CS 302 – Assignment #10

Purpose: Learn concepts regarding skip lists.
Due: Tuesday (4/16) → Must be submitted on-line before class.
Points: Part A → 75 pts, Part B → 50 pts

Assignment:

Part A:

Design and implement a template C++ class, ***skipList***, to implement a Skip List[1] data structure. The ***skipList*** class will implement a skip list for storing and counting words for a concordance. A main will be provided that performs a series of tests. Refer to the UML descriptions for implementation details.

Part B:

Create and submit a brief write-up including the following:

- Name, Assignment, Section.
- Summary of the ***skip list*** data structure.
- Compare the skip list data structure to using a Red-Black tree and hash table. Include the associated trade-offs in time and space complexity.
- Explain why a skip list does not need re-balancing or rebuilding operations.
- Explain what a randomized algorithm is. What are the trade-offs compared to an algorithm without randomization?
- Big-O for the various skip list operations (insert(), findEntry(), getMaxWord()).
- Time the program using the **`plots.txt`** test file (note the skip list statistics). Compare the times with the times from the red-black tree used in previous assignments. Run the skip list multiple times with the **`plots.txt`** test file and comment on the changes in the skip list statistics. What is the probability of getting a degenerate skip list, where all layers have the same number of nodes as there are nodes in the skip list? What is the Big-O for skip list operations (insert(), findEntry(), getMaxWord()) in the degenerate case? Is it reasonable to accept the possibility of the degenerate case? Briefly explain.

Submission:

- Submit a compressed zip file of the program source files, header files, and makefile via the on-line submission by 10:00 AM (before class).
- Submit a copy of the write-up (PDF format).

All necessary files must be included in the ZIP file. The grader will download, uncompress, and type **make** (so you must have a valid, working *makefile*).

1 For more information, refer to: https://en.wikipedia.org/wiki/Skip_list

Make File:
You will need to develop a make file. You should be able to type:

```
make
```

Which should create the executable.

Class Descriptions

- Skip List Class
  The Skip List class will implement functions specified below. We will use the following node structure definition.

```
template <class myType>
struct nodeType {
      myType             keyValue;
      int                isSent;
      unsigned int       count;
      nodeType<myType>   *up;
      nodeType<myType>   *down;
      nodeType<myType>   *left;
      nodeType<myType>   *right;
};
```

skipList<myType>
-nodeCount: unsigned int
-totalCount: unsigned int
-height: unsigned int
-*head: nodeType<myType>
-*tail: nodeType<myType>
+skipList()
+~skipList()
+insert(myType): void
+getUniqueWordCount() const: unsigned int
+getWordCount(myType) const: unsigned int
+getMaxNode(unsigned int &, myType &) const: void
+printSkipList() const: void
+showSkipListStats() const: void
-coinFlip() const: int
-addLayer(): void
-findEntry(myType k): nodeType<myType>*
-insert(myType, nodeType<myType>*): bool

Function Descriptions

- The *skipList()* constructor should initialize the skip list to an empty state.
- The *~skipList()* destructor should recover all dynamically allocated memory.
- The *insert()* function should increment the word count if the item is already in the skip list and if not, use the private function to insert the passed item into the skip list.
- The private *insert()* function should insert the passed item into the skip list. The insert should create a tower of nodes with a 50% chance to increase the tower height by one each time. The *coinFlip()* function should be called to determine whether a new node in the tower should be created. Additional layers in the list should be initialized as necessary if the tower becomes larger than the skip list.
- The *getUniqueWordCount()* function should return the current total unique word count.
- The *getWordCount()* function should return the current word count for the passed word.
- The *getMaxNode()* function should search for the word string associated with the largest count and return, via reference, the word string and count.
- The *printSkipList()* function should print all entries in the base layer of the skip list and their counts, one per line.
- The *showSkipListStats()* function is a utility function to print the current skip list height, the expected height, the number of nodes in each level, and the percentage of nodes in each layer relative to the number of nodes in the base layer.
- The *coinFlip()* function should return either 0 or 1 (i.e., **`rand()%2`**).
- The *addLayer()* function should initialize a new topmost layer in the skip list with left and right sentinel elements.
- The *findEntry()* function should search the skip list for a given key, ***k***, and return a node in the base layer with a key equal to ***k*** or the greatest key in the skip list less than ***k***.

- Word Count Class
  The word count class will implement functions specified below.

