Student number: M00719709

LIBRARY MANAGEMENT SYSTEM: PROJECT REPORT

Introduction

I will be implementing a binary search tree for, get, addition and removal of the books. This data structure is selected to satisfy the library. The library requires a high-speed get, remove, and add time. A binary search tree has an average time complexity of $\Theta(logn)$ for all three operations which is sufficiently fast. The program I have designed will load all the books from the text file into the data structure before the user interface appears. Then the user can interact with the system. The user may choose to do the following: display all books, search, partial search, add, remove or remove a single damaged or lost book.

Throughout the report I will be exploring the different features the console-based system offers. Firstly, I will describe the program's flow, including a UML use-case diagram. Secondly, I explore the advantages and disadvantages of the data structure. In addition to this, I will be using pseudocode to demonstrate an analysis of the main algorithms I have used. Thirdly, there are the test cases grouped in a test table. Lastly, I will conclude with the advantages and limitations of my selected data structure followed by a future approach to this scenario.

The report will include:

- Project Design
    - JUSTIFICATION OF SELECTED DATA STRUCTURE AND ALGORITHMS
        - Pseudocode of the display, get and remove functions.
        - Algorithm analysis.
- TESTING (test cases) of functions, methods (samples and table of test cases).
- Conclusion
    - Advantages
    - Limitations
    - Future approach

PROJECT DESIGN

JUSTIFICATION OF SELECTED DATA STRUCTURE AND ALGORITHMS.

The binary search tree is likely to be randomly built. This is because the text file's books are not sorted by title. The height of the tree is more likely to be balanced. Therefore, there is a higher chance that the time for the operations (insert, get and remove) is $O(logn)$ (CORMEN, T. H. 2009). This means even in the worst case, the time taken for the operations is swift even for large amounts of books.

The binary search tree is efficient and has a get, remove, and insert time of $\Theta(logn)$. This is very fast and is second to a hash table's time complexity of $\Theta(1)$ (CORMEN, T. H. (2009).

Dynamic Data Structure:

Binary search tree's size grows as the number of nodes grow. For every new book added, there is a new instance of the 'Node' class added to the tree. Therefore, the size of the tree can grow or shrink at runtime.

Insertion and removal of data

The program will update the address present in the pointer (pointer to left or right node) of a node.

Memory Efficiency:

A binary search tree is less memory efficient than a single linked list or a stack or queue. Each node will have a key (book title), value (Book object) and a pointer to the next left node and right node. Whereas a linked list for instance, only has one value and one pointer to the next node. However, the size of the binary search tree grows as the number of books grows which means there is no memory being wasted. Therefore, a binary search tree is still efficient, even though a bit of extra memory is used for speed.

In addition to this, books are sorted by title as they are added or removed. There is no additional algorithm and memory required for sorting.

Below is the pseudo code and analysis of the algorithms I will be implementing (Barry D. Nichols, no date)**.**

Algorithm 1 Pseudocode for displaying all books

```
DISPLAY_A_Z(node)
If node != nullptr then
//call functions until nullptr. In sorted order
    display_A_Z(node→left)
    node → value.toString();
    display_A_Z(node→right)
end if
```

Algorithm 1: Analysis

Display function

$T(n) = 2T\left(\frac{n}{2}\right) + 1$

In this algorithm, worst case and best cases are the same, because it recursively traverses all nodes in the tree.

For the master method, $a = 2$, $b = 2$, $f(n) = 1$
Using the case $T(n) = \Theta(n^{\log b\, a}) = \Theta(n)$.
Time complexity is $\Theta(n)$

Algorithm 2: Pseudocode for: get and insert

```
Book get(root, key)
    if node == root then //found
        return root→value
    else if key < root→key
        if !root→left //insert
            root→left = new Node(key, Book())
            return root→left→value
        else //perform function again
```

```
            return get(root→left, key)
        end if
    else //The key is more than root
        if !root→right //insert
            root→right = new Node(key, Book())
            return root→right→value
        else
            return get(root→right, key)
        end if
    end if
```

Algorithm 2: Analysis

get and insert function.

In this algorithm, $T(n) = T\left(\frac{n}{2}\right) + 2$
Time complexity is $\Theta(\log n)$
In this algorithm, when every node has only a left child or every node has only a right child, it would be like a linked list and would take the time $O(n)$ to traverse.

And the best case is $O(1)$. If the root (first node) has the correct key. So only one compare is required to be finished.

