CS202, Fall 2023

Homework 4 - Balanced search trees and hashing

Due: 13/12/2023

Before you start your homework, please read the following instructions carefully:

FAILURE TO FULFIL ANY OF THE FOLLOWING REQUIREMENTS WILL RESULT IN A GRADE SCORE OF 0 (zero) WITHOUT ANY CHANCE OF REDEMPTION.

- See the course page for late submission policies and the Honor Code for Assignments.
- This assignment has two questions; please read all pages carefully.
- Upload your solutions in a single ZIP archive using the Moodle submission form. Name the file as studentID_name_surname_hw4.zip. Please use only English characters when naming your file. Your ZIP archive should contain **only** the following files:
    - **studentID_name_surname_hw4.pdf**, the file containing the answers to Questions 1.
    - **main.cpp, KmerNode.h, KmerNode.cpp (optional), HashTable.h, HashTable.cpp** files, which contain the C++ source codes and the **Makefile**. Do not add unnecessary files like the auxiliary files generated from your preferred IDE.
    - Do not forget to put your name, student ID, and section number in these files. Comment on your implementation well. Add a header (see below) to the beginning of each file:

```
/**
 * Title: Hash Tables
 * Author: Name & Surname
 * ID:
 * Section:
 * Homework: 4
 * Description: description of your code
 */
```

- Your code must compile and be complete.
- Your code must run on the **dijkstra.cs.bilkent.edu.tr** server.
- You can submit your answer for Question 1 as scanned files, but ensure the answers are legible. You may lose points if the handwriting or drawings are unclear.
- For any questions related to the homework, you can send an email to your TA: *zulal.bingol@bilkent.edu.tr* with the subject line "CS-202: HW-4". Please follow the homework discussion forum on Moodle to be up to date with your possible questions.

Question 1 (30 points)

Assume that you have the following ***balanced-searched tree*** implementations:

a) **AVL tree**
b) **2-3 tree**
c) **2-3-4 tree**

Starting with an empty balanced search tree, insert the following keys into the tree in the ***given order***:

60, 96, 4, 6, 35, 78, 5, 7, 82, 99, 20

After inserting, delete the following keys from the tree in this order: 96, 82, 20, 5.

Note: while deleting an internal node, its in-order successor should be used as the substitute if needed.

Show the underlying tree for the AVL, 2-3, and 2-3-4 implementations after each insertion and deletion.

Question 2 (70 points)

For this question, you will use hash tables to store the k-mers you created for Homework 2. A k-mer is defined as *k* consecutive non-overlapping characters in a given text. Assume that the input text contains only English letters 'a'...'z', 'A'...'Z' and has no blank space. All k-mers should be stored as lower-case letters only. Please refer to Homework 2 to see an example of kmer.

In this homework, you will use two different hash collision techniques (separate chaining and quadratic probing) to store the kmers and their starting positions (0-indexed) in the input text in two different hash tables and compare the time taken for hash table retrievals.

Example: Input text = “FuzzyWuzzywasabearFuzzyWuzzy”
→ All of the 3-mers generated from this text should be as follows:
‘fuz’, ‘zyw’, ‘uzz’, ‘ywa’, ‘sab’, ‘ear’
→ The positions of the k-mers:
fuz: (0, 18), zyw: (3, 21), uzz: (6, 24), ywa: (9), sab: (12), ear: (15)

Please carefully read the following explanations for the details:

KmerNode Class (6 points):

Define a KmerNode class with at least three fields: one for the kmer (string) and one for a vector (integer) that stores the starting positions of these specific kmers in the input text. You will need one extra field for a pointer for the separate chaining strategy. You don’t need to define two different KmerNode classes; you can use the same class for both hash table versions. Your hash tables will store these KmerNode objects, not only the kmers as strings.

<u>HashTable Class (60 points):</u>

Define a HashTable class and implement the generic functions of the class, such as a constructor and a destructor. Your hash table should store KmerNode objects. The hash table will have two functions for separate chaining and quadratic probing. In your HashTable class, define the following functions:

Separate Chaining Functions:

Your hash table's size can be 7 for the operations done with the separate chaining strategy.

