Programming Assignment 4 (Dictionaries using Hashing and BST, Nearest Neighbor, and Range Counting)

Department of Computer Science, University of Wisconsin – Whitewater
Data Structure (CS 223)

1 Overview

We are essentially going to:

- **Solve the Dictionary problem using Hashing**, and compare its performance versus a Binary Search Tree.
- **Solve the Nearest Neighbour problem in One Dimension using Binary Search Tree**, and compare its performance versus a brute force strategy.
- **Solve the Range Counting problem in One Dimension using Binary Search Tree**, and compare its performance versus a brute force strategy.

To this end, **your task is to implement the following methods**:

- `search`, `insert`, and `remove` methods in `Hashing.h/Hashing.java`
- `search` and `insert` methods in `BST.h/BST.java`
- `getPredecessor`, `getSuccessor`, `getLCA`, `nearestNeighbour`, and `rangeCount` in `BSTApplications.h/BSTApplications.java`

The project also contains additional files (which you do not need to modify).

Use `TestCorrectness.cpp/TestCorrectness.java` to test your code.

For each part, you will get an output that you can match with the output I have given to verify whether your code is correct, or not. Output is provided separately in the `ExpectedOutput` file. Should you want, you can use `www.diffchecker.com` to tally the output.

You can use `TestTime.cpp/TestTime.java` for the bulk test in each part. It showcases how the choice of a good data structure/algorithm vastly improves your performance. A comparison analysis is not required for this project; hence, you do not need to run this code for grade purposes.

2 Dictionary Problem

Problem: Given a collection of integers, we want to support the following operations:

- Search a number. Returns *true* if number is present in the collection, else returns *false*.

- Insert a number to collection. Returns *true* on successful insertion (number is not already present), else returns *false*.

- Delete a number from the collection. Returns *true* on successful deletion (number is present), else returns *false*.

We have seen two techniques to solve this problem – *binary search trees* and *hashing*. We are going to measure their relative performances in various scenarios.

- **Hashing with Linked List as chain**: This is the same as the version of hashing that has been taught in class, i.e., Hashing with Separate Chaining. **You are going to implement** the *search, insert,* and *remove* functions in: `Hashing.h/Hashing.java`.

- **Binary Search Tree**: **You are going to implement** the *search* and insert functions in: `BST.h/BST.java`. I have provided the code from *remove*; you do not need to write that.

2.1 Hashing Implementation Details

We search/insert/delete in a hashtable in the following way. First use the *getHashValue* method to get the hash value. Now use this hash value to get hold of a hash table entry, which is a linked list. Finally, use appropriate linked list functions as described below.

- **[gethashValue]** Uses the hash function $(37 * val + 61)\%TABLE\_SIZE$. For search, insert, and delete you must use this method; DO NOT change this method.

- **[search]** Complete the `search` method in `Hashing.h/Hashing.java` files.

  First obtain the hash value for the *key* using gethashValue function. Get the linked list from the *hashTable*[ ] for this hash value. Now, use a loop to search this linked list, and return *true* if the linked list contains the *key*, else return false.

- **[insert]** Complete the `insert` method in `Hashing.h/Hashing.java` files.

  Remember that your hash table should contain a number only once. Otherwise, it occupies too much space, and it may also lead to unexpected results. So, before inserting, make sure that the number already does not exist on the linked list.

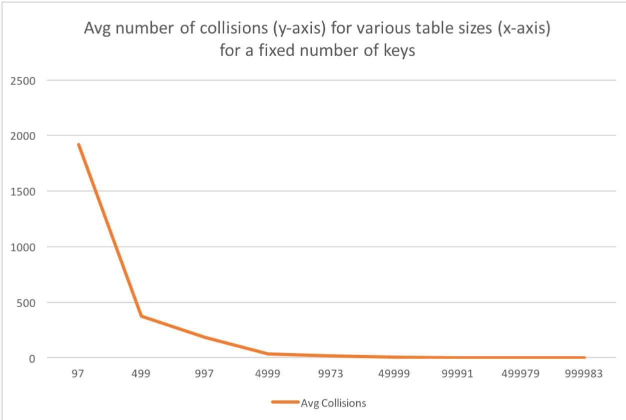
  Therefore, first use *search* to check if the hash table already contains *val*. If it does, then simply return false, else obtain the hash value for the *key* using gethashValue function. Get the linked list from the *hashTable*[ ] for this hash value. Now, use *insertAtEnd* function of the linked list to insert the value. Finally, return *true*.

