Programming Assignment 3 (Dictionaries using Hashing and BST)

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

1 Dictionary Problem

We will **solve the Dictionary problem using Hashing and Binary Search Tree**, and compare their performance. Specifically, given a set of integers, we want to support the following operations:

- Search a number. Returns *true* if the number is in the collection, else returns *false*.
- Insert a number. Returns *true* on successful insertion, else returns *false*.
- Delete a number. Returns *true* on successful deletion, else returns *false*.

We have seen two techniques to solve this problem:

- **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**: This is the same as the version of Binary Search Tree that has been taught in class. **You are going to implement** the `search`, `insert`, `remove`, `removeLeaf`, and `removeNodeWithOneChild` methods in `BST.h/BST.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 or not your code is correct. 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; however, it is suggested that you still run this code as there may be an edge-case-error which the TestCorrectness code fails to catch.

2 Hashing

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. These functions have been written and you do not need to modify them:

- `gethashValue`: Uses the hash function $(37 * val + 61)\%TABLE\_SIZE$.

- `getList`: The hashtable is an array of linked list. So, this method simply returns the linked list at a particular index of the hash table.

We will use the inbuilt linked-list of C++ or Java. This is an implementation of a doubly-linked-list (with links going both forward and backward). It supports all the standard operations (inserting at front or end, deleting head or tail, traversing the list, etc.) In the next sections (C++ or Java), I'll highlight some of the usages (not all may be required). Since we will deal with integers, I will only discuss integer linked lists, but lists of any type can be created.

Check the `UnderstandingLinkedList.cpp/UnderstandingLinkedList.java` file for examples.

2.1 C++

- Syntax to create an integer list: `list<int> *nameOfList = new list<int>();` Note that we are using dynamic allocation over here. Although that is not required in general, for this assignment, that's how the code has been set up.
- To get the size of the list, the syntax is: `nameOfList->size();`
- To add a number at the end, the syntax is: `nameOfList->push_back(15);` To add a number at the beginning, the syntax is: `nameOfList->push_front(15);`
- To remove the last number, the syntax is: `nameOfList->pop_back();` To remove the first number, the syntax is: `nameOfList->pop_front();`

To traverse the list (say for retrieving a value or searching or printing or deleting), one can use an iterator as shown next. Here, `print` prints the list and `remove` removes the node at index.

```
void print(list<int> *numbers) { // printing the content
    for (list<int>::iterator it = numbers->begin(); it != numbers->end(); ++it)
        cout << *it << " ";
}
void remove(list<int> *numbers, int index) { // remove node at index
    if (numbers->size() == 0)
        return; // nothing to remove
    if (index == 0)
        numbers->pop_front(); // remove node at index 0
    else if (index == numbers->size() - 1)
        numbers->pop_back(); // remove node at last index
    else {
        int i = 0;
        for (list<int>::iterator it = numbers->begin(); it != numbers->end();
                ++it) {
            if (i == index) { // if i = index, we are at desired node
                numbers->erase(it); // delete current node
                return;
            }
            i++;
        }
    }
}
```

- `list<int>::iterator it = numbers->begin()` declares an iterator *it* on the list *numbers* and the iterator points to the first number on the list.
- `it != numbers->end()` ensures that the iterator traverses until the last node.
- `*it` retrieves the value of the node at which the iterator is pointing, and thus `cout << *it` prints the value of the node at which the iterator is pointing.
- `++it` moves the iterator to the next node.
- `numbers->erase(it)` deletes the current node, i.e., the node at index.

2.2 Java

- Syntax to create an integer list: `LinkedList<Integer> nameOfList = new LinkedList<Integer>();`
- To get the size of the list, the syntax is: `nameOfList.size();`
- To add a number at the end, the syntax is: `nameOfList.addLast(15);` To add a number at the beginning, the syntax is: `nameOfList.addFirst(15);`
- To remove the last number, the syntax is: `nameOfList.removeLast();` To remove the first number, the syntax is: `nameOfList.removeFirst();`

To traverse the list (say for retrieving a value or searching or printing or deleting), one can use an iterator as shown next. Here, `print` prints the list and `remove` removes the node at index.

```
static void print(LinkedList<Integer> numbers) { // printing the content
    Iterator<Integer> it = numbers.iterator();
    while (it.hasNext())
        System.out.print(it.next() + " ");
}
static void remove(LinkedList<Integer> numbers,
            int index) { // remove node at index
        if (numbers.size() == 0) // get the size of the list
            return; // nothing to remove
        if (index == 0)
            numbers.removeFirst(); // remove node at index 0
        else if (index == numbers.size() - 1)
            numbers.removeLast(); // remove node at last index
        else {
            int i = 0;
            Iterator<Integer> it = numbers.iterator();
            while (it.hasNext()) {
                it.next();
                if (i == index) { // if i = index, we are at desired node
                    it.remove(); // delete current node
                    return;
                }
                i++;
            }
        }
    }
```

