CS-202
Data Structures
Assignment 3
Hashing, Heaps, Sorting, Tries

Due: 11:55 pm on Friday, Apr 02, 2021

In this assignment, you are required to implement a Hashing Class through Chaining and Linear Probing, a Min Heap, and the Insertion Sort and Merge Sort algorithms. The last part requires that you implement a trie data structure. You may test your implemented data structures using the test cases provided to you.

Plagiarism Policy

The course policy about plagiarism is as follows:

1. Students must not share the actual program code with other students.
2. Students must be prepared to explain any program code they submit.
3. Students cannot copy code from the Internet.
4. Students must indicate any assistance they received.
5. All submissions are subject to automated plagiarism detection. Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee.

Late Submission Policy

You are allowed 5 "free" late days during the semester (that can be applied to one or more assignments; the final assignment will be due tentatively on the final day of classes, i.e., before the dead week and cannot be turned late. The last day to do any late submission is also the final day of classes, even if you have free late days remaining). If you submit your work late for any assignment once your 5 “free” late days are used, the following penalty will be applied:

- 10% for work submitted up to 24 hours late
- 20% for work submitted up to 2 days late
- 30% for work submitted up to 3 days late
- 100% for work submitted after 3 days (i.e., you cannot submit your assignment more than 3 days late after you have used your 5 free late days)

Smart Pointers

In order to implement this assignment you must make use of smart pointers. Since certain functions require specialized syntax, a tutorial code, **SmartPointer.cpp**, has been shared with you. Within this tutorial file, you can explore the necessary syntax for the equivalent of the methods you would have used in raw dynamic pointers, which will be necessary in certain parts of this assignment.

PART 1: Hashing

Task 1:
Hashing Function

In this task you will be implementing a bitwise hash function as shown below. Write your implementation in the **HashFunction.cpp** file which also provides the required definitions.

Bitwise Hash:

```
Initialize bitwise_hash = 0
For every $s_i$ in str
bitwise_hash ^= (bitwise_hash << 5) + (bitwise_hash >> 2) + $s_i$
```

Required Functions:
Write the implementation for the following functions as described here:

unsigned long **bitHash**(string value)

- Takes a string and produces a corresponding hash key based on the bitwise hash operation described earlier.

unsigned long **divCompression**(unsigned long hash, long tableSize)

- Takes a hash key and compresses it to fit within the range bounded by the size of the hash table.

Task 2:
Chaining

In this task you will be implementing a hash table of a fixed size that uses chaining to resolve any collisions. Refer to **Chaining.hpp** which provides all the required definitions. Write your implementation in **Chaining.cpp** using the boiler code. In order to implement the chaining aspect, you will need to use the provided **LinkedList** implementation.

