University of Calgary
Faculty of Science
Department of Computer Science

UNIVERSITY OF CALGARY

CPSC 331- Data Structures, Algorithms, and Their Analysis

Assignment 3
Hash Tables, Binary Trees and Asymptotic Notations
Spring 2022

1 Objectives

The objectives of this assignment is three fold: (1) implementing advanced data structures such as hash table with chaining to solve a practical problem, (2) gaining experience in developing algorithms to manipulate binary search trees, and (3) analyzing related algorithms to find to their asymptotic notations.

2 Background

A school decided to build a database of its students. Typical operations are thus needed, including, adding new students, removing students when they graduate, and most importantly querying this database to retrieve various information about their students, and producing customized statistical reports about them. One of the school's administration's interest is to find the average age of its students by keeping track of their names and corresponding ages. The system should then automatically increase the students' ages on their birthdays.

3 The Problem

You need to solve the following two parts of this assignment:

3.1 Part 1

You are responsible for creating the school's database. For this, you will create a hash table with chaining in which the keys are names and the values are ages. Your hash table should have the following functionality:

1. Checking whether a student is in the database [method: search]
2. Checking a given student's age [method: getAge]
3. Adding students with their ages/modifying a student's age [method: insert]
4. Removing a student from the database [method: delete]
5. Incrementing a student's age by one (on their birthday) [method: increment]
6. Viewing the whole database [method: toString]

To implement these, you'll also need a hash function [method: hashValue]. More detail on all these things is in the implementation details.

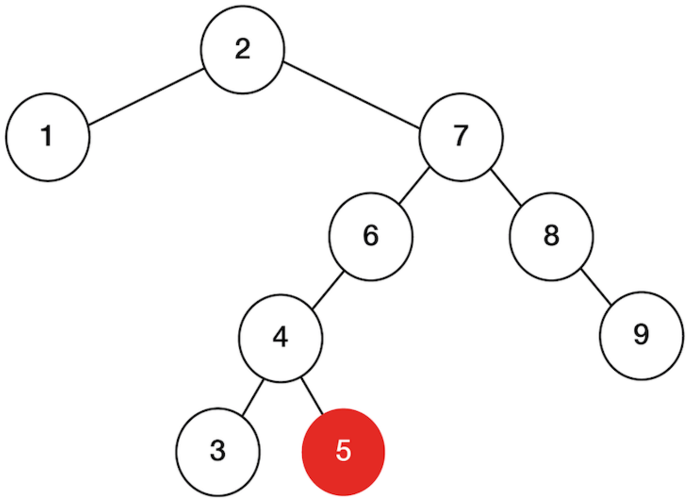

Figure 1: The node with value 5 is the median node because the BST has 4 nodes with smaller values, these are the nodes with values: $1, 2, 3$, and 4 and 4 nodes with larger values, these are the nodes with values $6, 7, 8$, and 9.

3.2 Part 2

One of the useful data structure is the Binary Search Tree (BST). As discussed in the course, this is a special case of a binary tree where the key in each node is greater than all keys in the left subtree and smaller than all keys of the right subtree. In this part of the assignment, you need to find the **median-valued** node in a BST. Generally speaking, the *median* node n of a given BST B is the node such that the number of nodes in B with value smaller than the value at n is equal to the number of nodes in B with value larger than the value at n. For example, in the BST below, the node with value 5 is the median because there are 4 nodes with smaller values than 5, these are the nodes with values: $1, 2, 3$, and 4 and 4 nodes with larger values, the nodes with values $6, 7, 8,$, and 9.

The problem consists then of the following:

Given Suppose that a binary search tree (BST) T has m nodes, all distinct, and of depth d, where m is an odd number. Note that T is not necessarily full. As defined in the class, the *depth of a tree* T is the maximum of the depths of all the nodes in T, where the *depth of a node* is the number of edges that must be crossed to travel from that node to the root of the tree. Additionally suppose that each node of B has a property *size* where *node.size* is the number of nodes in the subtree for which *node* is the root. For example, if r is the root node of B, then $r.size = m$ because the tree with root r is B and B has m nodes. For any node n with no child nodes, we have $n.size = 1$ since the subtree with root n contains only the node n. The following tree can be labeled with size as below:

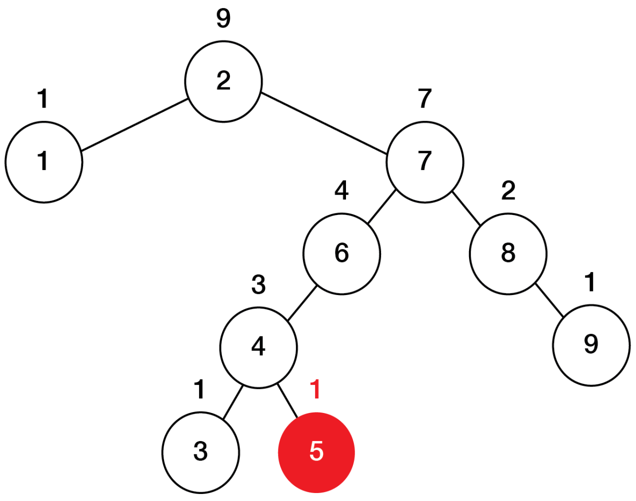

Figure 2: The BST above with nodes labeled by their size

Problem Design an efficient algorithm which runs in $O(d)$ time that finds a node *median* of B so that the number of nodes with value smaller than the value of *median* equals the number of nodes with value larger than the value of *median*. In other words, find the median-valued node of T (note that since the number of nodes m of the tree T is odd, there is exactly one median-valued node). Prove the correctness of your algorithm and analyze its running time.

4 Implementation Details

The following sections are helpful in your implementation[1]

4.1 Hash Tables with chaining

You've learned about hash tables with chaining during lectures. In this kind of hash table, keys are passed through a hash function which produces indices for an array. The key-value pair is then placed in a linked list at this index of the array. See the below diagram for an illustration of a hash table with chaining:

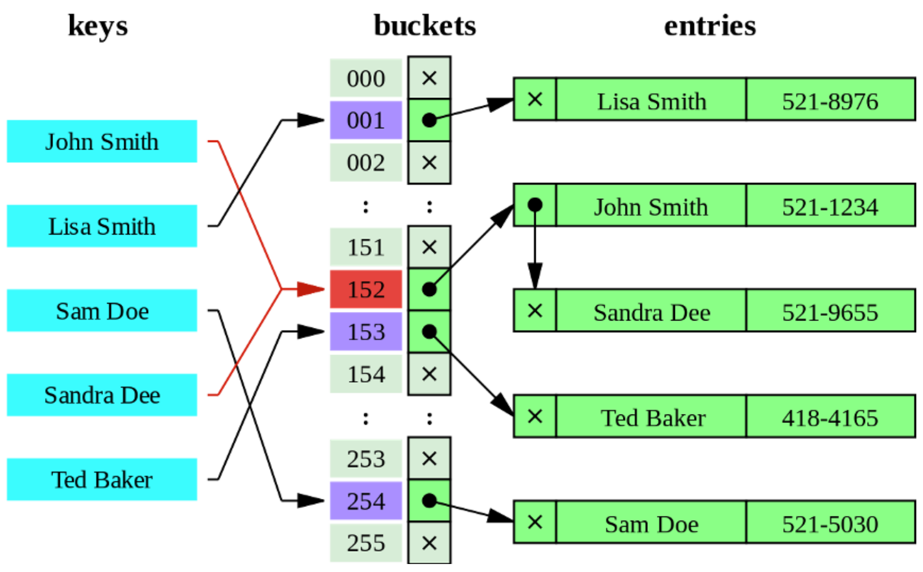

Figure 3: An example of a hash table with chaining.

You'll implement your hash table in Java, building upon skeleton code given in HashTable.java and Student.java.

[1]Please follow the directions of your TA in the tutorials sessions, and as posted on D2L shell of the course

The keys of the hash table will be the names of the students and will be of type String. The values are the ages and are of type int. The size of your array will be the constant LEN, which will be set to 8.

4.2 Hash Function Implementation

Your hash function will take a String and convert each character to its ASCII value. It will then take the sum of all these values modulo LEN. For example, for the string "cat", we have:

Table 1: Example of Converting Character to ASCII Values

Character	c	a	t
ASCII Value	99	97	116

The sum of these values is 99 + 97 + 116 = 312 and 312 modulo 8 is 0, so the hash function will return 0 in this case. To find the ASCII value of a character in Java, you can utilize the way Java stores char values and simply assign a variable of type int with the character you want to convert. See asciiIntDemo.java for an example of applying this. To take the modulus, you can use the % operator. For example, 312 % 8 returns 0.

4.3 Linked List Implementation

In assignment 2, you implemented linked lists from scratch. In this assignment, you can use the native Java implementation **java.util.LinkedList.** The type of object stored in your linked list will be Student, where the Student.java class is provided on D2L. Documentation for Java LinkedList can be found at:

docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html

Geeks for Geeks has a fairly comprehensive page about Java LinkedList here:

geeksforgeeks.org/linked-list-in-java/

Below are a few LinkedList methods you are likely to use:

1. Initializing: LinkedList$\langle ObjectType \rangle$ l = new LinkedList$\langle ObjectType \rangle$
2. Append to the end of the list: *l.add(element)*
3. Get element at specified index: *l.get(index)*
4. Remove by index: *l.remove(index)*

5. Length: *l.size()*

Note: Java LinkedList nodes don't seem to have **.next** pointers as would usually be expected so if you need to iterate through the nodes of a LinkedList, you can do so in a similar fashion to an array using the **l.get(index)** method (I believe doing so actually increases the time complexity but don't worry about it)[2] .

