2025-1 Spring Semester

Kangwon National University

Data Structure Project

이름: 누르 다야나 아이니 빈티 모하마드 루슬리

학번: 202411965

과목: 4471010 자료구조

Abstract

This report compares two abstract data types (ADTs): Binary Search Tree (BST) and Priority Queue Min Heap in solving the symmetric difference problem between two sets. The reference solution uses BST which have average case efficiency but can have possibilities to be skewed and not bushy and have worst case efficiency. Also, set will have duplicate data and BST handling duplicate priorities awkwardly. The proposed methodology uses Min Heap and Heap Sort to sort both sets and uses two pointers to compare the unique elements. This method guarantees a stable time complexity and simpler logic as well as more efficient space memory usage.

Contents

Chapter 1: Introduction

Selected Data Structure and Description Summary

Priority queue min heap, has ability to efficiently retrieve the smallest element. A min heap is a complete binary tree where the smallest value is always at the root. This allows for constant-time access to the minimum element (O (1)) and efficient insertion and deletion operations in logarithmic time (O (log n)). It is especially useful in applications like sorting, duplicate removal, and set operations (symmetric difference), offering both time and space efficiency.

Importance of Selected Data Structure

A Min Heap is a data structure based on a complete binary tree, where the smallest value is always located at the root node. It guarantees O (log n) time complexity for insertion and deletion operations. This structure is frequently used to implement a priority queue and plays a central role in various algorithms that require repeated selection of minimum values. In practice, min heaps are widely used in operating system task scheduling, network packet processing, and real-time event priority management, making them a fast and efficient tool for handling data in many fields.

Problem Description

Given two sets of numbers, A and B, *the problem is to find the size of the symmetric difference*. The number of elements that are in A but not in B (A - B) and in B but not in A (B - A). The input includes the number of elements and elements of each set, and the output is the total number of elements in the symmetric difference. For example, if A = {1, 2, 4} and B = {2, 3, 4, 5, 6}, then A - B = {1} and B - A = {3, 5, 6}, making the symmetric difference size 4.

Reference Solution Summary

The reference solution uses a Binary Search Tree (BST) to compute the size of the symmetric difference between sets A and B. First, it inserts the elements of A into the BST, and then, while reading elements of B, it checks if each element exists in A to count common elements. It then calculates the number of elements unique to each set and adds them.

Idea of Purpose Methodology

The reference solution uses a BST to store elements of A and compare with B, which has an average time complexity of O (log n) for insertion and search, but can degrade to O(n) in the worst case if the tree becomes unbalanced. This performance instability is a limitation. Therefore, this project proposes using a min heap, which guarantees O (log n) time complexity for both insertion and deletion, ensuring more stable and consistent performance when calculating the symmetric difference and minimize the space used. Plus, heaps handle duplicate priorities much more naturally than BST.

Chapter 2: Reference Solution Analysis

Detailed Description of Selected Data Structure

The ADT used on the problem was Binary Search Tree (BST).

Binary Search Tree is a set of nodes and a set of edges that connect those nodes. BST have constraint where there is only exactly one path between any two nodes. The one node is called the root. Every node except the root has exactly one parent. A node that has no child is called a leaf. Every node can either has 0,1 or 2 children only.

The property of BST is that every key in the left subtree is less than root key and every key in the right subtree is greater than root key. And it satisfied the transitive order. Also, no duplicate keys are allowed. Which means there are must be no same data in one tree.

This structure allows basic operations such as search, insert, and delete to be performed with average time complexity of O(log n). However, if the tree becomes skewed (not bushy), the time complexity can degrade to O(n).

Reference Solution Detailed Explanation

In the reference solution, elements of set A are first inserted into a BST. Then, for each element in set B is searched in the BST of set A. If element found in set A BST it will be counted as common element and get ignored. If element is not found in set A BST it will insert the element to the second BST which is set B BST to track the unique element of set B.

Finally, the size of symmetric difference is calculated by subtracting the number of common elements from the size of Set A and then add the number of unique elements from the Set B. This way it can filters out common elements and only combines the unique elements from both set to get symmetric difference.

There are few functions operation that are must know to understand the reference solution which are:

- **BSTNode* CreateNode**
  To create node in BST, first we need to do memory allocation by using malloc function. Then we must initialize data that are in the node such as key, left and right. After that the created node is ready for insertion.
- **BSTNode* Insert**
  To insert key in to the BST, first we must make sure that the root is not NULL and if it is happen to be NULL we can call the function CreateNode to create new node. Then, to satisfy the BST property we need to check either key is greater or smaller than the root. If the key is smaller than the root, key will be inserted at the left subtree but if

the key is greater than the root, key will be inserted at the right subtree. Lastly return the root.

- **Int Search**
  Now that all key data is inserted at the tree, we can search any key by using Search function. The concept of search is basically same like inserted where we must check if the key is smaller or greater then start search based on right or left subtree. We can use recursion calls to keep going down the tree until found the key.
- **Int CountNodes**
  In this function if root is not NULL it will recursively count the number of nodes in left and right subtrees and adds 1 for the current node. Thus, it will return the total number of nodes in the whole BST including the root.
- **Void FreeBST**
  This function is simply to free the memory after all works are done.