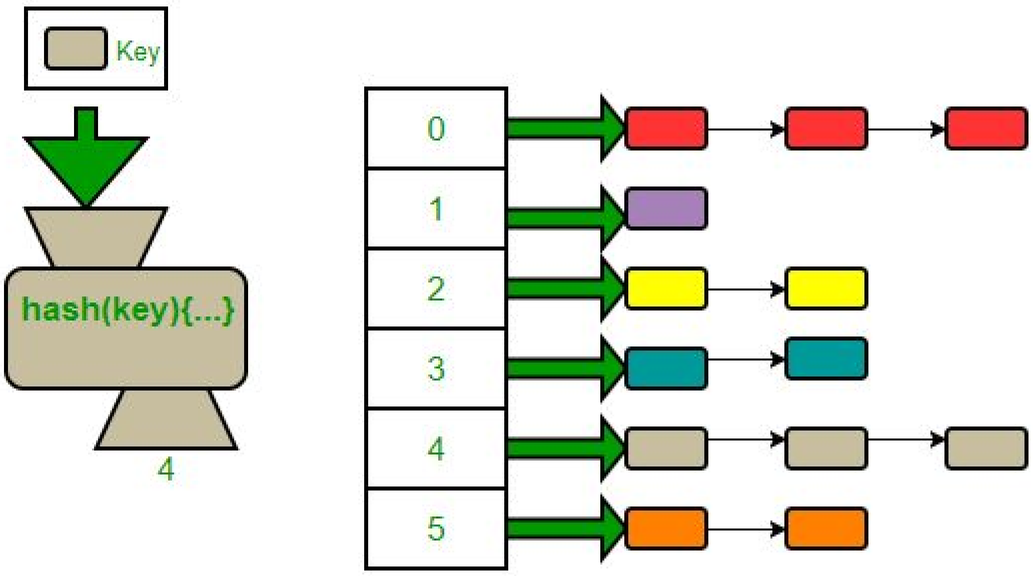


Figure 1: Chaining

Member Functions[1]:

Write the implementation for the following functions as described here:

HashC (int size)

- Constructor that creates a hash table of the given size.

unsigned long **hash**(string input)

- Returns a hash key corresponding to the given string. Make use of the functions you wrote in **HashFunction.cpp** to implement this method.

void **insertWord** (string word)

- Inserts the given word into the hash table.

shared_ptr<ListItem<string>> **lookupWord**(string word)

- Finds the given word in the hash table and returns a pointer to it. **NULL** is returned if the word does not exist.

void **deleteWord**(string word)

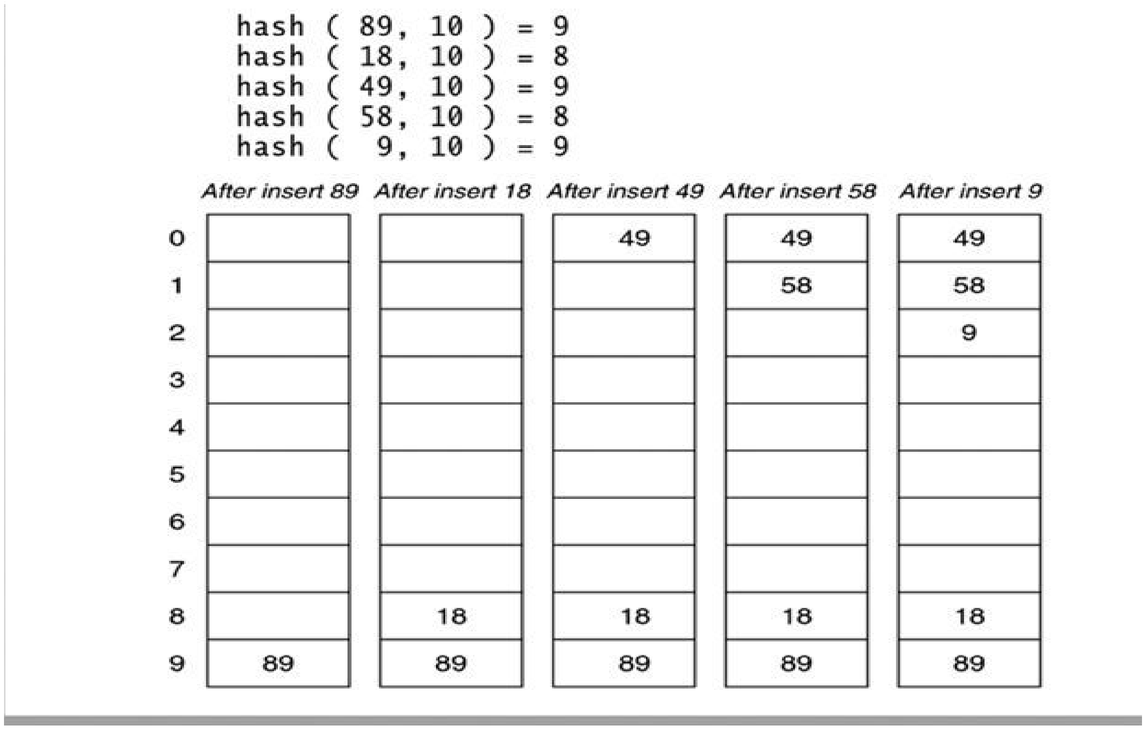
- Deletes the given word from the hash table if it exists.

