Binary search deals with splitting an array in two each cycle and throwing half of this array away. So in the worst case scenario (the maximum number of comparisons) this array will be reduced down to only one element.

For example:

A = [1, 2, 3, ... 8]

The length of the array is 8.

8 → 4 → 2 → 1

This shows that 3 iterations are needed for 8 to reach 1. Therefore, 3 comparisons are needed.

If we keep multiplying 8 by 1/2, we can get the expression:

$8 \times 1/2^n = 1$ where n is the number of comparisons.

Solving for n and generalizing this expression gives us:

No. of elements: $1/2^n = 1$

No. of elements = 2^1 →

$\log_2$(No. of ~~comparisons~~ elements)

therefore the number of comparisons is given by

$\log_2$(Number of elements)

for the worst case

proof: $A(N) = X + A(N/2)$

$A(N/2) = X + A(N/4)$

$A(N) = A(\frac{N}{4}) + 2X$

$A(\frac{N}{4}) = A + A(N/8)$

$A(N) = A(\frac{N}{8}) + 3A$

$A(N) = A(\frac{N}{2^i}) + iX$

$A(\frac{N}{2^i}) = T(1)$

$$\frac{N}{2^i} = 1$$

$$N = 2^i \rightarrow \log_2 N = \log_2 2^i$$

$$\boxed{i = \log_2 N}$$

3. No, there is no restriction the only restriction will be leaf are either has 0 or 2 number of children because it is a binary tree.

2. Full Binary Theorem:

If the number of nodes are N, the number of leaves are L and the number of internal nodes I are related in such a way that, if you know any one of them, you can determine other two. Also theorem states that if the number of internal nodes are I, then number of leaves will be

$$L = I + 1$$

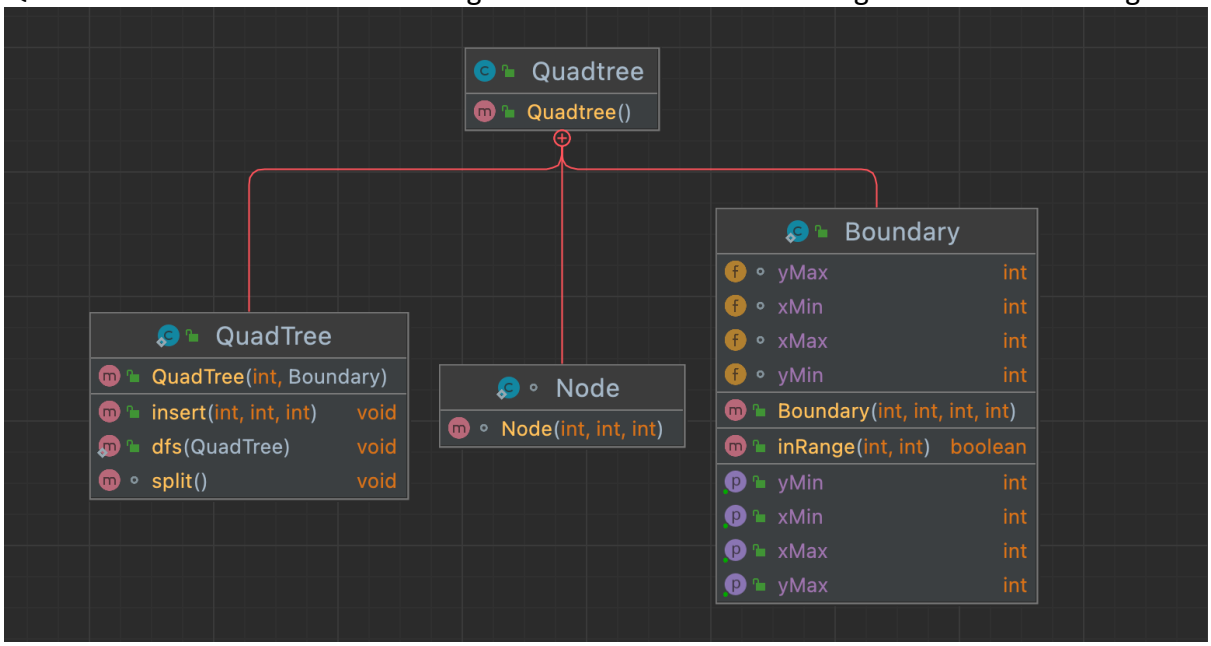
L → Leave, I → Internal node

$N = 2I + 1$

number of node

internal node

2. Quadtree implementation:
I use 4 data field for quadtrees.
Quadtree uses Node class for storing elements and also use dfs algorithm for traversing.