- *(8 points)* **`addKmer_chain(string kmer, int position):`** calculates the hash value of the input k-mer, creates a KmerNode object, and adds it into the hash table if not already there; otherwise, it finds the object in the table and adds its position in the text to the positions vector of the k-mer object. The hash function will be

  $h(x)$ = *(sum of the ASCII codes of each letter in the k-mer)* mod *(hashtable size)*

  If there is a hash collision, meaning that if there is already *another* k-mer object with the same hash value in the hash table, this function creates a KmerNode object for the new k-mer and adds the new node to the end of the linked list associated with this hash value.

- *(8 points)* **`fillHashTable_chain(string filename, int k):`** reads the input text from a file, and fills the hash table with KmerNode objects. In this function, you should detect all the k-mers in the input text and add them to the table using the `addKmer_chain` function (see above). This function also requires the parameter *k*.
- *(8 points)* **`KmerNode* findKmer_chain (string kmer)`**: calculates the hash value of the input k-mer, finds the KmerNode object, and returns it.

Quadratic Probing Functions:

Your hash table's size can be 71 for the operations done with the quadratic probing strategy.

- *(8 points)* **`addKmer_quadratic(string kmer, int position):`** calculates the hash value of the input k-mer, creates a KmerNode object, and adds it into the hash table if not already there; otherwise, it finds the KmerNode object in the table and adds its position in the text to the positions vector of the object. The hash function will be

  $h(x)$ = *(sum of the ASCII codes of each letter in the k-mer)* mod *(hashtable size)*

  If there is a hash collision, meaning that if there is already *another* KmerNode object with the same hash value in the hash table, this function creates a KmerNode object for the new k-mer, uses quadratic probing to calculate a new hash value, and adds the new node to the hash table with the new hash value.

- *(8 points)* **`fillHashTable_quadratic(string filename, int k):`** reads the input text from a file, and fills the hash table with KmerNode objects. In this function, you should detect all the k-mers in the input text and add them to the table using the `addKmer_quadratic` function (see above). This function also requires the parameter *k*.
- *(8 points)* **`KmerNode* findKmer_quadratic (string kmer)`**: calculates the hash value of the input k-mer, finds the KmerNode object, and returns it.

General Functions:

- ***(4 points)*** **`void printAll():`** prints each KmerNode object stored in the hash table and their positions. You can have different versions of this function for two hash collision strategies. In this case, you are allowed to create two different function names. Alternatively, you can have a single generalized function.
- ***(8 points)*** **`void go_to_positions(string inputFileName, int range, KmerNode *found):`** In general terms, this function accepts a KmerNode object, goes to the k-mer's positions in the input text file one by one, and prints out excerpts from the input text extending to the range given. You can assume that one of the relevant findKmer functions will be called before this function. Following the example above, the output of this function for 'zyw' with range = 2 will be
  ⟶ zyWuz (generated from position 3)
  ⟶ zyWuz (generated from position 21)

<u>main.cpp (4 points):</u> Write a main function to test your program. Create two different hash tables: one with the separate chaining strategy and one with the quadratic probing strategy. Fill both hash tables with the same content using the large text file you used in hw2 (Please refer to HW-2 for the details). Finally, choose a k-mer from your tables and call the **`findKmer_chain(string kmer)`** and **`findKmer_quadratic(string kmer)`** functions for the <u>same k-mer</u>. Measure and report the time to retrieve this specific k-mer in these different hash collision resolving strategies.

Implementation details: *(0 points, but mandatory)* In the end, write a basic **Makefile** that compiles all your code and creates an executable file named **`hw4`**. Your program should take two command-line arguments: `<k>` and `<input_file>` in the same order. Please ensure your **Makefile** works properly; otherwise, you will not get any points from Question 2. Put the implementation of your functions in **`KmerNode.cpp`** (optional) and **`HashTable.cpp`**, and their interfaces in **`KmerNode.h`** and **`HashTable.h`**, respectively. Do not include your main function in these files. Instead, submit your main function inside another file called **`main.cpp`**. Use the class and function names exactly as listed above (except for printAll function, read above). Your homework will be tested with different scenarios, so ensure all your functions work correctly. You are not given an example test code this time, so you should report your results with clear formatting in your main.