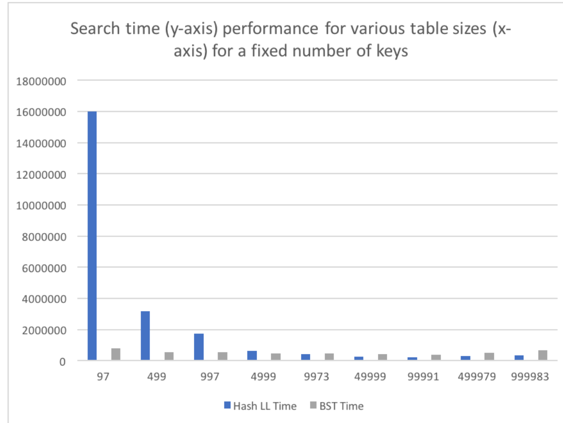
- **[delete]** Complete the `remove` method in `Hashing.h/Hashing.java` files.

  First obtain the hash value for the *key* using gethashValue function. Get the linked list from the *hashTable*[ ] for this hash value.

  Now, we have to remove the occurrence (if any) of *val* in the linked list. If the list is empty, then return $false$. If the value in the head equals *val*, then call *deleteHead* and return true. Otherwise, use a *tmp* variable to traverse the linked list. As long as *tmp*'s next is not null, you check if the value of *tmp*'s next equals *val*. If they are the same, then use $deleteAfter$ on *tmp* and return true, else move *tmp* to the next node. Once the loop terminates, return false.

Figure 1: Search time performances for various table sizes when 1 million values are randomly inserted and then another 1 million values are randomly searched.





2.2 BST Implementation Details

Implement the **search** method in `BST.h`/`BST.java` as follows:

- Assign a temporary variable *tmp* to the root
- while *tmp* is not null, repeat the following:
  - if value of tmp equals key then return *tmp*
  - else if value of tmp $<$ key, move *tmp* to *tmp*'s right child
  - else move *tmp* to *tmp*'s left child
- Return null

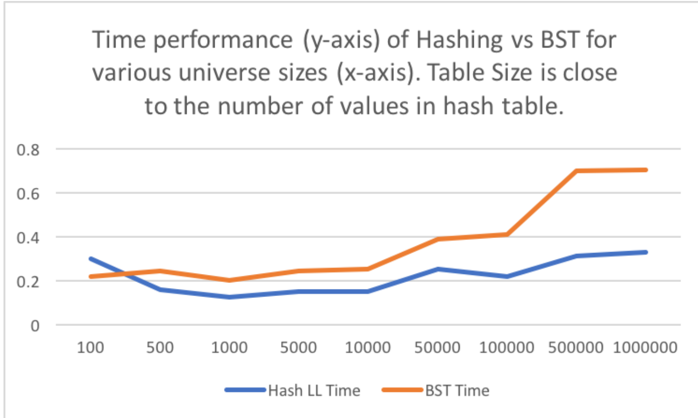
Implement the **insert** method in `BST.h`/`BST.java` as follows:

- If *size* is 0, then allocate memory for the *root*, increment size, and return the root.
- Otherwise, assign a temporary variable *tmp* to the root and another temporary variable *parent* to null
- while *tmp* is not null, repeat the following:
  - if value of tmp equals val then return null (indicating no node was created)
  - else if value of tmp $<$ val, set *parent* to *tmp* and move *tmp* to *tmp*'s right child
  - else set *parent* to *tmp* and move *tmp* to *tmp*'s left child
- Create a new BSTNode, named $newNode$ with value *val*. Assign $newNode's$ parent field to the local variable *parent*.
- If *parent*'s value is larger than *val*, then $newNode$ is the left child of *parent*, else $newNode$ is the right child of *parent*.
- Increment size and return $newNode$.

2.3 Comparative Analysis

Fig. 1 shows the search time performances for the three methods, under various table sizes. We see that as hash table size increases, the number of collisions go down (as one would expect), and Hashing behaves increasingly well. However, at small table sizes, the performance of BST

Figure 2: Comparison of search performances using hashing with linked lists and BST



is significantly better. Fig. 2 shows that if you have a large hash table (close to the size of the universe), then even a simple hashing with linked list will outperform BST almost always. Hence, if you have enough space for a large hash table, use a simple hashing with linked list implementation. In this case, balanced BST will be better (or we have to choose better hash functions).

