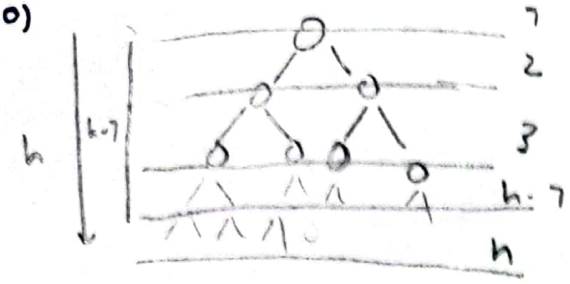
CSE 222 Homework #5
Berkan Atak
[redacted]

7)

a)



level	nodes
1	$1 (2^0)$
2	$2 (2^1)$
3	$4 (2^2)$
h-1	2^{h-2}
h	2^{h-1}

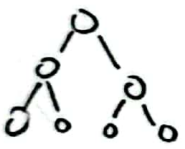
X is last line node.
we don't know how many node last line and we know tree is complete tree. we can say $1 \le X \le h \cdot 2^{h-1}$

Total depth $= 1.(2^0) + 2.2^1 + 3.2^2 + \dots + (h-1).2^{(h-2)} + X$

We can transform this expression to formula

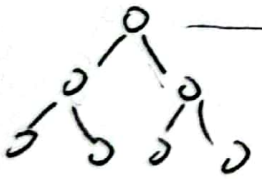
$$\sum_{n=1}^{h-1} n \cdot 2^{n-1} + X = \text{Total depth}$$

b)

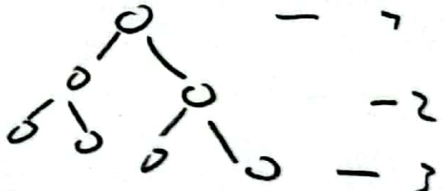


$$\text{average} = \frac{\text{Total comparison}}{\text{Node count}}$$

I assume that 1 node comparison count



Total comparison
1. 1
2. 2
3. 4 → node count



$$\text{average} = \frac{\text{Total comparison}}{\text{Node count}} = \frac{1.(2^0) + 2.2^1 + 3.2^2 + \dots + n.2^{n-1}}{2^0 + 2^1 + 2^2 + \dots + 2^{n-1}}$$

$$\frac{\sum_{n=1}^{depth} n \cdot 2^{n-1}}{\sum_{n=1}^{depth} 2^{n-1}} = \text{average}$$

or $\log n$

4) There are no restrictions. Only one node must have 2 children or zero children

If T has a total of N nodes, the number of leaves is $L = (N+1)/2$

If T has a total of N nodes, the number of internal nodes $I = (N-1)/2$

2) (50,50)

NE	SE	SW	NW

(20,75)

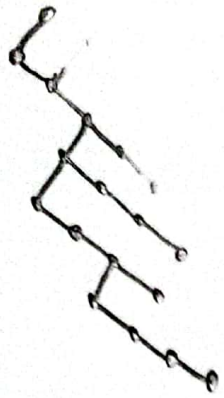
Direction
SW

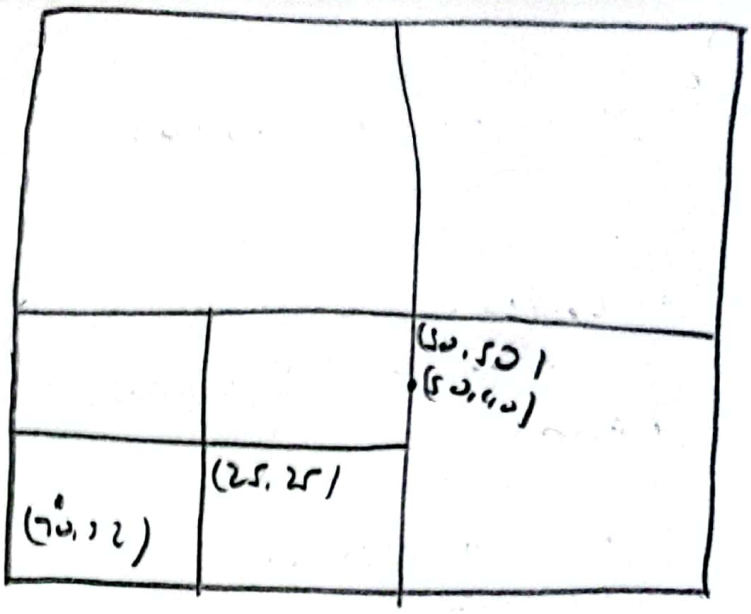
(90,10)

Direction
NE

(70, 72)