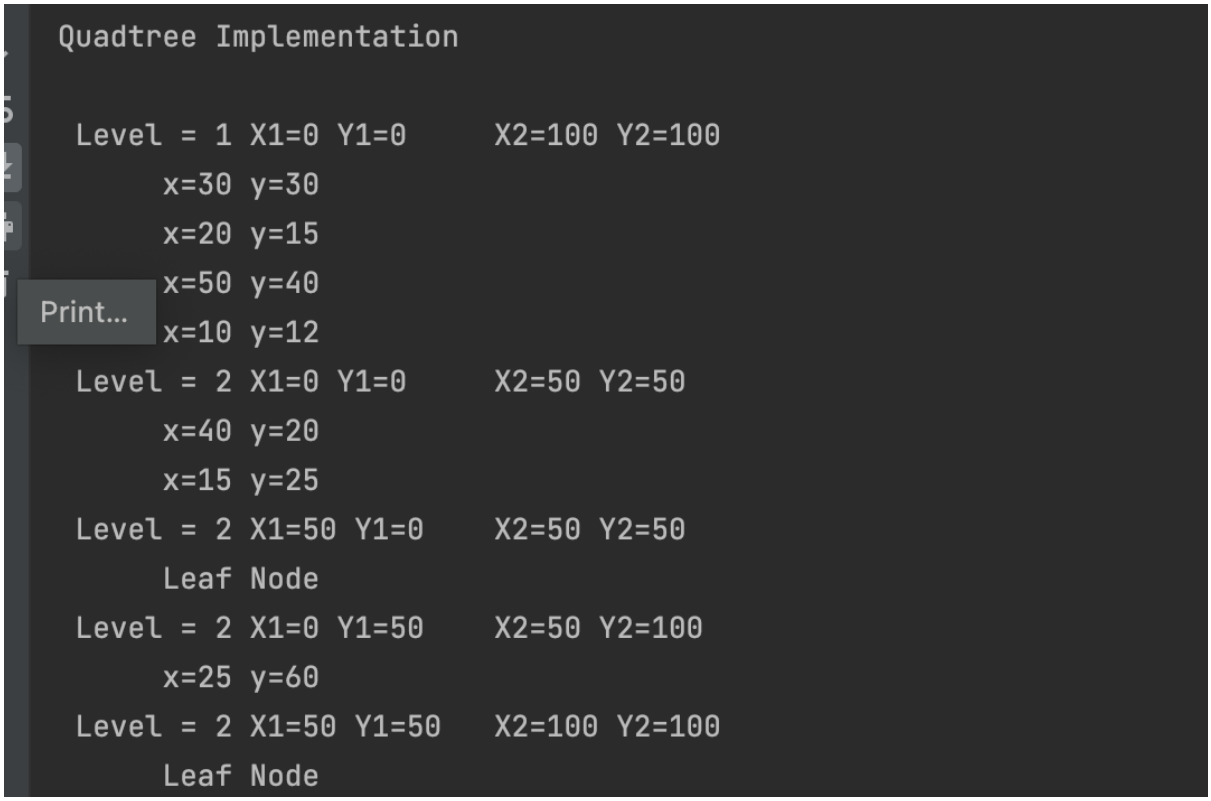



```
Quadtree.QuadTree anySpace = new Quadtree.QuadTree( level: 1, new Boundary( xMin: 0,  yMin: 0,  xMax: 100,  yMax: 100));
anySpace.insert( x: 30,  y: 30,  value: 1);
Quadtree.QuadTree.dfs(anySpace);
anySpace.insert( x: 20,  y: 15,  value: 1);
Quadtree.QuadTree.dfs(anySpace);
anySpace.insert( x: 50,  y: 40,  value: 1);
Quadtree.QuadTree.dfs(anySpace);
anySpace.insert( x: 10,  y: 12,  value: 1);
Quadtree.QuadTree.dfs(anySpace);
anySpace.insert( x: 40,  y: 20,  value: 1);
Quadtree.QuadTree.dfs(anySpace);
anySpace.insert( x: 25,  y: 60,  value: 1);
Quadtree.QuadTree.dfs(anySpace);
anySpace.insert( x: 15,  y: 25,  value: 1);
System.out.println();

Quadtree.QuadTree.dfs(anySpace);
```

