Advanced Algorithm

DR Moeini

Parsa Mohammadpour

Hesam Mumivand Fard

Mohammad Rahbarimoghadam

Saeid Terik

Second assignment

Fall 2024-2025

1- Prove that it doesn’t matter which node you start searching from in the binary search tree with the height of h, k times of calling TREE_SUCCESSOR, will be done in O (h+k).

The objective is to prove that starting from any node in a binary search tree of height h and performing k consecutive calls to the TREE-SUCCESSOR function will result in a runtime of O(k + h). We aim to analyze the execution time of this operation.

TREE-SUCCESSOR Function Behavior:

The TREE-SUCCESSOR function operates similarly to tree traversal. For each node x in a BST, the edge connecting x and its parent (x.p) is used when TREE-SUCCESSOR is called on x.p. This edge may be used again when TREE-SUCCESSOR is called on the largest element in x's subtree. Besides these instances and the initial step to find the tree's smallest element, this edge won't be traversed again.

Runtime Analysis:

The previous observation indicates that each edge in the BST is traversed at most twice during k consecutive TREE-SUCCESSOR calls. Given that the number of edges in a tree is one less than the number of nodes, the total runtime can be bounded by 2n in the worst case (when k equals n and we start from the smallest node), where n is the number of nodes in the tree.

However, a tighter bound can be achieved. We have a traversal with only k successors, so 2n can be reduced to 2k. Additionally, nodes with key values between A and B may be examined up to three times. (This occurs when we reach a node from its parent to find a successor, move to its left subtree children, revisit the node after traversing the left subtree, then move to its right subtree, and finally revisit the node to find the next successor. Thus, a node can be visited up to 3 times. 3k)

It's important to note that we may also visit nodes with key values outside the range between the start node (A) and the k-th successor (B). These nodes will be examined at most once, occurring along the path from A to R' or from B to R', where R' is the nearest common ancestor of A and B. The length of these paths is bounded by the tree's height, h; at most once for ascending and once for descending. 2h

Final Runtime Bound:

Therefore, the overall runtime can be bounded by 2k + 2h or, in the worst case, 3k + 2h. This bound represents the maximum number of node examinations, considering nodes within and outside the range between A and B, along with potential repetitions. This ultimately leads to a time complexity of O(k + h).

2- Assume that instead of having x.p (nodes parent) at each node that refers to the node parent, each node maintains the variable x.succ that refers to the node successor. By writing pseudocode of SEARCH, INSERT and DELETE in binary search tree show how they change. These actions must be done in O (h), where h is the height of the tree. (help: try to write the pseudocode that returns each nodes parent)

First, due to the question hint, we write the pseudocode for finding each node's parent. This function is called "***Find_Parent***" and it takes nodes value (the node that we want to find its parent) to the function along with the tree root, then in return, it returns the node that is the requested node parent. In this code we started by the root, and in each step, we store the previously seen node in a variable called parent then at the end, when we reached the node with the specified value, the same node that we wanted to get its parent, we return the parent. And because we are calling this function for a node in that is in the tree, we can assume that we have this node, but if we didn't want to assume this assumption, we would have checked at the end if the value of the node matches the required value or not, or even the node is null (which means we don't have this node in our tree) Pseudocode for this function, is as follows:

```
Find_Parent(root, val):
    parent = Null
    node = root
    while node is not Null and node.val != val:
            if node.val > val then:
                    parent = node
                    node = node.Left
            else then:
                    parent = node
                    node = node.Right
    return parent
```

in the above code, in the worst case we go through all the longest path from root to a leaf which is the tree height that we know is in O (log n) on average.

Now we move to the next function that the question ask which is “***SEARCH***”. This code will remain the same because in none of the algorithm step, we have used the parent in the code for simple BST. So now we only repeat the pseudocode again. We only mention that the Null will be returned in case that we don’t find a node with this value at tree. So, the pseudocode is as follows:

```
SEARCH(root, val):
    node = root
    while node is not Null and node.val != val:
            if node.val > val then:
                    node = node.Left
            else then:
                    node = node.Right

    return node
```

in the above code, in the worst case we go through all the longest path from root to a leaf which is the tree height that we know is in O (log n) on average.

