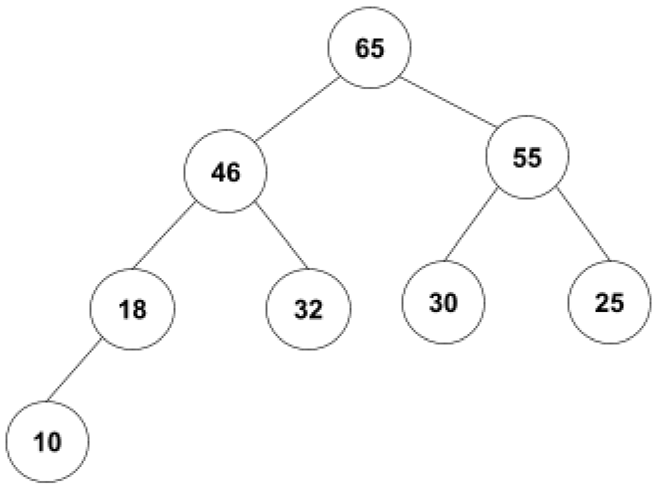
Problem Set 5

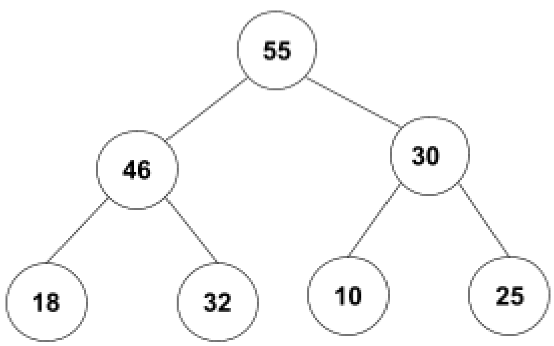
Problem 1: Heaps and heapsort

1-1)



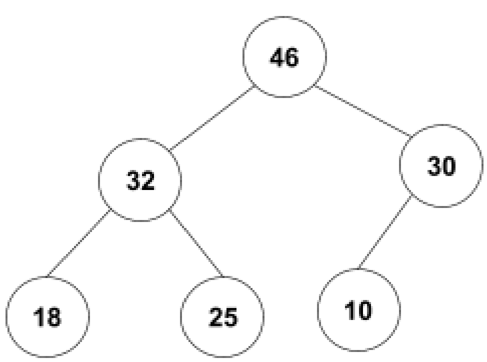
1-2) `{65, 46, 55, 18, 32, 30, 25, 10}`

1-3a) heap:



array:
`{55, 46, 30, 18, 32, 10, 25, 65}`

1-3b) heap:



array:
`{46, 32, 30, 18, 25, 10, 55, 65}`

Problem 2: Informed state-space search

2-1)

```
state a: 12
state b: 13
state d: 12
state h: 13
```

2-2)

```
state a: 12
state b: 13
state d: 14
state h: 16
```

Problem 3: Hash tables

3-1) linear

0	if
1	to
2	my
3	the
4	an
5	by
6	do
7	we

3-2) quadratic

0	
1	
2	my
3	the
4	
5	
6	an
7	

3-3) double hashing

0	
1	
2	my
3	the
4	do
5	an
6	by
7	we

3-4) probe sequence: 3 -> 5 -> 7 -> 1

3-5) table after the insertion:

0	function
1	
2	
3	our
4	
5	table
6	
7	see

It will be not colored

Problem 4: Determining if an array is a heap
4-1)

```
private static boolean isHeapTree(int[] arr, int i) {

    Queue<Integer> queue = new LLQueue<>();
    queue.insert(i);

    /**
     * The algorithm is similar to a level order traversal
     */
    while(!queue.isEmpty()){

        Integer currentIndex = queue.remove();

        int leftChildIndex = (2 * currentIndex) + 1, rightChildIndex = (2 *
currentIndex) + 2;

        /**
         * Break out of the loop if either of the children are bigger than the
parent
         */
        if((leftChildIndex < arr.length && arr[leftChildIndex] >
arr[currentIndex])
                || (rightChildIndex < arr.length && arr[rightChildIndex] >
arr[currentIndex])){
            return false;
        }

        /**
         * Insert the children indexes if they don't overflow
         */
        if(leftChildIndex < arr.length){
            queue.insert(leftChildIndex);
        }

        if(rightChildIndex < arr.length){
            queue.insert(rightChildIndex);
        }
    }

    /**
     * If we didn't break early, we return true
     */
    return true;
}
```

4-2)

I am assuming that we are trying to determine the efficiency of the entire tree with n elements.

