Hashtag Counter

using Advanced Data-Structure

Programming Project

Spring 2020

Project by:

Nikhil Tank

UFID: 9954-9157

email: nikhil.tank@ufl.edu

Hashtag Counter *using Advanced Data-Structure*

Programming Project - Spring 2020

Description:

The objective of the project is to implement a system to find the "n" most popular hashtags that appear on social media such as Facebook or Twitter. For the scope of this project hashtags will be given from an input file and output can be drawn output either in an output file or in console. With the change frequency of hashtag, there is a requirement of a system that can effectively facilitate the following operation:

1. Increase the frequency of a hashtag
2. Output "n" most popular hashtags.

Basic idea for the implementation is to use a max priority structure to find out the most popular hashtags. The following data structures were used:

Max Fibonacci heap: used to keep track of the frequencies of hashtags as a max priority queue.

1. Amortized cost for Increase key operation is O (1). This allowed faster performance if there will be a large number of hashtags appearing in the stream and we might need to perform increase key operation many times.
2. Amortized cost for Remove Max operation is O (log n), where n is the no. of nodes in the heap.

Hash table: The key for the hash table is the hashtag, and the value is the pointer to the corresponding node in the Fibonacci heap.

Assumptions:

1. No uppercase letters in the input stream

2. No spaces in the hashtags. (only one word per one hashtag)

3. One query has only one integer.

4. Query integer, n<=20. i.e. you need to find at most 20 popular hashtags.

5. Input file may consist of more than 1 million hashtags

Program Structure:

1. Header File: "fHeap.h"

The Header file contains the class definitions and functions for the Max Fibonacci heap. It contains the "Structure node" and "Class fHeap.

- "Struct node" is a collection of variables of different data types under a single name for our Max Fib. Heap. It is similar to a class in that, both holds a collection of data of different data types. It is collection of following:

  1. *hread* – It is a string for storing the value of hashtag of a node.
  2. *freq* – It is an integer for storing the value of frequency of a node.
  3. *degree* – It is an integer for storing the no. of Childs of a node.
  4. *parent* – It is a node pointer, pointing the address of parent node.
  5. *child* – It is a node pointer, pointing the address of child node.
  6. *leftSibling* – It is a node pointer, pointing the address of Left Sibling node.
  7. *rightSibling* – It is a node pointer, pointing the address of Right Sibling node.
  8. *childCut* – It is a Boolean data type storing the Child-Cut value at a node.

- "Class fHeap" is the building block, that leads to Object-Oriented programming for Max Fib. Heap. It's data members and member functions (methods) can be accessed and used by creating an instance of our priority queue.

  Data Members:

  1. *numNodes* – Integer variable storing the no. of Nodes in the Fib. Heap
  2. *MAX* – pointer for pointing to the max value frequency node in the Fib. Heap

  Public Member Functions (Method):

  1. *fHeap()* – method to initialization for Fibonacci heap.
  2. *node* getNode(string hread, int freq)* – Method to create and return a node so that it can be inserted into the Max Fib. Heap. It assigns a node hashtag name, frequency, and set's parent and child pointers to null; most important sets Child-Cut value to false.
  3. *void insert(node* newNode)* – It insert a new node in tree at root level this happens when hashtag was not found in HashMap.
  4. *void increaseKey(node* target, int key)* – It increase the value of frequency when a node was found in HashMap and now it require an update in the tree.
  5. *void Nodecut(node* target)* – When increase key operation leads to Child node Freq > Parent node Frequency. The node will be separated form parent.
  6. *void cascadingNodeCut(node* parenttarget)* – When parent node loses more than 1 child then the parent will also cut from it position.
  7. *void meld(node* targetA, node* targetB)* – Meld two subtrees together one making child and other it's parent (frequency target A > frequency target B)
  8. *void combine()* – Combine the tree with same degree at root level. Uses the meld operation inside. To optimize run time uses dynamic programming.
  9. *node* removeMAX()* – pop the MAX from the tree but trigger pairwise combine.

2. Implementation File: "fHeap.cpp"

The implementation of the Max Fibonacci heap class goes into the .cpp file. By doing this, if the class implementation doesn't change then it won't need to be recompile. This file contains the methods (functions that belongs to the class) of Fib. Heap class.

1. *fHeap()*
   This is a constructor to initialization the member function for Max Fibonacci heap. It sets number of nodes variable (numNodes) to zero and max frequency node pointer to null.

2. *node* getNode(string hread, int freq)*
   This Method to create and return a node so that it can be inserted into the Max Fib. Heap. It do the following sets:
   - Assigns the Structure (collection of variables of different datatypes) "Node" for a new node
   - Sets Hashtag value and Frequency of the node as per the new input corresponding to miss in HashMap.
   - Sets Degree for a new inserted node to 0.
   - Sets the Parent pointer and Child pointer to NULL.
   - Sets the Left Sibling and Right Sibling pointer to itself.
   - Sets the Child Cut Boolean to false.

3. *void insert(node* newNode)*
   This Method to insert a new node in tree at root level this happens when hashtag was not found in HashMap. The Fib. Heap doesn't constraint exact insertion position on root level, so the new node is inserted right to the max frequency node. Then it check if new inserted node is the new max.

