Iason Kotakis

Daniela Rosu

ITEC 2620 – Introduction to Data Structures

23/02/2021

CSC263 FALL 2006 ASSIGNMENT 1

**Written Question 1**

1. The best case for this algorithm is when all the first *n/2* elements of *A* are 1's. This is because as soon as the counter *numOnes* reaches *n/2* the algorithm will return *True*, and only half of the array will need to be checked. Therefore, the best case complexity of *HalfOnes* is:

In the case were the first half of the vector contains 0's instead of 1's, it would always be necessary to check *(n/2) + 1*, one more element, than in the previous case. This is because the condition *numZeros > n/2* will not trigger when there are *n/2* 0s, therefore, this is not a best case.

2. The worst case for this algorithm is when A contains an amount of 1s equal to *(n/2)-1* distributed in the first n-1 elements. This is because, in this situation, neither counter will fulfill the return condition until the last element, so the entire vector will need to be traversed. Therefore, worse case complexity will be:

3. To find the average case complexity, we have to determine the sum of the time complexity of all possible cases and divide it by the number of cases there are. The input vector *A* has a size *n*, with each element being either 0 or 1, therefore there are 2n possible cases. We will define *x* as the amount of 1s in *A* for this analysis. The algorithm can be stopped either by counting sufficient ones if *x ≥ n/2* or sufficient zeroes if *x < n/2*.

In the cases where *x ≥ n/2* the algorithm will stop after finding *n/2* amount of ones. Similarly if *x < n/2* the algorithm will stop after finding *n/2+1* zeroes. The time complexity in these cases depends on where the first *n/2* ones or *n/2+1* zeroes are distributed respectively.

In the best case, we have 2n/2 possibilities of having a time complexity of *T=n/2*. If we then assume that there is a 0 before the number of ones reaches *n/2*, the number of possibilities for this scenario are *2(n/2)-1 + n/2* and the time complexity for these cases is *T=n/2+1*. We add *n/2* to the possibilities, because the 0 we added can be in any position within the first *n/2* elements of *A*. If we were to continue adding zeroes before the final one, the next range of possibilities is *2(n/2)-2* with complexity *T=n/2+2*. The final step would be when the number of added zeroes is equal to n/2, in which case the possibilities would become 20 or 1, and the time complexity would be at it’s maximum of *T=n/2 + n/2* or *T=n.*

**Written Question 2**

1. Given a sorted array *A* of length *n* we can construct an ideally balanced binary search tree with the following algorithm:

ArrayToBST(A)

if A = *null*

return *null*

*mid* ← *n*/2

*T.data* ← A[*mid*]

*T.left* ← ArrayToBST(A[1..*mid*-1])

*T.right* ← ArrayToBST(A[*mid*+1..*n*])

return *root*

This algorithm builds takes the middle element of the array and using it as the root, and then assigns the sub-trees created from the slices of *A* before and after that element, to the children. Since the array is sorted, we know that every element that comes before the middle one will be smaller and every element that comes after will be larger. Each side of the tree will be recursively sliced in half, leaving the middle element as the root. Since the exact same process is done on each child, the amount of nodes on either side will only differ if *n* is an even number. In this case there would be no "middle" node, and the right side will have a slice that is one element smaller than the left side.

Every element of the array will be compared to null regardless of the contents of *A*, therefore this is the worse case. All the operations done on each element are performed in a constant time which we will call *“c”*. We can conclude from this that the worst case complexity of this algorithm will be:

or *O(n)*.

2. Given an unbalanced binary search tree *T*, we use the following algorithm to transform it into a balanced binary search tree:

BalanceBT(T)

A ← BSTToArray(T)

return ArrayToBST(A)

BSTToArray(T)

if T = *null*

return *null*

A ← Join( A, BSTtoArray(T.left) )

A ← Append( A, T.data )

A ← Join (A, BSTToArray(T.right) )

return A

This algorithm uses the previously made *ArrayToBST* algorithm to build a balanced binary search tree from a sorted array. Since a binary search tree should be already sorted even when unbalanced, we can store the nodes in a sorted array by saving the data from each one in order. Once we have a sorted array, we can use *ArrayToBST* to construct a balanced binary search tree.

The algorithm that builds a sorted array from the unbalanced tree makes a single comparison for every node in the tree, regardless of the input, so the worst case is to compare all the nodes. The operations in *BSTToArray* take a constant time to execute, which we will call *"c"*, therefore, the time complexity of *BSTToArray* is: or *O(n)*.

Finally, *BalanceBT* is the sequence of both algorithms, therefore, each element in both trees has been processed twice in a constant time. Knowing this, we conclude that time complexity of *BalanceBT* is:

or *O(n)*.

3. In order to sort an unordered array into a binary search tree, we perform comparisons between the elements to determine the order they should be in. For an input with size *n*, the amount of different possible arrangements in which the data might be found, or permutations, is *n!*. With every comparison we can decide between 2 possible arrangements, so if we make *x* comparisons, we can distinguish *2x* possible permutations. In order to distinguish all the possible permutations with comparisons, the following must be true: *2x ≥ n!.*

We can simplify this expression as *x ≥ log2(n!)* and thus conclude that the minimum amount of comparisons needed for a sorting algorithm is *log2(n!)*, or more simply *nLog2n.* Therefore the minimum time complexity of a comparison-based sorting algorithm is *O(nLog2n).*

**Written Question 3**

29

26

22

40

37

42

33

11

13

12

10

5

9

6

3

1. Insert an element with key 30 into the original tree.

29

26

22

40

33

42

30

11

13

12

10

5

9

6

3

37

b) Insert an element with key 2 into the original tree.

29

26

22

40

37

42

33

11

13

12

10

5

9

6

3

2

1. Delete the node with key 12 from the original tree.

29

26

22

40

37

42

33

11

13

10

5

9

6

3

1. Delete the node with key 26 from the original tree.

29

22

40

37

42

33

11

13

12

10

5

9

6

3

Q4.

1. **If we replaced property 2 of red-black trees (that every child of a red node is black) with the property that every grandchild of a red node is black, would every red-black tree still be balanced? Prove your answer.**

The tree could be balanced or it couldn’t be balanced; it all depends on the colour of the node. The most important thing is that the root should be black node according to the red-black property. However, sometimes to make it a red-black tree it is important to count the number of black nodes. The reason why it is balanced is that every new node is red so later that red node becomes a parent. Also, the child node becomes black as it has the parent node. But, having red-red nodes, which means parent-child relationship then isn't possible. However, every new root of the trees is black. But, as we add a new rode that has to be red, so the grandchild node can’t be black. Also, if a parent node is red, then the child node is black. So, later that grandchild depends on the colour of the child or parent nodes to balance the tree. Also, when finding the longest path the node has to be an alternatively red-black node. Also, when rotating the trees the parent and sibling must be black then the grandchild becomes red. The red-black tree should be balanced otherwise it would violate the properties and become the worst case. Also, it could not be a possible red-red node which is parent and child nodes. If a parent node is red, then the child node can’t be red. Therefore, to keep the tree balanced there should be a look after the number of nodes then it will be called a red-black tree.

1. **What if we replaced property 2 with the property that every child of a black node is red? Prove your answer.**

This can’t be possible in the red-black tree. If every child node turns out to be a red node, then there will be a node violation. Where in the tree it can’t be two red nodes. But, if there are two red nodes then it should either recolour or it should rotate the tree to balance the node. Also, if there is a colour violation, then there are two possibilities. Firstly, recolour the nodes. Secondly rotate the nodes. However, when rotating the nodes make sure that the colour is also balanced. When balancing the nodes and the colour black changes to red, or red changes to black. Then, there could be an imbalance. However, if the parent node is red, then the child node can’t be read. But, if the parent node is black then the child node can be black. Also, if the parent node is black, then the child can be red. However, we have to make sure that the black nodes are equal in the red-black tree. Also, by creating a red-black tree, there should be a balanced number of nodes in the tree. Moreover, the head of the node will always remain black, it can’t be read. Also, if the parent node is red, then the child node has to be black. However, in the tree, the black nodes must have the same numbers. Also, the numbers of NIL leaves are always black.