4.4 Method Preconditions and Postconditions

The following are the main preconditions and the postconditions of the operations involved. Again, you need to follow the directions of your TA in the tutorials sessions, and as posted on D2L shell of the course, in particular for any I/O format:

1. **HashValue(String s)**
   Precondition: s is a string composed only of alphanumeric characters.
   Postcondition: The sum of the ASCII values of s modulo LEN is returned.

2. **Search(String name)**
   Precondition:name is a string composed of alphanumeric characters.
   Postcondition: If the hash table contains a student by this name, then return true. Otherwise, return false.

3. **Insert(String name, int age)**
   Precondition: name is a string composed of alphanumeric characters and age is a positive integer.
   Postcondition: If the hash table does not contain a Student of this name, then a Student with attributes name and age is added to the appropriate linked list in the hash table. If a Student of this name is already in the hash table, then this Student has their age updated to the inputted age. Additionally, a message is printed indicating that a student is being inserted and their name and age. You may choose to use a more specific message when a student's age is being modified.

   Examples of possible messages:

   Inserting student Alice, age 10
   Changing age of Alice from 10 to 8

4. **Delete(String name)**
   Precondition: *name* is a string composed of alphanumeric characters.
   Postcondition: If the hash table contains a *Student* with this name, this Student is removed from the hash table. Otherwise, a message is printed indicating that no Student of this name was found in the hash table.

[2] *You may also choose to use the LinkedList.listIterator() method to iterate through the nodes.*

5. **GetAge(String name)**
   Precondition: *name* is a string composed of alphanumeric characters.
   Postcondition: If the hash table contains a *Student* with this *name*, then this student's age is returned. Otherwise, a message is printed indicating that no student of this name was found in the hash table and the value -1 is returned.

6. **Increment(String name)**
   Precondition: name is a string composed of alphanumeric characters.
   Postcondition: If the hash table contains a *Student* with this *name*, then this student's age is incremented by 1 and a message is printed to indicate this. Otherwise, a message is printed indicating that no student of this *name* was found in the hash table.

7. **toString()**
   Precondition: *arr* is an array of LinkedList of Student.
   Postcondition: A string is returned consisting of

   - One line per entry of the array, including empty linked lists (it is acceptable to have an extra newline at the end)
   - Each line takes the form

     ArrayIndex: [FirstStudentName:FirstStudentAge, SecondStudentName:SecondStudentAge, . . . , LastStudentName:LastStudentAge]

     Example: If Ahmad:25, Bob:20, Cathy:15, Desmond:20 are inserted (in that order), *toString()* produces the following:

     ```
     0:  [ ]
     1:  [Cathy:15]
     2:  [Desmond:20]
     3:  [Ahmad:25, Bob:20]
     4:  [ ]
     5:  [ ]
     6:  [ ]
     7:  [ ]
     ```

     Implementation Hint: It's intended that you use the built-in *toString()* for LinkedList, which will implicitly call the *toString()* for Student, which has already been implemented.

4.5 BST Median Algorithm and Running Time Analysis Details

The following steps are guidelines to develop your *BST-Median* algorithm and the corresponding asymptotic proof and correctness:

1. Write your algorithm in *pseudocode*.
2. Use the notation conventions of the lecture slides where possible (e.g. node.key, node.value, node.left, etc).
3. When writing your proof of correctness and analysis of running time, use the vocabulary from the lecture slides where applicable (e.g. left subtree, right child, descendant, leaf node).
4. Below are some binary search tree diagrams with the .size parameter labelled above each node. In each tree, the median-valued node is indicated in red. You can use these to come up with ideas for how your algorithm might be constructed.
5. Once you have written an algorithm, you can use these as test cases to check your algorithm's accuracy (note that getting the correct answer on these test cases does not prove that your algorithm is correct, so don't use them for your proof of correctness).

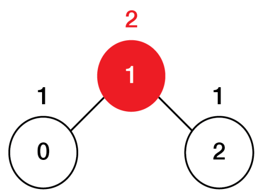

Figure 4: The BST above with nodes labeled by their size — The median-value node is shown in red.

5 Grading

Grading will be assigned for all aspects, including efficient programming, documentation style, and the identification of runtime/storage complexity. The total grade of this assignment will initially be set to 51 points but will be factored in the final grade as per the outline of the course. The following guidelines will be used[3]:

[3]Code efficiency will be considered in grading each of the implementation questions.

