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

Mert Gürşimşir
1901042646

PART 1

(1) - a

Total depth:

- Minimum: ($\sum_{i=1}^{h-1} i2^{i-1}$) + h
- Maximum: $\sum_{i=1}^{h} i2^{i-1}$

Node number at height h is 2^{h-1} because at each level, number of nodes is multiplied by 2. So for first level, there is 1 node and its depth is 1. For second level, there are 2 nodes and depth of each one is 2. So depth value is increasing by 1 every time.

We should split solution in 2 parts as maximum and minimum because in a complete tree, last level may have 2^{h-1} nodes at most but 1 node at least. So we should consider these 2 situations.

(1) - b

For the best case, there is only 1 comparison that is target at root → $\theta(1)$

For the worst case, there are number of "height" comparisons. Height is logn (n: number of nodes) for complete tree. So worst case is → θ(logn)

We can say number of comparisons for a successful search operation in a binary search tree is O(logn).

(1) - c

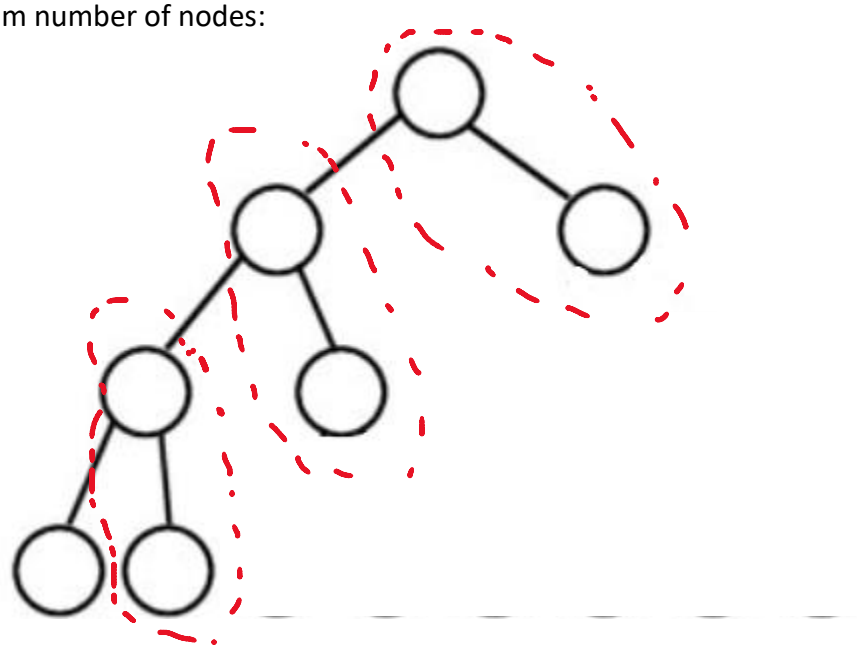
Yes there is:

- Maximum number of nodes: 2^h - 1
- Minimum number of nodes: 2h - 1

A full binary tree is a binary tree where all nodes have either 2 children or 0 children (the leaf nodes).

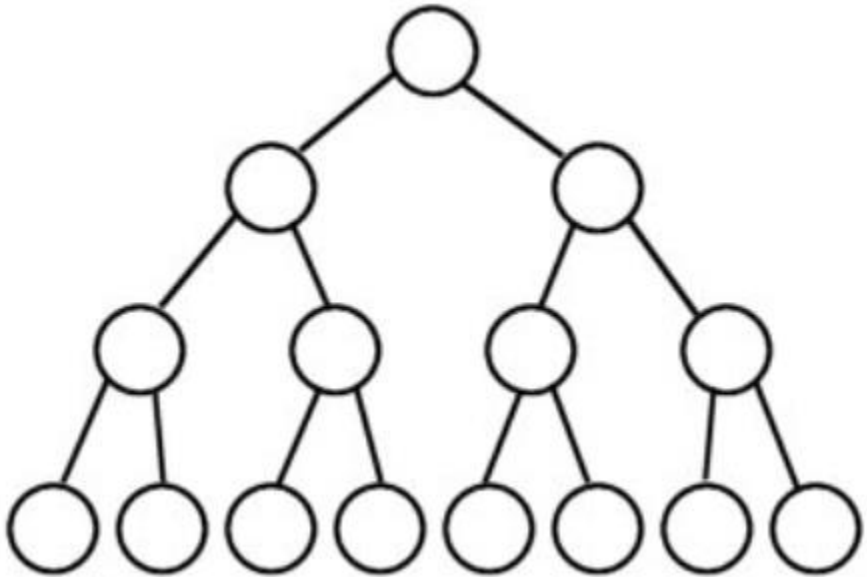
So if height is 4:

- Minimum number of nodes:



We can group them like these and there are h-1 number of pairs and also 1 more pair. So this can be represented as 2h - 1

- Maximum number of nodes:



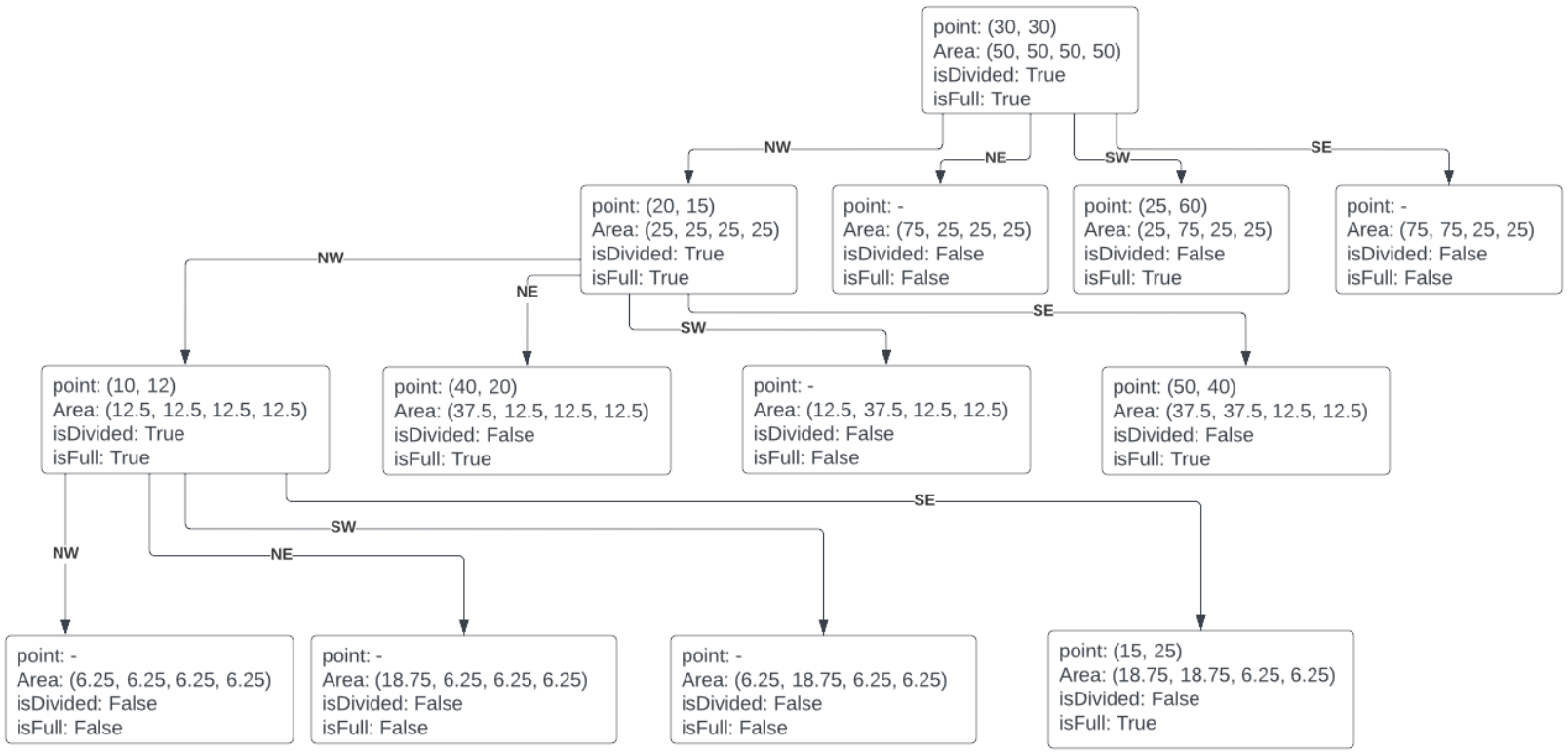
We have total of 2^h - 1 nodes is full binary tree for maximum.

Number of leaves is 1 more than the number of internal nodes in a full binary tree because full binary tree has either 0 or 2 children. At the end, all nodes have to end with at least 2 leaves (number of leaves = number of internal nodes + 1). So if there is n nodes:

- Number of internal nodes : (n-1) / 2
- Number of leaves : (n+1) / 2

(2)

In quadtree structure, every node contains information for the next nodes. Also information of the point can be included in a node.



Nodes traversed during each insertion:

- (30, 30) : Directly added to root

- (20, 15) : Only (30, 30) node is traversed during insertion. Since that node is full, we have to divide the area to 4 quadrants. Point (20, 15) is proper for northwest of the area. So it is added there.

- (50, 40) : (30, 30) and (20, 15) are traversed during insertion. We looked at first node and it is full. So we have to go northwest of the area because (50, 40) is there. We go to the northwest but that node is also full. So we have to split that northwest quarter into 4 again. (50, 40) is at the southeast of the northwest, so it is placed there.

- (10, 12) : (30, 30) and (20, 15) are traversed during insertion. We looked at first node and it is full. So we have to go northwest of the area because (10, 12) is there. We go to the northwest but that node is also full. Then we go to the northwest of the northwest and that area is proper.

- (40, 20) : (30, 30) and (20, 15) are traversed during insertion. We looked at first node and it is full. So we have to go northwest of the area because (40, 20) is there. We go to the northwest but that node is also full. Then we go to the northeast of the northwest and that area is proper.

