GIT Department of Computer Engineering
CSE 222/505 - Spring 2020
Homework 6
Report

Elif Goral
171044003

Problem Solution Approach

a) Hash Table for Binary Tree:

For that purpose, I use my binary tree and binary search tree classes from book. Because of tree classes are generic, My HashTableChainBinaryTree class use comparable types for use tree. I declare (BinarySearchTree<Entry<K,V>>[] table) which express all the table index's tree. My Entry class also should comparable. For this purpose, I implement Comparable class. And override compareTo method. In compareTo method o compare the key values. In my main class I override get(),isEmpty(),put(),remove() and size() methods.

get() method: *That method search the value of given key. Firstly, calculate index with hashCode. Then control the index if it is smaller than 0, add with table's length. Then control the table's index element is null or not. Then control the key for finding existing knowledge with my binary search tree's contains method. If table's index element has key, I returned that key's value. Otherwise return Exception which express the key is not here.*

put() method: *That method puts the element to the binary search tree which is on table's index value. Firstly calculate index with hashCode and control the index is smaller than 0 or not. If index is smaller than 0, add with table's length. If the table's index's value is null which means there is no element in that index, I create a binary search tree. Otherwise I call the add method of binary search tree and try to add given key and given value. If add method returns false, that means that key already exist in binary search tree. Than I find the value and delete the key from the tree and add my new value and return deletedValue. Then I increase the numKeys. If numKeys bigger than (LOAD_THRESHOLD * table.length), I rehash the table and return null.*

Remove() method: *That method remove given key from table. Firstly calculate index with hashCode and control the index is smaller than 0 or not. If index is smaller than 0, add with table's length. If the table's index's value is null which means there is no element in that index, return null. Then I call the find method, if key will be found, I save the deleted value and deleted key and delete them from the tree decrease the numKeys and return deleted value. If method can not find the key, throws an exception and says there is no exist that key.*

b) Hash table for Open addressing with double hashing

That hash table almost the same with book's hash table with open addressing. There is two difference between the book and my own code.

First difference is that I created an extra function for hashing. And in that extra hash function I use (prime – (key.hashCode() % prime)) algorithm. I take it from Internet. Internet says, this is the most popular algorithm. And I chose 31 for the prime number.

*Second difference is when I write find method Firstly generate hash1 value with my hash1() function. Then generate hash2 value with my hash2() function. than as you said I calculate subsequent probe locations with (hash(x) + i * second_hash(x)). that is my index now. if index is smaller than 0, I added with table's length. Otherwise I started to search table's index, if it is not null and its key value is not equals to given parameter key, I continue to search. Every loop turned, I increase the 'i' value and calculate again the index. then control the index value. If index value is bigger and equal than table's length, I assign o to index. If index value is smaller than 0, I added index value with table's length.*

Other functions are same as book's hash table for open addressing.

Hash Table for Chaining (size = 10)(Binary Tree)

```
put method starting...
put method ending...
printing the hash table
index 11
key:1627 value:22722

index 30
key:17301 value:9479
key:8312 value:163

index 42
key:27110 value:15004

index 65
key:2893 value:31002

index 68
key:8956 value:31873

index 74
key:12194 value:12550

index 78
key:1694 value:4074

index 87
key:12611 value:3740

index 94
key:23122 value:21716


put method:
Time taken: 2 miliseconds
Time taken: 1873200 nanosecond
hash table's length: 10
```

```
removing that elements...
key: 2893       value: 31002
key: 1694       value: 4074
key: 12611      value: 3740
key: 17301      value: 9479
key: 23122      value: 21716
key: 12194      value: 12550
key: 8956       value: 31873
key: 8312       value: 163
key: 27110      value: 15004
key: 1627       value: 22722
java.lang.Exception: there is no that key.
java.lang.Exception: there is no that key.
printing the hash table

remove method:
Time taken of get: 1 miliseconds
Time taken of get: 132700 nanoseconds
Time taken of remove: 0 miliseconds
Time taken of remove: 324500 nanoseconds
removed data number: 10
not removed data number: 10
hash table's length: 0
```

Hash Table for Open Addressing (size = 10) (Double Hashing)

```
put method starting...
put method ending...
printing the hash table
index 13
key: 7280       value: 3593
index 14
key: 16267      value: 18514
index 19
key: 26768      value: 6406
index 23
key: 9512       value: 4591
index 53
key: 22457      value: 24429
index 54
key: 25190      value: 22845
index 55
key: 24796      value: 5591
index 62
key: 21948      value: 19961
index 75
key: 16720      value: 13295
index 94
key: 30993      value: 23492

put method:
Time taken: 1 miliseconds
Time taken: 1013100 nanosecond
hash table's length: 10
```

```
removing that elements...
key: 22457      value: 24429
key: 16267      value: 18514
key: 7280       value: 3593
key: 21948      value: 19961
key: 25190      value: 22845
key: 24796      value: 5591
key: 16720      value: 13295
key: 9512       value: 4591
key: 30993      value: 23492
key: 26768      value: 6406
printing the hash table

remove method:
Time taken of get: 0 miliseconds
Time taken of get: 27600 nanoseconds
Time taken of remove: 0 miliseconds
Time taken of remove: 252400 nanoseconds
removed data number: 10
not removed data number: 10
hash table's length: 0
```

Class Diagram: