INF-1101: Data structure and algorithms

Assignment 1 report

Amund H. Strøm

February 28. 2022

1 Introduction

In this assignment the main objective was to create two different sorted **set ADT's** that support the same operations and compare the performance between them. The secondary objective was to create a spam filter that checks if a mail is spam or not, using the different operations created in the two different **set ADT's**.

2 Technical Background

By creating an ADT (abstract data type), it means to create a set of value and operations which describes the behavior of an object or class. The implementation is not necessary for the user, only what operations that can be performed. An example would be the ADT's that were created in this assignment. It can create a list, add elements to the list, remove elements, etc. The implementation of these operations is not important for the user, but only what kind of operations that are possible for this specific ADT. As for the meaning of a **set** ADT, it means that we create a list that don't have any repeats.

One of the ADT's uses a doubly linked list to store data, which is a list that uses pointer to connect elements to each other. This means that if you would like to access something that are in the middle of the list, you need to start at either the beginning or the end of the list, and then travers through the list. The difference between a linked list and a doubly linked list is that the doubly linked list has pointers that will point to the previous element in the list, for each node. This has the advantage of faster traversing, because you can start at the end of the list, if necessary. But has the disadvantage of taking more space in the memory. Bellow you can see an illustration of a doubly linked list

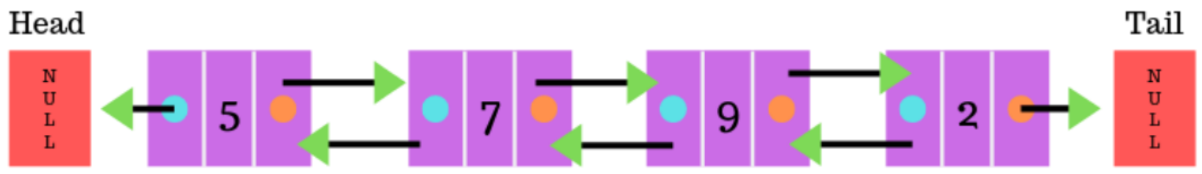


Figure 1: Illustration of a doubly linked list

The second ADT uses a binary search tree (BTS) to store data, which is a tree that will put each element to the right or left of itself, depending on if the value is bigger or smaller than itself. To access the data, there is a single "root" that stands at the top of the tree that needs to be traversed first, then it is possible to traverse downwards to access the rest of the tree. Here is an illustration of a BTS.

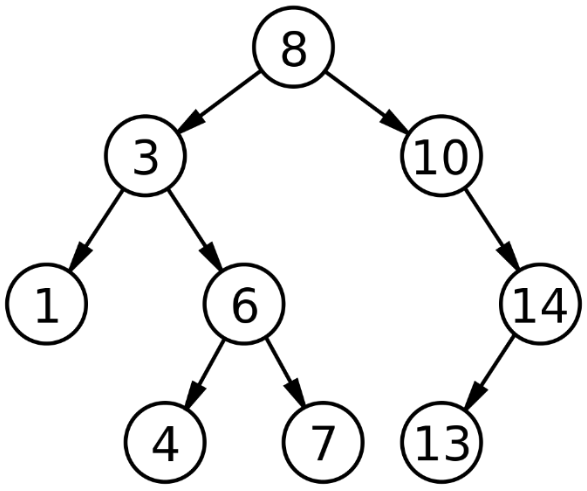
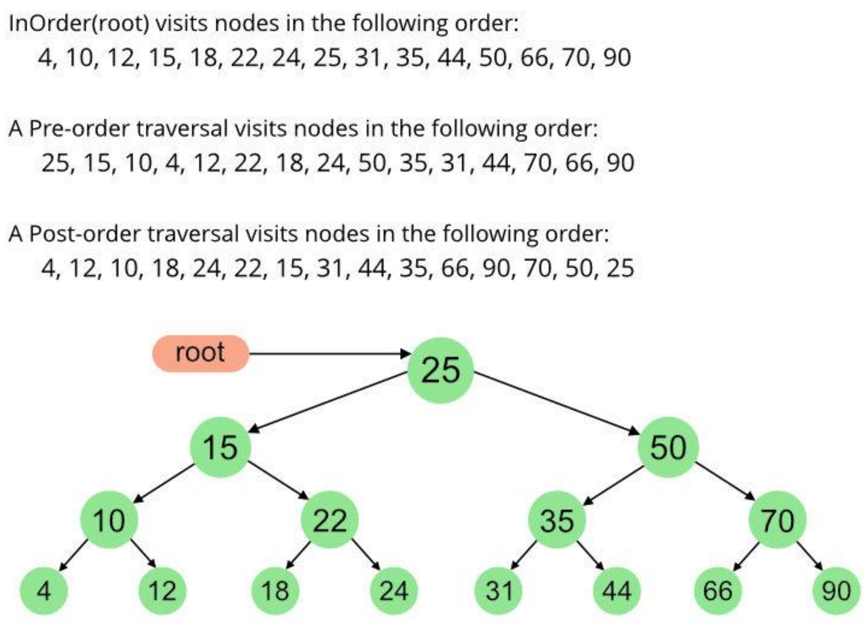


