CS 251 - Project 3: Hashtables and Heap

Out: Feb/4/2019, 8am
Due: Feb/25/2019, 8am

Part 1: Chaining Hashtable (30 points)

Hashtable with Separate Chaining:
Implement a generic Hashtable with **Separate Chaining**. This means our hashtable maintains an **array of Lists**, where the keys are stored by generating a hashvalue, obtained by:

hashvalue = hashcode(key) % capacity

This hashvalue is used to index the array, and the (key, value) pair is stored in the corresponding List at that index. Keys with same hashvalue are stored at the same index which means the same List.

Hashcode: Hashcode follows the rule: if A and B are equal, then hashcode(A) = hashcode(B). You may use any hash function you like for part 1.

Functions to implement:

- **Hashtable(capacity):** Constructor. Accepts the capacity of the table as a parameter. Initialize an array of length capacity then for each index, initialize an empty list. Capacity is a prime number.

- **get(key)**: Returns the value associated with the key if found, return null otherwise.

- **put(key, value):** Adds the (key, value) pair to the table. If key already present in the hashtable, overwrite the original value. Resize the hashtable if the load factor becomes greater than 0.5 after inserting this (key, value) pair. The new capacity should be the first prime number greater than twice the previous capacity and all the keys in the hashtable would need to be rehashed and stored in this new hashtable. In the provided framework, we hard code a list of prime numbers for you to use. **You can assume you will never run out of prime numbers to use.**

- **remove(key):** Removes the key from the table, if it's there. Return the value of the key if key exists in the hashtable, return null otherwise.

- **containsKey(key):** Returns whether the key is in the hashtable.

- **size():** Returns the total number of (key, value) pairs in the table.

- **replace(key, value):** Replaces the value for the specified key only if it is currently mapped to some value. Return the previous value associated with the specified key, or null if there was no mapping for the key.

- **Any other function you like! Those extra functions will not be tested. Functions above are the only functions that will be tested.**

(Key, Value) Type: Keys have type K and Values have type V.

Part 2: Hash Anagrams (30 points)

We then play with anagrams using the hash table in Part 1. Your job is to design a hash function which results in the least collisions for non-anagrams and in all anagrams of the same letters mapping to the same hash index.

Anagram: An anagram is a word or phrase formed by rearranging the letters of a different word or phrase, using all the original letters exactly once. e.g., “name” = “mean”, “debit card” = “bad credit”, etc…

Hashcode: When we query the hash code for two anagrams, we want to ensure they are equal. Also, hash codes for non-anagrams should be distinct. However, we can tolerate some collisions for non-anagrams.

getCollision(int hashIndex): You need to write a method getCollision() to return the number of collisions (i.e., chain length) of the specified hash table index (0 <= hash index < N).

The input for this part is a sequence of words/phrases. The maximum number of characters for a word/phrase is 32. You need to hash those words/phrases into a hash table of size N. In a normal well-designed hash table, each hash index chain should have the same number of entries. In this case, however, a hash index chain should ideally have only anagrams of the same letters or only non-anagrams. It depends on the hash code function with which you accomplish this.

Competition: We will run a competition across the whole class of CS251. The program(s) producing the least number of collisions for non-anagrams and having no anagrams mixed with non-anagrams in a hash table entry wins!

Input /output file for part 1 and part 2:

Input format: Use String as key type and Integer as value type

- The first integer 1 or 2 indicates this is the input file for part 1 or 2;
- The second integer n implies that there are n lines below;
- The character ‘p’ (for put) followed by: key value. Call put(key, value). put this key, value pair to hashtable. Print previous value of the specified key in this hashtable, or null if it did not have one

- The character ‘g’ (for get) followed by a key. Call get(key) and print the value of the key. Print “null” if key is not found.
- The character ‘r’ (for replace) followed by: key value. Call replace(key, value); Print previous value associated with the key. Print “null” if there was no mapping for the key;
- The character ‘d’ (for remove) followed by a key. Call remove(key). Print previous value associated with the key. Print “null” if there was no mapping for the key;
- The character ‘s’ (for size). Call size(). Print hashtable size()
- The character ‘c’ (for containsKey) followed by a key. Call containsKey(key). Print 1 if true, 0 if false.
- The character ‘m’ (for getCollision) followed by a integer index i. Print the chain length at that hash index -- if empty, then print 0.

Input	Expected output
1 10 p key1 5 g key1 g key2 r key2 7 r key1 7 s c key1 c key2 p key1 6 m 2	null 5 null null 5 1 1 0 7 1

Note: the parameter 2 for m (getCollision) is a made-up number which happens to be hash index for key1 in this case.

Part 3: Heap (40 points)

A ***heap*** is a binary tree *T* that stores a collection of elements with their associated keys at its nodes and that satisfies two properties:

(1) ***Heap-Order Property***: In a heap *T* , for every node *v* other than the root, the key associated with *v* is greater than or equal to the key associated with *v*’s parent;

(2) ***Complete Binary Tree Property*:** A heap *T* with height *h* is a ***complete*** binary tree, that is, levels 0,1,2,...,*h*−1 of *T* have the maximum number of nodes possible (namely, level *i* has 2^i nodes, for $0 \le i \le h - 1$) and the nodes at level *h* fill this level from left to right.

In this section you need to

(a) implement priority queue with an array-based heap.

(b) implement heap sort.