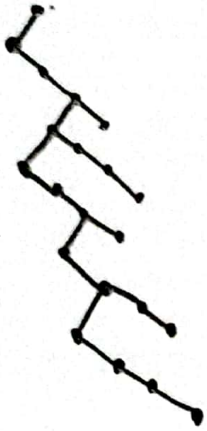
Direction
SW

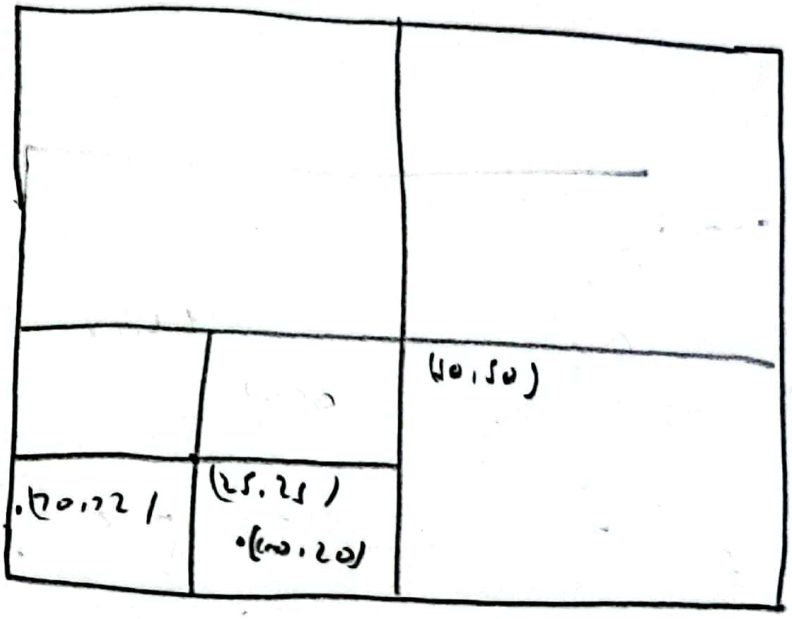




(40, 20)

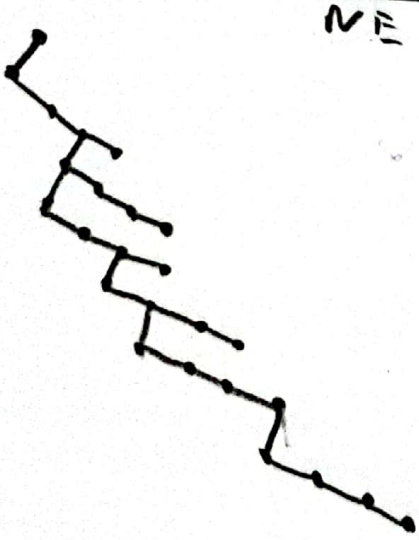
Direction
SE

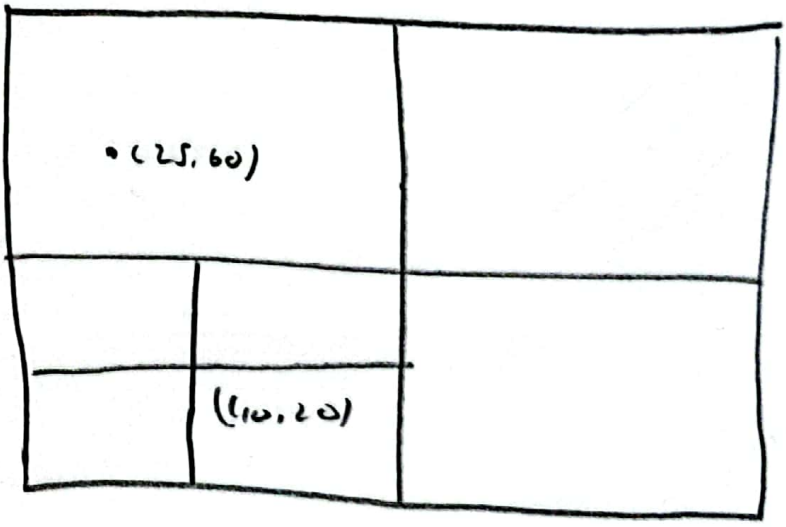




(25, 60)