[1] In this assignment, you are free to declare any helper functions as per your need but the already declared function declarations should not be modified.

Task 3:
Linear Probing

In this task you will be implementing a hash table using open addressing with linear probing. Refer to **LinearProbing.hpp** which provides all the required definitions. Write your implementation in **LinearProbing.cpp** using the boiler code.

$$h(k,n) = k \% n$$



```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	After insert 18	After insert 49	After insert 58	After insert 9
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Figure 2: Linear Probing

Member Functions:

Write the implementation for the following functions as described here:

HashL (int size)

- Constructor that creates a hash table of the given size.

unsigned long **hash**(string input)

- Returns a hash key corresponding to the given string. Make use of the functions you wrote in **HashFunction.cpp** to implement this method.

void **resizeTable**()

- Resize the hash table once the load factor becomes too large.

void **insertWord** (string word)

- Inserts the given word into the hash table.

shared_ptr<ListItem<string>> **lookupWord**(string word)

- Finds the given word in the hash table and returns a pointer to it. **NULL** is returned if the word does not exist.

void **deleteWord**(string word)

- Deletes the given word from the hash table if it exists.

PART 2: Heaps

In this part you will be implementing a Min Heap data structure. Refer to **heap.hpp** which provides all the required definitions. Write your implementation in **heap.cpp** using the boiler code.

Member Functions:

Write the implementation for the following functions as described here:

MinHeap (int cap)

- Simple constructor that creates a MinHeap of the given capacity.

void **insertKey**(int k)

- Inserts the new key into the heap.

void **deleteKey**(int i)

- Deletes the key stored at index i.
- Remember to ensure that the heap structure is maintained after deletion.

int **parent**(int i)

- Returns the index of the parent of the node at index i.

int **left**(int i)

- Returns the index of the left child of the node at index i.

int **right**(int i)

- Returns the index of the right child of the node at index i.

int **extractMin**()

- Returns the minimum element of the heap.
- The element must also be removed from the heap before returning.

void **decreaseKey**(int i, int new_val)

- Decreases key value of key at index i to the new value provided.
- You may assume that new_val will always be less than the current at index i.

int **getMin**()

- Returns the minimum element of the heap.

PART 3: Sorting

In this part of the assignment, you will have to implement two sorting algorithms, namely insertion sort and merge sort. For both of these parts, you have been provided with an unsorted vector of longs as the only parameter to both of these functions respectively. Both of these functions should return a sorted vector of longs upon successful execution. Alongside testing whether your functions successfully return a sorted vector, the test cases will also be testing the time it takes for your functions to execute. If the time taken exceeds the benchmark time provided in the test cases, the test cases will fail. You are allowed to make as many helper functions as you want.

We have provided a file called generator.cpp that generates a sequence of numbers between 1 and n based upon user input. It will generate either sorted numbers or random numbers based upon user input and then ask the user to select the type of sorting algorithm (insertion sort or merge sort) to apply on the sequence of numbers generated. It will then be testing your implementation of insertion sort and merge sort and you shall be able to see the results of your implementation on your screen. However, please note that this file has only been provided to help you visualize the results of your implementation and is not the final test file. Please also keep in mind that your functions should work for negative numbers as well. The final test cases will be testing that.

The header file sorts.h has declarations for all sort functions. You must implement all of these functions in sorts.cpp. You have also been given an implementation of the Linked List data structure in LinkedList.h and LinkedList.cpp files which **must** be used in merge sort. You must write all your code in sorts.cpp only and are not allowed to alter any other files. To see the output of your sorting algorithms, use generator.cpp. Additionally, you have been provided final test cases separately for both the parts as well (explained below) which you **must pass** in order to receive any credit.

