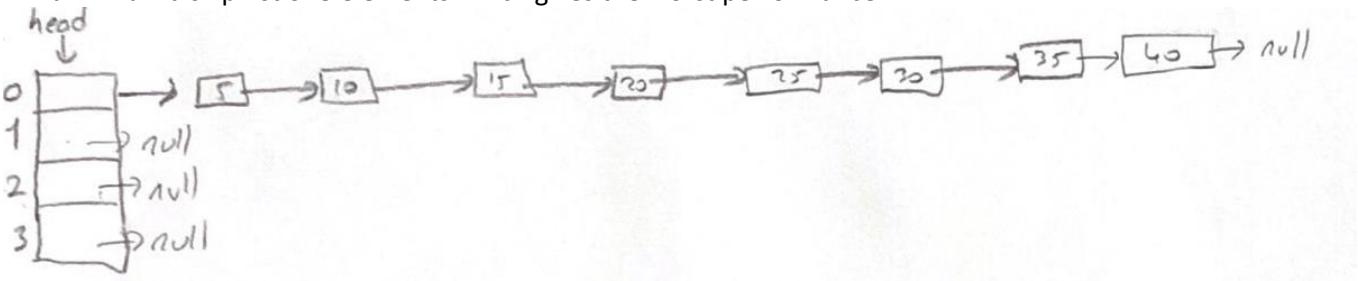
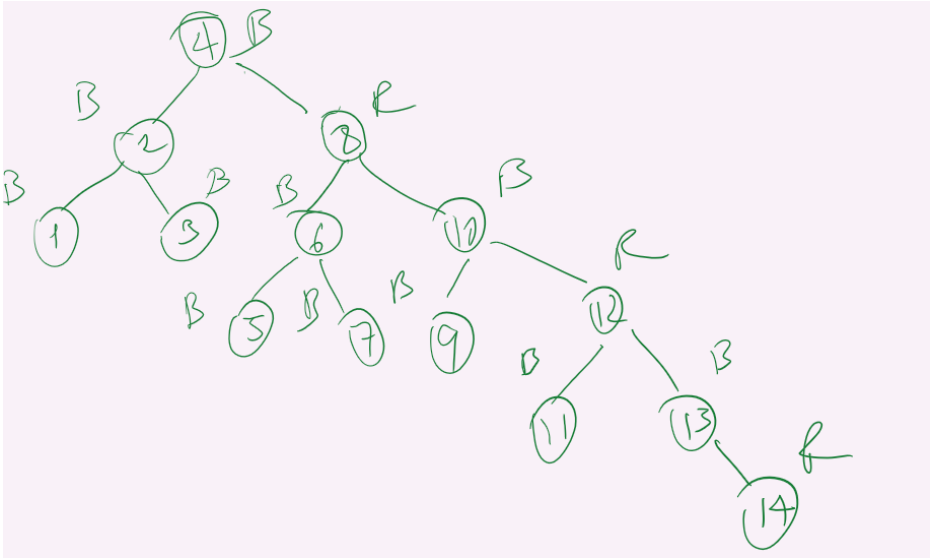
Session 1: You have 40 Minutes

1. [10 pts.] BBST

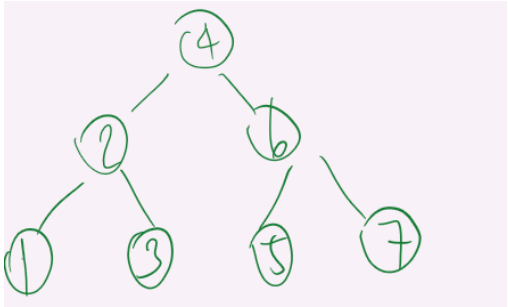
   a. Draw a skip-list of 8 elements which gives the worst performance.



   b. Draw a worst Red-Black tree of height 6 (i.e., draw a Red-Black tree with height 6 which stores the minimum number of elements). Indicate the color for each node.



   c. Draw a worst 2-3 tree of height 3 (i.e., draw a 2-3 tree with height 3 which stores the minimum number of elements).



