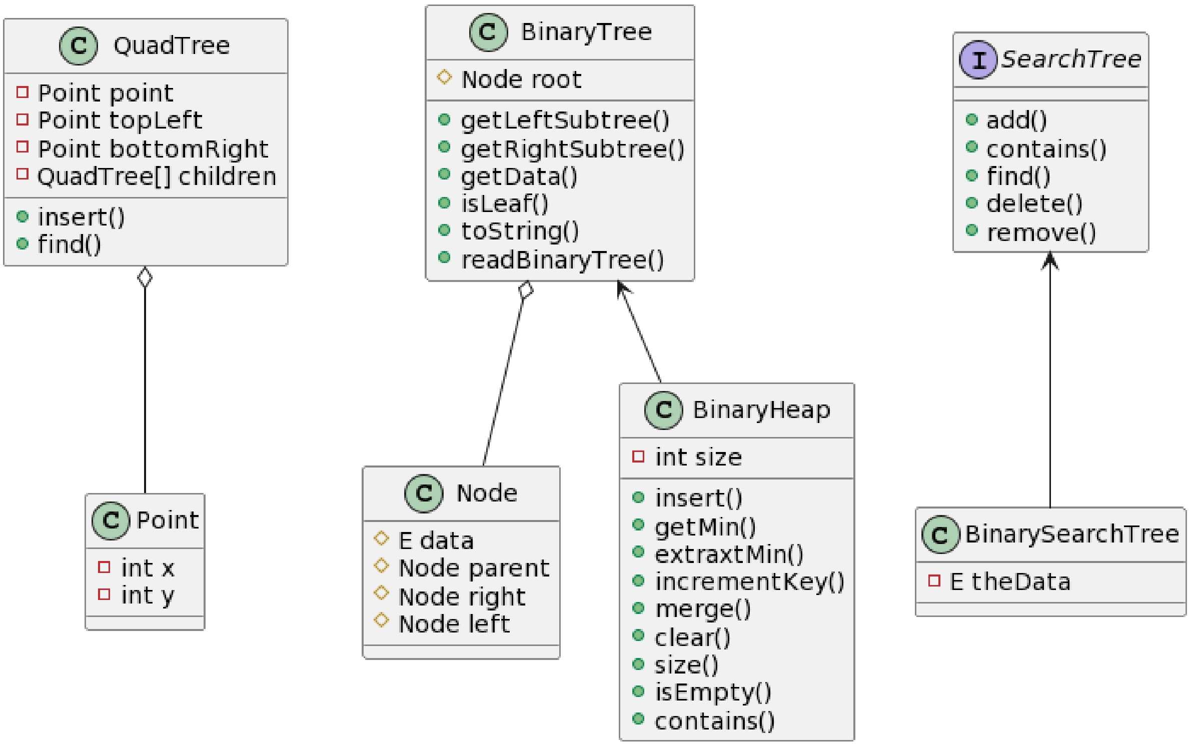
Q3.

Implement a binary min-heap which extends the BinaryTree class using node link structure. Make sure the heap satisfies the structural property of being a complete tree besides the min-heap order property. Also add two extra method incrementKey and merge. The method incrementKey increments the key value of an element. The method merge merges two heap structures and produce another heap.

Q4.

Implement an array based binary search tree which implements the SearchTree interface.

2. CLASS DIAGRAMS



3. PROBLEM SOLUTION APPROACH

Q1.

a) First, total depth depends on the total number of nodes in tree. However, for perfect trees it can be defined with the height of the tree. So, we can define the total depth of complete binary tree by using perfect trees. with its height. Think a complete binary tree which has N node, and its height is h (logN + 1). If we ignore the nodes at last level, then the remaining tree is perfect tree. So, we can find the solution as sum of two values which are the total depth of the perfect tree and sum of the depth of the nodes at the last level.

b) The number of comparisons for successful search in a binary search tree depends on the depth of the target node. There must be n comparison to find a target node at depth n. So average number of comparisons for a successful search operation is the sum of depth of all nodes divided by the number of nodes. In other words, it is total depth of the nodes divided by the number of nodes.

c) In full binary tree the number of leaf node is 1 more than the number of internal nodes. The total number of nodes is the sum of internal and leaf nodes which is always resulted in odd number. So, the total number of nodes in full binary tree must be odd.

Q2. The quadtree is a tree data structure in which each internal node has exactly four children. Quadtrees are used to partition a two-dimensional space by recursively subdividing it into four quadrants/regions.