(a) Implement priority queue

The array-based binary tree representation is especially suitable for a complete binary tree T . We recall that in this implementation, the nodes of T are stored in an array A such that node v in T is the element of A with index equal to the level number $f(v)$ defined as follows:

- If v is the root of T, then $f(v)=1$
- If v is the left child of node u, then $f(v) = 2f(u)$
- If v is the right child of node u, then $f(v) = 2f(u)+1$

With this implementation, the nodes of T have contiguous indices in the range [1, n] and the last node of T is always at index n, where n is the number of nodes of T .

Here is the ADT of a priority queue P:

size():	Return the number of elements in P.
empty():	Return true if P is empty and false otherwise.
insert(e):	Insert a new element e into P.
min():	Return a reference to an element of P with the smallest associated key value (but do not remove it); an error condition occurs if the priority queue is empty.
removeMin():	Remove from P the element referenced by min(); an error condition occurs if the priority queue is empty.

Your implementation should include functionalities listed above.

(b) **Implement heap sort**

Heap sort include two phrases: construct a heap which takes O(n log n) operations, and destruct that heap which takes O(n log n) operations. To construct a heap, you need to iteratively read an entry *e* from the input file and use *insert(e)* to extend the heap. To destruct a heap, you need to iteratively call *removeMin()* until the heap is empty. Sorted items should be ascending order.

Input format:

- The first integer 3 indicates this is the input file for part 3;
- The second integer n implies that there are n lines to follow;
- The character ‘i’ (for insert) followed by 1 integer. Insert this value into the heap;

- The character ‘m’ (for minimum). Call min() and print the value of the returned element. Print “empty” if heap is empty;
- The character ‘r’ (for delete). Call removeMin(); Print “empty” if heap is empty;
- The character ‘s’ (for sorted). Print the sorted contents of current heap.

Input	Expected output
3 5 i 7 i 5 i 3 r m	5
3 7 m i 5 i 4 i 3 i 2 i 1 s	empty 1 2 3 4 5

Part 4: Programming Environment and Grading

Assignments will be tested in a Linux environment. You will be able to work on the assignments using the Linux workstations in HAAS and LAWSON (use your username and password).

4A: Java

- Compiler we use: javac (v10.0.2)
- File to submit: HashTable.java, Heap.java

Your project must compile using the javac compiler (v10.0.2) on data.cs.purdue.edu. Main.java is provided to you so that you can generate output file on your own to see if your code works correctly, you may change it as your wish. We will run JUnit tests which looks like test/TestHashTable.java to grade your work. However, not all grading test cases are provided. You may write your own JUnit test cases.

Compiling process: **Following commands assume you are at project root and you have NOT changed project file structure!**

- To compile Main:
  javac src/*
- To run Main:
  java -cp src/ Main InputFilePath OutputFilePath
  Note: You should replace InputFilePath OutputFilePath with actual file path in the above command.

- To compile JUnit Tests:
  javac -cp src/:lib/* test/*
- To run All JUnit Tests: You can choose one of two commands below
  1. java -cp src/:lib/*:test/ TestRunner
  2. java -cp src/:lib/*:test/ org.junit.runner.JUnitCore TestSuite
- To run Single JUnit Test:
  java -cp src/:lib/*:test/ org.junit.runner.JUnitCore TestHashTable
  java -cp src/:lib/*:test/ org.junit.runner.JUnitCore TestHeap

Grading process:

1. Compiling your program with JUnit test files using javac v10.0.2.
2. Running JUnit tests. It read input files and manipulates data structures implemented by you and see if it works as expected. Any forms of difference between you output and expected output will cause points deduction.
3. Inspecting your source code.

4B: C++

Compilation will be done using g++ and makefiles. You must submit all the source code as well as the Makefile that compiles your provided source code into an executable named “program”.

Your project must compile using the standard g++ compiler (v 4.9.2) on data.cs.purdue.edu. For convenience, you are provided with a template Makefile and C++ source file. You are allowed to modify such file at your convenience as long as it follows the I/O specification. Note some latest features from C++14 are not available in g++ 4.9.

The grading process consists of:

1. Compiling and building your program using your supplied makefile.
2. The name of produced executable program must be “program” (must be lowercase)
3. Running your program automatically with several test input files we have pre-made according to the strict input file format of the project and verifying correct output files thus follow the above instruction for output precisely – do not “embellish” the output with additional characters or formatting – if your program produces different output such as extra prompt and space, points will be deducted.

4. Inspecting your source code.

Input to the programming projects will be via the command line:

program input-test1.txt output-test1.txt

The file output-test1.txt will be tested for proper output.

Important:
1. If your program does not compile, your grade will be 0.
2. Plagiarism and any other form of academic dishonesty will be graded with 0 points as definitive score for the project and will be reported to the corresponding office.

Part 5: Submit Instructions

The project must be turned in by the due date and time using the turnin command. Follow the next steps:

1. Login to data.cs.purdue.edu (you can use the labs or a ssh remote connection).
2. Create a directory named with your **username** and copy your solution there.
   C++: makefile, all .cpp, .h files.
   Java: HashTable.java, Heap.java
3. Go to the upper level directory and execute the following command:
   turnin -c cs251 -p project3 your_username
   (Important: previous submissions are overwritten with the new ones. Your last submission will be the official and therefore graded).
4. Verify what you have turned in by typing **turnin -v -c cs251 -p project3**
   (Important: Do not forget the -v flag, otherwise your submission would be replaced with an empty one). If you submit the wrong file you will not receive credit.