2. [10 pts.] Give short answers.
   a. For a given domain of keys and a hash table, it is not possible to avoid collisions completely if the number of all key values is larger than the table size. Give two approaches to decrease collision probability for real-life applications.
      - Select table size to be a prime number.
      - Select hash functions that evenly distribute the keys into indices.
      - Select larger hash tables

   b. In hashing, two methods to handle collisions are open addressing and chaining. Compare the load factor value of these methods for similar run-time performance.

For chaining

$$C = 1 + \frac{L}{2}$$

$$1 + \frac{L_c}{2} = \frac{1}{2}\left(1 + \frac{1}{1 - L_o}\right)$$

$$2 + L_c = 1 + \frac{1}{1 - L_o}$$

$$1 + L_c = \frac{1}{1 - L_o}$$

For open addressing

$$C = \frac{1}{2}\left(1 + \frac{1}{1 - L}\right)$$

3. [20 pts.] Consider partial definition of the set interface below. Implement the hash set class using open addressing with linear probing. You only need to give class header, necessary data fields and implementation of the **add** method in the set interface. You do not need to consider rehashing. The method **add** should return false if the table is full. Do not use any helper function.

```
public interface Set<E> {
      public boolean add(K key);
      . . . .
}
```

```
public class HashSet<K> implements Set<K> {
      private int size;
      private E[] table;
      private final E DELETED;

      public boolean add(K key){
            if(size >= table.lenght)
                  return false;
            int index = key.hashcode() % table.lenght;
            if (index<0)
                  index += table.lenght;
            while (table[index]!= null && !DELETED.equals(table[index]
                              && !key.equals(table[index])
                  index = (index + 1)% table.lenght;
            if (key.equals(table[index])
                  return false;
            table[index]=key;
            size++;
            return true;
      }
      . . . .
}
```

Session 2: You have 30 Minutes

4. [15 pts.] Consider the following sequence: 5, 3, 6, 9, 2, 4, 1. Sort these numbers using the following algorithms. Show all intermediate steps (each swap operation) in the algorithm.
   a. Shell sort with gap sequence 5, 3, 1.

5, 3, 6, 9, 2, 4, 1
4, 3, 6, 9, 2, 5, 1
4, 1, 6, 9, 2, 5, 3 5-sorted
4, 1, 6, 3, 2, 5, 9
3, 1, 6, 4, 2, 5, 9
3, 1, 5, 4, 2, 6, 9 3-sorted
1, 3, 5, 4, 2, 6, 9
1, 3, 4, 5, 2, 6, 9
1, 3, 4, 2, 5, 6, 9
1, 3, 2, 4, 5, 6, 9
1, 2, 3, 4, 5, 6, 9 1- sorted

   b. Quick sort. Use the middle element as pivot. Use the partition algorithm discussed in class. If the number of elements is even, there are two middle elements. Use the left one.

5, 3, 6, **9**, 2, 4, 1
1, 3, 6, 5, 2, 4, **9**
1, 3, **6**, 5, 2, 4
4, 3, 1, 5, 2, **6**
4, 3, **1**, 5, 2
1, 3, 4, 5, 2
3, **4**, 5, 2
2, 3, **4**, 5
2, 3,
1, 2, 3, 4, 5, 6, 9

   c. Heapsort. Create the heap by inserting all elements from left the right.

5
5, 3
6, 3, 5
9, 6, 5, 3
9, 6, 5, 3, 2
9, 6, 5, 3, 2, 4
9, 6, 5, 3, 2, 4, 1 heap
6, 3, 5, 1, 2, 4 | 9
5, 3, 4, 1, 2 | 6, 9
4, 3, 2, 1 | 5, 6, 9
3, 1, 2 | 4, 5, 6, 9
2, 1 | 3, 4, 5, 6, 9
1, 2, 3, 4, 5, 6, 9

