Süleyman Gölbol

SOLUTIONS

1. Analyze the following code segment and give the running time in theta (θ) notation. Explain your answer.

```
sum = 0;
for (int i=1; i<myList.size(); i*=2)
    sum += myList.get(i);
```

At each iteration i is multiplied by two, the loop iterate for 1, 2, 4, 8, …, n/2, n (if the size of the list, n = 2^k). The number of iterations is log(n).

a) If **`myList`** is an instance of **`ArrayList`**

In ArrayList, get operation takes Teta(1) time. Total time for all iterations is Teta(log n)

b) If **`myList`** is an instance of **`LinkedList`**

In LinkedList, get operation should traverse the links until the ith element is reached. So, the running time is Teta(i). If you add this time for all i values, total time for all iterations is Teta(n). Note that

n+n/2+n/4+…+8+4+2+1=n-1 for n=2^k

2. Write a recurrence relation for the running time of the following recursive method.

```
int foo (ArrayList l){
    if (0==l.size()) return 0;
    int sum = 0;
    int x = l.get(l.size()-1);
    for (int i=0; i<l.size(); ++i)
        sum += x % i;
    l.remove(l.size()-1);
    sum += foo(l);
    return sum;
}
```

The for loop iterates Teta(n) times and each iteration requires Teta(1) time. So, the for loop takes Teta(n) time. For ArrayList, get and remove operations takes Teta(1) time for the last element on the list. Thus, the running time of the function except the recursive call takes Teta(n) time. The recurrence relation:

T(n) = T(n-1) + Teta(n)

T(0) = Teta(1)

3. State whether O(nlogn), θ(n), and Ω(n^2) are true or false for the running time of the following two methods where n is the number of elements in the **`LinkedList`**. Two incorrect answers cancel out one correct answer. Explain your answer.

a) The following method of **`LinkedList`** that converts it to **`String`**. Hint: Strings are immutable in Java.

```
public String toString() {
    Node<String> nodeRef = head;
    String result = "";
    while (nodeRef != null) {
       result = result +
          nodeRef.data.toString() + " ";
       nodeRef = nodeRef.next;
    }
    return result;
}
```

The running time of the method is Teta(n^2) since the n elements are add to the string and the string is copied into a new one for each addition. So,

- O(nlogn) is F
- θ(n) is F
- Ω(n^2) is T

b) The method **`listIterator`** of **`LinkedList`** which takes an integer as a parameter.

```
ListIterator listIterator (int index);
```

The running time of the method is O(n) since in LinkedList the links should be traversed until the element at the index is reached. So,

- O(nlogn) is T
- θ(n) is F
- Ω(n^2) is F

4. Write a generic Java class **`MyDeque`** that implements **`Deque`** interface. You should keep an instance of **`ArrayList`** class to store the elements in the deque. You should at least implement the following four methods exactly as defined below.

`boolean offerFirst(E item); //` Inserts the item at the front of the deque in **amortized** constant time.
`boolean offerLast(E item); //` Insert the item at the rear of the deque in **amortized** constant time.
`E pollFirst(); //` Removes the entry at the front of the deque and returns it in **θ(1)** time.
`E pollLast(); //` Removes the entry at the rear of the deque and returns it in **θ(1)** time.

```
public class MyDeque<E> implements Deque<E>{

  private ArrayList<E> data;
  private int front=0;
  private int rear=0;
  private int size=0;
  E dummy;

  puclic MyQueue()
    data=new ArrayList<E>(10);
    dummy=new E();
    for(int i=0; i<10; ++i)
      data.add(dummy);
    front=0;
    rear=data.size();
    size=0;
  }
```

```
public boolean offerFirst(E item){
   if (size=data.size())
     reallocate();
   if (front==0)
     front=data.size()-1;
   else
     front--;
   result=data.set(front,item);
   size++;
   return true;
 }
```

```
public boolean offerLast(E item){
  if (size=data.size())
    reallocate();
  if (rear==data.size()-1)
    rear=0;
  else
    rear++;
  result=data.set(rear,item);
  size++;
  return true;
}

private reallocate() {
  ArrayList<E> nd;
  nd=new ArrayList<E>(2*data.size());
  int k=0;
  for(int i=0; i<size; ++i)
    k=(front+i)%data.size();
    nd.add(data.get(k));
  for(int i=0; i<size; ++i)
    nd.add(dummy);
  size*=2;
  data=nd;
}
```

```
public E pollFirst(){
  if (size=0)
    return null;
  E result=data.get(front);
  data.set(front,dummy);
  if (front==data.size()-1)
    front=0;
  else
    front++;
  size--;
  return result;
}

public E pollLast(){
  if (size=0)
    return null;
  E result=data.get(rear);
  data.set(rear,dummy);
  if (rear==0)
    rear= data.size()-1;
  else
    rear--;
  size--;
  return result;
}
```

Grading

1. [20 pts]

The number of iterations is log(n) -> 4 pts.

a) If **`myList`** is an instance of **`ArrayList`**

In ArrayList, get operation takes Teta(1) time. -> 4 pts.

Total time for all iterations is Teta(log n) -> 4pts.

b) If **`myList`** is an instance of **`LinkedList`**

In LinkedList, running time od get is Teta(i). -> 4 pts.

Total time for all iterations is Teta(n). -> 4pts.

n+n/2+n/4+…+8+4+2+1=n-1 for n=2^k

2. [15 pts.]

For ArrayList, get and remove operations takes Teta(1) time for the last element on the list. -> 3 pts.

Thus, the running time of the function except the recursive call takes Teta(n) time. -> 3 pts.

T(n) = T(n-1) + Teta(n) -> 6 pts.

T(0) = Teta(1) -> 3 pts.

No recursion -> -10

3. [20 pts.]

a) The running time is Teta(n^2) -> 4 pts.

- O(nlogn) is F -> 2 pts.
- θ(n) is F -> 2 pts.
- $\Omega(n^2)$ is T -> 2 pts.

b) The running time is O(n) -> 4 pts.

- O(nlogn) is T -> 2 pts
- θ(n) is F -> 2 pts
- $\Omega(n^2)$ is F -> 2 pts

4. [50 pts.]

ArrayList -> 5
Deque -> 5
Declerations & Constructor -> 10
Offers -> 10
Pools -> 10
Reallocation -> 10

Not circular or not constant time -> -15