Hashtag Counter

COP5536 SPRING 2020 PROGRAMMING PROJECT

Neha Sharma | 4137-1452 | nehasharma@ufl.edu

Contents

Project Purpose

The client’s goal in implementing this software is to have a system to find the *n* most popular hashtags on social media given a list of hashtags and frequencies. Furthermore, the developers are instructed to implement this functionality using a priority structure – more specifically, a max Fibonacci heap. Along with the heap, the developers are to store hashtags and pointers to their frequencies in a hash table. After reading in hashtags from a given input file, the software must either (a) store most frequented hashtags in an output file or (b) output results to the console.

Project Requirements

Functional Requirements

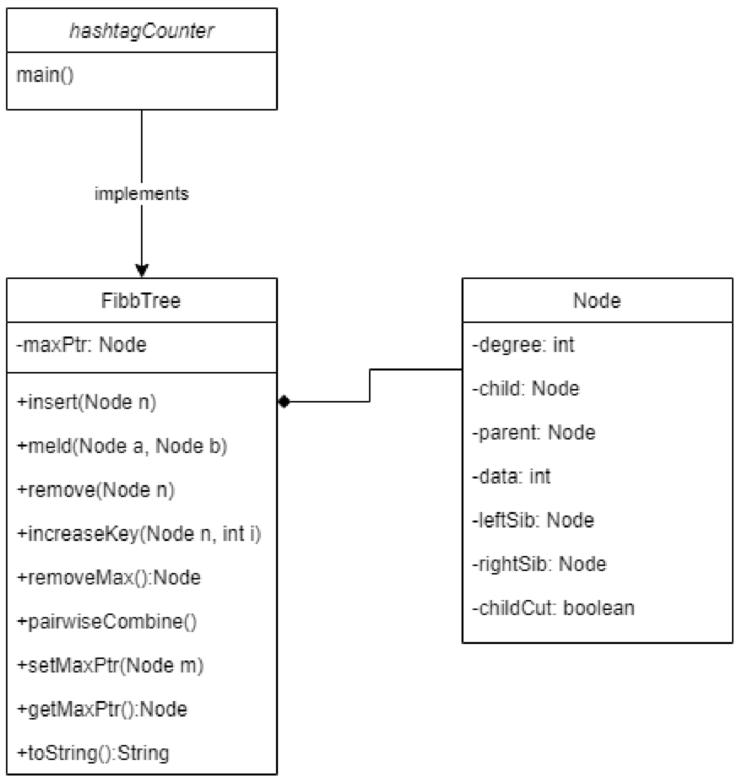
F1	The system must read input from an input file.
F2	The system must use a max Fibonacci heap.
F3	The system must utilize a hash table to store hashtag data.
F4	The system must implement all functions of a Fibonacci heap: • Insert • Remove Max • Meld • Remove • Increase Key
F5	The system must check if there is an output file specified in the command line.
F6	The system must create a new output file if an output file is specified in the command line.
F7	The system must write output to console if no output file is specified.
F8	The system must be able to discern hashtags (#) from query requests (given number).
F9	The system must terminate when it reads the world "stop".
F10	The system must output relevant data as a comma separated list with no spaces.

Non-Functional Requirements

NF1	The system must be optimized to perform the increase key operation many times.
NF2	The increase key operation must have an amortized complexity of O(1).
NF3	The insert operation must have an amortized complexity of O(1).
NF4	The remove max operation must have an amortized complexity of O(log n).
NF5	The meld operation must have an amortized complexity of O(1).
NF6	The remove operation must have an amortized complexity of O(log n).
NF7	The system must be optimized to perform at most 20 queries in a row.
NF8	The system must be optimized to input more than 1 million hashtags.

Software Design

This project was implemented using Java. There are three central classes in this project: `hashtagCounter`, `FibbTree`, and `Node`.



hashtagCounter

This class consists of the main method used to run the program. This main method takes input (and potentially output) file names as command line arguments. Depending on the number of arguments specified, the program may create a file for output. The program begins to read the input file and take a decision line by line. If the read line has a '#' as the first character, the program reads the string and frequency corresponding to it. The data is then either inserted into the Fibonacci heap and hash table as a new entry, or an existing entry is update (using either the insert or increase key functions). If the reader reads "stop", the program terminates. Otherwise, an assumption is made that the next read line is an integer x, and x many of the most frequent hashtags are outputted (using the remove max function).

`FibbTree`

The FibbTree class represents the Fibonacci heap and its functions. The class consists of one property: *maxPtr* which is the essential link from the heap to the rest of the program. The FibbTree object implements the following functions:

`void insert(Node n);`

The insert function takes in a node to insert to the right of the current *maxPtr.* If there is no current *maxPtr,* the newly inserted node becomes the new *maxPtr.* The function then compares the new node's data to the current *maxPtr* and reassigns if necessary.

