You have 30 minutes

1. (10 pts.) Assume you have 400 elements to store in a hash table and you would like to keep the expected number of comparisons less than three for finding an item. What would be smallest table size,

a) If you use open addressing with linear probing?

b) If you use chaining?

2. (10 pts.) Explain why quadratic probing reduces collision probability and number of probes over linear probing. Give an example.

3. (20 pts.) Give the number of comparisons performed while sorting [3, 4, 7, 6, 9] using following sorting algorithms. Give details of your work.

a) Insertion Sort

b) Shell sort with gap values 7, 3, 1

c) Quicksort using first element as the pivot

d) Mergesort (the number of elements in the left half is equal to ceeling of n/2)

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

Süleyman Gölbol
180142656
CSE 222 Midterm Examination

1) Expected number of comparisons

a) $c = \frac{1}{2}\left(1 + \frac{1}{1-L}\right) < 3$

For open adressing with linear probing

$1 + \frac{1}{1-L} < 6$ $\quad \frac{1}{1-L} < 5$

$L = \frac{400}{\text{Table size}}$

$1 < 5(1-L)$

$1 < 5\left(1 - \frac{400}{\text{Table size}}\right)$

$1 < 5 - \frac{2000}{\text{Table size}}$

$\frac{2000}{\text{Table size}} < 4$

$\frac{2000}{4} < \text{Table size}$

500 $\quad$ Should be 501

b) $c = 1 + \frac{L}{2}$

$1 + \frac{\frac{400}{\text{Table size}}}{2} < 3$

$\frac{400}{\text{Table size}} < \frac{2.2}{4}$

$\frac{100}{\text{Table size}} < 1$

$100 < \text{Table size}$

Minimum 101

Süleyman Gönen
180104265b
CSE122 midterm Examination
~~50~~.

2)

When we use quadratic probing;

Formula is index = indexOfStart + probing * probing % tableLength

Because of probing grows quadraticly, this makes increase the need of bigger table size. While the table size is big, also the elements are not many (for example less than $\frac{tableSize}{2}$)

it prevents the infinite loop possibility, because when it is close to full, program can stuck in infinite loop. But in linear probing, it easily becomes table size close to full and it increase the risk for infinite loop.

3) [3, 4, 7, 6, 9]

a) For insertion sort, there are 2 nested loop (for example for i and j) and every outer loop beginning, we are going to compare array[i] with array[j] and in inner loop j will increment all the time. If we say size is 5; for first, 4 comparisons; for second three comparisons and so for. So 4+3+2+1 = 10 comparisons.

b) For shell sort if gap value is 7, it won't compare because there no elements after 7 index. If gap value is 3 it will compare 3 and 6. And for 1 it will compare all (which is 5-1=4). 4+1 = 5 comparisons.

c) Quicksort (first element as pivot). Firstly we are going to compare pivot with middle, last element which is bigger. After we check and done, we are do it for left side recursively. $\frac{(5-1)}{2} = 2$ numbers needs 1 comparison and for right $\frac{5-1}{2} = 2$ numbers need 1 comparison so 1+1+2 = 4 comparisons.

d) Mergesort (left has $\frac{5}{2} = 2$ element and right has 3 elements) left[i] and right[j] will be compared so i=0 ⇒ 1 comparison
i=1 ⇒ 1 "
i=2 ⇒ 1 comparison
+

need of bigger table size. while the table ...
also the elements are not many (for example less than $\frac{tableSize}{2}$)

it prevents the infinite loop possibility, because when it is close to full, program can stuck in infinite loop. But in linear probing, it easily becomes table size close to full and it increase the risk for infinite loop.

3) [3, 4, 7, 6, 9]

a) For insertion sort, there are 2 nested loop (for example for i and j) and every outer loop beginning, we are going to compare array[i] with array[j] and in inner loop j will increment all the time. If we say size is 5; for first, 4 comparisons; for second three comparisons and so for. So $4+3+2+1 = 10$ comparisons.

b) For shell sort if gap value is 7, it won't compare because there no elements after 7 index. If gap value is 3 it will compare 3 and 6. And for 1 it will compare all (which is $5-1=4$). $4+1=5$ comparisons.

c) Quicksort (first element as pivot). Firstly we are going to compare pivot with middle/last element which is bigger. After we check and done, we are do it for left side recursively. $\frac{5-1}{2}=2$ numbers needs 1 comparison and for right $\frac{5-1}{2}=2$ numbers need 1 comparison so it $1+2=4$ comparisons.

d) Mergesort (left has $\frac{5}{2}=2$ element and right has 3 elements), left[0] and right[0] will be compared so i=0 ⇒ 1 comparison
i=1 ⇒ 1 "
i=2 ⇒ 1 comparison
3 comparisons