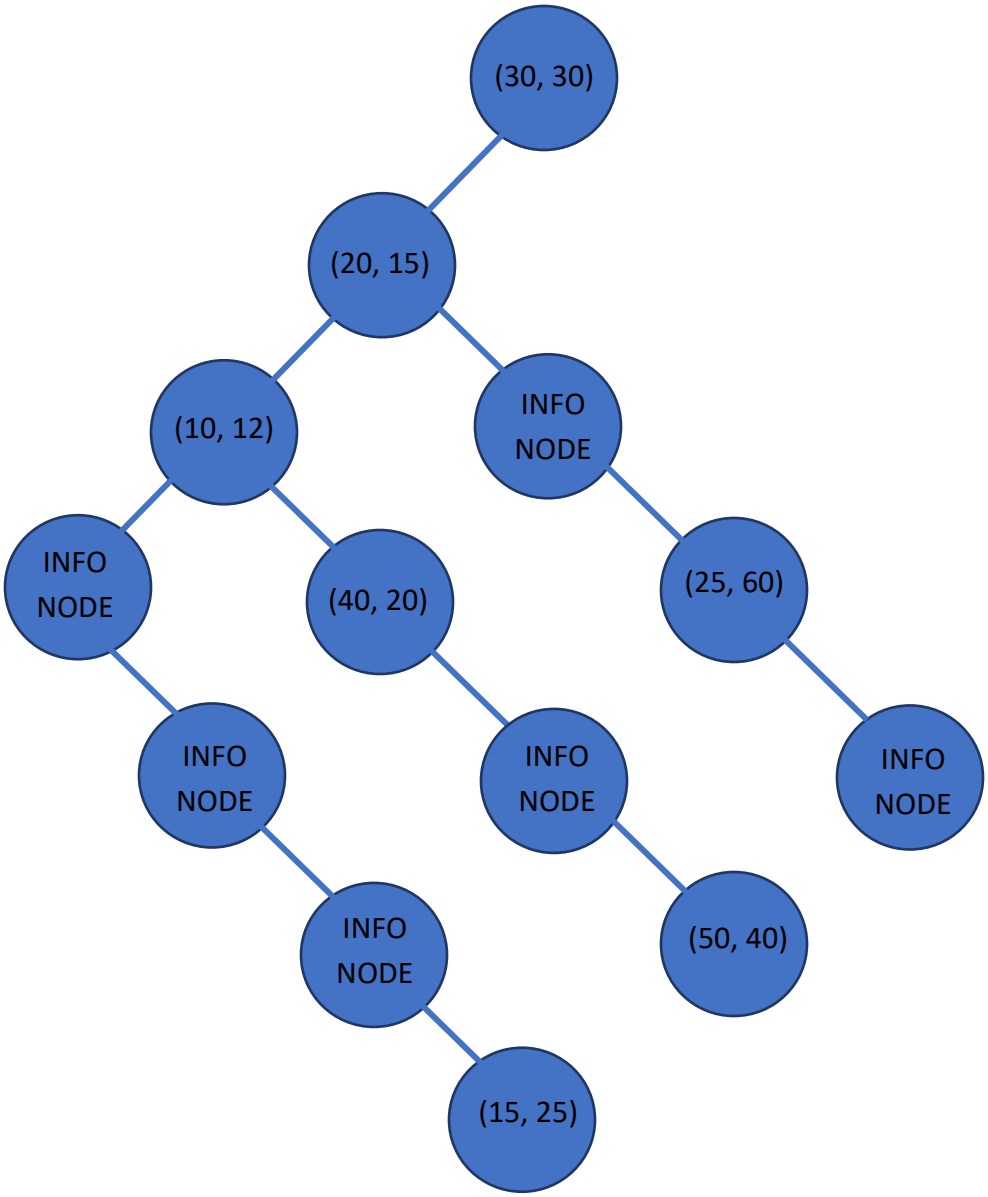
- (25, 60) : (30, 30) node is traversed. Since that node is full, we have to add this point to proper quadrant. Point (25, 60) is proper for southwest of the area. So it is added there.

- (15, 25) : (30, 30), (20, 15), and (10, 12) are traversed during insertion. First node is full so we go to the proper quadrant and it is northwest. Since that node is also full (20, 15), we go to proper quadrant which is northwest. Since that node is also full (10, 12), we divide that node into 4 quadrants again. (15, 25) is proper for the southeast part and it is added there.

```
Nodes traversed during insertion of (30, 30):

**********
Nodes traversed during insertion of (20, 15):
(30, 30)

**********
Nodes traversed during insertion of (50, 40):
(30, 30)
(20, 15)

**********
Nodes traversed during insertion of (10, 12):
(30, 30)
(20, 15)

**********
Nodes traversed during insertion of (40, 20):
(30, 30)
(20, 15)

**********
Nodes traversed during insertion of (25, 60):
(30, 30)

**********
Nodes traversed during insertion of (15, 25):
(30, 30)
(20, 15)
(10, 12)
```

"-" signs are inserted according to depth of the node.

```
Resulting tree after insertion of (30, 30):
(30, 30)

*********************************
Resulting tree after insertion of (20, 15):
(30, 30)
-(20, 15)
-null
-null
-null

*********************************
Resulting tree after insertion of (50, 40):
(30, 30)
-(20, 15)
--null
--null
--null
--(50, 40)
-null
-null
-null

*********************************
Resulting tree after insertion of (10, 12):
(30, 30)
-(20, 15)
--(10, 12)
--null
--null
--(50, 40)
-null
-null
-null

*********************************
Resulting tree after insertion of (40, 20):
(30, 30)
-(20, 15)
--(10, 12)
--(40, 20)
--null
--(50, 40)
-null
-null
-null

*********************************
Resulting tree after insertion of (25, 60):
(30, 30)
-(20, 15)
--(10, 12)
--(40, 20)
--null
--(50, 40)
-null
-(25, 60)
-null

*********************************
Resulting tree after insertion of (15, 25):
(30, 30)
-(20, 15)
--(10, 12)
---null
---null
---null
---(15, 25)
--(40, 20)
--null
--(50, 40)
-null
-(25, 60)
-null
```


(30, 30)
(20, 15)
(10, 12)
INFO NODE
INFO NODE
(40, 20)
(25, 60)
INFO NODE
INFO NODE
INFO NODE
INFO NODE
(50, 40)
(15, 25)

(3)

HEAP - ANALYZING METHODS & EXPLANATION OF METHODS

METHOD	EXPLANATION	ANALYSIS
endNode	This method is to find the node that new node is going to be inserted. It uses LinkedList to store nodes for efficiency. Method adds nodes to LinkedList one by one and removes them in order until it finds a node that either left or right nodes are empty. This process happens for height times and height of a complete binary tree is logn.	$\theta(logn)$
lastInHeap	This method finds the node at the end of the heap. Works very similar to endNode method. Difference is this method finds bottom, rightmost node. So finding the last node also happens for height times and height of a complete binary tree is logn.	$\theta(logn)$
add	This method inserts given element to the heap. It uses endNode method to identify the inserting place. Then new element is inserted. We have to go till to up (at worst case) to correctly place new element. This process also takes logn times. Since we use endNode method, it is θ instead of O.	$\theta(logn)$
remove	This method removes the proper element. It uses lastInHeap method to identify the last element in heap. Then it changes that node with the root node and delete the last node. Then root node's data has to be in proper place so it traverses until its proper place. This process also takes logn time at worst case. Since we use lastInHeap method, it is θ instead of O.	$\theta(logn)$
merge	This method merges two given heaps. It stores datas in seperate heap and adds elements of two heaps one by one (n: node number of one heap, m: node number of another heap). We have to traverse all n nodes and insert (takes logn) them. So overall, running time is nlogn.	$\theta(nlogn)$ + $\theta(mlogm)$

mergeAdd	This method is used by merge method. Adds all the elements starting from given node to given heap. We have to traverse all n nodes and insert (takes logn) them. So overall, running time is nlogn.	$\theta(nlogn)$
incrementKey	Uses search method to find the node of the given element. Then traversed it to proper position. Method depends on search operation so it takes linear time for the worst case.	$O(n)$
search	Looks all the nodes one by one and stop sif it finds the element in the heap. Constant fort he best case, and linear for the worst case.	$O(n)$
toString	Creates StringBuilder and uses overloaded toString method.	$\theta(nlogn)$
toString (overloaded)	Running time depends on the indentation part. Height is logn for the heap and we have to add depth of each node to find number of indentations. We can say this method needs nlogn time (height is logn, there are n nodes).	$\theta(nlogn)$

(4)

METHOD	EXPLANATION	ANALYSIS
enlarge	Increases the capacity of the tree array. Makes copy of each element so it needs linear time.	θ(n)
find	Finds target and returns it. Calls recursive overloaded find method.	O(logn)
find (overloaded)	Searchs the array by eliminating half of the tree in every step recursively according to relationship between elements. So it needs logn time for the worst case (element doesn't exist), constant for the best case (element at the root).	O(logn)
contains	Finds target and returns true if it exists.	O(logn)
contains (overloaded)	Searchs the array by eliminating half of the tree in every step recursively according to relationship between elements. So it needs logn time for the worst case (element doesn't exist), constant for the best case (element at the root).	O(logn)
add	Inserts given item to the tree. Looks the array starting from beginning and traverse "left" (2*index + 1) if item is less than array[index] and "right" (2*index + 2) if item is bigger than array[index]. If there is no enough space, array is enlarged. Best case constant, worst case amortized logn.	O(logn)
delete	Deletes given target from the tree. Finds the item in logn times. Than array has to be changed after removing the item. For best case, target has no children and it is constant time. For worst case, target has 2 children. We need to find minimum at right for this situation. Finding minimum at right requires logarithmic time for the worst case. Then code adds elements to buffer and then adds them to the original array properly. This process requires nlogn time for the worst case.	O(nlogn)