First, point class is implemented as inner class in QuadTree class to represent 2D point data. Quadtree is a special node which has 4 children to represent other quadtrees. It’s important to emphasize that each node are sub-quadtrees. There are two point data fields to keep the boundary of region as top-left and bottom-right, and one point field for to insert a point inside the quadtree. The quadtree implementation done with manipulating the existing node as three different interpretations. A node can be interpreted as region node, point node or empty node. Region nodes are inner nodes, and they are parent of point nodes and empty nodes. They have 4 child nodes to represent each subregion and each subregion can contain point or sub subregions. The point nodes are the leaf nodes that contain only 2D point data. The empty nodes are the leaf nodes to indicate an empty place for new point insertions.

Insertion done with of subdividing of quadrants/regions till find an empty node or empty region. In each subdividing operation, algorithm eliminates unrelated regions by selecting one of the four quadrant which are TP (top-left), TR (top-right), BL (bottom-left), BR (bottom-right). If subdividing ends at a point node, then convert that point node to region node to fit two points into one region. On the other hand, subdividing ends at an empty node convert empty node to point node by setting its point field to the target point.

Searching target point is similar to insertion. First find its region by subdividing. If the subdividing is ended at an empty node, then the target point does not exist in the quadtree. On the other hand, if subdividing is ended at point node than compere the target point and the existing one to get the result.

Q3. BinaryHeap is a min-heap and implemented by using node-link structure. Each child larger or equal to its parent and each parent smaller or equal to its child. After operation like insert or extract some adjustments are required to satisfy the min-heap property. That’s why traversing both upward and downward is necessary. Therefore, I used a Node structure which has references for children and parent nodes. On the other hand, to maintain structural property of complete binary tree, insertion and deletion must be done by modifying last node (most right node at the last level). For that matter, there exist two possible solutions. First one is keeping the last node and using its parent for inserting or deleting keys. The second one is using the data of total number of nodes at the tree to find the next insertion position or last inserted item for deletion. First solution usually takes constant time, but overall time complexity is O(logN). Second solutions always start at root and traverse down by recursively and has θ(logN) time complexity. I prefer to implement second solution, because I find it much more challengeable and different perspective for learning purposes.

Insertions are done by adding a new node as last node of the tree and rearranging the nodes by upward traversing to maintain min-heap property. For extraction, the value of the root node saved to return later and root data change with the data of the last node. After saving data of last node, remove the last node and make sure the min-heap property isn’t violated. For that traverse downward by checking min-heap property of the sub trees.

Q4. Binary trees can be represented by arrays. To maintain parent-child relationship, array contains the tree items as level ordered tree traversal. So, with this order child of item at index i can be found by multiplying by two and adding 1 or 2 for left-right child. Similarly, the parent of item at index i can be found by subtracting one and dividing by two. Of course, arrays are fixed size memory spaces, after some point we need to resize it. For resizing I choose the increase the capacity of the array as 2*c - 1 (c: current capacity of the tree). With this design array size increases according to height of the tree. So, for degenerate tree structure it is resulted as so many empty memory spaces. On the other hand, for complete binary tree structure each memory cell used efficiently.

The class BinaryHeap also has the extra methods merge and incrementKey. The method merge first converts two heap structure to arrays by doing level order traversing and then inserts these keys into a new heap. The method incrementKey takes a new key value which supposed to has higher priority than current one. First a search algorithm is applied to find the current key. If the key is found, then current key is replaced with the new key and some adjustments are done to maintain min-heap property.

4. TIME COMPLEXITY OF METHODS

BinaryHeap		
Method	**Time Complexity**	
	Best Case	**Worst Case**
insert	Θ(1)	Θ(logn)
extractMin	Θ(1)	Θ(logn)
incrementKey	Θ(1)	Θ(n)
contains	Θ(1)	Θ(n)
merge	Θ(n)	
size	Θ(1)	
clear	Θ(n)	
isEmpty	Θ(1)	

BinarySearchTree			
Method	**Time Complexity**		
	Best Case	**Worst Case**	**Average Case**
add	Θ(1)	Θ(n)	Θ(logn)
contains	Θ(1)	Θ(n)	Θ(logn)
find	Θ(1)	Θ(n)	Θ(logn)
delete	Θ(1)	Θ(n)	Θ(logn)
remove	Θ(1)	Θ(n)	Θ(logn)

5. TEST CASES

Test Cases for class QuadTree

T1. Insert a point which is inside the boundary of the quadtree.

T2. Insert a point which is out of quadtree region.

T3. Search both an existing and non-existing item.

```java
public class TestQuadTree {
    public static void test1() {
        // define an QuadTree which is bounded (0, 0) as topLeft, (100, 100) bottomRight
        QuadTree t = new QuadTree(0, 0, 100, 100);
        System.out.println("Initial quadtree");
        System.out.println(t);
        insertPoint(t, 30, 30);
        insertPoint(t, 20, 15);
        insertPoint(t, 115, 25); // invalid point
        insertPoint(t, -5, 17); // invalid point
        insertPoint(t, 0, -35); // invalid point
        insertPoint(t, 50, 40);
        insertPoint(t, 10, 12);
        insertPoint(t, 40, 20);
        insertPoint(t, 25, 60);
        insertPoint(t, 15, 25);
        // find the points in QuadTree
        System.out.println(t);
        System.out.printf("Find (15, 25): %b\n", t.find(15, 25));
        System.out.printf("Find (50, 40): %b\n", t.find(50, 40));
        System.out.printf("Find (30, 30): %b\n", t.find(30, 30));
        System.out.printf("Find (50, 70): %b\n", t.find(50, 70));
        System.out.printf("Find (150, 870): %b\n", t.find(150, 870));
    }

    private static void insertPoint(QuadTree t, int x, int y) {
        boolean r = t.insert(x, y);
        System.out.println(t);
        System.out.printf("INSERT (%d, %d): ", x, y);
        System.out.println(r ? "SUCCESS\n" : "FAILURE\n");
    }
}
```

Test Code

Test Cases for class BinaryHeap

T1. Insert item to the heap.

T2. Get the min value of the heap with extractMin method.

T3. Increment a key priority.

T4. Merge two heap.

```
public class TestBinaryHeap {
    public static void test1() {
        BinaryHeap<Integer> h = new BinaryHeap<>();
        int[] items = {12, 334, 53, 89, 1, 4, 54, 123, 324, 54, 12, 4, 4, 32, 23, 14, 89, 11, 7

        System.out.println(h);
        System.out.println("Initially the heap is empty\n");
        for (var k : items) {
            System.out.printf("INSERT %d\n", k);
            h.insert(k);
            System.out.println(h);
        }
        System.out.printf("\nsize: %d\n", h.size());

        int target1 = 12; // not exist in tree
        int target2 = 25; // exist in tree
        System.out.printf("contains %-3d: %b\n", target1, h.contains(target1));
        System.out.printf("contains %-3d: %b\n", target2, h.contains(target2));

        // extract the top item of heap and add it to the arrayList to observe
        // the items ordered in increasing order (HeapSort)
        ArrayList<Integer> list = new ArrayList<>();
        while (! h.isEmpty()) {
            System.out.printf("MinValue: %d\nExtract min\n\n", h.getMin());
            list.add(h.extractMin());
            System.out.println(h);
            System.out.printf("new size: %d\n\n", h.size());
        }

        System.out.printf("contains %-3d: %b\n", target1, h.contains(target1));
        System.out.printf("contains %-3d: %b\n", target2, h.contains(target2));

        // all the heap content
        System.out.println(list);
    }
```

Test Code 1

```java
public static void test2() {
    // create two heap
    BinaryHeap<String> h1 = new BinaryHeap<>();
    h1.insert("meat");
    h1.insert("peanut");
    h1.insert("egg");
    h1.insert("cheese");
    h1.insert("hamburger");
    h1.insert("hazalnut");
    h1.insert("almond");
    h1.insert("milk");
    System.out.println("heap1: ");
    System.out.println(h1);

    BinaryHeap<String> h2 = new BinaryHeap<>();
    h2.insert("apple");
    h2.insert("orange");
    h2.insert("banana");
    h2.insert("avocado");
    h2.insert("blueberry");
    h2.insert("strawberry");
    h2.insert("watermelon");
    h2.insert("pineapple");
    System.out.println("heap2: ");
    System.out.println(h2);

    var merged = h1.merge(h2);
    System.out.println("Merge heap1 and heap2: ");
    System.out.println(merged);

    if (merged.incrementKey("egg", "sausage"))
        System.out.println("Key value 'egg' incremented to 'sausage'");
    else
        System.out.println("Key value 'egg' cannot be incremented to 'sausage'");
    System.out.println(merged);
    if (merged.incrementKey("blue", "purple"))
        System.out.println("Key value 'blue' incremented to 'blue'");
    else
        System.out.println("Key value 'blue' cannot be incremented to 'blue'");
    System.out.println(merged);
}
```