Please do keep in mind that it is not compulsory to use generator.cpp at all and you can solve this entire part without using it even once. It has only been provided to aid you with visualizing the output of your sorting functions. However, **it is compulsory** to pass the individual test cases for both insertion sort and merge sort, given in the individual test case files respectively.

Note: You must implement the actual insertion sort and merge sort algorithms as taught in the class. You must also make sure to follow all the instructions related to individual tasks given below, otherwise you may end up losing marks.

Task 1: Insertion Sort

In this task, you will be implementing an array based implementation of insertion sort, as discussed in class. You will first be transferring longs from the unsorted vector to an array, sorting the array and then transferring back elements from this sorted array back to a vector (old or a new one). You will then be returning this sorted vector.

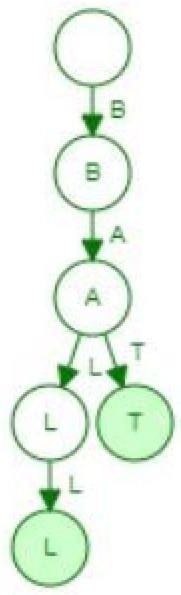
Task 2: Merge Sort

In this task, you will be implementing a Linked List based implementation of Merge Sort. You will first have to transfer all the elements of the provided unsorted vector into a linked list and then sort the elements within that linked list using the merge sort algorithm as taught in class. You will then be transferring back elements from this sorted linked list into a vector (old or new one) and then returning back this sorted vector.

Note: You must make sure that tests for these sorting algorithms pass on a Linux (Ubuntu) machine. We will be testing your implementation on Ubuntu Linux so you will only receive credit if your test cases pass on Ubuntu.

PART 4: Tries

In this part, you will be practically implementing the trie data structure. According to wikipedia, a trie is a type of search **tree**, a **tree** data structure used for locating specific keys from within a set. These keys are most often strings, with links between nodes defined not by the entire key, but by individual characters. In our case, we will be using words from the English language as keys and Alphabets of the english language as individual characters. Each node will be storing a character and a vector of nodes as its children nodes. For example, a trie with keys **bat** and **ball** will look like this:



There are three really important things to note here:

1) At each level, characters are stored in an alphabetical order e.g at Level 3, both L and T are children of A but L must appear before T which is why the word **ball** is stored before the word **bat** even if the word **bat** is inserted before the word **ball.** This is just how a standard trie works

2) For two keys having n same starting characters (where n is an integer > 1), the first n characters are not duplicated while inserting the second key within the trie e.g if ball is inserted as the second key within the trie while bat already exists, the characters ‘b’ and ‘a’ are not duplicated. Instead, it creates a new branch from a

towards the left and stores the first ‘l’ within ball as a child of a towards the left of ‘t’ from bat.

3) While the above image shows only capital letters within the trie, your implementation should work on both capital and small letters. However, you do not need to handle this separately. As long as your functions have been implemented correctly, they will be general enough to handle all sorts of characters (even non-english characters). So you do not need to worry about this.

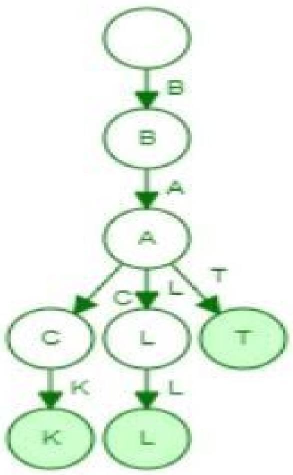
You have been provided **trie.hpp** and **trie.cpp** files. Member function declarations of the trie class as well as the structure for a Node have been provided in the trie.hpp file. You are not allowed to make any changes in the existing function declarations or the node structure otherwise your test cases will not pass. You **must** implement all the member functions declared in trie.hpp. Additionally, you are allowed to create as many helper functions as private members of the trie class.

Member Functions:

Write the implementation for the following functions as described here:

void **insertWord** (string word)

- Inserts a word from the English Language into the trie
- You must make sure that it follows the rules mentioned above after it has been successfully implemented.