3 BST Applications

Using the pseudo-codes given for each, your task is to complete the `getPredecessor`, `getSuccessor`, `getLCA`, `nearestNeighbour`, and `rangeCount` in `BSTApplications.h/BSTApplications.java`

The following BST has been used for testing the correctness of your code.

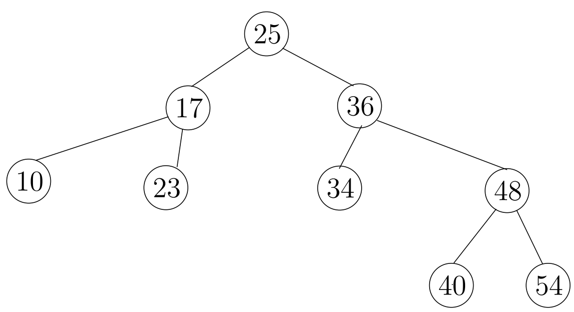


Figure 3: Binary Search Tree Used for Correctness Test

3.1 Predecessor and Successor

Given a set of numbers $\mathcal{N}$, *predecessor*(x) is the highest number y in $\mathcal{N}$ such that $y \leq x$, and *successor*(x) is the smallest number z in $\mathcal{N}$ such that $z \geq x$. Thus, if for the set $\{6, 9, 10, 13, 22, 31, 34, 88\}$, the predecessor of 31 is 31 itself, whereas the predecessor of 30 is 22, and the predecessor of 5 is not defined. Likewise, the successor of 31 is 31 itself, whereas the successor of 15 is 22, and the successor of 89 is not defined.

Implement the `getPredecessor` method. Here is how you find predecessor using BST:

- Assign a temporary variable *tmp* to the root of BST
- Let predecessor be *null*
- while *tmp* is not null, repeat the following:
  - if value of tmp equals key then return *tmp*
  - else if value of tmp < key, do the following:
    * set predecessor to *tmp*
    * set tmp to the right node of tmp
  - else set tmp to the left node of tmp
- Finally return predecessor

Use similar ideas to implement the `getSuccessor` method for finding the successor.

3.2 Nearest Neighbor Search

Nearest neighbour search (NNS) is one the most important problems in Computer Science. It can be defined simply as follows: given a set $\mathcal{P}$ of points and a query point q, find the point in $\mathcal{P}$ that is closest to q (breaking ties arbitrarily). It's importance is not only limited to algorithms and data structures, but can be traced into numerous other areas such as (but not limited to): *Machine Learning, Security, Artificial Intelligence, Data Mining and Analytics, Databases, DNA sequencing, and etc.*

Observe that the nearest neighbour is either the predecessor or the successor. Thus, the nearest neighbour of 30 is 31 and that of 24 is 22. So, our approach is to find both, and simply pick the the one which is closer to the number (breaking ties arbitrarily).

Implement the `nearestNeighbour` method for find the closest value to *key* as follows:

- Find the predecessor and the successor of *key*
- If predecessor is null, then successor's value is the nearest neighbor
- If successor is null, then predecessor's value is the nearest neighbor
- If neither of them is null, then return the one whose value is closer to *key*

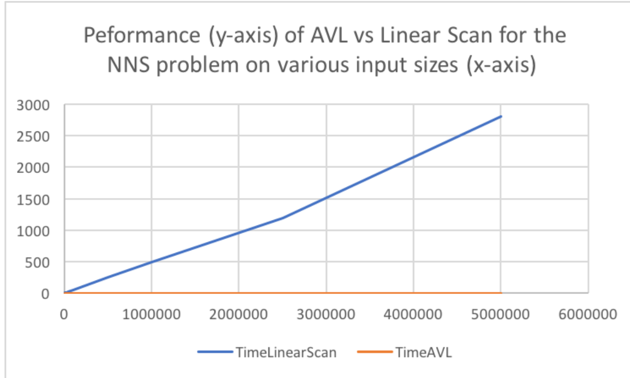
Comparative Analysis. Note that we can also find nearest neighbour by scanning through the entire list of numbers and finding the closest one. This, however, is going to have $O(N)$ complexity as opposed to the $O(H)$ complexity using BST. Although H is N in the worst case, Figure 4 clearly depicts the difference – BST is much faster!

3.3 Lowest Common Ancestor

Define the lowest common ancestor (LCA) of x and y as the lowest node (i.e., the node with minimum height) that contains both x and y in its subtree. For example, in Figure 5, the LCA of 20 and 55 is 50, LCA and 20 and 35 is 25, LCA of 55 and 95 is 80, and LCA of 10 and 25 is 25.

Implement the `getLCA` method for computing the LCA.

Figure 4: Nearest Neighbour Search Comparison



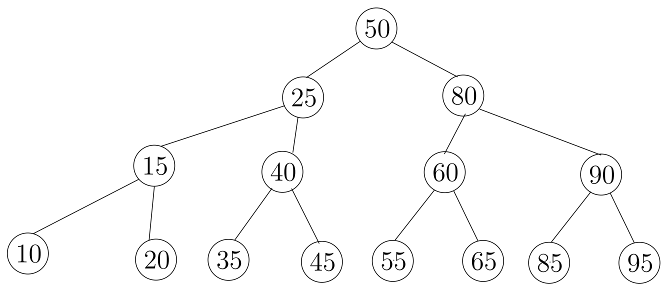


Figure 5: A Binary Search Tree

- If $x > y$, then swap them
- Assign a temporary variable tmp to the root of BST
- while tmp is not null, repeat the following:
  - if value of tmp is smaller than x then go to the right node of tmp
  - else if value of tmp is larger than y, then go to the left node of tmp
  - else break
- return tmp

3.4 Range Counting

In one-dimensional range counting, given a set $\mathcal{P}$ of numbers and two numbers L and R, where $L \leq R$, the task is to report the count of numbers in $\mathcal{P}$ that lie within L and R both inclusive.

The simplest way to visualize the set $\mathcal{P}$ is a list that stores the numbers and to find all the numbers within $[L, R]$ we simply traverse the list with a for-loop and keep a counter which is incremented every time we find a new number within $[L, R]$. Unfortunately, this takes $O(N)$ time.

Here's how the BST based algorithm works. For example, let $L = 15$ and $R = 86$, then the numbers from the BST lying in this range are $\{15, 20, 25, 35, 40, 45, 50, 55, 60, 65, 80, 85\}$; hence, the count is 12. Successor of L is 15 and predecessor of 86 is 85; thus, LCA is 50. We start from 50 and traverse to 15; there are 3 nodes ≥ 15. Also, the subtree size of 40 (right child of 25) is 3, and subtree size of 20 (right child of 15) is 1. Hence, counter value is $(3 + 3 + 1) = 7$. Now, we traverse from 50 to 85, and see 2 nodes ≤ 85 and > 50. Also, the subtree size of 60 (left child of 80) is 3. Hence, counter value becomes $(7 + 2 + 3) = 12$, as desired.

Implement `rangeCount` method as follows.

- If $L > R$, then return 0
- Otherwise find the LCA of L and R (note that this LCA contains all values in the range $[L, R]$ and any value outside the subtree of LCA does not contain $[L, R]$)
- if LCA is null, then there is no value in the range; so, return 0
- if LCA's value equals both L and R, then there is only one value, so return 1
- Initialize a counter to 0
- If LCA's value $> L$ and $< R$, then there is at least one value, so set counter to 1
- Else If LCA's value equals L or R, then there is at least one value, so set counter to 1
- If LCA's value $> L$, then we may find more values to LCA's left. Do the following: [a]
  - Set *tmp* to the left of LCA
  - As long as *tmp* is not *null*, do the following:
    - if L equals *tmp*'s value, then increment counter by (1+ sub-tree size of tmp's right) and break out of the loop
    - if $L <$ *tmp*'s value, then increment counter by (1+ sub-tree size of tmp's right) and set *tmp* to *tmp*'s left
    - else set *tmp* to *tmp*'s right
- If LCA's value $< R$, then we may find more values to LCA's right. So, do the following:
  - Set *tmp* to the right of LCA
  - As long as *tmp* is not *null*, do the following:
    - if R equals *tmp*'s value, then increment counter by (1+ sub-tree size of tmp's left) and break out of the loop
    - if $R >$ *tmp*'s value, then increment counter by (1+ sub-tree size of tmp's left) and set *tmp* to *tmp*'s right
    - else set *tmp* to *tmp*'s left
- return count

[a] subtree size is found using the *getSubtreeSize* function that accepts a node and returns its subtree size.

Caution: For the BST methods, you must achieve a complexity of $O(H)$, where H is the height of the tree. For hashing, your code must achieve complexity proportional to the length of the linked list being scanned for the particular value concerned.

Otherwise, (say when your code ends up scanning the entire tree/hash table), you'll get partial credits (even if your code is correct).

You should test your methods thoroughly, and it is a good idea to use other test-cases (other sequence of numbers than the one I have given).