For the master method, $a = 1$, $b = 2$, $f(n) = 2$
So, using the case $T(n) = \Theta\left(n^{logb(a)} * logn\right) = \Theta(\log n)$
Time complexity is Θ(log n)

Algorithm 3: Pseudocode for: remove

```
Book remove(node, key)
//book/node doesn't exist
    If !node then
        return nullptr
    //search
    if key < node→key then
        node→left = remove(node→left, key)
    else if key > node→key then
        node→right = remove(node→right, key)
    //found node to delete?
```

```
        else
        //Node to delete has no right
            if !node→right then
                Node temp = node→left
                delete node
                return temp
            end if
            //Node to delete has no left
            if !node→left then
                Node temp = node→right
                delete node
                return temp
            end if
        //node to delete has two children
        Node temp = node
        node = new Node( min(temp→right))
        node→right = remove_min(temp→right)
        node→left = temp→left
        delete temp
        end if
    return node
```

Algorithm 3: Analysis

Remove Node function

In this algorithm, $T(n) = T\left(\frac{n}{2}\right) + C$

Here C is the time to determine its candidate node instead of its deletion node.

Remove function's time complexity is $\Theta(\log n)$. Also, the worst case could be $O(n)$. As the binary search tree is not a self-balancing tree (e.g. 2-3, red-black, AVL). There is a possibility that binary search tree could have only left or only right nodes and take a time of $O(n)$ to traverse through the tree and remove a book.

The best case can be $O(1)$. If the root has only one book or if there are no books at all, then the function takes only one time.

For the master method, $a = 1, b = 2, f(n) = C$
So using the case $T(n) = \Theta(n^{logb(a)} \log n) = \Theta(\log n)$
Time complexity is Θ(logn)

In the remove function, the remove_min and min function do not change the overall time complexity of the algorithm.

remove_min function

This algorithm finds the deepest left node and deletes its pointer. Releasing its memory back into the free store. It takes about the time of its height. Hence, the time is $O(h) = O(logn)$

The worst case is about $O(n)$ because all nodes could have only a left child or all nodes could have only a right child.
The time is $T(n) = T\left(\frac{n}{2}\right) + 1$
So using the case $T(n) = \Theta(n^{logb(a)} \log n) = \Theta(\log n)$
Time complexity is $\Theta(logn)$

min function

This algorithm finds the min node.
The recurrence is $T(n) = T\left(\frac{n}{2}\right) + 1$
So using the case $T(n) = \Theta(n^{logb(a)} \log n) = \Theta(\log n)$
Time complexity is $\Theta(logn)$

The worst case and the best case of min are the same as remove_min.

Algorithm 5 Pseudocode for partial search of a book

```
PartialSearch(node, userInput, firstLetterCapital)

count = 0;

if node != nullptr then

//call functions until nullptr. In sorted order

    PartialSearch(node→left, userInput, firstLetterCapital)

    //userInput has a pos (position) in node’s key (AKA book title)

    if node→key.find(userInput) != pos || node→key.find(firstLetterCapital) != pos

        node → value.toString();

    end if

    PartialSearch(node→right, userInput, firstLetterCapital)

end if
```

Algorithm 5: Analysis

Partial search function

$T(n) = 2T\left(\frac{n}{2}\right) + 1$

In this algorithm, worst case and best cases are the same, because it recursively traverses all nodes in the tree. f(n) can be 1 + c (c being the nested if statement being true, so c = 2 in this case) so worst case could be f(n) = 3. However, this would not have an effect on the overall time complexity.

For the master method, $a = 2$, $b = 2$, $f(n) = 1$
Using the case $T(n) = \Theta(n^{\log b\, a}) = \Theta(n)$.
Time complexity is $\Theta(n)$

Algorithm 6 Pseudocode for removal of damaged book

```
RemoveDamagedBook()

title

Print "Enter book title to remove"

Input.ignore(newlines) // avoid input skip possibility

Input title

Book foundBook = bookTree[title]; // get book

if foundBook.getTitle() == "" then   //book doesn't exist

    Print "This book does not exist to begin with!"

else

    if foundBook.getQty() == 0 then

        Print "This book is out of stock"

    else

    FoundBook.damagedBook();

    Print "Successfully removed one of " + title + " from stock. There is now " + foundBook.getQty() + " in stock left."

    //-1 quantity

    BookTree[title] = foundBook

    end if

end if
```

Algorithm 6: Analysis

Removal of damaged book function

In this algorithm, $T(n) = T\left(\frac{n}{2}\right) + \Theta(\log n)$ if a book is not found.
For the master method, $a = 1$, $b = 2$, $f(n) = \Theta(\log n)$
So, using the case $T(n) = \Theta\left(n^{logb(a)} * logn\right) = \Theta(\log n)$
Time complexity is Θ(log n)

$T(n) = T\left(\frac{n}{2}\right) + \Theta \log^2 n$ if a book is found.

