Due: Nov. 20 (11/20), 13:00

Overview

This assignment consists of two parts: implementing a Cuckoo Hash and implementing a balanced binary search tree.

General Notes

- *Read this homework guideline carefully.* If you do not follow the guidelines, you may receive a 0 regardless of whether your code works or not.
- Do not use any IDEs (Eclipse, IntelliJ IDEA, etc.)
  - We recommend Sublime Text (Linux/Mac/Windows), Notepad++ (Windows), or TextWrangler (Mac).
  - IDEs often create a "package" of your code, which breaks the autograder.
  - **If you know how to fix the package problem**, you can use any IDE you want. However, we will not answer any questions related to this problem since we have already recommended a solution.
- Do not change any method or class signatures. If your code changes any class or method names or signatures, you will receive an automatic 0.
- Make sure your code compiles. Non-compiling code will automatically receive a 0. If your code does not compile, it may be better to just comment out the incorrect code and return a dummy value (something like null or -1) so the rest can compile.
- To ensure that your code will be accepted by the autograder, you should submit your code on LearnUS, download it again, recompile it and check the provided test suite. This way, you know the file you are submitting is correct.
- If you use any code from the textbook, make a reference to the textbook through comments. If your code is flagged as copied and does not have a legitimate reference, you may get a 0 for plagiarism.

Cuckoo Hashing

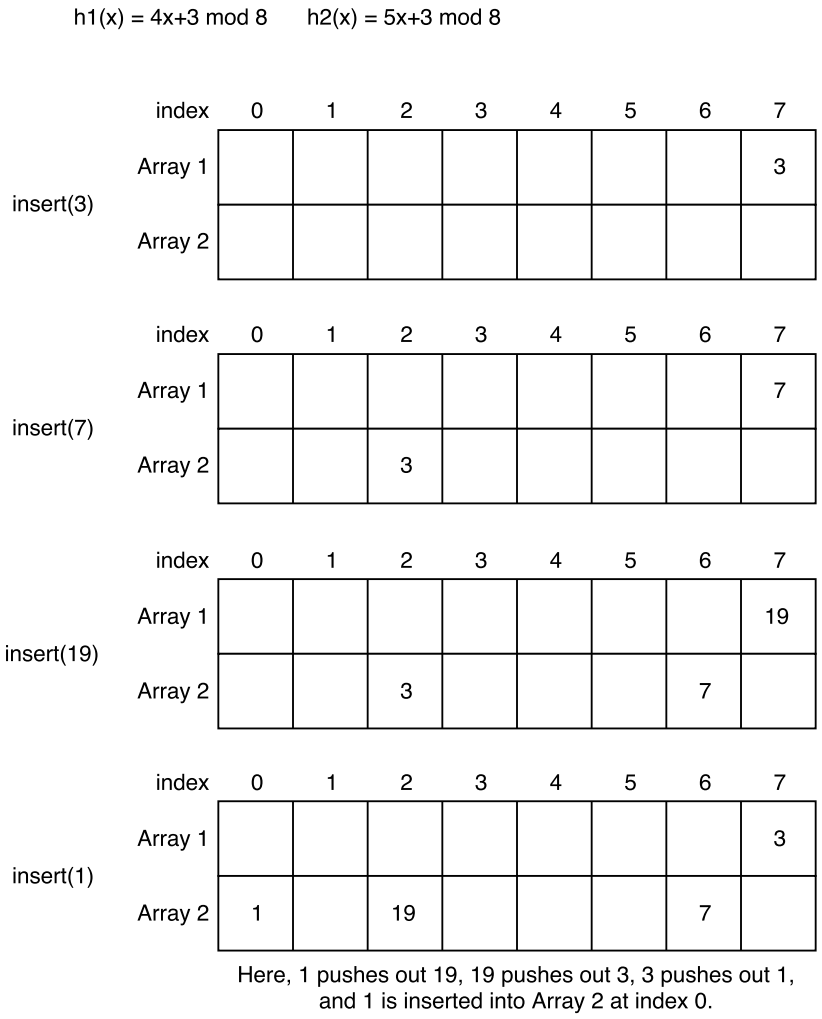
Cuckoo Hashing is a well known hashing scheme that has some desirable properties. Its name comes from the egg laying procedure of the Cuckoo bird. The bird sometimes lays its eggs in other nests. When the eggs hatch, they push out the eggs that were there. In a Cuckoo Hash, element containment and deletion are both worst-case $O(1)$ operations, while insertion can be done in *amortized* $O(1)$ time.