3b

Transitions Animations Slide Show Review View Help Share Comme


7-sort

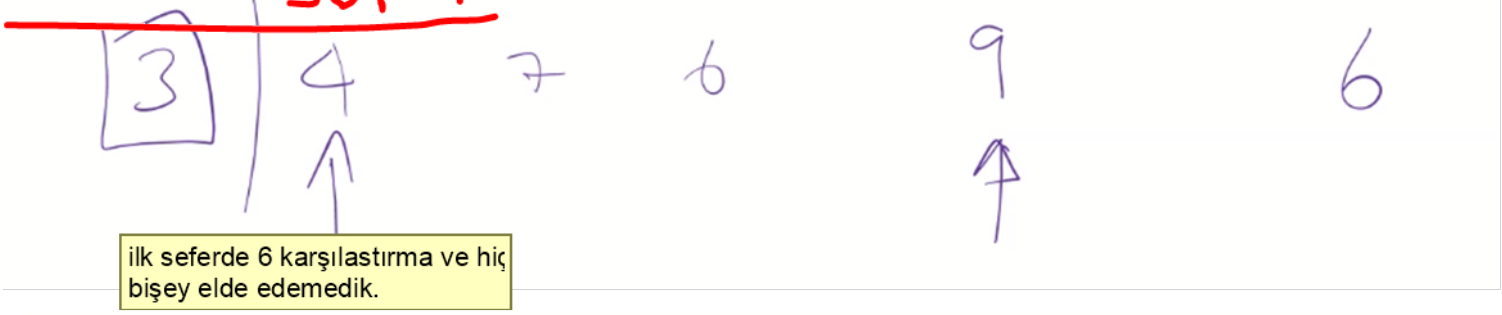
3-sort

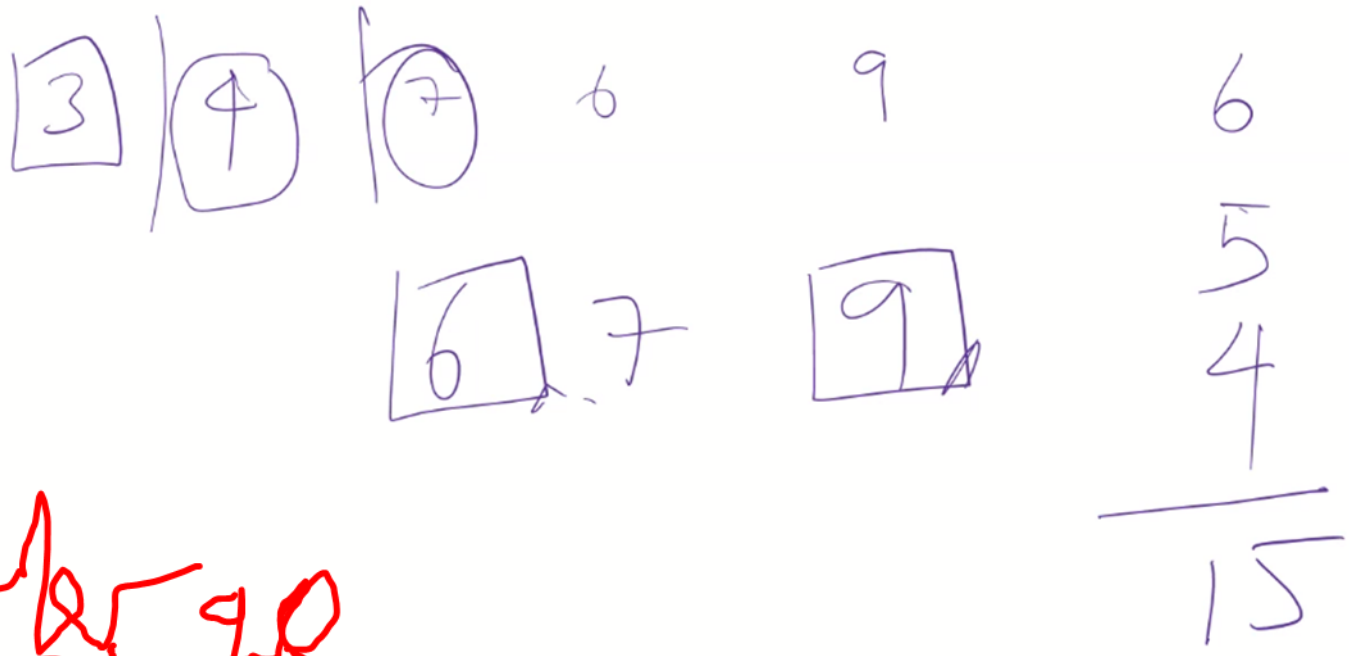
3 4 7 6 9

1-sort

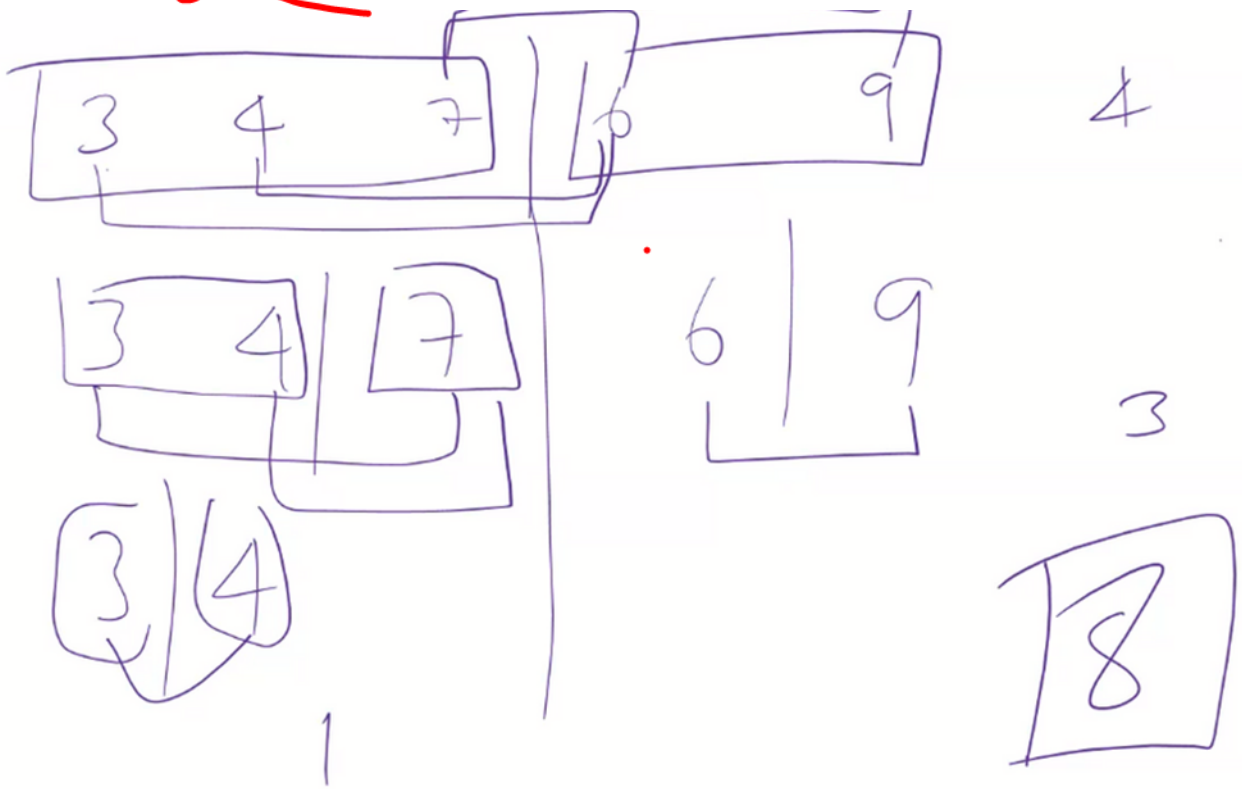
2+5 = 7

# quicksort




# Merge



You have 40 minutes

4. (30pts.) Write a method **`findPred`** in the class **`BinarySearchTree`** to find predecessor of a value passed as a parameter in the binary search tree. Note that the predecessor is the largest element which is smaller than the given value. The value passed as may not be avaliable in the binary search tree. Your function should return the predecessor or null reference if the value is not bigger than the smallest element in the binary search tree. Analyze the worst-case and best-case performance of your method and give examples for both cases.

5. (30 pts.) Write a method **`update`** in the class **`PriorityQueue`** which uses array based binary heap structure. The method takes an array index to locate the element and the new priority value as parameters. It should update the priority value of the element and perform required operations to maintain heap order property. Analyze the worst-case and best-case performance of your method and give examples for both cases.

Süleyman Gölbol
1801042656
CSE 222 Midterm Examination

4)

```
public E findPred(BST<E extends Comparable<E>> root,
                  Node<E> item, Node<E> parent){
    if(item.data.compareTo(root.data)==0 && root.left.right==null)
        return root.left.data; // if this is first call
