Data Structure Assignment 2

Yair Lahad (205493018) Igal Boxerman (207553231)

i. int backtrackingSearch1(Array A, int x).

This function iteratively searches for x in an unsorted array by performing the following actions in its $i\ th$ iteration:

- Perform i search steps
- Backtrack $i - 1$ steps

Let's use the iterative method in order to understand how the function run.

$i = 1$: we get array[1] and we need to do 0 backtracks = 1 atomic operation.

$i = 2$: search from array[1] to array [3] and after 1 backtrack array[2] = 3 atomic operations.

$i = 3$: search from array[2] to array[5] and 2 backtracks to array[3] = 5 atomic operations.

For the i'th iteration we get 2(i-1)+1=2i-1

In the worst case the value x is not in the array so we will run on all array sized n and get arithmetic series which sum is

$\boldsymbol{s_n = \frac{n*(2n-1+1)}{2} = n^2 = \theta(n^2)}$ ***as we proved in class***

ii. backtrackingSearch2(Array A, int x)

when analyising the psedu code given we conclude that for the i'th iteration we get a run time of $1 + 2(i-2) + 2(i-1) + \cdots + 2*(1) =$

$$1 + 2*(1 + 2 + \cdots + i - 2) = by\ the\ sum\ of\ arithmetic\ series$$

$$= 1 + 2\left(\frac{(1 + (i-2)) * (i-2)}{2}\right) = i^2 - 3i + 3$$

Each iteration, we advanced 1 cell of the array, and all the other steps are canceled by thier backtrack with same length.

Therefore for the worst case, the x is not in the array therefore we will have n iteration and we will get:

$$\sum_{i=1}^{n} i^2 - 3i + 2 = (from\,summation\,rules) = \sum_{i=1}^{n} i^2 - 3\sum_{i=1}^{n} i + \sum_{i=1}^{n} 3 =$$

$$= (from * and **)we\ get = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} - \frac{3n^2}{2} - \frac{3n}{2} + 3n = \frac{n^3}{3} - n^2 + \frac{5n}{3} = \theta(n^3)$$

$$*\ we\ know\ that \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$$

$$**\ we\ know\ that \sum_{i=1}^{n} i^2 = \frac{n*(n+1)(2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$$

Therefore, for the worst case we get $\boldsymbol{\theta(n^3)}$.

iii. **BacktrackingSearch(A, x, $n1, n2$) - for $A.\,length > n1 > n2 \geq 0$**

In this function, we advance n1 steps forward and n2 steps backward.
For maximum steps we need the slower advanced pace and most backtracks number that can be .
Beacuse $n_1 > n_2$ and both are integers, for maximum backtracks: $n_2 = n_1 - 1$.
Let $1 \leq a \leq n-1 : n_1 = n - a$ (we chose n-1 because $n_2 \geq 0, 1$ because $n > n_1$)
In each step (a search action and a backtrack action) we advance $n_1 - n_2$ cells forward. So the number of iterations is $\frac{n}{n_1 - n_2}$ (Equation 1)
Every iteration (step) costs $n_1 + n_2$ (Equation 2).
From equation 1 and equation 2, which are the number of steps and the cost of each step, we get the total time complexity of the function:

$$\boldsymbol{T(n) = \left(\frac{n}{n_1 - n_2}\right) \cdot [n_1 + n_2] = \frac{n\,[(n-a)+((n-a)-1)]}{1} = 2n^2 - n \cdot (2a+1) = \Theta(n^2)}$$

2. **Analyze the runtime of Backtrack method in case of deletion and insertion.**

i. Unsorted array:

a. Insert

In unsorted array the insert method inserts an element at the end of the array, so the backtrack action delete this element. Deletion in unsorted array is $\Theta(1)$. Because there are 2 cases. One that the element is last then we simply remove the value in that index $= \Theta(1)$. And the second case is that the value isn't in last index, then we need to switch the element and the last element which is $\Theta(1)$ and delete the last element which is also $\Theta(1)$. In total we get $\Theta(1)$.

