Süleyman Gölbol
1801042656
SG.

I hereby pledge on my honor that I will strictly adhere to academic integrity codes and the work done on this examination is solely my own and I will not receive / give any help from / to anybody or source during this examination.

1)

```
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>=0){
            if (list.get(i).compareTo(current)>0){
                E temp= list.get(i+1);
                list.set(i+1, list.get(i));
                list.set(i, temp);
                i--;
            }
        }
    }
}
```

In the worst case of insertion sort, the algorithm will take $O(n^2)$ time, because in inner and outer loop we increment or decrement numbers by 1. If we were divide / multiply by 2, it would contain logn in the time complexity. Also I moved comparing current and i'th index outside of loop condition and inside of loop that to even comparing won't be give a number bigger than 0, while condition always be executed if i is bigger or equals 0.

Süleyman Gölbol
1801042656

2)

To compare the shortest path we can use Djaviat's solution. In that solution if graph is dense graph we can use adjacency matrix, if it is a sparse graph we can use adjacency list for a better running time.

Adjacency matrix	AdjacencyList
$\Theta(n^2)$	$\Theta(m)$
↳ square of number of vertices.	↳ number of edges (without priority queue)

To implement it, we can use a better solution than Djaviat's solution. If we use a priority queue (it can be a heap), because of tree traversal rules, it will take to compare as $O(\log n)$ and because we are going to do for every vertex it'll take $O(n \log n)$. (for adjacency list)

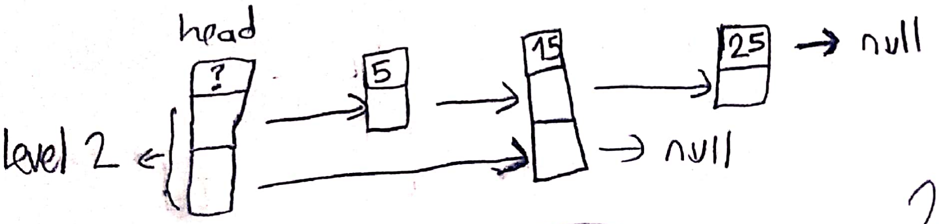
For adjacency matrix, we can use it dense graph and will take $O(n^2)$

Süleyman Gölbol
1801042656

3)

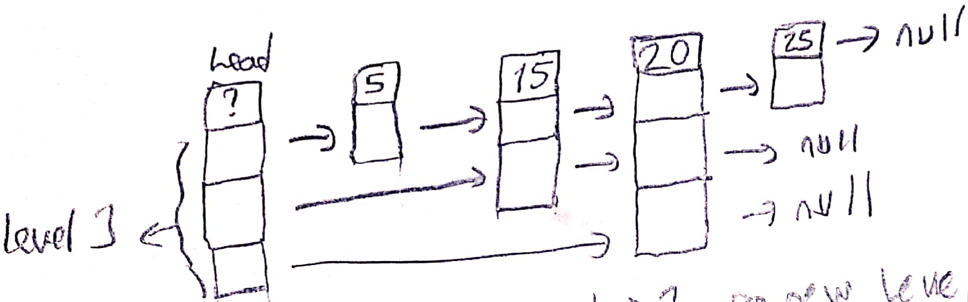
a) If we say that skip list level is $= n$, then the rule is for maximum elements is $\boxed{2^n - 1}$

The reason it become exponential Maximum size is because skip list insertions are logaritmic. For example %50 of elements will be in first level, and %25 of elements will be in second level and it goes like that by dividing by 2.