deleteToBuffer	Used by delete method. Adds element to buffer from original array. Requires amortized constant time to add and linear for elements.	θ(n)
findMax	Finds maximum in the tree starting from given index. Requires logarithmic time.	θlog(n)
findMin	Finds minimum in the tree starting from given index. Requires logarithmic time.	θlog(n)
remove	Removes given element from the tree and returns true if operation is successful. Uses delete method. Requires same time.	O(nlogn)
emptyBuffer	Empties the buffer by assigning its size to 0. Requires constant time.	θ(1)
addBuffer	Adds given item to buffer array. Requires amortized constant time.	θ(1)
enlargeBuffer	Increases the capacity of the buffer array. Requires linear time for copying elements one by one.	θ(n)
toString	Returns a string representation of the tree in array format. Requires linear time to traverse all the element sone by one.	θ(n)
TreeString	Creates StringBuilder and uses overloaded toString method.	O(nlogn)
TreeString (overloaded)	Returns a string representation of the tree in tree format. Running time depends on the indentation part. Height is logn for average for the BST and we have to add depth of each node to find number of indentations. We can say this method needs nlogn time (height is logn, there are n nodes).	O(nlogn)

PART 2

SYSTEM REQUIREMENTS

To use heap, you need to create heap for the type you wanted:

```
Heap<Integer> tester = new Heap<Integer>();
```

Then you can add to heap by:

```
tester.add(6);
```

You can remove from the heap by (The element with highest priority will be removed):

```
tester.remove();
```

You can create another heap structure (tester2) and merge (tester and tester2) by (new heap structure will be returned):

```
tester.merge(tester2)
```

You can increment key of an element (first parameter element whose key is incremented, second parameter element after incremented). Element to be incremented must exist in the heap and element has to be incremented:

```
tester2.incrementKey(8, 80);
```

To use binary search tree, you need to create binary search tree for the type you wanted:

```
BinarySearchTree<Integer> tester = new BinarySearchTree<Integer>();
```

Then you can add to BST by:

```
tester.add(25);
```

You can delete an element from the BST by:

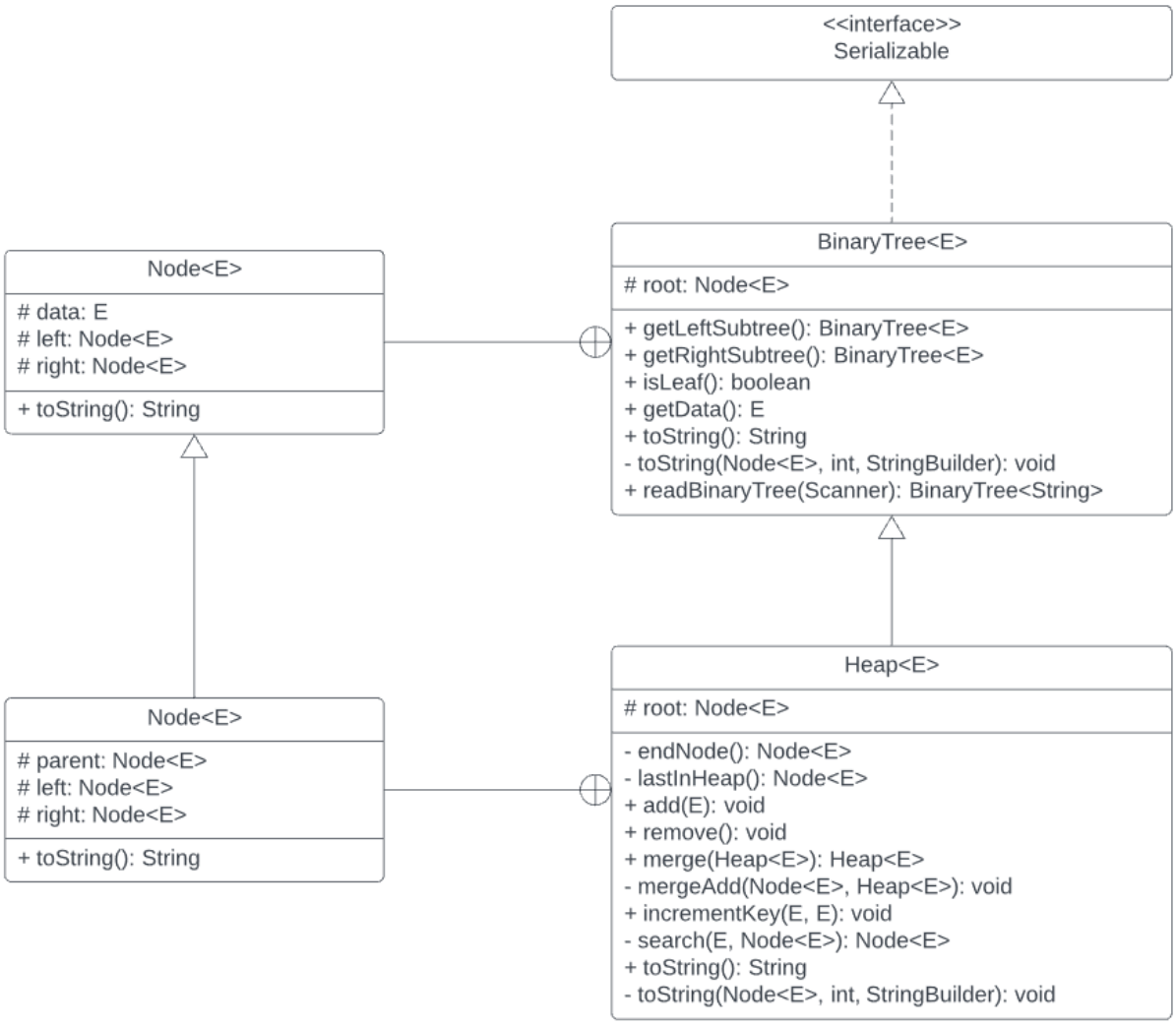
```
tester.delete(25);
```

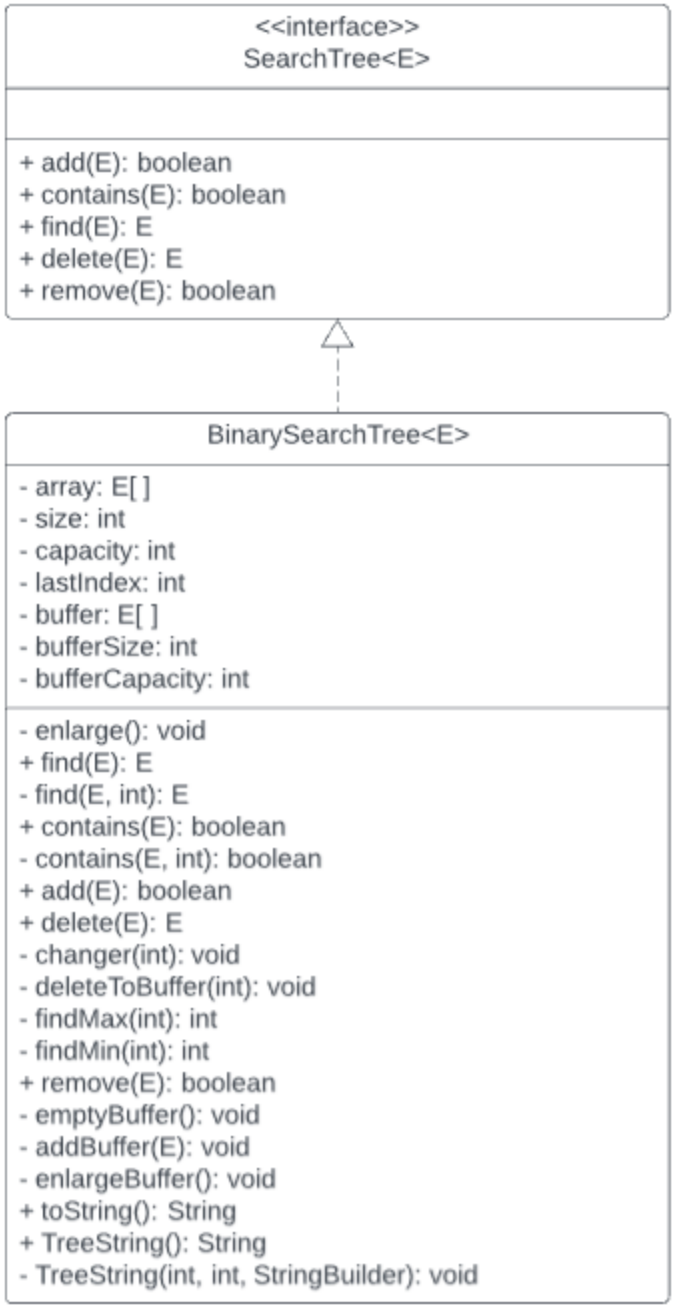
You can check if BST contains an element by “contains” and “find” methods. Contains returns true or false, find returns found value:

```
tester.contains(25)
```

```
tester.find(38)
```

CLASS DIAGRAMS




<<interface>>
SearchTree<E>
+ add(E): boolean
+ contains(E): boolean
+ find(E): E
+ delete(E): E
+ remove(E): boolean
BinarySearchTree<E>
- array: E[ ]
- size: int
- capacity: int
- lastIndex: int
- buffer: E[ ]
- bufferSize: int
- bufferCapacity: int
- enlarge(): void
+ find(E): E
- find(E, int): E
+ contains(E): boolean
- contains(E, int): boolean
+ add(E): boolean
+ delete(E): E
- changer(int): void
- deleteToBuffer(int): void
- findMax(int): int
- findMin(int): int
+ remove(E): boolean
- emptyBuffer(): void
- addBuffer(E): void
- enlargeBuffer(): void
+ toString(): String
+ TreeString(): String
- TreeString(int, int, StringBuilder): void

PROBLEM SOLUTION APPROACH

At first I need to learn the concepts very well before starting implementation so I read the book carefully and watch some videos about Trees. After I feel comfortable with the concepts, answering the first question did not take so long.

