CSCI 102 assignment 5 – Hash maps

Mar 3, 2025

1 Using maps

In this part of the assignment, you will see maps in action.

- Build a `main` method in `UnsortedMap` where you create a `Map` variable that points to an `UnsortedMap` that has values of type `String` with keys of type `DoublyLinkedList<Double>`. Add entries with keys [3.43,5.432], [7.1], [812.4, 12.76, 123.4] and values "One", "Two", "Three". Print out the result of `get([7.1])`. Try to add "four" with key [7.1] and print the result of `get([7.1])` again.

Now let's do the same thing with a Hashmap. We'll need to implement a hash code though!

- Override `hashCode` in `DoublyLinkedList` to get a hash code that turns the elements in the list into a string and then hashes the concatenated string with commas in between and brackets on the ends– i.e. a list that has the doubles `3.43` and `5.432` should have the hash of the string "[3.43,5.432]". To get the string of an element, use the built in Object method `toString()`.
- Write another method `hashCodeAlternative` for `DoublyLinkedList` that implements a polynomial hash code.
- Build a `main` method in `HashMapSC` where you create a `Map` variable that points to an `HashMapSC` that has values of type `String` with keys of type `DoublyLinkedList<Double>`. Add entries with keys [3.43,5.432], [7.1], [812.4, 12.76, 123.4] and values "One", "Two", "Three". Print out the result of `get([7.1])`. Try to add "four" with key [7] and print the result of `get([7.1])` again. You should reuse the code from the first part of this problem!

2 Hashing trees

Building a hash code for a list was not so hard: we just look at the elements in order of traversal. Here we'll look at building a hash code for trees using traversals.

- Give an example of two different trees that have the same preorder traversal, two binary trees with the same inorder traversal, two trees with the same postorder traversal, and two trees with the same breadth-first-search traversal. Given your answers, is looking at the elements of a tree in order of these traversals a good strategy for building a hash code?
- Say we implemented a method `preOrderString` that prints out a string **with brackets** as such:

  ```
  root.toString()(preOrderString(first_child))(preOrderString(second_child))...
  ```

  where `preOrderString(root.left)` outputs `preOrderString` for the subtree rooted at `root.left`. Can two different trees have the same output `preOrderString` (assume `E.toString()` cannot contain brackets)? If so, give an example; if not, explain why not.
- Implement `preOrderString` for `LinkedTree<E>`.
- Override `hashCode` and `equals` in `LinkedTree<E>` with a more sensible implementation.
- (Bonus) Why is this not a good hash code for **binary** trees? Give an example. How could we modify it so it works fine for binary trees?
- (Bonus) Give an example of where this is a bad hash code if `E.toString()` contains brackets.

3 Open addressing

- Implement a hash map `HashMapOA` that resolves collisions with open addressing with linear probing. Don't worry about correctly implementing `values`, `keySet`, or `entrySet` (fill them with whatever you want); just correctly build a constructor and implement `get`, `put`, and, `remove` (implement `put` using `get` and `remove`). For a placeholder, use an entry which has both key and value set to null. Don't worry about the case where every index in the list has an entry.
- Build a `main` method in `HashMapOA` where you create a `Map` variable that points to an `HashMapOA` that has values of type `String` with keys of type `DoublyLinkedList<Double>`. Add entries with keys [3.43,5.432], [7.1], [812.4, 12.76, 123.4] and values "One", "Two", "Three". Print out the result of `get([7.1])`. Try to add "four" with key [7] and print the result of `get([7.1])` again. You should reuse the code from the first part of this problem! (Note this `main` method probably won't test your collision resolution!)

Please submit your code and answers to the questions in a zipped folder on Brightspace by Mar 17.