We are traversing through the entire elements once in the worst case where we don't exit early and when the tree is in fact a max heap. The remove and insert operations of the queue take constant time in each iteration. There will also be a constant number of comparisons in each iteration. It is O(n) in the worst case time complexity pertaining to traversing the entire tree

The best case is if the root of the tree has a smaller value than one of its children. It is O(1) time complexity when we exit early

Problem 5: Comparing data structures

A HashTable would work well in this case for the reasons outlined below

- A BST is less efficient than HashTable for searching and it is O(log n) for a BST Vs O(1) for a HashTable lookup. A HashTable provides an array like random access which will make retrieval as efficient as possible and accessing a product by its name is a constant time operation

- By picking a strategy like double hashing while keeping the size of the hash array a prime, we can avoid overflow. The disadvantages of a HashTable here is that all of the elements need to be copied and rehashed to resize. Rehashing is at the order of O(n).

- Using separate chaining, we can create an index for the first n characters of a record and records in that category can be hashed to the same bucket. Retrieving them all is then a constant time operation

- Performing a range search is easier with a BST but that can be achieved by indexing as outlined above

- Insertion and Deletion are constant time operations in a HashTable as long as insertion doesn't warrant resizing

With a good choice of a hash function and array size, the benefits of a HashTable far outweigh the cost in this case

Problem 6: A non-recursive DFS

```
/*
* dfTrav - an iterative version
*/
private static void dfTravIter(Vertex v) {

   /**
    * Start out by pushing the root into stack
    */
   Stack<Vertex> vertices = new LLStack<>();
   vertices.push(v);
```

```java
    System.out.println(v.id);
    v.done = true;
    v.parent = null;

    while(!vertices.isEmpty()){

        Vertex toBeVisited = vertices.pop();

        /**
         * Visit the vertex if it is not visited yet
         */
        if(!toBeVisited.done){
            System.out.println(toBeVisited.id);
            toBeVisited.done = true;
        }

        Edge edge = toBeVisited.edges;

        while(edge != null){

            /**
             * Assign parent reference
             */
            Vertex adjacent = edge.end;
            adjacent.parent = toBeVisited;

            /**
             * Add the adjacent vertices if they are not visited yet
             */
            if(!adjacent.done){
                vertices.push(adjacent);
            }

            /**
             * Advance the LLList of edges
             */
            edge = edge.next;
        }

    }
}
```

Problem 7: Graph traversals

7-1) depth-first traversal:

Denver, Seattle, O’Hare, Atlanta, Washington, New York, Boston, L.A., San Jose

7-2) path from Denver to Boston in depth-first spanning tree:

Denver -> O’Hare -> Atlanta -> Washington -> New York -> Boston

7-3) breadth-first traversal:

Denver, Seattle, O’Hare, San Jose, Atlanta, Washington, New York, Boston, L.A.

7-4) path from Denver to Boston in breadth-first spanning tree:

Denver -> O’Hare -> Boston

Problem 8: Minimal spanning tree

(Denver, Seattle) - 900

(Denver, O’Hare) - 1100

(O’Hare, Atlanta) - 700

(Atlanta, Washington) - 600

(Washington, New York) - 250

(New York, Boston) - 200

(Denver, San Jose) - 1200

(San Jose, L.A.) - 400

Problem 9: Dijkstra's shortest-path algorithm
9-1)

Atlanta	inf	inf	inf	1800	1800	1800			
Boston	inf	inf	inf	2000	2000	2000	2000	2000	2000
Denver	0								
L.A.	inf	inf	inf	3100	1600				
New York	inf	inf	inf	1900	1900	1900	1900	1900	
O'Hare	inf	1100	1100						
San Jose	inf	1200	1200	1200					
Seattle	inf	900							
Washington	inf	inf	inf	1850	1850	1850	1850		

9-2)

Denver -> O’Hare -> L.A. with weight 3100

Denver -> San Jose -> L.A. with weight 1600

Problem 10: Directed graphs and topological sort

10-1)

It is a DAG

Topological sort : d, c, a, b, e, f

10-2)

It is not a DAG.

Cycles:

c -> d -> b -> c
c -> d -> f -> c

Problem 11: Alternative MST algorithm

(Boston, New York) - 200

(New York, Washington) - 250

(San Jose, L.A.) - 400

(Washington, Atlanta) - 600

(Atlanta, O'Hare) - 700

(Seattle, Denver) - 900

(Denver, O'Hare) - 1100

(Denver, San Jose) - 1200