- For example when **insertWord(“back”)** is called on the trie shown above, the resultant trie will look like this:



Note: We will not be testing the insertion of two words such that one word is a complete substring of another word. For example after inserting “Hello”, we will not be expecting your implementation to cater to the insertion of “Hell” or Vice versa since “Hell” is a complete substring of “Hello”. However, your implementation should still cater to the insertion of two words such that they both have a common substring. For example your implementation should cater to the insertion of words “Hello” and “Hellish” since both have “Hell” as a substring but none of these two words is a complete substring of each other.

bool **search** (string word)

- Searches for a word within the trie. Returns true if a word is found within the trie, otherwise returns false
- For example, **search(“Ball”)** returns true whereas **search(“Base”)** returns false

string **longestSubstr** (string word)

- Returns the first n characters of the provided argument as a string after searching for the word within the trie. If not even a single character is found within the trie, it returns an empty string.
- For example **longestSubstr(“Base”)** returns “Ba” whereas **longestSubstr(“random”)** returns “”.

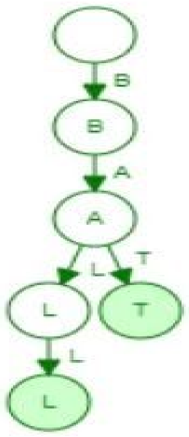
vector<string> **getTrie** ()

- Traverses the entire trie and returns all the words present within the trie, in alphabetical order, in a vector
- For example, applying this function on the above trie will return a vector with the following strings:

  “Back”
  “Ball”
  “Bat”

void **deleteWord** (string word)

- Removes the word provided as an argument to it, from the trie. However, it must ensure not to delete the characters which are common ancestors to several words.
- For example, when **deleteWord(“Back”)** is called on the above trie, the resultant trie will look like this:



- As you can see, the characters ‘B’ and ‘A’ did not get deleted when deleteWord(“Back”) was called because they were ancestors to the words “Ball” and “Bat”.

Helper Functions

While it is highly encouraged to think of helper functions yourself, we would strongly suggest you to make the following helper function before making any of the above functions as it will remain useful to you while implementing several of the above functions:

findChar ()

Make a function that searches if a particular character exists within the trie at one particular level. When successfully implemented, this function should tell you if that character is found or not. The return type and the parameters of this function are totally up to you to decide.

Trie Visualization

The following link is being provided to you to better help you out with visualizing tries. We would recommend you to visit this link and try out different trie operations such as insertion and deletion and then see how the trie looks like as a result of these operations. This will help you better understand what's expected from you in this part.

https://www.cs.usfca.edu/~galles/visualization/Trie.html

Note: Please keep in mind that your implementation will be tested against hidden test cases that will not be disclosed to you. However, as long as your implementation is generic enough and you have used the correct approach while writing these functions, alongside ensuring that you have followed all the rules of a standard trie, your implementation should pass any and all hidden test cases and you do not need to worry about this at all.

Testing

To test the implementation of your code, compile and run the following test files:

Task	Test File
Chaining	test_chaining.cpp
Linear Probing	test_linearprobing.cpp
Heaps	test_heap.cpp
Insertion Sort	test_insertion_sort.cpp
Merge Sort	test_merge_sort.cpp
Tries	test_tries.cpp

Use the following commands:

Part 1:

Compile: g++ test_chaining.cpp -pthread -std=c++11
Run: ./a.out

Compile: g++ test_linearprobing.cpp -pthread -std=c++11
Run: ./a.out

Part 2:

Compile: g++ test_heap.cpp -std=c++11
Run: ./a.out

Part 3:

Compile: g++ test_insertion_sort.cpp -pthread -std=c++11
Run: ./a.out

Compile: g++ test_merge_sort.cpp -pthread -std=c++11
Run: ./a.out

Part 4:

Compile: g++ test_tries.cpp -pthread -std=c++11
Run: ./a.out

Marks Distribution

Task	Marks
Chaining	5
Linear Probing	20
Heaps	15
Insertion Sort	5
Merge Sort	15
Tries	40

Submission Guidelines

Zip the complete folder and use the following naming convention:

PA3_<roll_number>.zip

For example, if your roll number is 22100074, then your zip file name should be:

PA3_22100074.zip