`void meld(Node a, Node b);`

The meld function takes in two nodes acting as pointers to their respective Fibonacci heaps. The function then applies a meld technique based on the siblings of node pointers a and b. Finally, the function iterates through the top-level linked list and determines a new *maxPtr* if necessary.

`void remove(Node theNode);`

This remove function takes in a specific node to remove from the Fibonacci heap. First, it removes the node from its doubly linked list. It then changes the node's child and parent pointers if necessary. Following, it removes ties from parents and children. If they exist, child nodes are incorporated into the top-level list using meld.

`void increaseKey(Node n, int scalar);`

The increaseKey function takes in a node which needs to have its data updated, and the integer scalar by which the data is increased. The function first performs the update of the data, and then may proceed to perform a cascading cut if necessary.

A cascading cut is performed in the case that (a)* the node that has been updated now has a higher value than its parent, therefore violating max heap properties and (b) the newly removed node has a parent that has already lost a child. If this is the case, the function repeatedly moves up the tree, removing nodes until they have either reached a parent whose *childCut* value is false (thereby setting it to true), or until the top-level list has been reached.

The increaseKey function is called every time a hashtag is read from the input file that already exists in the *hashtags* hash table, since it is necessary to update an existing data point by increasing the value by the given scalar.

* It should be noted that if condition (a) occurs, the node will be removed from the tree regardless of whether a cascading cut is needed.

`Node removeMax();`

The remove max function returns and removes the node with the highest data value. The function fully removes the *maxPtr* from the linked list and reassigns its siblings, parents, and children accordingly. If the removed node had children, the children are incorporated into the top-level linked list using meld. A temporary *maxPtr* is set using one of the siblings of the old *maxPtr.* After completing the meld, the function iterates through the top-level linked list and finds the correct new *maxPtr.* The Fibonacci heap is then consolidated using pairwiseCombine.

`void pairwiseCombine();`

A pairwise combine is done following a remove max. The intent of a pairwise combine is to create more of a balanced Fibonacci heap structure vertically along with horizontally. Each node in the top-level linked list is analyzed according to degree. No two nodes may exist in the top-level list with the same degree; therefore, subheaps with the same degree are consolidated into one.

The function keeps track of subtree degrees using a hash table *treeTable.* Because the degrees/extent to which the degrees can grow are unknown to the developer, a hash table is the most efficient way to store degrees (over an array for example, where the largest degree would need to be known and memory could be wasted with unused indices).

If two subtrees exist with the same degree, they are consolidated and put back into their correct respective place in the *treeTable.* This process continues until there are no longer any two trees in the top-level list that have the same degree. Since each node from the "old" Fibonacci heap is removed in the process of this consolidation, all "new" trees are taken from the *treeTable* and inserted into the Fibonacci heaps as new subtrees.

`void setMaxPtr(Node maxPtr);`

This setter function is implemented for the purpose of testing from the main method.

`Node getMaxPtr();`

This setter function is implemented for the purpose of testing from the main method.

`String toString();`

This toString function is implemented for the purpose of testing.

Node

The Node class serves as the building block for the Fibonacci Heap. The properties of node along with their getters and setters are:

- `private int degree;`
    - `public int getDegree();`
    - `public void setDegree(int degree);`
- `private Node child;`
    - `public Node getChild();`
    - `public void setChild(Node child);`
- `private Node parent;`
    - `public Node getParent();`
    - `public void setParent(Node parent);`
- `private int data;`
    - `public int getData();`
    - `public void setData(Node data);`
- `private Node leftSib;`
    - `public Node getLeftSib();`
    - `public void setLeftSib(Node leftSib);`
- `private Node rightSib;`
    - `public Node getRightSib();`
    - `public void setRightSib(Node rightSib);`
- `private Boolean childCut;`
    - `public boolean getChildCut();`
    - `public void setChildCut(boolean childCut);`

Known Errors

1. There are inconsistencies in the removal and resetting of children and parents. There are two locations in which a child/parent relation might be updated (pairwise combine or the cascading cut functionality in the increase key):
    a. This is causing errors in the reinsert of “popped out” nodes. Therefore, while the first few queries come out correct, after incorporation of more node nesting, there is a bug in the system that either (a) outputs the wrong answer or (b) freezes the program.
    b. This can be assuaged with more intensive debugging on all the instances a child or parent is set. There needs to be a check done on every reallocation of a pointer’s parents and siblings.
2. There is a specific case that is causing a pointer to point to a “null” value which is causing problems in execution and is stalling the read process.
3. The time complexity of receiving the string value of the highest frequency node is O(n); n being the number of values in the hash table. This can be detrimental to the efficiency of the code, especially since the access time for a hash table should be O(1). The reasoning for this complexity is that since the strings themselves are keys of the hash table and we are attempting to access keys through the values, we must loop through the table until we find a match.

References

Makefile Instructions: https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html#java