- `Iterator<Integer> it = numbers.iterator();` declares an iterator *it* on the list *numbers* and the iterator points to the first number on the list.
- `while (it.hasNext())` ensures that the iterator traverses until the last node.
- `it.next()` retrieves the value of the node at which the iterator is pointing as well as moves the iterator to the next node. Thus `System.out.print(it.next())` prints the value of the node at which the iterator is pointing and moves the iterator to the next node.
- `it.remove()` deletes the current node, i.e., the node at index.

2.3 Few things that you should know but not use for this assignment

Both in C++ and Java, there are inbuilt functions that allow you to search a list for a particular number or remove an occurrence of a particular number. However, these are riddled with issues or totally necessary for our purposes because of the overhead involved. For example, to remove an number in Java, one has to create an Integer object for the number and then use it as argument to the remove method. In C++, one has to use the find method in the algorithms header to get an iterator to the occurrence of a number, and then remove the iterator.

Remember that over here we want to learn the usage of iterator; so, you are prohibited to use in-built methods other than the ones mentioned previously.

2.4 Pseudo-code

search

- First obtain the hash value for the *key* using `getHashValue` function.
- Use `getList` to get the linked list from the $hashTable[\,]$ for this hash value.
- Now, iterate through this linked list using an iterator.
  If the iterator's value equals *key*, then the list contains the *key*; so, return *true*.
- Once the iteration completes, return $false$.

insert

- Remember that your hash table should contain a number only once. So, first use *search* to check if the hash table already contains *val*. If it does, then return $false$.
- Obtain the hash value for *val* using `getHashValue` function.
- Use `getList` to get the linked list from the $hashTable[\,]$ for this hash value.
- Now, insert the value at the end of the linked list, and return *true*.

remove

- First obtain the hash value for *val* using `getHashValue` function.
- Use `getList` to get the linked list from the $hashTable[\,]$ for this hash value.
- Now, iterate through this linked list using an iterator. If the iterator's value equals *val*, then the list contains *val*; so, delete using the iterator and return *true*.
- Once the iteration completes, return false.

3 BST Implementation Details

Let's understand the class structure. `BSTNode.h/BSTNode.java` contains the `BSTNode` class that represents a node in a binary search tree. The class has 4 variables – $left$ & $right$ (which respectively represent the left & right children of a node), *value* (stores the value of the node), and *parent* (which represents the parent of a node). `BST.h/BST.java` contains the `BST` class that represents the tree; it has two variables – *root* (which represents the root of the tree) and a *size* (which captures the number of nodes in the tree). Note the types – *value* and *size* are integers, whereas the others are a BSTNode reference (in Java) or a BSTNode pointer (in C++) because they point to a BSTNode.

The `search` and `insert` methods work as their names suggest. The `remove` method first searches for the node containing *val*, the value to be deleted. If *val* does not exist, there is nothing to delete. Otherwise, if the node has both children (Case 3), then a reduction to Case 1 or 2 is carried out; to this end, use the `findMax` method that accepts a node as argument and finds the node with the maximum value in the subtree of the node. Now, either the `removeLeaf` method or the `removeNodeWithOneChild` method is called. See the project folder for an explanation video.

- A node is the root if it equals root or its parent is null
- A node is a leaf if its left child and right child are both null. A node has a left child if its left child is not null. A node has a right child if its right child is not null.
- A node is a left child if its parent's value is larger or equal, else it is a right child.

search

- 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

insert

- If *size* is 0, then allocate memory for the *root*, increment size, and return the root.
- Set a temporary variable *tmp* to the root and a 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*.

remove

- Find the node to be deleted, say *nodeToBeDeleted*, by calling `search` with *val*
- if *nodeToBeDeleted* is null, there's nothing to be deleted; so, return *false*
- if *nodeToBeDeleted* has a left child and a right child, then do the following:
  - find the node having the maximum value in *nodeToBeDeleted*'s left subtree
  - set *nodeToBeDeleted*'s value to the maximum node's value
  - set *nodeToBeDeleted* to the maximum node
- if *nodeToBeDeleted* is a leaf, call `removeLeaf` with *nodeToBeDeleted* as argument, else call `removeNodeWithOneChild` with *nodeToBeDeleted* as argument
- decrement size and return *true*;

removeLeaf

- if *leaf* is the same as *root*, then
  - C++ programmers: delete *root*
  - set *root* to *null*
- else, do the following:
  - Assign a temporary variable *parent* to *leaf*'s parent
  - if *leaf* is a left child, then set *parent*'s left to null, else set *parent*'s right to null
  - set *leaf*'s parent to null
  - C++ programmers: delete *leaf*

removeNodeWithOneChild

- declare a temporary variable child (of type BSTNode)
- if *node* has a left child,
  - set *child* to *node*'s left and set *node*'s left to null
- else, do the following:
  - set *child* to *node*'s right and set *node*'s right to null
- if *node* is the same as *root*, then
  - set *root* to *child* and set *child*'s parent to null
  - C++ programmers: delete node
- else, do the following:
  - if *node* is a left child, then set *node*'s parent's left to *child*, else set *node*'s parent's right to *child*
  - set *child*'s parent to *node*'s parent and set *node*'s parent to null
  - C++ programmers: delete node

4 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 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.

Needless to say that in actual "real-world" applications, one has to worry about the data, as data is not truly random. So simple hash functions are often not good and neither are unbalanced binary search trees. One has to either choose better hashing schemes or implement balanced binary search trees.

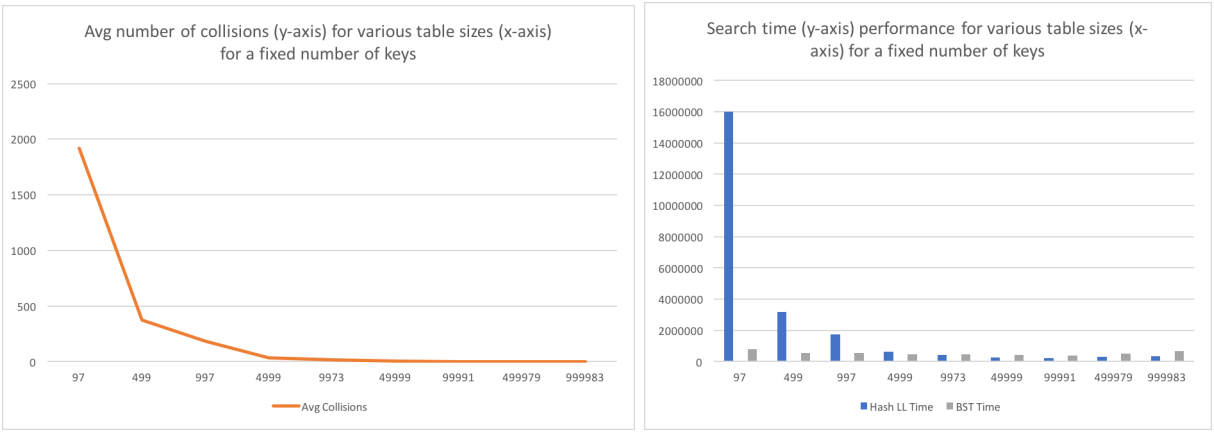

Figure 1: Search time performances for various table sizes when 1 million values are randomly inserted and then another 1 million values are randomly searched.

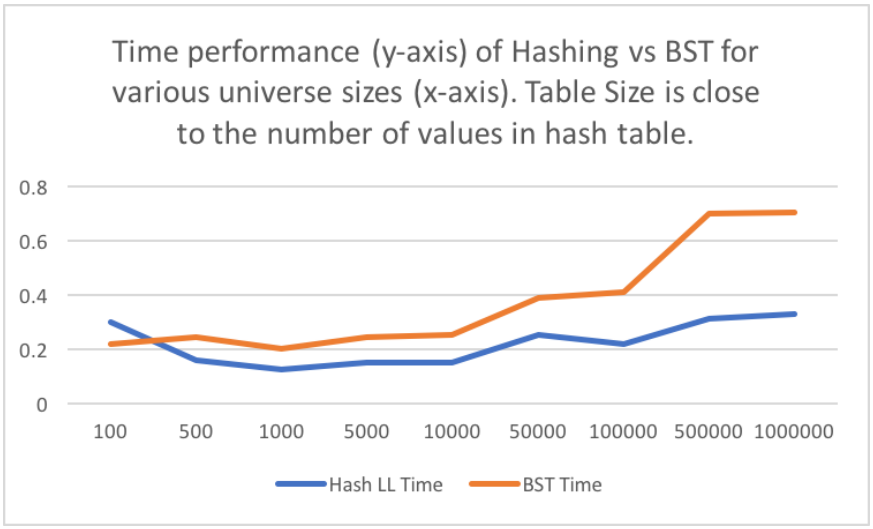

Figure 2: Comparison of search performances using hashing with linked lists and BST