Definition

A Cuckoo Hash is defined by an integer $N > 1$, two hash functions $h_1(x) = (\texttt{a1} \cdot x + \texttt{b1}) \bmod \texttt{N}$ and $h_2(x) = (\texttt{a2} \cdot x + \texttt{b2}) \bmod \texttt{N}$, and two constants, $0 < \texttt{threshold} \leq 1$ and $\texttt{chainLength} > 0$. The Cuckoo Hash contains two arrays A_1 and A_2, each of size `N`. When an element x is inserted, we insert it into position $h_1(x)$ in array A_1. If there is an element, y, already in this position, we bump y out, insert x, then insert y into position $h_2(y)$ of array A_2. If there is already an element, z, at *that* position, we bump z out, insert y, then insert z it at position $h_1(z)$ into array A_1. This repeats until one of two situations happens:

- An element is placed in a previously unoccupied spot.
- An insertion causes a chain of bumped elements of length at least `chainLength`.

If we encounter a chain no shorter than the `chainLength` or we meet the condition $\frac{e}{2\texttt{N}} \geq \texttt{threshold}$ (where e is the number of elements currently in the hash), we perform a resizing operation. To resize, we set $\texttt{N}' = 2\texttt{N}$ and choose two new hash functions $h_1'(x) = (\texttt{a1}' \cdot x + \texttt{b1}') \bmod \texttt{N}'$ and $h_2'(x) = (\texttt{a2}' \cdot x + \texttt{b2}') \bmod \texttt{N}'$. We then reinsert (with the new hash functions) every element into two new tables, A_1', A_2', both of size $\texttt{N}'$.



Implementation Notes

- The methods to be implemented are:
  - `CuckooHash(double threshold, int chainLength, int N)`: initializes an empty cuckoo hash with the given constants and the initial length of the hash tables.
  - `boolean contains(int x)`: returns whether or not the given integer is in the hash table.

  - `void delete(int x)`: if integer `x` is in the hash table, deletes `x` from the hash table. Otherwise, do nothing.
  - `void insert(int x)`: if integer `x` is not in the hash table, inserts `x` from the hash table. Otherwise, do nothing.

- The `contains(int x)` and `delete(int x)` methods should take $O(1)$ time, and `insert(int x)` should take amortized $O(1)$ time.

- You are free to declare additional variables, private methods, or modify the constructor without altering its method signature.

- The `HashParameter getParams()` method returns a `HashParameter` object specifying the current hash functions and the number of elements. This method is already implemented for you for grading purposes. Since this method collects the current state of the instance variables `a1`, `a2`, `b1`, `b2`, and `N`, make sure to update those variables whenever a resizing operation is performed.

- You can use the `java.util.Random` class inside `CuckooHash.java`.

- For this assignment, there is no need to implement generics for the Cuckoo Hash.

- The chain length is defined by the number of bumped out elements during an insertion. So, if you insert an element and it immediately finds an open spot, that chain is length 0. If you insert an element and it bumps an element out which then bumps another out before finding an open spot, that chain is length 2. We activate the resize as soon as we reach `chainLength` bumps.

