Advanced Software Engineering Assignment 1

N0786072

Stephen Anderson

2021

Contents

[Task 1 – Complexity Analysis of Binary Search Tree 2](#_Toc65768927)

[Destructor 2](#_Toc65768928)

[Copy Constructor Taking BST as Argument 2](#_Toc65768929)

[Copy Assignment Taking BST as Argument 2](#_Toc65768930)

[Move Constructor Taking BST as Argument 2](#_Toc65768931)

[Move Assignment Taking BST as Argument 2](#_Toc65768932)

[Lookup Function Taking a Key as Argument 2](#_Toc65768933)

[Insert Function Taking Key and Value as Arguments 2](#_Toc65768934)

[Remove Function Taking Key as Argument 2](#_Toc65768935)

[Display Entries in Key Order 3](#_Toc65768936)

[Display Tree as ASCII Graph 3](#_Toc65768937)

[Task 2 – Justifying Implementation Choices 3](#_Toc65768938)

[Task 2a – Performance Guarantees 3](#_Toc65768939)

[std::list 3](#_Toc65768940)

[std::map 3](#_Toc65768941)

[std::unordered\_map 4](#_Toc65768942)

[Task 2b – Analysing Royal Software Engineer’s Algorithm 4](#_Toc65768943)

[Analysis 4](#_Toc65768944)

[Combinations 5](#_Toc65768945)

# Task 1 – Complexity Analysis of Binary Search Tree

## Destructor

Performs the deepDelete() function which iterates through every node and performs the C++ generated delete(pointer) function which means that it has a worst case and average time complexity of O(n). The best case is where there is only the root node to delete which would give a complexity of O(1).

## Copy Constructor Taking BST as Argument

Performs the deepCopy() function which must iterate through every node and its children so it always has O(n) time complexity. The best case scenario would be where there is just the root node which would give a time complexity of O(1).

## Copy Assignment Taking BST as Argument

Clears current BST structure with deepDelete() if appropriate then uses deepCopy() to return a second data structure that is equivalent in value but is not a pointer to the original tree. The deepDelete() function has a worst case time complexity of O(n). If both deepDelete() and deepCopy() are performed it would give a time complexity of O(2n) which is equivalent to O(n), so both the worst case and average complexity is O(n) and the best case is O(1).

## Move Constructor Taking BST as Argument

Copies data from original tree’s root to the new tree’s root then makes the original a null pointer. This function always changes two pointers and nothing else which gives it a constant complexity therefore O(1).

## Move Assignment Taking BST as Argument

Copies value of root to a new tree’s root then deletes the original root if appropriate. This always changes one pointer, and sometimes changes two. The worst case and average scenarios would be O(2) which is equivalent to O(1). The best case scenario does not take the longer route through the if statement which gives it complexity O(1). That means that it has a constant complexity of O(1).

## Lookup Function Taking a Key as Argument

Performs a recursive search function. It has an average time complexity of O(log2(n)) assuming that the tree is balanced. The worst case is where the tree is entirely unbalanced down one side with the desired node is at the end and so effectively acts as a linked list which has a complexity of O(n). The best case is where the root is the desired node which has complexity O(1).

## Insert Function Taking Key and Value as Arguments

Performs recursive insertion function. Has the same time complexities as the lookup as it is essentially performing the same function but inserting a new item instead of returning a search result. The only difference is that the best case is where the tree is empty but it still has the same best case complexity of O(1).

## Remove Function Taking Key as Argument

Performs recursive removal function or sets root to be null. The recursive function has the same worst case and average complexities as the lookup and insert functions (again assuming a balanced tree) of O(n) and O(log2(n)) respectively. The best case of the wrapper function is where the root is already empty and so the tree is also empty. However that is not really appropriate for determining time complexity in this scenario so a better best case scenario is where the root has no children and has the correct key; the root is made to be null, giving a time complexity of O(1).

## Display Entries in Key Order

Performs in-order tree traversal where it visits every node which means that it has a time complexity of O(n) with a best case being where n = 1 giving a complexity of O(1).

## Display Tree as ASCII Graph

Performs pre-order tree traversal visiting every node, so it has the same complexities as the in-order tree traversal.

# Task 2 – Justifying Implementation Choices

## Task 2a – Performance Guarantees

### std::list

The C++ standard library implements this as a doubly-linked list. This means that it stores elements in non-contiguous memory but stores the location of the first and the last elements in the list.

Due to the way the insert() method is implemented, where it requires an iterator to the desired position, it allows for the method to have a constant time insertion. The container also allows for easy insertion to the front or the back of the list using push\_front() and push\_back() where the container can just adjust the pointers to the start or end elements.

In a similar way to insert(), erase() is constant time but requires an iterator to locate where the desired element to remove is stored in memory. But because the container is a doubly linked list, by accessing the desired element the locations in memory of the elements either side in the list are known and can have pointers adjusted accordingly.

The main drawback to using a list is that whilst the methods themselves are constant time, the std::find() method is usually required to return an iterator which locates an element in memory. It is possible to create a method that performs a similar function if desired. However, because the container is a doubly-linked list, finding an element in memory to return an iterator actually has a worst case time complexity of O(n). This means that using insert() or erase() with a given position that isn’t the first or last element actually has an amortised average complexity of O(n).