                               // it will return leftdata because
                               // smaller one is there

    if (item.data.compareTo(parent.data) < 0){
        Node<E> tempParent = root;
        root = root.left;
        findPred(root, item, tempParent);

    }
    else if (item.data.compareTo(parent.data) > 0) {
        Node<E> tempParent = root;
        root = root.right;
        findPred(root, item, tempParent);

    }
    return parent.data;
}
```

In best case item will be in root so it will return left child's data, if left child has right child recursively it will compare with root's parent which is bigger and check tree recursively. Best case is $\Omega(1)$ and worst case is $O(n)$ (Not $O(\log n)$ because in worst case tree is not balanced so it will go until left-left.... or right, right, right....) so h will be n in worst case.


node
value
pred

Süleyman Gölbol
180104256
CSE 222 Midterm Examin

5)

```
public void update (int index, E newValue){
    int leftIndex = 2*index +1;
    int rightIndex = 2*index+2;
    pQueue[index] = newValue;
    for (int i=0; i< pQueue.length ; i++){   //heapify
        if (newValue.compareTo( pQueue[leftIndex]) <0){
            E temp = pQueue[leftIndex];
            pQueue[leftIndex] = newValue;
            pQueue[index] = temp;
        }
        if (newValue.compareTo(pQueue[rightIndex]) > 0){
            E temp = pQueue[rightIndex];
            pQueue[rightIndex] = newValue;
            pQueue[index] = temp;
        }
    }
}
```