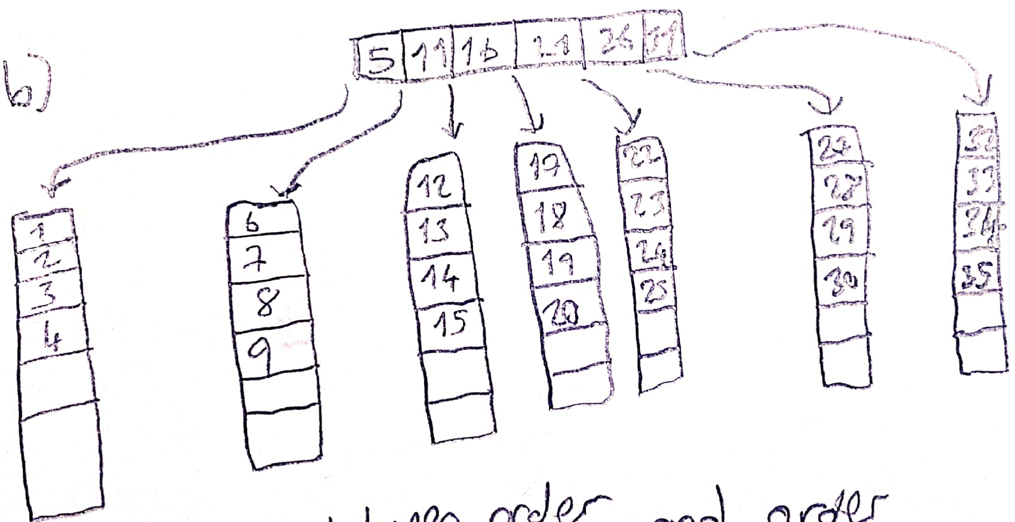
$2^2 - 1 = 3$ max elements

Adding 20



4 > 3 so new level created and new element inserted.

b)



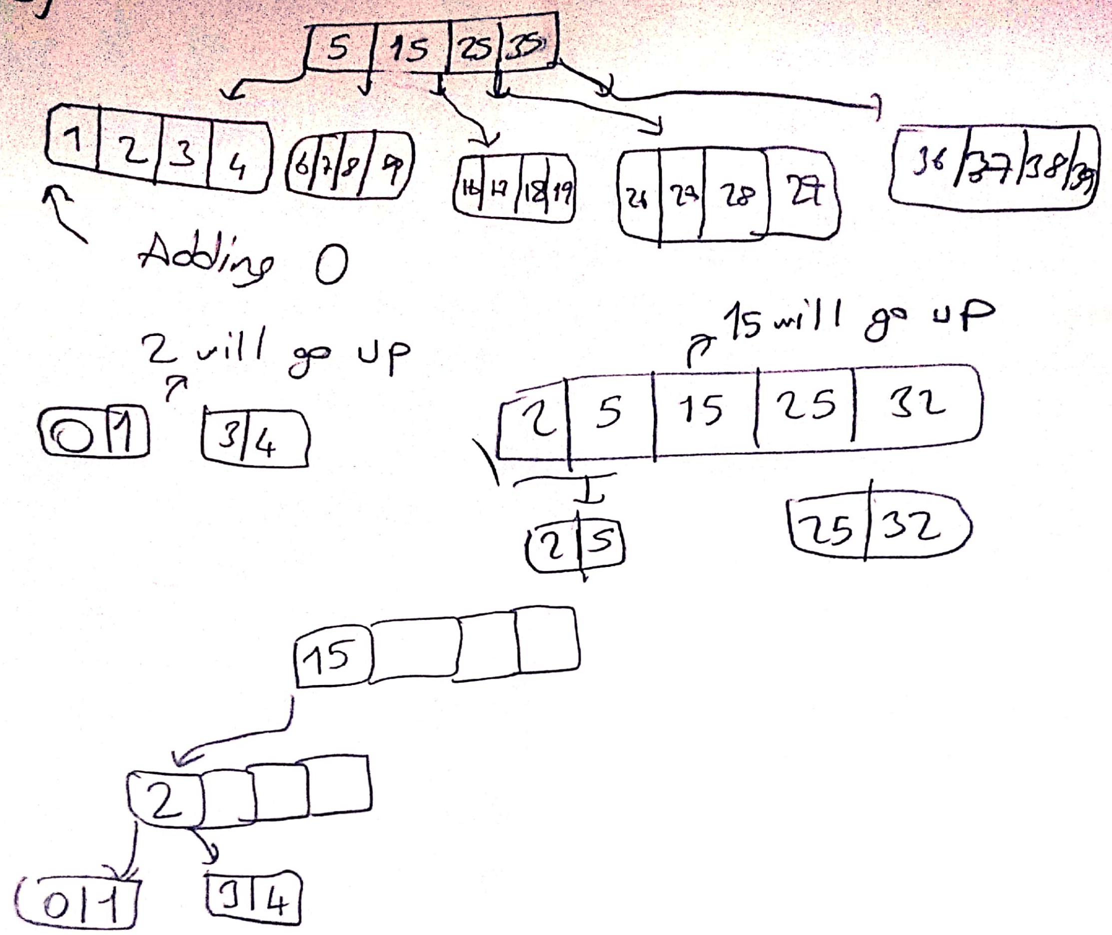
Order → number of children.
order holding data + 1