Result:

```
Level = 1 X1=0 Y1=0     X2=100 Y2=100
     x=30 y=30
Level = 1 X1=0 Y1=0     X2=100 Y2=100
     x=30 y=30
     x=20 y=15
Level = 1 X1=0 Y1=0     X2=100 Y2=100
     x=30 y=30
     x=20 y=15
     x=50 y=40
Level = 1 X1=0 Y1=0     X2=100 Y2=100
     x=30 y=30
     x=20 y=15
     x=50 y=40
     x=10 y=12
Level = 1 X1=0 Y1=0     X2=100 Y2=100
     x=30 y=30
     x=20 y=15
     x=50 y=40
     x=10 y=12
Level = 2 X1=0 Y1=0     X2=50 Y2=50
     x=40 y=20
Level = 2 X1=50 Y1=0    X2=50 Y2=50
     Leaf Node
Level = 2 X1=0 Y1=50    X2=50 Y2=100
     Leaf Node
Level = 2 X1=50 Y1=50   X2=100 Y2=100
```

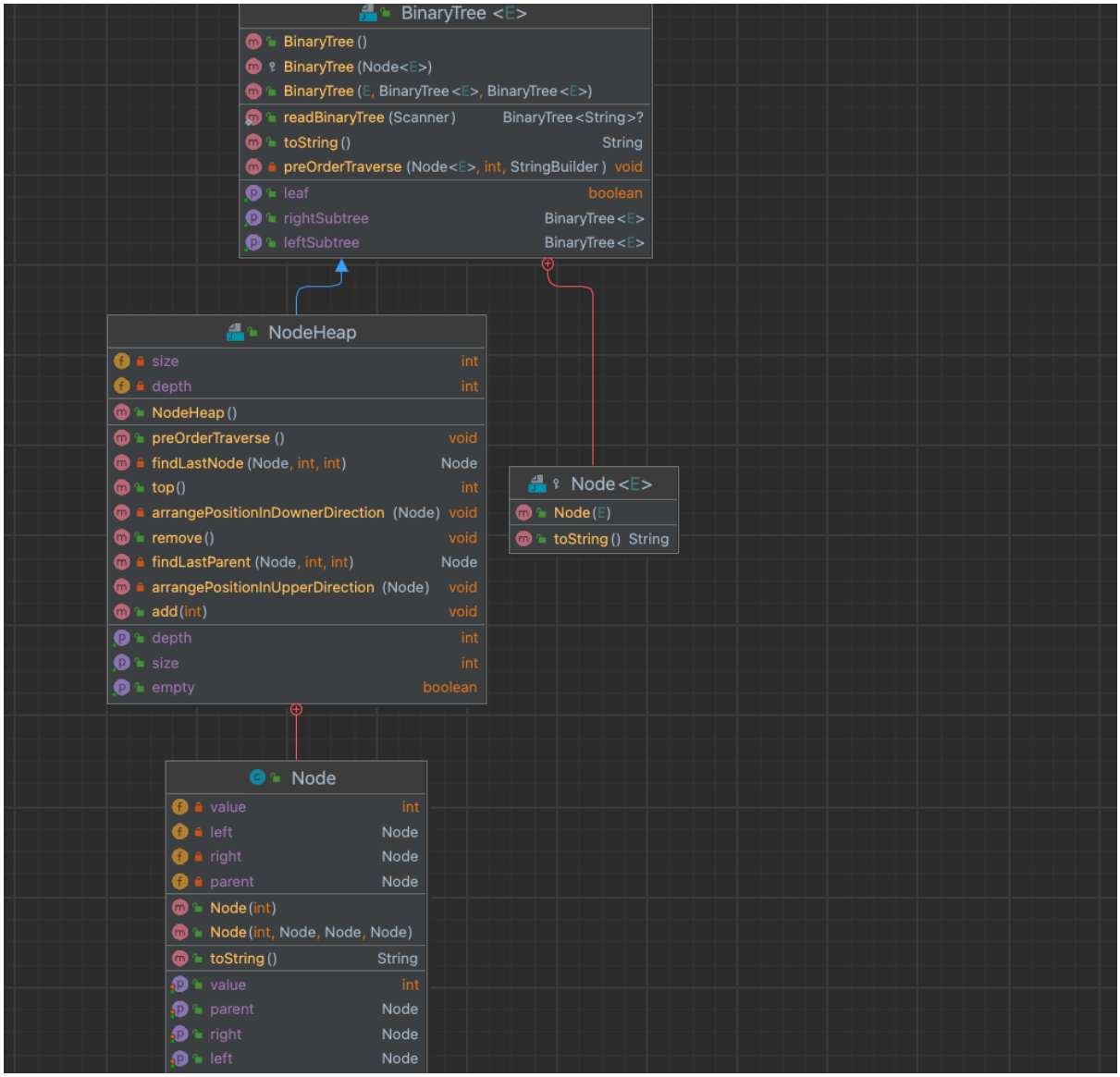
```
 Level = 2 X1=50 Y1=50   X2=100 Y2=100
      Leaf Node
 Level = 1 X1=0 Y1=0     X2=100 Y2=100
      x=30 y=30
      x=20 y=15
      x=50 y=40
      x=10 y=12
 Level = 2 X1=0 Y1=0     X2=50 Y2=50
      x=40 y=20
 Level = 2 X1=50 Y1=0    X2=50 Y2=50
      Leaf Node
 Level = 2 X1=0 Y1=50    X2=50 Y2=100
      x=25 y=60
 Level = 2 X1=50 Y1=50   X2=100 Y2=100
      Leaf Node
Quadtree Implementation

 Level = 1 X1=0 Y1=0     X2=100 Y2=100
      x=30 y=30
      x=20 y=15
      x=50 y=40
      x=10 y=12
 Level = 2 X1=0 Y1=0     X2=50 Y2=50
      x=40 y=20
      x=15 y=25
 Level = 2 X1=50 Y1=0    X2=50 Y2=50
```