b. Delete

In Unsorted array the delete method removes an element in the array, so the backtrack action needs to insert the element back. The backtrack method does that by inserting a new element with the old value and switching between the cells values. So, we get in total $\Theta(1)$.

ii. Sorted array:

a. Insert

In sorted array the insert method inserts an element at specific index, so the backtrack action delete this element. In sorted array we delete the element and move some cells in order to maintain the order of the array.
in the worst case: we need to delete the last cell therefore run on all other indexes until we reach the n-1 index and remove it $= \Theta(n)$.

b. Delete

In sorted array the delete method removes an element in the array, so the backtrack action needs to insert the element back. In order to insert the element in the right index without destroying the sorted array, we need to find the specific index, which we have in the stack command, but, the other elements need to move according to their value. So, for worst case, if the element to insert is the first, all the other elements need to move by one. So, we get that the runtime is $\Theta(n)$.

iii. BST:

a. Insert

In BST the insert method inserts a node to the tree as a leaf, so the backtrack action delete this node. As we know which node at the tree is inserted, and we have a pointer to its parent node, we can simply change the value of his son's pointer to be null in order to remove the correct node, therefore if we will look at the code:

```
Node curr=com.getNode(); // function of O(1)
if(curr.parent.key>curr.key)
    curr.parent.left=null;
else
   curr.parent.right=null;
```

We can see that it includes only 4 atomic operations therefore the runtime is $\Theta(1)$.

b. Delete

In BST the delete method removes a node in the tree, so the backtrack action needs to insert the element back. There are 3 options for a deleted node:

case 1 – the deleted node had no sons:

In that case we just set the node's parent son pointer back to the node, which in total takes $\Theta(1)$.

case 2 – the deleted node had one son:

In that case we need to make two actions. first, set the parent son to be the deleted node, and then update the son's parent to be the deleted node. Those are 2 action which every action is $\Theta(1)$, so in total is $\Theta(1)$.

case 3 – the deleted node had two sons:

In that case we do two actions, first we switch between the deleted node key and value that was deleted and his successor which we got at the command stack. All of this takes $\Theta(1)$.

Then we preform "backtrack delete" with successor node, so it sets as the node's son. This action is $\Theta(1)$, as we explained above. In total we get $\Theta(1)$.

iv. AVL-tree:
   Similarly to BST, we don't need to search for the correct node to handle, we receive it from stack, and only need to change pointers, and fix future imbalances by rotation operations, which we learned in class that their cost is $O(1)$.

   a. Insert
      When backtracking we need to delete the insertion. If we insert to the stack a pointer to the node which affected by the insertion (and now is unbalanced) and the kind of the rotation we have made - we reduce the time complexity because we don't need to search and balance the tree. In other words, we just rotate the tree back to the previous shape and then remove the new node. This pointer helps us to save log(n) steps of searching which node we need to balance, but we still need to update the heights of all the parents nodes, therefore in the worst case we need to update log n nodes, which costs $\Theta(\log n)$.after we delete we need to rotate, Rotating costs $\Theta(1)$ (as shown in class) and the action of deleting cost $\Theta(1)$, hence, the backtrack action in total costs $\Theta(\log n)$.
   b. Delete
      Our backtrack of the delete is to insert a new node, which is similar to BST insertion which costs $\Theta(1)$, and in the worst case, we need to fix imbalances by the proper rotation which as we learned, each rotate costs $\Theta(1)$.
      But as we insert a new node, we need to balance the tree. The worst case is that the root is unbalanced, so we have to "climb" the route to the top of the tree (let h be the tree's height). Hence, the rotation might be at the top and costs $\Theta(1)$. As shown in class, when inserting a node to the tree, you need to balance only one unbalanced node.
      Yet, we still need to update the height of all the inserted node parents so in the worst case we must update log n nodes, therefore it costs $\Theta(\log n)$.
      Overall, the runtime is: $\Theta(\log n)$.