In second question, I searched about quadtree structure for 2D point data. There are so many different informations in the internet so it was hard for me to decide which one is correct. Then I decided the one that is more sensible. After that I added my points and track the behaviour of the quadtree.

In thirth question, I worked on binary heap first. After I understood the one with implemented with arrays, I implemented my own binary heap with node-link structure. It was hard to decide on more efficient implementation among different thoughts of myself but implementing the node-link structure was harder. It was a problem for me to going upwards in the tree. I decided on keep the parent information to make things easier for me and that solved my problem.

In fourth question, I worked on binary search tree. After I understood the one with implemented with node-lnik structure, I implemented my own binary search tree with array. Adding to tree and finding element in the tree were not too hard but deleting from the tree was challenging. I solved that problem with keeping another buffer array and exchanging elements between original and buffer arrays.

TEST CASES

HEAP:

```
Heap<Integer> tester = new Heap<Integer>();
System.out.printf("TESTING THE HEAP STRUCTURE\n");
System.out.printf("--------------------------\n");
System.out.printf("Adding 6, 18, 20, 28, 29, 37, 26, 76, and 32 to tester heap.\n");
tester.add(6);
tester.add(18);
tester.add(20);
tester.add(28);
tester.add(29);
tester.add(37);
tester.add(26);
tester.add(76);
tester.add(32);
System.out.printf("Heap after insertions:\n%s", tester);

System.out.printf("\n\n2 elements will be removed.\n");
tester.remove();
tester.remove();
System.out.printf("After removals:\n%s", tester);


System.out.printf("-------------------------------------------------\n");
Heap<Integer> tester2 = new Heap<Integer>();
System.out.printf("\nAdding 8 to second tester heap.\n");
tester2.add(8);
System.out.printf("Removing one element from second tester heap.\n");
tester2.remove();
System.out.printf("After these 2 operations second tester heap:\n%s", tester2);

System.out.printf("\n\nAdding 8, 30, 74, 89, 66, and 39 to second tester heap.\n");
tester2.add(8);
tester2.add(30);
tester2.add(74);
tester2.add(89);
tester2.add(66);
tester2.add(39);
System.out.printf("Second heap after insertions:\n%s", tester2);

System.out.printf("\n\nIncrementing key value of element 8 to 80 at the second heap.\n");
tester2.incrementKey(8, 80);
System.out.printf("Second heap after key incrementation:\n%s", tester2);

System.out.printf("-------------------------------------------------\n");
System.out.printf("Merging first and second heaps.\nResult:\n%s", tester.merge(tester2));



System.out.printf("-------------------------------------------------\n");
Heap<String> tester3 = new Heap<String>();
System.out.printf("\nAdding Palahniuk, Tarantino, Camus, Hetfield, Angie, and Murakami to last tester heap.\n");
tester3.add("Palahniuk");
tester3.add("Tarantino");
tester3.add("Camus");
tester3.add("Hetfield");
tester3.add("Angie");
tester3.add("Murakami");
System.out.printf("Last heap after insertions:\n%s", tester3);
tester3.remove();
tester3.remove();
tester3.remove();
System.out.printf("\n\nAfter 3 removals:\n%s", tester3);
```

BINARY SEARCH TREE:

```
BinarySearchTree<Integer> tester = new BinarySearchTree<Integer>();
System.out.printf("TESTING BINARY SEARCH TREE\n");
System.out.printf("-------------------------\n");
System.out.printf("Adding 25, 20, 36, 10, 22, 30, 40, 5, 12, 28, 38, and 48 to tree.\n");
tester.add(25);
tester.add(20);
tester.add(36);
tester.add(10);
tester.add(22);
tester.add(30);
tester.add(40);
tester.add(5);
tester.add(12);
tester.add(28);
tester.add(38);
tester.add(48);
System.out.printf("\nArray representation of the BST:\n%s\n",tester);
System.out.printf("Tree representation of the BST:\n%s\n",tester.TreeString());

System.out.printf("Deleting 25 from the tree.\n");
tester.delete(25);
System.out.printf("\nArray representation of the BST:\n%s\n",tester);
System.out.printf("Tree representation of the BST:\n%s\n",tester.TreeString());

System.out.printf("Deleting 48 from the tree.\n");
tester.delete(48);
System.out.printf("\nArray representation of the BST:\n%s\n",tester);
System.out.printf("Tree representation of the BST:\n%s\n",tester.TreeString());

if (tester.contains(25)) System.out.printf("Tree contains 25.\n");
else System.out.printf("Tree doesn't contain 25.\n");

if ((new Integer(38)).equals(tester.find(38))) System.out.printf("Tree contains 38.\n");
else System.out.printf("Tree doesn't contain 38.\n");

BinarySearchTree<String> tester2 = new BinarySearchTree<String>();
System.out.printf("-----------------------------------------------------------------\n");
System.out.printf("Adding lay, house, rat, lay, jack, that, milked, and cow to tree.\n");
tester2.add("lay");
tester2.add("house");
tester2.add("rat");
tester2.add("lay");
tester2.add("jack");
tester2.add("that");
tester2.add("milked");
tester2.add("cow");
System.out.printf("\nArray representation of the BST:\n%s\n",tester2);
System.out.printf("Tree representation of the BST:\n%s\n",tester2.TreeString());

System.out.printf("Deleting rat from the tree.\n");
tester2.delete("rat");
System.out.printf("\nArray representation of the BST:\n%s\n",tester2);
System.out.printf("Tree representation of the BST:\n%s\n",tester2.TreeString());

System.out.printf("Deleting house from the tree.\n");
tester2.delete("house");
System.out.printf("\nArray representation of the BST:\n%s\n",tester2);
System.out.printf("Tree representation of the BST:\n%s\n",tester2.TreeString());

if (tester2.contains("milked")) System.out.printf("Tree contains milked.\n");
else System.out.printf("Tree doesn't contain milked.\n");

if ("mert".equals(tester2.find("mert"))) System.out.printf("Tree contains mert.\n");
else System.out.printf("Tree doesn't contain mert.\n");
```