### std::map

The C++ standard library implements the map container as a red-black tree with each element containing an std::pair of a key and a value. As it is a red-black tree it can self-balance which leads to significantly better time complexities compared to unbalanced trees.

For example, the worst case time complexity when searching an unbalanced binary tree is O(n) where it is so grossly unbalanced that it effectively acts as a singly-linked list. This is impossible to occur in a self-balancing tree as it can never become that grossly unbalanced which leads to a worst case time complexity of about O(log2(n)).

This means that using the in-built find(key) method is fairly efficient as the tree is always close to balanced.

The erase() and insert() methods can be a little more complex than just a simple search with some adjustments of pointers as the tree may need to be rebalanced when changing the elements. This is done by recolouring and rotating elements in the tree. Recolouring occurs when inserting a new element (always coloured red) into a position where its parent is also red.

There are a couple of scenarios that can play out if recolouring is required.

If the parent of the newly adjusted element has a sibling (the element at the same level in the tree with a shared parent) that is also red, then elements are swapped from being red to black where possible (the root element always remains black) so that every descending path through the tree has an equal number of black elements. This scenario is the worst-case as it needs to go through up to half of the tree to recolour elements leading to an average complexity O(log2(n)).

The other scenario that can occur is where a rotation is required to make it so the newly adjusted node becomes a sibling of the parent it was adjusted under and the two red elements are on an equal level instead of being directly related to each other. This scenario is the best-case in terms of complexity as whilst it may have some seemingly complex restructuring it always changes the same number of elements leading to a complexity of O(1).

These scenarios mean that the average and worst case time complexity of insert() and erase() are the same, being O(log2(n)). This is because although the scenarios that can occur have differing time complexities it is always required to navigate through the tree beforehand which has a complexity of O(log2(n)). At worst, this would be doubled due to recolouring but that still scales in an equivalent manner.

### std::unordered\_map

The C++ standard library stores unordered maps in what is effectively an unsorted hashmap. Elements are stored internally in buckets which are themselves singly-linked lists.

Because it is a hashmap, bucket locations are stored in an array-like structure that can be accessed using a default or user-created hash function which calculates the index. Unfortunately, collisions can occur if the hash function is not well designed for the data and when inserting new data, the same index is calculated many times for different keys. The probability of this occurring increases as the number of buckets (and the size of the hash table) increases.

If collisions do occur, then elements can be added to the same bucket and this is where the elements are chained together. Of course, if this happens a lot particularly to one bucket then the hashmap’s load factor can become quite large and as the load factor increases and decreases the number of available buckets also increases and decreases to make the hash table more efficient. Unfortunately, this process is not efficient as it requires going through every item and rehashing and finding a new place to insert the data, meaning it has time complexity O(n). Luckily this does not occur very frequently with a good hashing function.

Because of the various factors involved, the average complexity of finding an element – so as long as the load factor is low – is pretty much O(1). However, if the load factor is high and the hash function is unsuitable then this could potentially devolve to O(n) if all the elements are chained into one bucket.

This is also somewhat true for insertion and removal. As long as the load factor and hash function are acceptable then the time complexities are on average constant. But if the load factor is too high after insertion or too low after removal then the costly rehashing occurs, with the worst case time complexity to match.

## Task 2b – Analysing Royal Software Engineer’s Algorithm

### Analysis

The algorithm can be broken down into requiring two main data structures.

The first one is a data structure that can hold the randomly inserted bricks in such a way that can be searched efficiently. For this either a map or an unordered map are likely to be most appropriate, each with benefits and drawbacks.

Using a map would allow for quick searching and reasonably quick deletion of bricks once they’ve been found. However initial creation is slow for many elements.

An unordered map with an appropriate hash function would be a good choice as searching is fast and initial creation could be optimised to be more efficient than a normal map. However, deletion of found bricks would be slower but not always necessary as accessing elements is fast.

The list is least appropriate for this aspect of the task due to needing to perform a linear search with the possibility for each required item always being at the end of the list. The only advantages are that initial creation is fast and that as each brick is found, it can be quickly and easily removed from the list which improves subsequent searches.

The second data structure needs to have the ability to insert sorted data both at the front and back of the data structure. It also needs to maintain the sorted data, thus meaning that a list is probably the best data structure for this implementation. It can have data pushed both front and back whilst maintaining a user-defined order instead of a system-defined order such as a map or a semi-random order such as an unordered map.

### Combinations

#### Preferred Combination

I feel that an unordered map to store the raw data and a list to store the sorted data is likely the best option. This is because the unordered map has very fast random access and the number of buckets can be reserved (using the reserve() or rehash() methods) – the number of total bricks is known – which should allow for more efficient searching and initial insertion as the size of the hash table can be set.

After initial creation, once the size has been set, insertion and searching should both have a constant time efficiency – O(1).

I think that a list is the best of the options for the second data structure because it has constant time insertion at the front and back which is exactly what is required by the algorithm.

#### Alternate Combination

I still feel that for the second data structure a list is the best option for the same reasons as before.

However, the first data structure could instead be a map as opposed to an unordered map as it would offer reasonably quick searching after an initially slow insertion, an average of O(log2(n)). However, the insertion process could result in needing to rebalance and recolour the tree multiple times which has a complexity of O(log2(n)) as well. This is worse than the unordered map and the list but could still be viable.