Should be between $\frac{order}{2}$ and order → 7

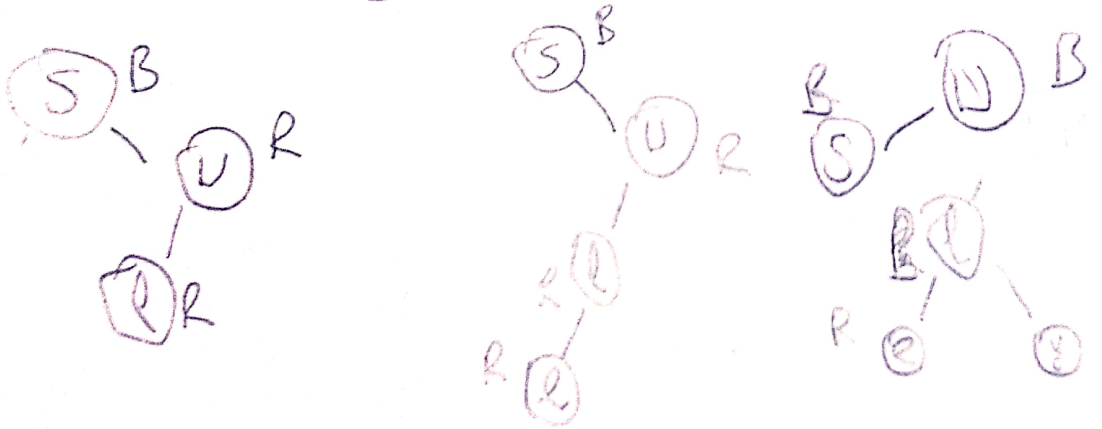
$\lceil \frac{7}{2} \rceil = 4$

Süleyman Gölb
1801042656
~~56~~.

c)



d) S u l e y m



Süleyman Gölbol
1801042656
~~56~~.

4)

In my implementation I represent vertices with integers because edgeIterator takes source parameter as integer and source is our beginning vertex.

```
public class SuleymanGraph implements Graph {

    MyHashSet <Edge> allEdges;

    public SuleymanClass (int size){
        Edge dummyVal = new Edge (-1,-1);
        allEdges = new MyHashSet<Edges>();
        for(int i=0; i<size; i++) allEdges.put(dummyVal);
    }
    Private static class MyHashSet extends HashSet <E>{
        @override
        public int hashCode (){
            return 31 * this.size() * this.size();
            // 31 is a prime number
            // so using this number with quadratic
            // probing helps us for prevent collisions.
        }
        @override
        public boolean equals (MyHashSet other){
            return hashCode() == other.hashCode();
        }
    }
    Iterator <Edge> edgeIterator (int source){

        return new Iterator<Edges> (){
            @override
            public Edge next(){
                boolean returnNextFlag = false;
                for (Edge temp: allEdges){
                    if(returnNextFlag == true)
                        return temp;
                    if (temp.getSource() == source)
                        returnNextFlag=true;
                }
            }
        }
```

	Final	Final
1	Runtime	0
1	Iter 1 + move	0
1	Iter 2 + move	0
1	Replace (remove+insert)	0
1	1	0
2	2BFS + queue	3
2	distance value handling	0
2	Run time	3
2	2	6
3	3.a	--
3	initial(3)	3
3	Final(2)	2
3	# of elements(3)	3
3	3.b	--
3	Root(3)	
3	Leaves(3)	
3	# of elements(2)	2
3	Sort (-1)	
3	3.c	--
3	Root(2)	2
3	Insert(3)	2
3	# of elements(3)	3
3	Sort(-1)	
3	3.d	--
3	True(8)	
3	Each mistake(-2)	
3	3.e	--
3	Where(3)	
3	Why one(5)	
3	**3**	17
4	Edge representation	0
4	HashSetGraph Declerations	3
4	Constructor	1
4	edgeIterator	0
4	Iter Declerations	
4	Constructor	
4	next	0
4	Edge Declerations	
4	Constructor	
4	hashCode	0
4	equals	1
4	4	5
	Format	
	F Total	28