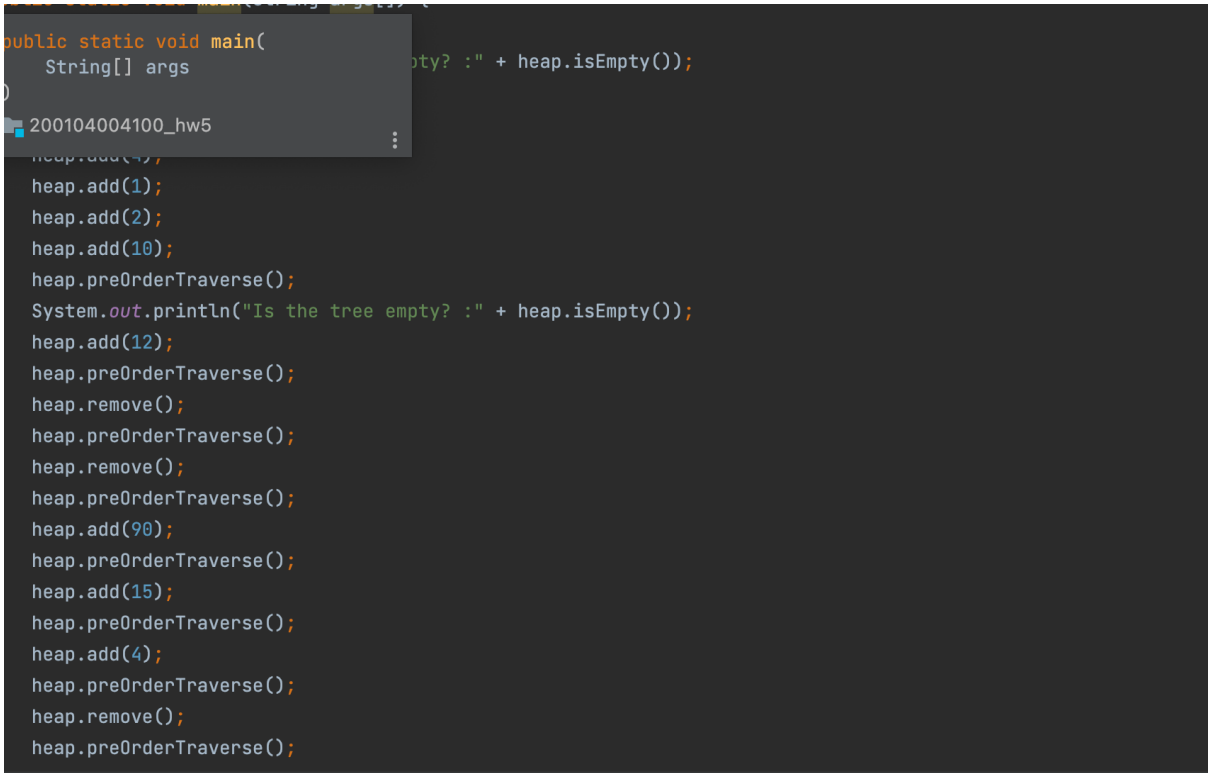


3.BH IMPLEMENTATION

Implement a binary heap (extend the BinaryTree class in the text book) class using nodelink structure. Your ternary heap should satisfy the structural property of being a complete tree besides the heap order property. I add some more data fields in the Node class such as parent Node.

NodeHeap class extends the Node class for implementing some additional data in BH implementation.