For the next function, we need to provide “***INSERT***” pseudocode. For this action we take two arguments as an input, which are tree root and the value that we want to insert. For this function, the approach that we take is to start from the root and then at each step for adding the new value, compare it from the current node (initially the root) that we have and then if you are going right (the value is bigger than the current node value) then store that node in a variable called predecessor, which is the node with maximum value among the nodes that their values are less than the current node, or in another word, the node which is the current node previous node in the in-order traversal presentation of the tree. And we also store that node’s previous value at a node called parent to be able to assign the new node as its parent’s left or right child. So then after we have Null for our node value, it means that we have reached to the section that we have to generate a new node with that value and set its successor and update the predecessor nodes successor and set it to the new value. In this implementation we assume that we don’t have that value in our tree. But checking this can be done in O (log n) as we can simply use “*SEARCH*” algorithm that we provided in previous section which will be done in O (log n) and if it returns” *Null*”, we add this node, otherwise we return some exception or “*Null*” value. The pseudocode of the following algorithm is as follows:

```
INSERT(root, val):
    node = root
    parent = Null
    predecessor = Null
    while node is not Null:
        parent = node
        if node.val > val then:
            node = node.Left
        else then:
            predecessor = node
            node = node.Right
    newNode = GENERATE_NODE(val)
    if parent != Null then:
        if parent.val > val then:
            parent.Left = newNode
        else then:
            parent.Right = newNode
    if predecessor != Null:
        newNode.succ = predecessor.succ
        predecessor.succ = newNode
    else then: // we have only came left
        newNode.succ = parent
    return
```

in the above code, in the worst case we go through all the longest path from root to a leaf which is the tree height that we know is in O (log n) on average.

Now for the last part, we will provide the pseudocode of the "*DELETE*" action. For this action, we will start from the root and then at each step we store the node value current value in parent variable and beside that if we go to the right based on the node value, we store the value of the node before updating it at the variable called predecessor. Then when we arrive to the node with the value equal to the required value in the input, we first set it predecessor's successor to this target node successor and then if it doesn't have any child, then we return, if it has one child, we replace its parent child (whichever it was, left or right) to that node, if it had both children, we get its successor, and remove that successor and then set that successor here. The pseudocode is as follows:

```
DELETE(root, val):
    node = root
    predecessor = Null
    while node is not Null:
        if node.val > val then:
            node = node.Left
        else then:
            predecessor = node
            node = node.Right
    predecessor .succ = node.succ
    parent = Find_Parent(root, node.val)
    pred_parent = Find_Parent(root, predecessor.val)
    if pred_parent.Left = predecessor then:
        pred_parent.Left = Null
    else then:
        pred_parent.Right = Null
    if parent.Left = node then:
        parent.Left = predecessor
    else then:
        parent.Right = predecessor
    del(node)
```

So, in the worst case, we will go until the tree depth which is O (log n) on average and call parent function which is in O (log n). So, our Pseudocode order will be O (log n) on average.

3- Consider that we have alphabet {a, b, c, d, e}. Assume that we have the following frequencies for this alphabet in a text:

$$f_a = 0.32, f_b = 0.25, f_c = 0.2, f_d = 0.18, f_e = 0.05$$

What is the number of expected bits for Huffman encoding of a text with 1000 character?

First, we briefly explain Huffman encoding. Then we show the result of the algorithm for this example.

Huffman algorithm: Huffman algorithm is a greedy algorithm designed for encoding and decoding efficiently. This algorithm encodes each alphabet member such that no two different characters are prefix of each other and also tries to minimum the number of total bits that are used for encoding the whole text. As we said, this algorithm is a greedy algorithm. But why? We will see about that in the algorithm description. This goal, having minimum number of the bits in the encoded document, we try to assign the minimum length of the encoded bit-string[1] (string containing zeros and ones) to the alphabet that has the minimum frequency.

Huffman algorithm description: In this algorithm, we want to make a tree from these alphabets, a tree like BST[2], but with differences, and keep alphabet characters in its leaves and then by applying a simple rule we assign a bit-string to each of these characters. Our rule for assigning the bit-string to characters is to start from the root, if you went on the left child, add zero to the bit-string (which is empty string at the beginning) and by going on the right child, add one to the bit-string. Do this action until you get to the character that you want.

So, in the algorithm, first we convert each character with its frequency to a BST node. These nodes contain the character and the frequency of that character. For this purpose, we use a min-heap[3] (minimum priority queue) to store these nodes and the comparison function used for this heap, is by comparing each node frequency. In other meaning, we store a min-heap of these nodes by comparing their frequencies. Then at each step of the algorithm, we get first element of the heap by getting calling the *pop*[4] function of the heap, we get the element of the heap with least frequency and then by doing this action again, we get the element with second least frequency from the heap. Then we merge these two nodes together. We define merging two node, like the follow:

Merging two nodes: Generate a new node and set the frequency of this new node to the summation of these two nodes. And then assign the node with the smaller frequency to the left child of this

[1] bit-string (binary string): the string containing only zeros and ones
[2] BST: binary search tree
[3] using heap is not mandatory, we also could use array and sort that array at each step, but it would have more time complexity.
[4] pop function of the min-heap, return the minimum element of the heap and then remove it. Then by setting the last element to the first element and by applying some functions like *bubble-down* and *bubble-up*, it re-generates the heap such that the first element is the minimum element of the heap.

new node and assign the node with higher frequency to the right child of this new node. Then put this new node to the heap.