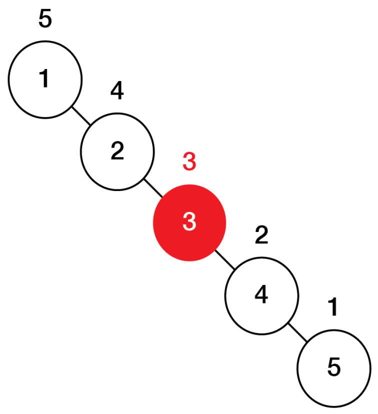

Figure 5: The BST above with nodes labeled by their size — The median-value node is shown in red.

Part 1 This part will carry 31 points (4 for each of the below methods and 3 for documentation):

1. hashValue()
2. search()
3. insert()
4. delete()
5. getAge()
6. increment()
7. toString

Part 2 This part will carry 20 points distributed as follows:

1. 10 points for a correct and efficient algorithm
2. 5 points for proof of correctness
3. 5 points for running time analysis

6 Grading policy

Students could be randomly selected to demonstrate their programs and to show that they understand their codes.

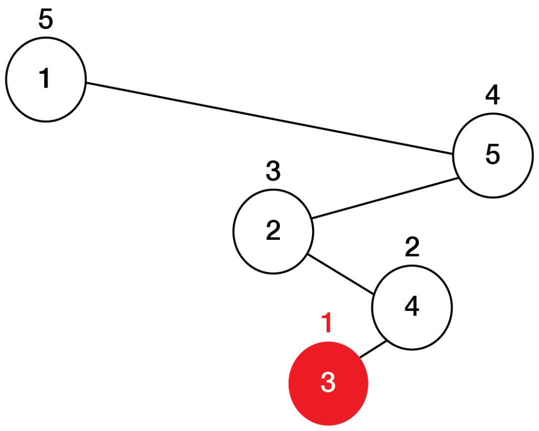

Figure 6: The BST above with nodes labeled by their size — The median-value node is shown in red.

The skeleton code provided demonstrates how the code will be tested, however the skeleton code does not test every aspect of the functionality, and additional tests will be part of the grading process. Work should be done individually. Any source of help MUST be indicated and cited

Grading will be done as follows:

1. You will receive less than 20% of the total grade if your program compiles, but it doesn't work at all.
2. You will receive 0 points if your program will not compile.
3. For identical code, in part or full, the UofC rules and regulations for plagiarism will apply.
4. After the due time, a %20 deduction will apply on every late day or part of the day.

It's better to have something working, even with little functionality, than a big program that crashes (or doesn't compile). We expect to see your own program, otherwise the above will apply.

7 To Submit

Submit all java files required to compile your project, including all the provided files, even if you have not modified them, and your written component files as a .zip archive. Also be

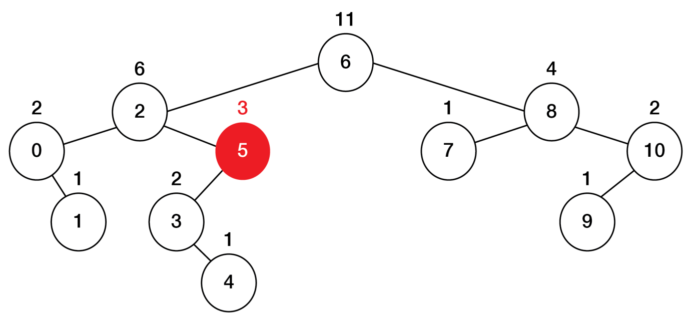

Figure 7: The BST above with nodes labeled by their size — The median-value node is shown in red.

sure to include any additional files you may have written (if any). Name your compressed file **Assign2-ID.zip**, where ID is your UCID.

For the written component please submit a legible document with your two proofs. You may typeset your document using word processing software such as Microsoft Word, LibreOffice Writer, or Latex, or you can hand-write your proofs. If you are writing them by hand submit a clear scan or photograph of your proofs. Please ensure that scans/photos of handwritten proofs are clearly readable. The written document should be named **Assign3Proof-ID**.

Your files should include the following:

For Part 1
1. HashTable.java
2. Student.java
3. Any additional java files necessary to compile and run your project (most students probably won't need any)
4. README if there's any information about compiling and running your project that you want us to know

For Part 2 PDF containing your algorithm, proof of correctness, analysis of running time

You are URGED to check your submission after loading it on D2L, unzipping it and ensuring the contents are as expected. Errors in submission after the deadline will be subject to the indicated penalties above.

8 Deadline

Sunday June 16, 2022 before 11:55 P.M. For each late day (in full or in part), a deduction of %20 will be applied.

9 Reference

`https://commons.wikimedia.org/w/index.php?curid=6471915`