In main function, first read the size of set A and B. Next, read and insert the element of set A in to BST. Then for every input of element of set B, it will do search such that, if the element is already in set A increment will happen in both sets. Else, it will be inserted into set B BST which is another BST. After that, it will count the total nodes that are in A but not in B and vice versa. And lastly print symmetric difference result and free all memory allocated.

Based on this reference solution we can see that the time and space complexity based on each function are:

Function	Average Time Complexity	Worst-case Time Complexity	Space Complexity
Insert	O(log n)	O(n) -if tree is skewed	O(n)
Search	O(log n)	O(n) -if tree is skewed	
CountNodes	O(n)	O(n) -visit every node one time	

Limitation of the Reference Solution

Based on this reference solution there are a few limited we can see here. The first is if the tree is not bushy and it became skewed it will fall into worst case where time complexity can be O(n) and not efficient. Second, to handle the duplicate data it uses two BST which is to store nodes A and nodes B this requires more conditional logic and make the code more complex also more memory space is needed.

Chapter 3: Proposed Methodology

Proposed Methodology Summary

The methodology that I would like to propose is by using the Min Heap. We can sort both sets use Min Heap then applies a two-pointer technique to efficiently calculate the symmetric difference. First sort the set A and set B using heap sort based on Min Heap. Then, compared the element in the sorted array and *extract only the different values* which are part of the symmetric difference.

Overcome the Limitation of Reference Solution

By using this method, it ensures the stability of O (log n) sorting performance and avoid the skewed case that can happen in BST. It performs a linear comparison of 2 array which give the overall time complexity remains O(log n). the conditional logic for this method is much more direct, avoid unnecessary nested loops, and less complicated compare to BST method. Lastly since this method use array-based heaps which improve the efficiency of space management or take less memory.

Proposed Methodology Detailed Description

From the main function, first we will read the size of set a and b (n &m) then read each set data and insert it into the array. Here we use the two indices to walk through both arrays the same time. After insertion now we will sort the array use Heap Sort. Sorting is a must for efficient array comparison. Then we can just use 2 pointers (i, j) to compare both array and count the unique data. If the data are equal then we just going to skip it because they are in both sets. If data in set A but not B symmetric different counter will increase. If data in set B but not in set A counter will increase too. And finally add the remaining elements to get final result of symmetric different.

Thus, in this method there are few functions that we must understand to solve the question:

- **MinHeap* CreateHeap**
  In this function we will do memory allocation for heap then initialize data in the heap and return it.
- **Void Insert**
  For insertion the value will be insert at the bottom line at the end or the last leaf which is at last index of the array. Size increase and to maintain the min heap property it will go through heapifyUp.
- **Void heapifyUp**
  Since data was inserted at the end of the leaf, we need to make sure the value satisfies the min heap property, else if it is not satisfied it will get swap with its above parent. This will loop until the data is bigger than its parent and less than its child.

- **Void swap**
  Swap function is simply to swap the element that needed by using temp variable.
- **Int extractMin**
  Next, to extract the min we can just extract the first index in the array and decrease the size. After data is remove, min heap needs to go through heapify down.
- **Void heapifyDown**
  Heapify down is also compulsory to maintain the min heap property. After deletion happened the new data that replace the old data need to be check to make sure the new data is the smallest value or not. First compare the left side. If the left data is less than root then data will go down to left and left root can be candidate to be new root. Same goes to right side. Until the data reach the index it will swap with possible candidate recursively.
- **Void HeapSort**
  Now that all data are correctly inserted into min heap, we need to sort it. A min heap keeps the smallest element at the top but it does not store elements in sorted order. Only the heap[1] is the smallest. So, it will do insertion first, then to sort the array and after that it will call extractMin function to extract element in order since extractmin remove the smallest element and returns it. Then, the value is stored into arr[i] which is sorted in ascending order.

Chapter 4: Performance Test and Analysis Result

Acmicpc.net Performance Test Result Link

- http://boj.kr/58e1f991c29740878d7658a16b7c6261

Performance Evaluation and Analysis Result

Problem	Original ADT	Time / Space	Proposed ADT	Time / Space
1269: symmetric difference	BST	284ms 13524kb	Min heap	104ms 3468kb

Min Heap takes 180ms less time which is 2.73 times faster than BST and save 10074kb memory space which is 3.9 times less memory.

Min Heap method not only improves time efficiency by nearly 3x, but also drastically reduces memory usage by almost 4x compared to the BST implementation. This demonstrates its clear advantage for this problem.

Chapter 5: Conclusion

Significance and Limitation of Proposed Methodology

In conclusion, this report proposed the methodology of using Min Heap instead of BST to avoids the BST's worst case time complexity issue O(n) caused by imbalanced BST and provides a stable performance of O (log n) through heap sort. This makes it highly significant. Moreover, the logic is more straight forward and simple to implement and at the same time offers excellent memory management. However, for very small or extremely large data this method may perform worse.

Application of the Proposed Methodology in Real Life

This Min Heap-based approach is well-suited for sorting large datasets and computing set differences or symmetric differences. It can be effectively applied in various fields such as real-time event processing, log analysis, network packet classification, and priority-based task queues. Especially in systems where stable performance and low complexity are critical, this method can serve as a more suitable alternative to BST.