Süleyman Gölbol

1. (30 pts.) Analyze the worst-case efficiency of the following insertion sort implementation. Modify the body of this implementation so that the performance is O(n^2) in the worst-case.

```
public static void insertionSort (LinkedList<E> list) {
    for (int j = 1; j < list.size(); j++) {
        E current = list.get(j);
        int i = j-1;
        while ((i > -1) && ((list.get(i).compareTo(current)) == 1)) {
            list.set(i+1, list.get(i));
            i--;
        }
        list.set(i+1, current);
    }
}
```

Question 1

```
public static void insertionSort (LinkedList<E> list) {
    Iterator<E> itr = list.iterator();
    j = 0;
    while (itr.hasNext()) {
        E current = itr.next()
        ıtr.remove();
        int i = j-1;
        ListIterator<E> revItr = list.listIterator(j);
        while (revItr.hasPrev()) {
            E shift = revItr.prev();
            if (!shift.compareTo(current))>0))
                break;
        }
        revItr.next();
        revItr.insert(current);
        ++j;
    }
}
```

2. (10 pts.) Consider the method takes a node and an integer k as parameters and finds the number of nodes which are k edge away from the given node in an unweighted undirected graph. (*A node is k edge away from another node if the shortest path between the nodes includes k edges.*) Explain how this method can be implemented. Compare the performance of the method for Adjacency List and Adjacency Matrix representations of the graph.

Shortest paths from a node to all other nodes can be found by using BFS traversal. During BFS, each node's distance can be recorded in the queue besides the node and calculated by adding one to distance of its parent. Using bradth first traversal, the algorithm will count the nodes with distance k and terminates when the next node taken form the queue has the distance value equal to k+1.

Since the basis of the algorithm is BFS, the running time is O(n+m) for adjacency list and O(n^2) for adjacency matrix. So, the performance of adjacency list is better for sparse graphs and the performance is asymptotically same for dense graphs.

3. BBST (32 pts)
a) (8 pts.) Explain when the maximum level in a skip list may need to be incremented during an insertion. Give an example insertion operation that makes a level-3 skip list to be a level-4 skip list. Show the skip list before and after insertion.

When the number of elements is incremented from 7 to 8, level of the skip list is incremented from 3 to 4.

b) (4 pts.) Draw a B-tree of order 7 and height 2 with minimum number of elements. Show empty cells as well.

The root includes one value, and two children of the root includes three values each. The values of left node are smaller than the value at the root and the values at the right node are larger than the value at the root.

c) (4 pts.) Draw a B-tree of order 4 and height 2 with maximum number of elements. Insert an element smaller than minimum element into this tree. Show the tree after insertion.

The root includes three values, and four children of the root will include three values each. The values at each node are sorted. The value at a leaf are between corresponding two values of the root.

3. BBST (32 pts)
d) (8 pts.) Insert first six letters of your name one by one into initially empty Red-Black tree. If your name is ‘Cem Can Paksu’ insert C, e, m, C, a and n. Note that uppercase letters comes before lowercase letters.

e) (8 pts.) In AVL trees, after an insertion at a leaf, the balance criteria of more than one node may be violated.

- Explain where such nodes are located in the tree?

These nodes are located on the path fron the root to the insertion point.

- It is sufficient to handle such violations by fixing at only one node (by single or double rotation). All other violations disappear afterwards. Explain where the fix is performed and why it is enough.

The fix is performed at a node v closest to the insertion position and the balance is violated. The height of the subtree rooted at v is same before the insertion and after the insertion. So, for the nodes from root to v, the balance valus do not change with insertion.

Grading

1. Runtime 5

    Iter 1 + move 3 + 5 = 8

    Iter 2 + move 3 + 7 = 10

    Replace (remove+insert) 7

2. BFS + queue: 4

    distance value handling: 2

    Run time: 4

3. a) initial: 3, final: 2, no of elements: 3

    b) root: 3, leaves: 3, no of elements: 2 sort:-1

    c) root: 2, no of elements: 3, insert:3 sort: -1

    d) each mistake: -2

    e) where: 3, why one: 5

4. Graphs and Hashing (40pts.) Your task is to design and implement a graph representation using `HashSet` in java.

a) The vertices are represented as integers in the `Graph` interface in the book. Explain how the edges are represented in your design.

Edges are represented as set of edges where each edge keeps the source and destination.

HashSet<Edge>

4.

b) Write a Java class to implement the **`Graph`** interface in the book together with any inner classes required. Include only the declarations of data fields, constructors for the classes, and the following methods:

- **`Iterator<Edge> edgeIterator(int source)`** method of the **`Graph`** interface
- **`next()`** method of the **`Iterator`** class.
- **`hashCode()`** method that will be used by your **`HashSet`**. Do not forget the method that has to be implemented together with **`hashCode`**.

We need to write **`HashSetGraph`** which implements the Graph interface usign HashSet. In **`HashSetGraph`**, we should implement **`edgeIterator`** method.

This class should include an inner class **`Iter`** that implements Iterator interface to iterate the nodes adjacent to a given source node. In this class, we shlud implement **`next`** method.

We should also implement the **`Edge`** class and implement **`hashCode`** and **`equals`** methods.

```java
public class HashSetGraph extends AbstractGraph {

  Set<Edge> edges;

  public HashSetGraph(int numV, boolean directed) {
    super(numV, directed);
    edges = new HashSet(numV);
  }

  public Iterator<Edge> edgeIterator(int source) {
    return new Iter(source);
  }

  private class Iter implements Iterator<Edge> {
    private int source;
    private int index;

    public Iter(int source) {
      this.source = source;
      index = -1;
      advanceIndex();
    }
    @Override
    public boolean hasNext() {
      return index != getNumV();
    }
    @Override
    public Edge next() {
      if (index == getNumV()) {
        throw new java.util.NoSuchElementException();
      }
      Edge returnValue = new Edge(source, index);
      advanceIndex();
      return returnValue;
    }
    private void advanceIndex() {
      do {
        index++;
      } while (index!=getNumV() &&
               !edges.contains(new Edge(source, index)));
    }
  }
}
```

```
public class Edge {
  private int source;
  private int dest;

  public Edge(int source, int dest) {
        this.source = source;
        this.dest = dest;
}

  public boolean equals(Object obj) {
        if (obj instanceof Edge) {
            Edge edge = (Edge) obj;
            return (source == edge.source && dest == edge.dest);
        } else {
            return false;
        }
  }
  @Override
  public int hashCode() {
        return (source << 16) ^ dest;
  }
}
```

XOR (yada)

left shift (2^{16} [illegible])

Grading

4. 40 points

Edge representation: 5

`HashSetGraph`

Declerations: 3

Constructor: 3

`edgeIterator: 3`

`Iter`

Declerations: 3

Constructor: 6

`next: 6`

`Edge`

Declerations: 2

Constructor: 2

`hashCode` : 4

`equals: 3`