So, at each step, we remove two nodes from heap, and then add another node in that heap hence we have n-1 step totally. All these actions are done in O (1) except for heap actions. Heap actions like adding element and removing elements are done in O (lg n). So, the overall time complexity of this algorithm by using heap, is O (n lg n). We also have to generate a heap from these nodes at the beginning. This action is done in O (n). So, the overall time complexity is O (n + n lg n) which is same as O (n lg n).

Now let's move on to our example. By the algorithm, first we make nodes with frequency of each character and setting that character value in that node. Then we have to make a heap from these nodes. But for simplification, we just use the sorted array. Because their difference was in the time complexity not in the functionality of the algorithm. So, our sorted array is as follows:

Node	a	b	c	d	e
Freq	0.32	0.25	0.2	0.18	0.05

So as the algorithm requires, we have to get two nodes with minimum frequencies from this array, which are d and e, and then merge them together with each other. So, the merges node is as follows:

Merging e and d, will result in a new node f which has frequency of 0.05+0.18 = 0.23. SO, after inserting this new node to the array, the array is as follows:

Node	a	b	f e d	c
Freq	0.32	0.25	0.23	0.2

Now, if we apply this algorithm for the next step, we will have a new node g, which has the frequency of 0.23+0.2 which is equal to 0.43. So, the sorted array is as follows:

Node	g c f e d	a	b
Freq	0.43	0.32	0.25

Now we move forward to another step for the algorithm. So, we now make a new node from merging nodes a and b and name that new node to h. The frequency of the h is 0.25+0.32 which is equal to 0.57. So our sorted array will be as follows:

Node	h a b	g c f

		e d
Freq	0.57	0.43

Now, for the last step we get these last two remaining nodes and merge them together. So, now get these two nodes and merge them together and make the final node, the root of the tree, and then we will explain the final tree and assign the bit-string to each leaf (each alphabet character) and then compute the final answer. So, first let's build final tree. Our tree will be as follows:

Node	i h g a b c f e d
Freq	1

So, now we will have to assign the bit-string to each character. So, we apply the bit-string assignment algorithm, which was starting from the root with an empty string, and then for each time that go to the left child, we add a zero to the bit-string and each time we go to the right child, we add one to the bit-string. Then bit-string of each character is as follows:

character	bit-string
a	00
b	01
c	10
d	111
e	110

Now, for the last step, we have to find expected number of repetitions in the document and multiply that with number of required bit-string that is assigned to each character. Then we will compute some extra computation to show that this algorithm, Huffman algorithm, is more efficient.

So, due to the provided frequency of the characters, we will compute the expected number of bit used to encode each part of the document that is related each character. These computation for each character is as follows:

- character a:

$$f_a = 0.32 \Rightarrow E[number\ of\ 'a'\ character\ occurance\ in\ 1000\ character]$$
$$= 320$$
$$\Rightarrow E[number\ of\ bits\ used\ for\ character\ 'a'\ in\ 1000\ letter\ length\ document]$$
$$= 320 \times 2 = 640$$

- character b:

$f_b = 0.25 \Rightarrow E[number\ of\ 'b'\ character\ occurance\ in\ 1000\ character]$
$= 250$
$\Rightarrow E[number\ of\ bits\ used\ for\ character\ 'b'\ in\ 1000\ letter\ length\ document]$
$= 250 \times 2 = 500$

- character c:
  $f_c = 0.2 \Rightarrow E[number\ of\ 'c'\ character\ occurance\ in\ 1000\ character]$
  $= 200$
  $\Rightarrow E[number\ of\ bits\ used\ for\ character\ 'c'\ in\ 1000\ letter\ length\ document]$
  $= 200 \times 2 = 400$

- character d:
  $f_d = 0.18 \Rightarrow E[number\ of\ 'd'\ character\ occurance\ in\ 1000\ character]$
  $= 180$
  $\Rightarrow E[number\ of\ bits\ used\ for\ character\ 'd'\ in\ 1000\ letter\ length\ document]$
  $= 180 \times 3 = 540$

- character e:
  $f_e = 0.05 \Rightarrow E[number\ of\ 'e'\ character\ occurance\ in\ 1000\ character]$
  $= 50$
  $\Rightarrow E[number\ of\ bits\ used\ for\ character\ 'e'\ in\ 1000\ letter\ length\ document]$
  $= 50 \times 3 = 150$

So, the expected value of total number of used bits, is equal to 640+500+400+540+150 = 2230 bits.

But, if we wanted to use fixed length encoding like ASCII nut with length of 3 bits for all characters, we would have the exact value of 1000×3 = 3000 bits. So, obviously, by using Huffman algorithm, we have improved the expected number of bits in the document.