Programming Assignment 6 (Trie, and Generic Dynamic Heap)

Department of Computer Science, University of Wisconsin – Whitewater
Data Structure (CS 223)

1 Overview

We are essentially going to:

- **Implement Insertion and Search in a Trie**
- **Implement Insertion and Deletion in a Generic Dynamic Heap**
- **Implement Heap Sort** and **Find the top-k relevant data**

To this end, **your task is to compelte the following class and methods**:

- `locus`, `insert`, and `search` methods in `Trie.h`/`Trie.java`
- complete the class in `StudentComparator.h`/`StudentComparator.java`
- `insert` and `extract` methods in `GenericHeap.h`/`GenericHeap.java`
- `readStudents` in `Student.h`/`Student.java`
- `heapSort` and `topK` methods in `HeapApplications.h`/`HeapApplications.java`

The project also contains additional files (which you do not need to modify).
Use `TestCorrectness.cpp`/`TestCorrectness.java` to test your code.
Output is provided separately in the `ExpectedOutput` file. Should you want, you can use `www.diffchecker.com` to tally the output.

2 Operations on Strings

In this assignment, you are going to be dealing with strings. You have already dealt with strings in the previous assignment, but over there, you read characters and ordered characters based on string order. Here, you are going to:

- Get hold of the character at index i of a string.

  If the string is str, use $str.at(i)$ in C++ and $str.charAt(i)$ in Java to get the character.

- Compare two strings $str1$ and $str2$ to get their relative lexicographic (dictionary) rank.

  Use $str1.compare(str2)$ in C++ and $str1.compareTo(str2)$ in Java to get their relative order.

  If the function returns a negative value, then $str1$ is smaller than $str2$. If the function returns a positive value, then $str1$ is larger than $str2$. If the function returns 0, then $str1$ equals $str2$.

3 Trie

You are going to use a trie to build a spell checker. The idea is that you build a trie out of all the words in the dictionary by inserting them one-by-one. Assume that words end in the sentinel $ to avoid a prefix match. Otherwise, if you insert `abcd`, every prefix of `abcd` will yield a match. Now, given a sentence, you tokenize to get all the words. Then, you match each word in the trie; if you get a mismatch that word does not exist in the dictionary, and hence, is an incorrect spelling.

The TrieNode class depicts each node in the trie, where each node is equipped with a hashtable. Say you are at a node u and v is a child of u, such that the edge label from u to v is the character C. Then the hashtable at u will hash C to the node v. TrieNode contains the following two fields:

- `edges`: This hash table maps a parent TrieNode to its child(ren) TrieNode(s) based on the character(s) labeling the edge(s).
- `depth`: The depth of the root node is 0, and the depth of any other node is one more than the depth of its parent.

Now, onto more details about the hash table structure.

3.1 Hashtable

A hash table entry comprises of a *key* and *value* pair, whereby you can search the hash table for a particular key, and if it exists, retrieve the corresponding values back. You are not required to create a hash table as they have already been created, but it is good to know about them. For your code, *ht* is the `edges` field in a `TrieNode` object.

C++. To create a hash table object *ht* whose keys are of type *char* and values are of type *TrieNode* pointers, the syntax is unordered_map<char, TrieNode*> ht. What you will need:

- To insert a value v with key k, the syntax is ht[k] = v;
- To check whether or not *ht* contains a key k, the syntax is: if(ht.find(k) != ht.end()). The statement inside the if evaluates to *true* if *ht* contains k.
- To retrieve the value v corresponding to a key k, the syntax is: TrieNode *v = ht[k]; Note that the key k (, and the corresponding value) may not exist in the hash table; hence, the returned value may be NULL.

Java. To create a hash table object *ht* whose keys are of type *char* and values are of type *TrieNode*, the syntax is Hashtable<Character, TrieNode> ht = new Hashtable<Character, TrieNode>(). What you will need:

- To insert a value v with key k, the syntax is ht.put(k, v);
- To check whether or not *ht* contains a key k, the syntax is: if(ht.containsKey(k)). The statement inside the if evaluates to *true* if *ht* contains k.
- To retrieve the value v corresponding to a key k, the syntax is: TrieNode v = ht.get(k); Note that the key k (, and the corresponding value) may not exist in the hash table; hence, the returned value may be null.

3.2 Pseudo-code

Recall that either during insertion, we match the argument string in the trie as long as we do not get a mismatch or until we have read the entire string. If we have read the entire string, then there is nothing new to insert. If we cannot match completely then we insert the remaining characters one at a time. In the latter case, we say the last node that was reached after a successful match is the *locus*. Your first task is find the locus.

Finding the Locus

- Set a TrieNode temporary variable *tmp* to the *root*. (In C++, *tmp* is a pointer.)
- for ($i = 0$ to $i < str.length()$), do the following:
  - Let c be the character at position i of *str*.
  - Let *child* be the child of *tmp* such that the edge from *tmp* to the *child* is labeled by c; use the hashtable *edges* stored in *tmp* to get hold of *child*.
  - If the child is null, then *tmp* is the locus; so return *tmp*.
  - Else, move *tmp* to *child*
- The entire string is matched at this point; so, the locus is given by *tmp*; return it.