Figure 2: Illustration of a binary search tree

If we compare a BTS to a doubly linked list, it has the advantage of faster traversing, since you don't have to look through every element, but only to the right or left of each node, depending on if the element you are looking for is bigger or smaller than the current node. Adding elements to a tree is also faster, since there are far more places an element can be stored, and for the same reason that it is faster to traverse through. Removing elements can be an issue but are possible and are slower than a doubly linked list. I have not made this possible in my ADT, so there is no need for explaining. The BTS will also take more space in the memory, since there are three pointers for each node, that is its left and right "child" and "parent" which will point to the node above it. Compared to the linked list that only have a next and previous pointer. In some cases there is possible for a tree to have a worst-case scenario where it evolves to a doubly linked list. Where each element added to the tree is greater than the previous. This will result to the tree being equally slow as the linked list.

There are also three different methods for traversing a binary search tree, that is In-order, Pre-order and Post-order. They all follow different "rules" for traversing and will create different results. Below is an illustration.


InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

Figure 3: Illustration of different tree traversal results

These different methods are all used in different situations. In-order traversal visits nodes in ascending order and is used for returning a sorted list of data for a tree. Pre-order traversal visits the root first, then the left and right subtree, and is used for returning an exact copy of a tree. Post-order traversal visits the leaves of the left subtree, then the right subtree, and finally the root. This is used for destroying a tree, freeing it form the memory.

3 Design and Implementation

There are three parts of the implementation that needs explaining, that is the creation of the two different set ADT's, and the spam filter that uses the operations supported by the ADT's. Also, the spam filter only supports sorted sets, this is important to bear in mind. The different operations for the set ADT are:

- Create new set
- Destroy set (freeing it from the memory)
- Get the size of a set
- Add new element to a set
- Checking if an element is in a set
- Iteration over a set
- Union of two sets
- Intersection of two sets
- Difference between two sets
- Get a copy of a set

3.1 Doubly linked list

The assignment gave us pre-code for a fully working doubly linked list, it was up to us students to understand this, and put the code good to use. When understood, it was as simple as to use the different operations the came with the linked list to create a new ADT, which the only difference was that it could not have any repeats and it should be sorted, therefor we end up with a sorted set ADT. To achieve this we check if an element is already in the set before adding it, then sorting it.

3.2 Binary search tree

As mentioned briefly in the technical background, every node in a binary search tree (BTS) consists of a pointer to its left and right child, and a pointer to its parent. The first element in the tree is called the "root", and if you wish to access the tree, you always need to start at the root. If we wish to add elements to a tree, we compare the element we are adding to the root, if it is smaller, we add it to the left subtree, and if it is bigger, we add it to the right subtree. Do this method until you reach the end of the tree, also called the leaf, and add it to the left or right. Using this logic, in best case scenarios we will end up with a balanced tree, where the tree is completely sorted, or in worst case scenarios where we end up with a doubly linked list.

To make the binary search tree support the same operations as the doubly linked list, it was necessary to use the different tree traversal methods. As explained in the technical background, they each had their own use. For destroying the tree and freeing it from the memory, post-order traversal was used. The algorithm used follow three different rules inside of a loop.