RUNNING COMMAND AND RESULTS

HEAP:

```
TESTING THE HEAP STRUCTURE
--------------------------
Adding 6, 18, 20, 28, 29, 37, 26, 76, and 32 to tester heap.
Heap after insertions:
6
-18
--28
---76
----null
----null
---32
----null
----null
--29
---null
---null
-20
--37
---null
---null
--26
---null
---null


2 elements will be removed.
After removals:
20
-28
--32
---null
---null
--29
---null
---null
-26
--37
---null
---null
--76
---null
---null
------------------------------------------------
```

"-" signs are inserted according to depth of the node.

```
Adding 8 to second tester heap.
Removing one element from second tester heap.
After these 2 operations second tester heap:
null


Adding 8, 30, 74, 89, 66, and 39 to second tester heap.
Second heap after insertions:
8
-30
--89
---null
---null
--66
---null
---null
-39
--74
---null
---null
--null


Incrementing key value of element 8 to 80 at the second heap.
Second heap after key incrementation:
30
-66
--89
---null
---null
--80
---null
---null
-39
--74
---null
---null
--null
------------------------------------------------
```

```
Merging first and second heaps.
Result:
20
-26
--29
---30
----null
----null
---66
----null
----null
--28
---89
----null
----null
---80
----null
----null
-32
--37
---39
----null
----null
---74
----null
----null
--76
---null
---null
-----------------------------------------------

Adding Palahniuk, Tarantino, Camus, Hetfield, Angie, and Murakami to last tester heap.
Last heap after insertions:
Angie
-Camus
--Tarantino
---null
---null
--Hetfield
---null
---null
-Murakami
--Palahniuk
---null
---null
--null
```

```
After 3 removals:
Murakami
-Palahniuk
--null
--null
-Tarantino
--null
--null
```

BINARY SEARCH TREE:

```
TESTING BINARY SEARCH TREE
--------------------------
Adding 25, 20, 36, 10, 22, 30, 40, 5, 12, 28, 38, and 48 to tree.

Array representation of the BST:
25 - 20 - 36 - 10 - 22 - 30 - 40 - 5 - 12 - null - null - 28 - null - 38 - 48

Tree representation of the BST:
25
-20
--10
---5
----null
----null
---12
----null
----null
--22
---null
---null
-36
--30
---28
----null
----null
---null
--40
---38
----null
----null
---48
----null
----null

Deleting 25 from the tree.

Array representation of the BST:
28 - 20 - 36 - 10 - 22 - 30 - 40 - 5 - 12 - null - null - null - null - 38 - 48
```

```
Tree representation of the BST:
28
-20
--10
---5
----null
----null
---12
----null
----null
--22
---null
---null
-36
--30
---null
---null
--40
---38
----null
----null
---48
----null
----null

Deleting 48 from the tree.

Array representation of the BST:
28 - 20 - 36 - 10 - 22 - 30 - 40 - 5 - 12 - null - null - null - null - 38
```

```
Tree representation of the BST:
28
-20
--10
---5
----null
----null
---12
----null
----null
--22
---null
---null
-36
--30
---null
---null
--40
---38
----null
----null
---null

Tree doesn't contain 25.
Tree contains 38.
------------------------------------------------------------------
```

```
Adding lay, house, rat, lay, jack, that, milked, and cow to tree.

Array representation of the BST:
lay - house - rat - cow - jack - milked - that

Tree representation of the BST:
lay
-house
--cow
---null
---null
--jack
---null
---null
-rat
--milked
---null
---null
--that
---null
---null

Deleting rat from the tree.

Array representation of the BST:
lay - house - that - cow - jack - milked

Tree representation of the BST:
lay
-house
--cow
---null
---null
--jack
---null
---null
-that
--milked
---null
---null
--null
```

```
Deleting house from the tree.

Array representation of the BST:
lay - jack - that - cow - null - milked

Tree representation of the BST:
lay
-jack
--cow
---null
---null
--null
-that
--milked
---null
---null
--null

Tree contains milked.
Tree doesn't contain mert.
```