4. *void increaseKey(node* target, int key)*
   This Method increase the value of frequency by a value "key" when a node was found in HashMap and now it requires an update in the tree. If the target node is not parent and frequency of parent is greater than itself then the node is cut from the tree, using Nodecut method and method cascadingNodeCut is called for parent.

5. *void Nodecut(node* target)*
   This method checks the parent node degree and if degree is found to be 1 then it resets the parent child to null, else its redirection the sibling doubly linked and eliminating our target node from that list. Now it does the following updates for cut node:
   - Sets the Left Sibling pointer of target to max node.
   - Sets the Right Sibling pointer of target to old right sibling of max.
   - Sets the Child Cut Boolean of target to false.
   - Decrease degree of old parent by 1 and then sets the Parent pointer of target to NULL.

6. *void cascadingNodeCut(node* parenttarget)*
   This Method is called on the parent of the target node which was called in Nodecut. This method is only relevant when our "parent-target" has a parent. If true, then it checks the child cut value of "parent-target" if found False then it toggles it, else the parent have already lost one child in past and by this one more child lost now requires this parent node to be cut from its position and recursively calls "cascadingNodeCut" and "Nodecut" again.

7. *void meld(node* targetA, node* targetB)*
   This Method melds two subtrees together one making child (targetB) and other its parent (targetA) (frequency target A > frequency target B). This is function we are going to use in combine method.

8. *void combine()*
   This Method combine the tree with same degree at root level. The length of the pointer array used to check if there are nodes of same degree is derived using the Fibonacci Formula (length = $\log_{phi}$n), where *phi* is the golden ration and n is no. of node in the Fib. Heap. The check for degree is done using dynamic programming. It uses the meld operation when it founds 2 subtrees of same degree.

9. *node* removeMAX()*
   This method pop the MAX frequency node from the tree and trigger pairwise combine for the root level nodes.

3. Main function: "hashtagcounter.cpp"

This is the main file which support the implementation of project. This maintains the heap, read the input file line by line for appearing hashtags and user queries and any output generated is returned to the output file or to the console as per user requirement.

```
int main(int bname, char* file[])
```

The char pointer stores the file names. "file[1]" being the input file name and "file[2]" being the output file name. If "file[2]" is false then the generated output is displayed on console.

It uses `getline(inputFile, lread)` function to read the input file line by line and uses a map to store the hashtag and nodes pointer Fibonacci heap. The parsing and implementation of function is done as follows:

Case1: Input line is a hashtag

- If line starts with a #.
- Search for first white space and use it to separate hashtag name and frequency
- Case1: node corresponding to hashtag is already in HashMap
  - Then increase its frequency using the *increasekey* method
- Case2: node corresponding to hashtag is not in HashMap
  - Then insert hashtag and frequency in heap at a node.
  - Insert hashtag and node pointer in HashMap.

Case2: Input line is a query

- Check whether the Query integer, n<=20
- If true, create a pointer array of query integer size
- Remove max n time form the heap, and store the nodes in the pointer array.
- As per user requirement write the output array's hashtag into text file or on console.
- Insert back the removed nodes into the heap to keep heap invariant.

Case3: Input line is a "stop" / "STOP" command: terminate the operation of code.

4. makefile:

It contains a set of directives used by a make build automation tool to generate a target/goal binary file named "hashtagcounter".

Input / Output Files Format

1. Input format

Hashtags appear one per line in the input file and start with # sign. After the hashtag an integer will appear and that is the count of the hashtag (There is a space between hashtag and the integer). We need to increment the hashtag frequency by that count. Queries will also appear in the input file and once a query appears you should append the answer to the query to the output file. A query appears as an integer number (n) without # sign in the beginning. The answer to the query n is n hashtags with the highest frequency. These should be written to the output file. An input line with the word "stop" (without hashtag symbol) causes the program to terminate. The following is an example of an input file.

```
#saturday 5
#sunday 3
#saturday 10
#monday 2
#reading 4
3
```

2. Output format

For each query n, we need to write the n most popular hashtags (i.e., highest frequency) to the output file in descending order of frequency (ties may be broken arbitrarily). The output for a query is a comma separated list occupying a single line in the output file "`<outputfilename>.txt`" or in console. There is no space character after the commas.

Following is the output file for the above input file.

```
Saturday, reading, sunday
```

Running the program

The following are the steps involved in running the code on remote server:

- Send file to remote server (thunder.cise.ufl.edu, Ubuntu 18.044 LTS) using the SSH command: `scp -r folder username@thunder.cise.ufl.edu:/cise/homes/username`
- Log in to remote serve using the command: `ssh username@thunder.cise.ufl.edu`
- Move inside the folder directory and run: `make`
- Run the binary file using either of the two:
    - For output file: `./hashtagcoutner <inputfilename>.txt <outputfilename>.txt`
    - For output in console: `./hashtagcoutner <inputfilename>.txt`
- Receive the output file from remote server using the SSH command: `scp username@thunder.cise.ufl.edu:/cise/homes/username/folder/outputfilename.txt .\`