For the master method, $a = 1,\ b = 2,\ f(n) = \Theta \log^2 n$

So, using the case $T(n) = \Theta\left(n^{logb(a)} * logn\right) = \Theta(\log n)$

Time complexity is Θ(log n)

TEST CASE TABLE

Test Case #	Test Case Description	Test Data >	Expected Result	Actual Result	Pass/Fail
1	Check response when user enters a book title to direct search	>”” >” “	No book to be loaded.	No book was loaded.	Pass
2	Check if ISBN inputs would work when adding a new book to data structure	> “978032114653” > “978020163361” >“9780070446878” > “Δ^~µ¬øΩπ≈ø ∫” > “Θ”	ISBN numbers that are not valid are not added to the system. Only ‘‘9780070446878’ should be accepted.	Only ‘9780070446878’ was the valid ISBN number as expected.	Pass
3	Check if input for the quantity of a book works if non-integer or non-digit input is entered.	> “¯˘ø ∫¬≤∂” > “ “ > “” > “978032”	Only digit inputs to be accepted.	Only digit inputs worked	Pass
4	Check if new book’s title’s first letter is a capital letter.	> "ldeo" > "Tebana" >”” > "¯˘ø ∫¬≤∂" > "Θ"	Title with lower case is false. Only allowed Title with upper-case letter or special character first letter.	Test assertions with first letter capital letter or special characters were allowed.	Pass

CONCLUSION

Advantages

Although the binary search tree takes up more memory when compared to other data types such as an array, a binary search tree offers faster search speeds. Functionality such as sorting algorithms for an array are generally slower when compared to a binary search tree. One of the fastest and most efficient sorting algorithms being quicksort, best case $O(nlogn)$), is an example of a sorting algorithm for an array. Quicksort takes a longer time than a binary search tree’s sorting method. When given a larger amount of data to sort, a binary search tree is significantly faster at sorting. I chose to use a binary search tree because the sorting happens naturally when books are added or removed, at a time complexity of $\Theta(logn)$. This naturally sorting method is even faster than a merge sort’s time complexity of $\Theta(nlogn)$. There is also no extra dynamic data structure or algorithm required for sorting to take place. Being a sorted list of books is more user-friendly as it provides an easier navigation through the list of books after choosing to view or search.

Limitations

Another option for the data structure I could have used was a hash table. A hash table has a faster get, add and remove time complexity of $\Theta(1)$. However, a binary search tree takes up less memory space and is therefore more efficient. Furthermore, A binary search tree is also sorted, whereas a hash table is not, which is an advantage for a binary search tree.

Future approach

In the future, a self-balancing tree such as red-black, 2-3, AVL tree would be a better data structure to use. A self-balancing tree is a balanced binary search tree. When insertion or deletion of a node is done, the tree may rotate or shift nodes. This ensures the height of the tree is as short as possible. For all three functions get, add and remove, self-balancing trees have a worst case of $O(logn)$. Whereas a binary search tree has a worst case of $O(n)$, which is slower. The given text file's books are in random order, so the worst case of the binary search tree tilts towards $O(logn)$ (CORMEN, T. H, 2009). Be that as it may, there is a chance the library may use a sorted list of books in the text file, which would mean a time complexity of $\Theta(n)$ for the binary search tree.

Conclusion

To conclude, the program matches the task requested and offers the desired software product. The software passed the test cases and is ready to be used to manage the library. As proven by the data structure justification and the algorithm analysis I have written, the library management system will now provide a satisfactory search, add and remove time complexity of $\Theta\ (logn)$.

REFERENCE LIST

CORMEN, T. H., & CORMEN, T. H. (2009). Introduction to algorithms. Cambridge, Mass, MIT Press.

Dr Barry D. Nichols (no date) Binary Search Trees [PDF]. Available at: https://mdx.mrooms.net/pluginfile.php/2528433/mod_resource/content/0/binary_search_trees.pdf/ (Accessed: 1 April 2021).