5. [15 pts.] Consider the following algorithm given in the textbook.

```
1.  Initialize S with the start vertex, s, and V–S with the remaining vertices.
2.  for all v in V–S
3.        Set p[v] to s.
4.        if there is an edge (s, v)
5.              Set d[v] to w(s, v).
          else
6.              Set d[v] to ∞.
7.  while V–S is not empty
8.        for all u in V–S, find the smallest d[u].
9.        Remove u from V–S and add u to S.
10.       for all v adjacent to u in V–S
11.             if d[u] + w(u, v) is less than d[v].
12.                   Set d[v] to d[u] + w(u, v).
13.                   Set p[v] to u.
```

a. Implement the algorithm in lines 2-6 so that the running time of the code is O(n) (where n is the number of vertices) if adjacency list data structure is used? What is the running time of your code if adjacency matrix is used?

```
for (int i = 0; i < graph.getNumV(); i++){
      pred[i]=start;
      dist[i]=Double.POSITIVE_INFINITY;
}
Iterator<Edge> edgeIter = graph.edgeIterator(start);
while (edgeIter.hasNext()) {
      Edge edge = edgeIter.next();
      int v = edge.getDest();
      dist[v]= edge.getWeight();
}
```

The running time is θ(n) for both adjacency matrix and adjacency list.

b. During the execution of the algorithm, how many times the condition of the if-statement in line 11 is evaluated? State the value for adjacency list and adjacency matrix and try to be as exact as possible.

m is the number of edges. Line 11 is executed twice for each edge, so it is executed 2m times. This value does not change for adjacency list and adjacency matrix.

Session 3: You have 30 Minutes

6. [10 pts.] Give short answers.
   a. Although the worst-case performance of insertion sort is not good compared to mergesort, it is preferred in some situations. Give two cases where you may prefer insertion sort over mergesort.
      - If the array is almost sorted.
      - If the number of elements is very small

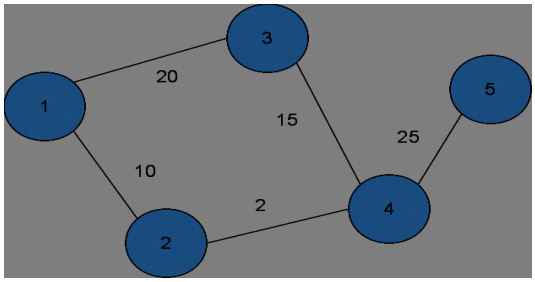
   b. The worst-case run-time performance of mergesort is asymptotically optimal (best possible performance) while the worst-case run-time of quicksort is quadratic. Give two reasons why quicksort is preferred for real-life applications, despite this fact.
      - Mergesort requires extra memory space
      - Merge operation in mergesort usually requires more comparison and assignment than partition operation in quicksort.
      - Average running time of quicksort is better than average running time of mergesort

7. [20 pts.] Write a method that takes a weighted Graph as parameter. The graph represents cities and distances between them. Your method prints the most distant two cities in the Graph and the distance between them. The most distant cities have the maximum shortest path distance among all city pairs. So, you should calculate the shortest path distances between all pair of cities and look for the largest distance. For the graph given below, the most distant cities are 3 and 5 with a distance of 40. You may use Dijsktra's method in your implementation directly. You do not need to implement the following method. Analyze the running time and space complexity of your method. Assume adjacency matrix is used in the analysis.

```
static void dijkstrasAlgorithm(Graph graph, int start,
                               int[] pred, double[] dist)
```

Graph Interface methods:

```
Iterator<Edge> edgeIterator(int source)
Edge getEdge(int source, int dest)
int getNumV()
void insert(Edge edge)
boolean isDirected()
boolean isEdge(int source, int dest)
```




```
void distantCities( Graph cities) {

        int numV = graph.getNumV();
        int[] pred = new int[numV];
        double[] dist = new double[numV];
        double largestDist = 0;
        int source = -1;
        int destination = -1;
        for (int i = 0; i < numV; i++){
             dijkstrasAlgorithm(graph, i, pred, dist);
             for (int j = 0; j < numV; j++){
                  if (dist[j] > largestDist){
                       largestDist = dist[j];
```

```
                    source = i;
                    destination = j;
                }
            }
        }
        String s = "" + largestDist + source + destination;
        System.out.println(s);
}
```