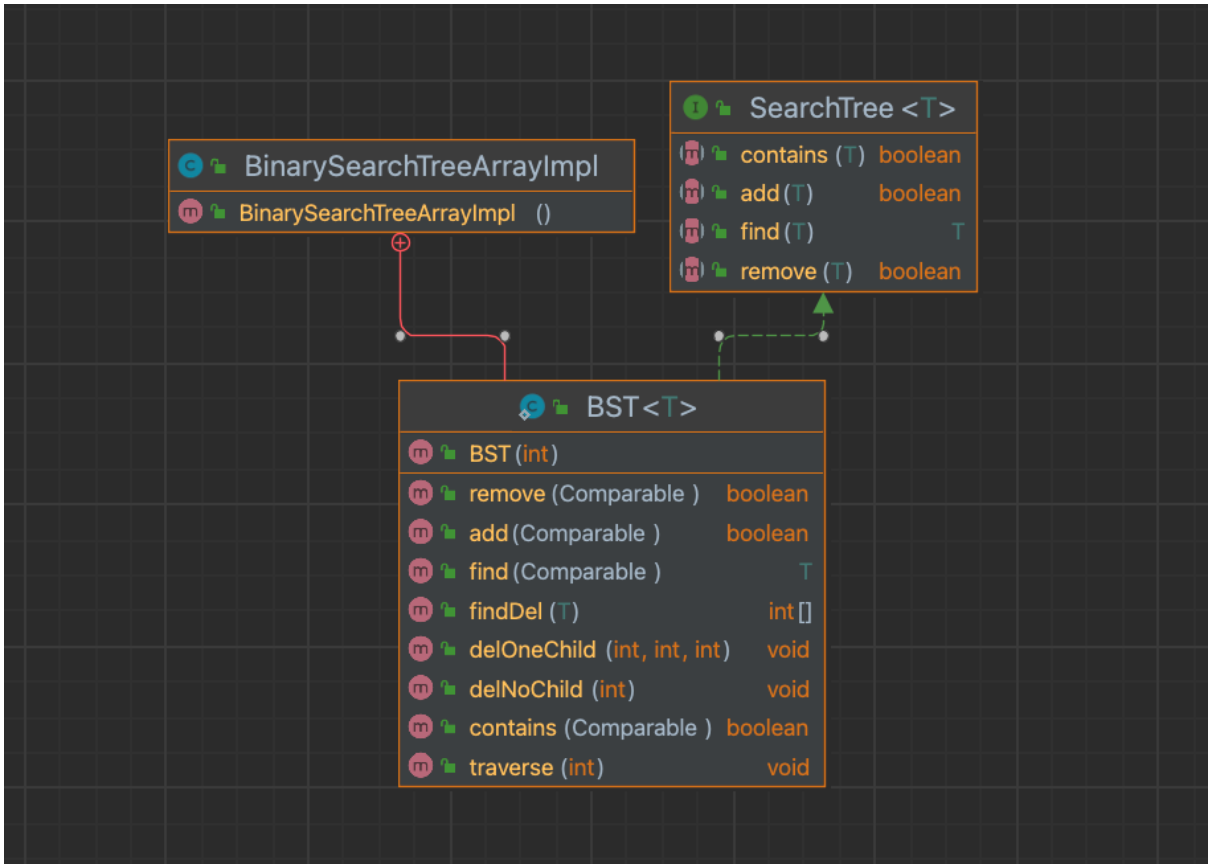

BinaryTree <E>
BinaryTree ()
BinaryTree (Node<E>)
BinaryTree (E, BinaryTree <E>, BinaryTree <E>)
readBinaryTree (Scanner) BinaryTree <String>?
toString () String
preOrderTraverse (Node<E>, int, StringBuilder) void
leaf boolean
rightSubtree BinaryTree <E>
leftSubtree BinaryTree <E>
NodeHeap
size int
depth int
NodeHeap ()
preOrderTraverse () void
findLastNode (Node, int, int) Node
top() int
arrangePositionInDownerDirection (Node) void
remove() void
findLastParent (Node, int, int) Node
arrangePositionInUpperDirection (Node) void
add(int) void
depth int
size int
empty boolean
Node <E>
Node(E)
toString() String
Node
value int
left Node
right Node
parent Node
Node(int)
Node(int, Node, Node, Node)
toString() String
value int
parent Node
right Node
left Node


public static void main(
String[] args
)
200104004100_hw5
pty? :" + heap.isEmpty());
heap.add(1);
heap.add(2);
heap.add(10);
heap.preOrderTraverse();
System.out.println("Is the tree empty? :" + heap.isEmpty());
heap.add(12);
heap.preOrderTraverse();
heap.remove();
heap.preOrderTraverse();
heap.remove();
heap.preOrderTraverse();
heap.add(90);
heap.preOrderTraverse();
heap.add(15);
heap.preOrderTraverse();
heap.add(4);
heap.preOrderTraverse();
heap.remove();
heap.preOrderTraverse();

```
Is the tree empty? :true

Add operation takes  1 swaps.

Add operation takes  0 swaps.

Add operation takes  0 swaps.

Add operation takes  0 swaps.

Add operation takes  2 swaps.