Now onto insertion.

Insertion

- Let *parent* be the locus of *str*; let *depth* be its depth.
- Starting a for-loop from index *depth* until the end of the string, do the following:
  - Let c be the character at the index of *str* given by the for-loop counter.
  - Create a new TrieNode *child* whose depth is one more than the loop counter.
  - Insert this newly created node as the child of *parent* by using the hashtable *edges* stored at *parent*. Specifically, insert the *child* as value into *edges* keyed by c.
  - Move *parent* to *child*:

Finally searching a string.

Matching a String

A string is found if the string is entirely matched, which implies that a string is found if the locus has the same depth as the length of the string itself, else it is not found.

4 Generics & Comparators

In this assignment, you are going to use a heap to order a bunch of comparable items. To ths end, you are going to use a generic heap coupled with an appropriate comparator. The idea is that you can create a heap of items of any sort (and not just integers), where an item can be compared against another using a comparator. A couple of examples:

- You will order strings in a heap. For this, you will use the standard technique by which strings are ordered.
- You will order students, each having a string name and an integer grade. Students will be ordered based on their grades, and in case of a tie, based on their names.

Thus, for the price of one heap, you can create two (and in fact, infinitely many by just using an appropriate comparator). Question is how we use generics and comparators.

I'll sketch the idea using Selection Sort; check the given `GenericSelectionSort` file. You need to understand that to implement the heap by using pretty much the same concept; the code has been reasonably commented to help you understand.

- The `GenericSelectionSort` class can sort any collection of items as long as they are comparable. The class accepts the type of items that you want to sort as arguments, as well as a comparator using which you will sort the items.
- To see how the comparator works, check `StringComparator` class. This method simply returns the integer obtained by comparing two strings. You'll use the idea to complete the `StudentComparator` class as follows:

  Student Comparator

  Here's how you compare Student $arg1$ and $arg2$

  - If $arg1$'s grade does not equal $arg2$'s grade, return $arg1$'s grade – $arg2$'s grade
  - Otherwise, return the value obtained by comparing $arg1$'s name to $arg2$'s name

  Do not deviate from the structure given `StringComparator`; usage of comparators needs the exact format prescribed there

- We compare the two objects using `comparator(currentValue, minValue)` in C++ and `comparator.compare(currentValue, minValue)` in JAVA. This will return us a negative number if $currentValue$ is smaller, else it will return us a non-negative number. Notice the obliviousness of the selection sort method about how the objects are compared. All it cares is that the objects are comparable, and presumes that the comparator is correct.

5 Generic Dynamic Heap

Use the notes posted on Heap to implement the `insert` and `extract` methods in the `GenericHeap.h`/ `GenericHeap.java` files. Instead of implementing a standard heap (with a given maximum capacity), we shall use a dynamic array to implement a (dynamic) heap, i.e., we do not make any assumptions on the size of the heap. Therefore, at the class level, you have a *heapArray* variable, which is a dynamic array (vector in C++ and ArrayList in Java). Additionally, since the heap is generic, we replace `getMinimum` with the more generic term `top`; likewise, we replace `deleteMinimum` with the more generic term `extract`.

Notice the generic nature of the heap – not only can you create heaps of any objects, but also by using an appropriate comparator, you can convert the default min-heap nature to a max-heap. This is how inbuilt Java and C++ heaps (known as priority queues) are implemented; we will do this in a future assignment, but this should be a nice warm-up.

To this end, adapt and use the pseudo-code in the Canvas notes by making the following changes:

Accessing, Inserting, Deleting, and Swapping Contents of Heap-array

- You can obtain the current size of the heap using the $size()$ method. The last value in the heap is at index $size() - 1$.
- To read a value of heapArray at index i use the following syntax:

  C++: heapArray.at(i)

  Java: heapArray.get(i)
- To insert a new value v use the following syntax:

  C++: heapArray.push_back(v)

  Java: heapArray.add(v)
- To delete the last value in the heap, use the following syntax:

  C++: heapArray.pop_back()

  Java: heapArray.remove(size() - 1)
- To update the value at index i to the value v, use the following syntax:

  C++: heapArray.at(i) = v

  Java: heapArray.set(i, v)
- To swap the contents of the heap at two indexes x and y, call the swap method with arguments x and y, i.e., `swap`(x, y)

Comparing items in Heap-array for swapping

Finally, remember that we are going to store generic objects in the heap; hence, you cannot compare the values using $<$ or $>$ symbol. Instead, you need to compare them using comparators as discussed in the beginning of this document.

Note that $leftKey$, $rightKey$, and $currentKey$ are of generic objects of type T. Say we are comparing $leftKey$ and $rightKey$. To check if $leftKey$ is smaller than $rightKey$, use the following syntax. Use the same ideas to compare other values in the heap.