- If after inserting an element the threshold $\frac{e}{2N} \geq$ `threshold` would be met, resize the Cuckoo Hash and then insert the new element. For example, suppose $N = 2$, `threshold` $= 0.5$ and e is currently 1. Since if we inserted a new element we would have $\frac{e}{2N} = \frac{2}{4} = 0.5 \geq$ `threshold`, we should resize the Cuckoo Hash and then insert the new element.

- Whenever a resizing is activated, you should reinsert elements **in the order of their arrival**. Note that spending $O(n \log n)$ time for a single resizing operation will not make the insert function run in amortized $O(1)$ time.

- Your hash functions must satisfy $0 < \texttt{a} < \texttt{N}$, $0 \leq \texttt{b} < \texttt{N}$.

Balanced Binary Search Trees

Binary search trees allow searching for elements in $O(\log n)$ time. In this homework, you will implement a generic balanced binary search tree. You are free to choose the balancing method among the methods learned in class. Since each entry should be comparable by their keys, the generic type `T` implements the `Comparable` interface. For this implementation, you can assume that two entries are the same if they have the same key. The methods to be implemented are:

- `BinarySearchTree()`: initializes an empty binary search tree.
- `boolean contains(T entry)`: returns `true` if the binary search tree contains the given entry. Otherwise returns `false`.
- `void put(T entry)`: if `entry` is not in the binary search tree, inserts `entry` into the binary search tree. Otherwise, do nothing.
- `void remove(T entry)`: if `entry` is in the binary search tree, deletes `entry` from the binary search tree. Otherwise, do nothing.
- `void clear()`: removes all entries from the binary search tree.
- `int getSize()`: returns the number of entries in the binary search tree.
- `T getNext(T threshold)`: returns the minimum element that is greater than the key of the given threshold object.
- `T getPrev(T threshold)`: returns the maximum element that is no greater than the key of the given threshold object.

The methods `contains(T entry)`, `put(T entry)`, `remove(T entry)`, `getNext(T threshold)`, and `getPrev(T threshold)` should run in $O(\log n)$ time, where n be the total number of entries in the binary search tree. The methods `clear()` and `getSize()` should take $O(1)$ time. Use the `compareTo()` method to compare two type `T` objects.

General Directions

- Write your name and student ID number in the comment at the top of the files in src/main directory.
- Implement all of the required methods.
- You should not import anything that is not already included in the file.
- Pay careful attention to the required return types and edge cases.
- All the codes we provided can be found in src/base directory. If you are unsure what a class/method exactly does, please refer to the code.

Submission Procedure

Submit the files in the src/main directory, excluding Main.java, as a zip file. You *must* make a zip file for submission using the Gradle build tool (refer to Compiling section).

For this assignment, you should submit only the following three files:

- CuckooHash.java
- BinarySearchTree.java
- your_student_id_number.txt

You must rename 2023xxxxxx.txt to your actual student ID number. Inside that text file, you must include the following text. Please be sure to write all the following text including the last period.

In completing this assignment, I pledge that I have not given nor received any unauthorized assistance.

If this file is missing, you will get a 0 on the assignment. It should be named *exactly* your student id, with no other text. For example, *2023123456.txt* is correct while something like *2023123456_pa4.txt* will receive 0.

Compiling

You can test your Java code using the following command:

```
% ./gradlew -q runTestRunner
```

You can also make a zip file for submission using the following command. The zip file named with your student id (the name of the .txt file) will lie in the "build" directory:

```
% ./gradlew -q zipSubmission
```

We provide an empty Main class for testing using standard input/output:

```
% ./gradlew -q --console=plain runMain
```

Since this file (src/main/Main.java) is not for submission, you can use any package in the file.

On Windows, use `gradlew.bat` instead of `./gradlew`.

Testing

We have provided a small test suite (src/test) for you to check your code. You can test your code by compiling and running the tester program.

Note that the test suite we will use to grade your code will be much more rigorous than the one provided here (and not necessarily a superset of the provided tests). You should consider making your own test cases to check your code more thoroughly.