Heap with size 6 and depth 2
10 5 9 1 2 4
Is the tree empty? :false

Add operation takes  2 swaps.
Heap with size 7 and depth 2
12 5 10 1 2 4 9

Remove operation takes 2 swaps.
Heap with size 6 and depth 2
10 5 9 1 2 4

Remove operation takes 1 swaps.
Heap with size 5 and depth 2
9 5 4 1 2
```

```
Remove operation takes 1 swaps.
Heap with size 5 and depth 2
9 5 4 1 2

Add operation takes  2 swaps.
Heap with size 6 and depth 2
90 5 9 1 2 4

Add operation takes  1 swaps.
Heap with size 7 and depth 2
90 5 15 1 2 4 9

Add operation takes  1 swaps.
Heap with size 8 and depth 3
90 5 15 4 2 4 9 1

Remove operation takes 2 swaps.
Heap with size 7 and depth 2
15 5 9 4 2 4 1
```

4.
UML DIAGRAM:



In this part, I used Search Tree interface to implement BST.

```
BinarySearchTreeArrayImpl.BST bst= new BinarySearchTreeArrayImpl.BST<Integer>( size: 10);
bst.add(14);
bst.add(16);
bst.add(19);
bst.add(2);
System.out.println("Current node:");
bst.traverse( i: 0);
System.out.println();
System.out.println("Finding 2:");
System.out.println(bst.find( d: 2));
bst.remove( d: 2);
System.out.println("After Delete: ");
bst.traverse( i: 0);
System.out.println("Is three contains 19:");
System.out.println( bst.contains(19));
bst.traverse( i: 0);
```

I am adding 14,16 ,19 and 2 consequently and I am calling the traverse function. Result will be:

```
Leaf Node
Current node:
2 14 16 19
Finding 2:
2

Deleting 2
After Delete:
14 16 19 Is three contains 19:
true
14 16 19
```