CS 580 HW 4

Ankur Dhoot

February 22, 2017

Q1

We'll use a balanced search tree (such as a red-black tree) to support the efficient execution of the operations. The keys in the red-black tree will be the car serial numbers. We'll assume that we have car objects that contain the serial number and price as member variables.

Insert: This is completely straightforwad using the RB-INSERT operation given in CLRS 13.3. We'll assume the tree T is a global variable and already initialized.

To insert, we simply take the serial number and the price, create a car object, and insert it into the tree using RB-INSERT which runs in O(logn).

To Find-Price, we simply use the normal binary tree search algorithm following an appropriate path from the root to our destination node (if it exists). The height of a red-black tree is O(logn) so Find-Price runs in O(logn).

Since RB-Delete requires a pointer to the node to be deleted, we first find the node in O(logn) and then call RB-Delete which runs in O(logn) for a total deletion run time of O(logn).

To multiply the price of all the cars in a given serial range by a fixed rate, we start at the root and work our way done as follows: If the given node lies within our range, multiply its price by the rate. Then recurse on its left and right child. If a node does not lie within the range, it is either above or below the range. If the node lies below the range, all possible nodes that could lie within the range must be in its right subtree so we recurse on the right child. Analogously if the node lies above the range. It's easy to see this procedure runs in time O(h + m) where h is the height of the tree (O(logn)) and m is the number of nodes that lie within the given range. If the number of nodes within the given range is $\Omega(logn)$, then the runtime is O(m).

Algorithm 1 Insert Car into RB tree

```
1: function INSERT(serialNum, price)
2:     z = new Car
3:     z.key = serialNum
4:     z.price = price
5:     RB-Insert(T, z)
6: end function
```

Algorithm 2 Find node given serial number (key)

```
1: function FIND-NODE(numb)
2:     x = T.root
3:     while x $\neq$ NIL and numb $\neq$ x.key do
4:         if numb $<$ x.key then
5:             x = x.left
6:         elsex = x.right
7:         end if
8:     end while
9:     return x
10: end function
```

Algorithm 3 Find Car Price

```
1: function FIND-PRICE(numb)
2:     x = find-node(numb)
3:     return x.price
4: end function
```

Algorithm 4 Delete Car from RB tree

```
1: function DELETE(numb)
2:     z = find-node(numb)
3:     RB-Delete(T, z)
4: end function
```

Algorithm 5 Insert Car into RB tree

```
1: function MULT-PRICE(rate, numb1, numb2) x = T.root mult-price(x,
   rate, numb1, numb2)
2: end function
```

Algorithm 6 Multiply price of cars in RB-Tree

```
function MULT-PRICE(x, rate, numb1, numb2)
    if x == NIL then
        return
    end if
    if numb1 $\leq$ x.key $\leq$ numb2 then
        x.price = x.price * rate
        mult-price(x.left, numb1, numb2)
        mult-price(x.right, numb1, numb2)
    else if x.key < numb1 then
        mult-price(x.right, numb1, numb2)
    else
        mult-price(x.left, numb1, numb2)
    end if
end function
```

Q2

We can solve this problem in constant amortized time using a stack and a deque where the underlying implementation of the deque supports insertion / deletion at either end in O(1) (e.g using a single doubly linked list) (the LinkedList class in Java is such an implementation of a deque). The tricky part comes in maintaining the task with the minimum id among all the tasks remaining in the list. We do this by using a stack with the following invariant: The task at the top of the stack is the task with minimum id among all elements remaining in the list.

To maintain this invariant, we do the following: Every time a new task is inserted into the deque, either at the left or the right end, we check if the id for that task is less than the current minimum (which is at the top of the stack). If so, we push that task onto the stack. When deleting from the linked-list, we check if the element being deleted has the minimum id among all elements in the list. If so, we pop that task from the stack and the new task at the top of the stack now has the minimum id among all remaining tasks in the list.

Let $\phi(D_i)$ denote the number of elements in the stack at the ith step. Then $\phi(D_0) = 0$ and $\phi(D_i) \geq 0$ for all i.

The amortized cost for insert-left or insert-right:

$\hat{c_i} = c_i + \phi(D_i)$ - $\phi(D_{i-1})$ is at most 1 + 1 = O(1) since $\phi(D_i)$ - $\phi(D_{i-1})$ is O(1) (we can only add at most one element from the stack in which case $\phi(D_i)$ - $\phi(D_{i-1}) = 1$)

The amortized cost for delete-left or delete-right:

$\hat{c_i} = c_i + \phi(D_i)$ - $\phi(D_{i-1}) \leq 1$ (if we remove an element from the stack $\phi(D_i)$ - $\phi(D_{i-1}) = -1$ else 0).

The amortized cost for identify-min:

$\hat{c_i} = c_i + \phi(D_i)$ - $\phi(D_{i-1}) = c_i = 1$ ($\phi(D_i)$ - $\phi(D_{i-1}) = 0$ since we don't add or remove from the stack)

Thus, all operations can be supported in constant time.

In the following algorithms, we assume that a stack s and deque dq have been initialized.

```
function INSERT-LEFT(x)
    if ! s.isEmpty()  then
        if x.id < s.peek().id then
            s.push(x)
        end if
    else                                         ▷ No items in list
        s.push(x)
    end if
    dq.addFirst(x)
end function
```

```
function INSERT-RIGHT(x)
    if ! s.isEmpty()  then
        if x.id < s.peek().id then
            s.push(x)
        end if
    else                                         ▷ No items in list
        s.push(x)
    end if
    dq.addLast(x)
end function
```

```
function DELETE-LEFT
    if ! dq.isEmpty()  then
        x = dq.removeFirst()
        if x == s.peek() then
            s.pop()
        end if
        return x
    end if
end function
```

```
function DELETE-RIGHT
    if ! dq.isEmpty()  then
        x = dq.removeLast()
        if x == s.peek() then
            s.pop()
        end if
        return x
    end if
end function
```

Algorithm 7 Return pointer to task with min id among all tasks in list

```
function IDENTIFY-MIN(x)
    return s.peek()
end function
```