C++: `if (comparator(leftKey,rightKey) < 0)`
JAVA: `if (comparator.compare(leftKey,rightKey) < 0)`

6 Sorting Strings Using Generic Heap

Our first application of a heap is to sort an array of strings, i.e., order them lexicographically (dictionary order). Normally you could sort strings using a selection sort kind of approach. However, that would be bad (just like sorting numbers using them is bad). To see why that is the case, let's consider the scenario where we have m strings, each of length N characters. Suppose we want to sort them using selection sort. We know that will have a nested for-loop, one going over each string

and other to find the minimum string. However, we will have another loop (either explicitly if you write it or implicit if you use the string compare method discussed before); this loop will run N times to find the order between two strings.[1] Therefore, the complexity is ultimately $O(m^2N)$; it would be the same in case of insertion sort.

Using a heap, we can reduce this complexity down to $O(mN \log m)$. Let's see why. You will have at most m nodes in the heap (one for each string), and to compare a string in a node with its parent or children, you will need $O(N)$ time. Therefore, *insert* and $deleteMin$ operations costs you $O(N \log m)$, whereas $getMinimum$ is still an $O(1)$ time operation. To sort, you will carry out m of these operations each. Hence, the overall cost is $O(mN \log m)$, which is a substantial improvement.

Following is the pseudo-code. Use it to complete the `heapSort` method of the `HeapApplications.h`/ `HeapApplications.java` file.

Heap Sort

- Create a string heap via a constructor call to the `GenericHeap` class; use the `StringComparator` in this case. Refer to the `GenericSelectionSort` class method `stringSort` for usage.
- Insert all the strings of the array into the heap using a for loop.
- for ($i = 0$ to $i < arrayLen$), do the following:
  - set $array[i]$ to the smallest/topmost string in the heap
  - extract the smallest/topmost from the heap

7 Finding k-most important items

In many scenarios, we are interested to find the top-k numbers (i.e., the k highest numbers) from a given set of numbers. This is has numerous applications related to web searching (such as the 10 most relevant websites for a Google search, or the 20 most popular items for an Amazon search). The obvious way to do this would be to first sort the array, and then report the last k numbers in the sorted array. This, however, has a complexity of $O(n \log n)$, where the length of the array is n. Typically, k is much smaller than n; hence, we want to design an algorithm that is faster for smaller values of k (such as when k is a constant), but no slower than sorting for larger values of k (such as when k approaches n).

Specifically, our goal is to *design an algorithm that can find the k largest items in an array of n items*, where each item can be compared to another based on some criteria. Our algorithm will achieve a complexity of $O(n \log k)$ time, assuming any two items can be compared in $O(1)$ time (if that's not the case, we have to multiply the time needed to compare two items).

The main idea is as follows. Note that the we need the k largest items, and the complexity has a $\log k$ factor; so, immediately, you can guess that the size of the heap should not exceed k. So, what you do is insert the first k items in the array into the heap.

Now, think of the next item in the array; if it is larger than the smallest in the heap then the smallest in the heap cannot be one of the k largest items. On the other hand, if the next item is smaller than the smallest in the heap, then the next item cannot be one of the k largest items.

[1] Recall how you compare strings lexicographically. You match the strings one character at a time, until you find a mismatch (or exhaust one of the strings). The lexicographically smaller string is the one with the smaller mismatched character (or the shorter one if you exhaust one string).

Therefore, either way we can discard one of them – in the first case, remove the minimum and insert the next one, whereas in the second case, ignore the next one. Keep doing this for the remaining items in the array and the current heap. At the end, the items left in the heap form the k largest.

Use the following pseudo-code and complete the `topK` method of the `HeapApplications.h`/ `HeapApplications.java` file, which finds the k-largest items and returns them in an array. Note that the output array must be correct, and you must achieve the claimed complexity for full credit. Here, items are objects of the `Student` class.

Read Students

Complete the `readStudents` method in the `Student` class. This method will read the student at the given filepath into a dynamic array (vector in C++ and ArrayList in Java) of type Student. Then it will return the dynamic array.

The `students.txt` file has a student record in each line. Each record contains the student's name followed by the student's grade, separated by a space.

topK

- If k is more than the size of *students*, then set k to the size of *students*
- Create a Student heap via a constructor call to the `GenericHeap` class; use the `StudentComparator` in this case. Refer to the `GenericSelectionSort` class method `studentSort` for usage.
- Create a `StudentComparator` object.
- Insert the first k students into the heap.
- for ($i = k$ to $i < size\ of\ students$), do the following:
  - let *min* be the topmost student in the heap
  - let *current* be the student at index i of *students*
  - Use the student comparator object to compare *min* and *current*. If (*min* is smaller than *current*), then:
    - extract the topmost student from the heap
    - insert *current* into the heap
- Create a Student dynamic array (vector in C++ and ArrayList in Java).
- At this point, notice that the heap contains the top-k students. Use `top` and `extract` to extract the students in the heap and add them into the dynamic array.
- Return the dynamic array.