- If possible, move to the left child, and jump to the start of the loop
- If possible, move to the right child, and jump to the start of the loop
- Free the leaf node, go to parent of leaf, and set the value of the leaf to NULL

These next two algorithms for tree traversal were considerable much harder to create, since they should support iteration over the tree and return one after one element at a time, without destroying anything. This meant that the algorithm must return the correct element, then stop until it is called again.

To make a copy of a tree, pre-order traversal was necessary. It follows five different rules, where some rules are inside a loop.

- If possible, move to the left child, and return the previous element
- If possible, move to the right child, and return the previous element
- Start a loop, that will "climb out" of a subtree
  - Save the position of current node (will be the previous node)
  - Move to the parent of current node

    - If current node is equal to NULL, return the previous element
    - If possible, move to the right, and return the previous element

For returning the sorted values of the tree, in-order traversal was used. This was the most difficult algorithm to create and may not seem that obvious. It follows five different rules, depending on how you count, which are all inside a loop.

- If possible, move to the left child, and jump to the start of the loop
- If right child of the current node, is equal to the previous node, move to parent of current node, and jump to the start of the loop
- If left child of current node is equal to the previous node, or left child of current child is equal to NULL
    - If possible, move to the right
    - If not possible
        - If current node is the rightmost node, return the previous element
        - Move to parent of current node
    - Return the previous element

3.3 Spam filter

We got some pre-code that converted strings into tokens, so we could add the different mails to sets without issues. To make the spam filter it was necessary to complete this math equation:

$$M \cap ((S1 \cap S2 \cap \ldots \cap Sn) - (N1 \cup N2 \cup \ldots \cup Nm)) \neq \varnothing$$

If we put this into words, you need to find the intersection of all the spam mail (red), and the union of all the non-spam mail (green), take the difference between these two sets, and then use this set to find the intersection between the mail we are checking if are spam or not (orange). If the intersection is NOT empty, the mail is classified as spam.

To complete the math equation, an approach was to add each of every spam, non-spam and normal mail to a double pointer. Using this method, we get three different double pointers that holds all the spam mail, another that holds all the non-spam mail, and another that holds all the normal mail. We can then iterate through the spam double pointer and take the intersection of every set inside it and create a new set for this result. We do the same with the non-spam double pointer but take the union of every set and create a new set for this result. Finally, we take the difference between these two results, and add that to a new set. Effectively we now have one double pointer that holds every normal mail, and a set that holds the difference between spam and non-spam. The last and final step is to iterate through the double pointer and take the intersection between each normal mail and the set that holds the difference between spam and non-spam, to confirm if the mail is spam or not.

4 Discussion

When creating the algorithm for the different tree traversal for the binary search tree, I choose to do so with loops and if-statements, but it is possible and better to do so with the use of a recursive function. But I did not realize how to make it return one and one element at the time, and I also didn't realize before late in the assignment that I could simply create a new function inside of the source file, I thought that I was limited to the functions that was in the header file. And there are probably many different versions of these algorithm that have better worst- and best-case scenario than what I have created. I know of one flaw in the pre-order algorithm. To return the last element in the tree, it must iterate through the entire right side of the tree, until it reaches the root and then goes to the parent of the root, till it finally reaches NULL. A solution is to make it return the last element when it reaches the rightmost node, the same way I have done it in in-order traversal, but I did not figure out how.

I have not compared the performance between the two different ADT's, because I did not have time

5 Conclusion

To sum up, the point of this assignment was to create two different set ADT's and compare the performance between them, something I did not do. The two different ADT's created was a doubly linked list, and a binary search tree. Then the secondary part of the assignment was to use the different operations supported by the ADT's to create a spam filter that checks if mail is spam or not.

Acknowledgement

I worked together with Morten Jansen at some parts in the doubly linked list ADT, and spam filter.

References

[1] Figure 1: https://learnersbucket.com/tutorials/data-structures/doubly-linked-list-implementation-in-javascript/

[2] Figure 2: https://en.wikipedia.org/wiki/Binary_search_tree#/media/File:Binary_search_tree.svg

[3] Figure 3: https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/