Test Code 2

Tests Cases for class BinarySearchTree

- **T1.** Create a binary search tree by using add methods
- **T2.** Try to add an item which is already exist in the tree
- **T3.** Check if tree contains given item
- **T4.** Search the given item in the tree
- **T5.** Remove an item from the tree
    - i. Remove a node which is not exist in the tree
    - ii. Remove a node which has no child
    - iii. Remove a node which has one child
    - iv. Remove a node which has two children
- **T6.** Destroy the tree

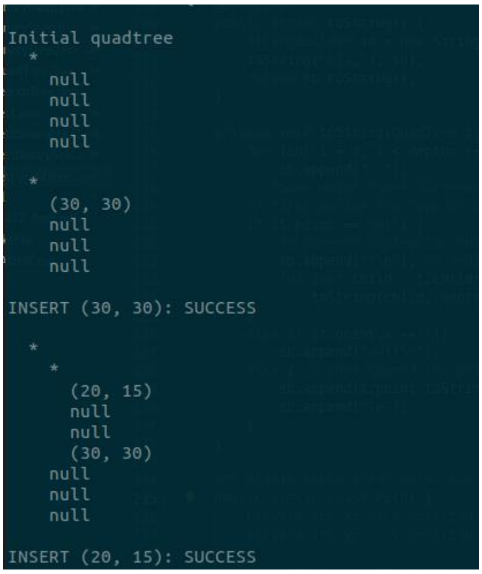
```java
public static void test1() {
    BinarySearchTree<Integer> t = new BinarySearchTree<>();
    int[] vals = {36, 25, 9, 49, 61, 1, 4, 16};

    System.out.println(t);
    System.out.println("Initially tree is empty\n");
    for (var v : vals) {
        System.out.printf("ADD %d\n\n", v);
        boolean r = t.add(v);
        System.out.println(t);
        if (r)
            System.out.printf("%d is added properly\n\n", v);
        else
            System.out.printf("%d isn't added\n\n", v);
    }

    int target1 = 12; // not exist in tree
    int target2 = 25; // exist in tree
    System.out.printf("contains %-3d: %b\n", target1, t.contains(target1));
    System.out.printf("contains %-3d: %b\n", target2, t.contains(target2));

    target1 =  112; // not exist in tree
    target2 =  49; // exist in tree
    System.out.printf("find %-3d: %b\n", target1, t.find(target1));
    System.out.printf("find %-3d: %b\n", target2, t.find(target2));
}
```

Test Code

6. RUNNING AND RESULTS

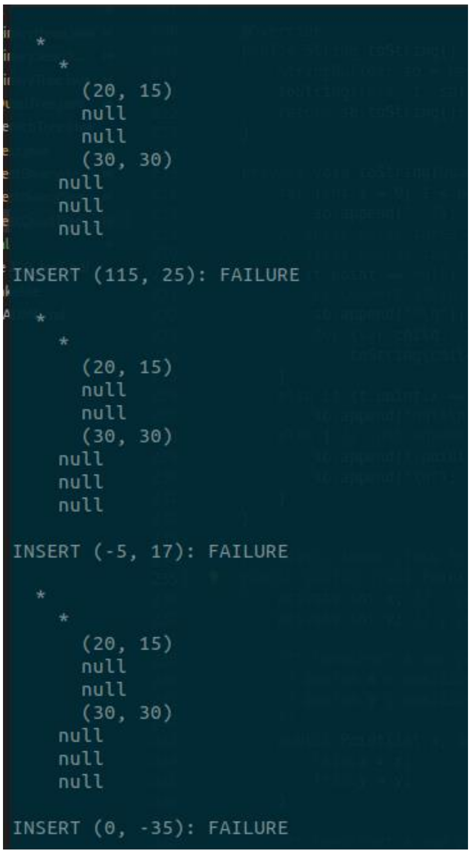
Test results for class QuadTree

T1.



```
Initial quadtree
 *
    null
    null
    null
    null

 *
    (30, 30)
    null
    null
    null

INSERT (30, 30): SUCCESS

 *
    *
      (20, 15)
      null
      null
      (30, 30)
    null
    null
    null

INSERT (20, 15): SUCCESS
```

T2.



```
 *
    *
      (20, 15)
      null
      null
      (30, 30)
    null
    null
    null

INSERT (115, 25): FAILURE

 *
    *
      (20, 15)
      null
      null
      (30, 30)
    null
    null
    null

INSERT (-5, 17): FAILURE

 *
    *
      (20, 15)
      null
      null
      (30, 30)
    null
    null
    null

INSERT (0, -35): FAILURE
```

QuadTree is bounded (0, 0) as top Left, (100, 100) bottom Right. So, the x value of point 115 is out of bound.