`wordCount: public skipList<string>`
`-totalWordCount: unsigned int`
`-docFileName: string`
`+wordCount()`
`+getArguments(int, char *[], string &, bool &): bool`
`+readDocument(const string): bool`
`+showDocumentFileName() const: void`
`+showTotalWordCount() const: void`
`+showUniqueWordCount() const: void`
`+showStats() const: void`
`+getMaxWord(unsigned int &, string &): void`
`+printWordCounts() const: void`
`+getWordCount(string) const: unsigned int`

Function Descriptions

- The *wordCount()* constructor should perform applicable initializations.
- The *getArguments()* function should read and validate the passed command line information. If no arguments are entered, it should display a usage message ("**`Usage: ./concordance -i <documentName> [-p]`**"). If the arguments are invalid, it should display an error message ("**`Error, command line arguments invalid.`**"). If the arguments are valid, it should return the document file name as a string and set the boolean value for the print option.
- The *readDocument()* function should attempt to open the passed file name, and if successful read the document one word. Each word should be added to to the skip list which will create new entries for unique words and update the count for duplicate words. The function should remove any brackets, numbers, or punctuation from the words (e.g., '(', '{', '[', ')', '}', ']', '!', '?', '.', ':', '$', '#', etc.). Additionally, numbers should be filtered out. For example, '**`Hello!`**' would be stored as '**`hello`**'. All words should be converted to lower case. This will count '**`Hello`**' and '**`hello`**' as the same word. Additionally, the word '**`1234`**', '**`p@asw0rd`**', '**`re-did`**', and '**`count12er`**' should be ignored (i.e. not stored in the skip list). The function should update the total word count (including duplicate words). See the sample output for examples. If the open and read are successful, the function should return true and false otherwise. The function should close the file.
- The *showDocumentFileName()* function should display the formatted document file name. See the sample output for formatting.
- The *showTotalWordCount()* function should display the formatted total word count. See the sample output for formatting.
- The *showUniqueWordCount()* function should display the formatted unique word count by calling using the word skip list *getUniqueWordCount()* function. See the sample output for formatting.
- The function *showStats()* should call the *showSkipListStats()* function.
- The *getMaxWord()* function should return the node with the largest word count, word count and word, by reference.
- The *printWordCounts()* function should call the print hash function.
- The *getWordCount()* function should return the count of the passed word (0 if not found) by calling the base class function. *Note*, since the words are all lower case, must search for only lower case words.

Refer to the example executions for output formatting. Make sure your program includes the appropriate documentation. See Program Evaluation Criteria for CS 302 for additional information. ***Note, points will be deducted for especially poor style or inefficient coding.***

Skip List Insertion Algorithm

Note, you must use this approach. Do not use other algorithms from the net. Code copied from the net will result in a zero and formal academic conduct write-up. As noted in the UML, the public insert should increment the count if the item is already in the skip list and if the key is not in the skip list, then call the private insert function to add the key.

Algorithm for Skip List Insert

Inputs: myType ***k***, the item to be inserted
Outputs: True, if item is inserted, false otherwise

Side-Effects: Node with key equal to ***k*** added to Skip List object or its count incremented.

```
// --- Part of public insert function
// Check if the key is already in the list, if so increment count
p ← findEntry(myType k)
if p→key == k do
    p→count++
    return true



// --- Part of private insert function
// Insert newNode with key k into list as successor of p
newNode←new nodeType

towerHeight ←0
q ← p

// Insert a copy of newNode in above lists as long as coinFlip is 1
while coinFlip() = 1 do
    if towerHeight >= height do
        addLayer()
    p ← q

    // Find first node left of q with a tower
    while p→up = nil do
        p ← p→left
    p ← p→up
    tower ← new Node

    // Insert tower with key k into list as successor of p
    q→up ← tower
    q ← tower
    towerHeight++

nodeCount++

return true
```

Example Execution:

Below is an example output for the test script and a program execution for the main.

```
ed-vm%
ed-vm% ./concordance -i words1.txt
------------------------------------------------------------
CS 302 - Assignment #10

Document Information:
---------------------
Document File Name: test/words1.txt

Total Word Count: 22496
Unique Word Count: 1749
Skip List Stats
       Current Height: 11
       Expected Height: 11
       Number of nodes in each level:
              Level 11: 2 nodes. (0.11%)
              Level 10: 3 nodes. (0.17%)
              Level 9: 3 nodes. (0.17%)
              Level 8: 6 nodes. (0.34%)
              Level 7: 11 nodes. (0.62%)
              Level 6: 26 nodes. (1.48%)
              Level 5: 52 nodes. (2.97%)
              Level 4: 100 nodes. (5.71%)
              Level 3: 204 nodes. (11.66%)
              Level 2: 421 nodes. (24.7%)
              Level 1: 850 nodes. (48.59%)
              Level 0: 1749 nodes. (100.0%)

Most Frequent Word is: 'the' occurring 2228 times.

------------------------------------------------------------
Game Over, thank you for playing.
ed-vm%
ed-vm%
ed-vm%
ed-vm% ./concordance -i words5.txt
------------------------------------------------------------
CS 302 - Assignment #10

Document Information:
---------------------
Document File Name: test/words5.txt

Total Word Count: 262140
Unique Word Count: 2
Skip List Stats
       Current Height: 4
       Expected Height: 1
       Number of nodes in each level:
              Level 4: 1 nodes. (50.0%)
              Level 3: 1 nodes. (50.0%)
              Level 2: 1 nodes. (50.0%)
              Level 1: 2 nodes. (100.0%)
              Level 0: 2 nodes. (100.0%)

Most Frequent Word is: 'line' occurring 131070 times.

------------------------------------------------------------
Game Over, thank you for playing.
ed-vm%
ed-vm%
ed-vm%
```