Direction
NE

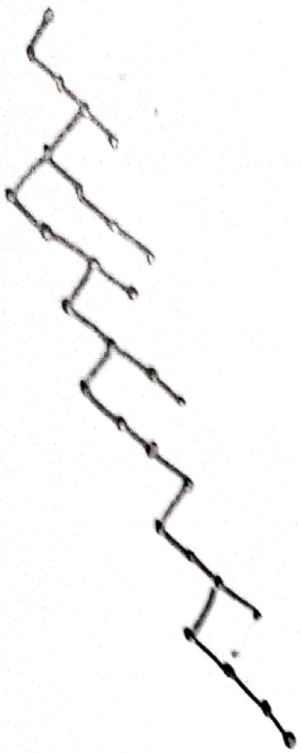


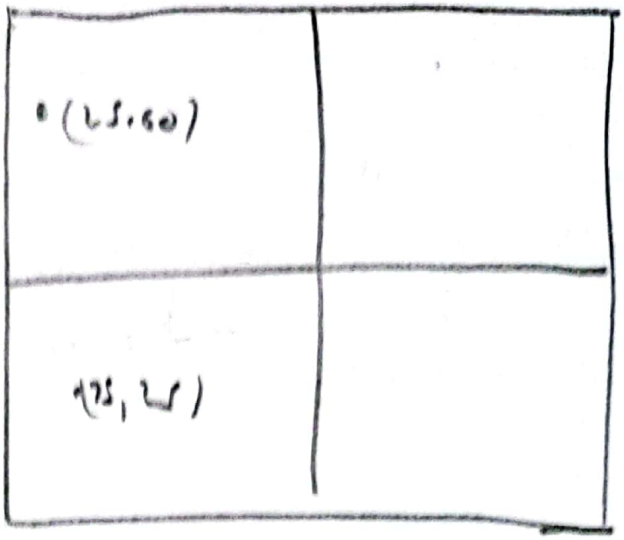


(75, 25)

Direction

SW





Time Complexity Analysis

#93 BinaryTreeNode.java

```
Public String toString(BinaryNode<E> node, int depth, StringBuilder sb) {
    For(int i=0; i<depth; ++i){
        sb.append(" ");
    }                                                    } O(n)

    if(node==null) {
        sb.append("null\n");                             } 1
    } else {
      sb.append(node.toString());
      sb.append("\n");
      toString(node.left, depth+1, sb);                  } T(n-1)
      toString(node.right, depth+1, sb);
    }
}
```

Best case

O(n)

Worst case

O(n)

BinaryHeap.java

```
Public void minHeap(){
    For(int pos = (size()/2); pos>=0; pos--){          n
        BinaryHeap(pos);
    }
    root = insertLevelOrder(Heap, root, 0);            n.logn
}                                                      logn
```

Best case

$\Omega(n \log n)$

Worst case

$\Omega(n \log n)$

```
private void BinaryHeap(int pos) {
  if(!isLeaf(pos) {
    if(Heap[pos] > HeapLeftChild(pos)]
       || Heap[pos] > Heap[RightChild(pos)]){
       if(Heap[LeftChild(pos)] < Heap[RightChild(pos)]){
       Swap(pos, LeftChild(pos));
       BinaryHeap(LeftChild(pos));
       }
       else{
       Swap(pos, RightChild(pos));
       BinaryHeap(RightChild(pos));
       }
    }
}
```

} T(n/2)

} T(n/2)

Best case

O(1)

Worst case

O(log n)

```
Public void Insert(int element) {
    if(size >= maxsize){
        return;
    }
    Heap[size] = element;
    int current = size;
    size++;
    while(Heap[current] < Heap[Parent(current)]){
        Swap(current, Parent(current));
        current = Parent(current);
    }
}
```

} O(1)

} O(n)

Worst case

O(1)

Avg case

O(n)

```
Public int removeM()
    int popped = Heap[front];
    size--;
    Heap[front] = Heap[size];
    BinaryHeapify(front);
    return popped;
}
```

Best case
O(log n)

Worst case
O(log n)

#96
ArrayTree.java

```
Public void add(E data){
    if (array[0] == null){
        array[0] = new Node(data);
        return;
    }
    recAdd(0, data);
}

private void recAdd(int root, E data){
    if(array[root] == null){
        return;
    }
    int res = data.compareTo((E) array[root].getData());
    if(res<0){
        if(array[root*2+1] == null){
            array[root*2+1] = new Node(data);
            return;
        }
        else{
            recAdd(root*2+1, data);
        }
    }
    else{
        if(array[root*2+2] == null){
            array[root*2+2] = new Node(data);
            return;
        }
        else{ recAdd(root*2+2, data);
```

Best case
O(1)

Worst case
O(log n)

```
Public boolean contains(E data) {
    return contains(0, data)
}

Private boolean contains(Integer root, E data){
  if(array[root] == null){
     return false;
  }
  int res = data.compareTo((E) array[root]);
  if(res == 0){
     return true;
  }
  if(res < 0){
     return contains(root * 2 + 1, data);
  else {
    return contains(root * 2 + 2, data);
  }
}
```

Best case: $O(1)$ — Worst case: $O(\log n)$

Best case: $O(1)$ — Worst case: $O(\log n)$

Best case: $O(1)$ — Worst case: $O(\log n)$

$O(1)$ $O(\log n)$ $O(\log n)$

Best case: $O(1)$ — Worst case: $O(\log n)$