T3.

```
  *
    *
      *
        null
        null
        (10, 12)
        (20, 15)
      (40, 20)
      (15, 25)
      (30, 30)
    (50, 40)
    (25, 60)
    null

Find (15, 25): true
Find (50, 40): true
Find (30, 30): true
Find (50, 70): false
Find (150, 870): false
```

Test results for class BinaryHeap

T1.

```
  null

Initially the heap is empty

INSERT 12
  12
    null
    null

INSERT 334
  12
    334
      null
      null
    null

INSERT 53
  12
    334
      null
      null
    53
      null
      null
```

```
  1
    12
      334
        null
        null
      89
        null
        null
    4
      53
        null
        null
      54
        null
        null

INSERT 123
  1
    12
      123
        334
          null
          null
        null
      89
        null
        null
    4
      53
        null
        null
      54
        null
        null
```

T2.

```
  89
    123
      324
        null
        null
      421
        null
        null
    334
      432
        null
        null
      789
        null
        null

new size: 7

MinValue: 89
Extract min

  123
    324
      789
        null
        null
      421
        null
        null
    334
      432
        null
        null
      null

new size: 6

MinValue: 123
```

```
heap1:
  almond
    egg
      milk
        peanut
          null
          null
        null
      hamburger
        null
        null
    cheese
      meat
        null
        null
      hazalnut
        null
        null

heap2:
  apple
    avocado
      orange
        pineapple
          null
          null
        null
      blueberry
        null
        null
    banana
      strawberry
        null
        null
      watermelon
        null
        null
```

T3.

```
Merge heap1 and heap2:
  almond
    apple
      egg
        peanut
          pineapple
            null
            null
          null
        milk
          null
          null
      avocado
        hamburger
          null
          null
        banana
          null
          null
    blueberry
      cheese
        orange
          null
          null
        meat
          null
          null
      hazalnut
        strawberry
          null
          null
        watermelon
          null
          null
```

```
contains 12 : false
contains 25 : false
[1, 4, 4, 4, 11, 12, 12, 14, 23, 23, 32, 53, 54, 54, 67, 89, 89, 123, 324, 334, 421, 432, 789]
```

After extracting all the keys, the keys are ordered as increasing order. So, extracting works properly.

T4.

```
Key value 'egg' incremented to 'sausage'
  almond
    apple
      milk
        peanut
          pineapple
            null
            null
          null
        sausage
          null
          null
      avocado
        hamburger
          null
          null
        banana
          null
          null
    blueberry
      cheese
        orange
          null
          null
        meat
          null
          null
      hazalnut
        strawberry
          null
          null
        watermelon
          null
          null
```

```
Key value 'blue' cannot be incremented to 'blue'
```

Test results for class BinarySearchTree

T1.

```
  null

Initially tree is empty

ADD 36

  36
    null
    null

36 is added properly
ADD 25

  36
    25
      null
      null
    null

25 is added properly
ADD 9

  36
    25
      9
        null
        null
      null
    null

9 is added properly
ADD 49

  36
```

T2.

```
4 is added properly
ADD 16

  36
    25
      9
        1
          null
          4
            null
            null
        16
          null
          null
      null
    49
      null
      61
        null
        null

16 is added properly
contains 12 : false
contains 25 : true
find 112: false
find 49 : true
```

T3.

```
  H
    D
      B
        A
          null
          null
        C
          null
          null
      F
        E
          null
          null
        G
          null
          null
    L
      J
        I
          null
          null
        K
          null
          null
      N
        M
          null
          null
        O
          null
          P
            null
            null

contains X: false
contains X: false
contains K: true
contains K: false
contains O: true
contains O: false
contains N: true
contains N: false
```

```
  H
    D
      B
        A
          null
          null
        C
          null
          null
      F
        E
          null
          null
        G
          null
          null
    L
      J
        I
          null
          null
        null
      null

Destroy the tree
  null
```