**Data Structures and Algorithms, Spring '23 section1 Dr. Hristescu**

# Lab 12



Assigned: Thursday, April 20
Prelab (Problem 0) due: beginning of lab Thursday, April 20
**Due: 10pm Monday, April 24 -- NO LATE submissions will be accepted**

**All methods have to be implemented AS DESCRIBED. Read and follow the description carefully. Implementations that are different than required will not receive credit.**

**Problem0 (Prelab):**

I. Compute the $n^{th}$ power of 32 efficiently Given a positive integer **n>=0** complete the following line with the following

**int nthpowerof32 =……;**

The solution should:

- be a simple arithmetic expression that does NOT contain any method calls
- contains 1 multiplication operation but no integers other than the number 1 and one other small 1-digit integer (but no 32)

II. Given an upper-case character **c**, complete the line of code that computes the character's mapping to the rank in alphabetic order. Do not use any "magic" numbers other than 1. Your solution has to be **encoding-independent** (not depend on any specific encoding), except for the fact that the characters are assigned subsequent encodings. Your mapping-independent approach should ensure that if tomorrow these characters' Unicode encoding changes to other consecutive mappings, your solution should not require any refactoring.

**Example:** char c= 'D'; // c is initialized to an upper-case letter, ex: 'D'
**int rank =……;** //complete this 1 line of code
System.out.println(rank); //displays 4 if c is 'D', 1 if c is 'A', and 26 if c is 'Z'

**Problem1:**

Write a program that keeps track of a collection of (key,value) pairs. The keys are short strings of alphabet characters from A to Z (**at most 6 chars**). Assume input is always specified using upper-case letters and is correct. Each key will be inserted in the collection with an associated **Integer** type value.

Write an implementation for the collection that uses a hash table.

- For the hash index, use the hash function *h(key) = hashCode(key) % arrayLength* (where arrayLength the length of the primary array (should be a prime number) and the algorithm that uses **Horner's rule** to compute *hashCode(key)*, as described in class and in the textbook and outlined below, to map a string to an integer.
- Resolve collisions by using **separate (linear) chaining**.
- Use Horner's rule:
  1. Map each character in **key** string to the corresponding integer. A maps to 1, B maps to 2, … Z maps to 26 (Problem0 b.)
  2. Compute the hashCode(**key**) = $\Sigma_{i=0 \text{ to size-1}}$ map($char_i$) * $32^{size-i-1}$ (where *size* is the length of the key string)

Example from class:

Mapping the String to an integer by:

1. Mapping the characters to integers: map('A')= 1, map('E')=5, map('T') =20
2. Use Horner's rule to compute *hashCode(key)*

hashCode( "EAT" ) = $5*32^2 + 1*32^1 + 20*32^0 = 5{,}172$
hashCode( "ATE" ) = $1*32^2 + 20*32^1 + 5*32^0 = 1{,}669$
hashCode( "TEA" ) = $20*32^2 + 5*32^1 + 1*32^0 = 20{,}641$

Compute the hash index using h(key) = hashCode(key) % arrayLength;

Compute the powers of 32 efficiently using the code developed for Problem0 a. Do NOT place this in a separate method, but use the expression directly in your code. The given HashTable class uses **3** as the length of the primary array (**arrayLength**). Do not change this value, nor should the integer 3 appear anywhere else in your code except for the given HashTable constructor.

Using ChainNode , HashException and HashTable that implements HashTableInterface . Implement the ChainNode's **toString** method ((key,value) format) and the **5** remaining **methods** in the HashTableInterface **exactly as specified**.

**Problem2:** Develop a test application with the following menu options:

**0.** Exit program.
**1.** **Insert** a (key, associated value) pair in the table.
**2.** **Delete** a pair from the table.
**3.** **Retrieve** and display the value associated with a key in the table.
**4.** **Display** the hashCode of a key**.**
**5.** **Display** the content of the table.

Note:

- For **1. Insert**, if the key has no association in the table, insert the pair before all other pairs/nodes in the chain.
- Unlike Java HashMap's **put** method, tableInsert rejects insertion if the key is already in the HashTable, i.e. tableInsert does not overwrite an existing association for a key. In this implementation the old association will need to be explicitly deleted by the user via a tableDelete before a new association can be inserted.

Use an associated value of type **Integer**. Your driver should declare the hashtable variable to be of type **HashTable<String, Integer>**. Test tableInsert by inserting at least 15 different keys (with associated values) and then deleting some of them to test tableDelete. Suggested (key,value) pairs to **start** your testing for tableInsert with ("ATE", -7), ("EAT", 5),("TEA",1000),("GRADE",100),("DSA",10), ("CAT",67), (DOGGY",100), ("CHAIRS",-45),("EAT",76), ("LETTER",6),( "LETTER",34), ("GRADE",0),("HASH",88),("EAT",34). **Continue by adding at least 6 more pairs of your choice, including existing keys (no change to table)**. Thoroughly test tableDelete.

**Problem3:** Lab12Conclusions: Summarize in a few sentences what you have learned from working on this lab.

**Submit:** Follow the instructions to turn in the lab electronically using **Lab12** for the name of your directory. Develop your own extensive input suite and test your solutions thoroughly on the provided data as well as your own additional input input. Submit all the code developed for this lab and all sample runs on the input suite you developed.