HASHTAG COUNTER USING FIBONACCI HEAP

Fall 2016

Advanced Data Structures (COP 5536)

Sri Akhilesh Joshi

UFID: 83666029

srijoshi@ufl.edu

Introduction:

Social Networking has taken the world by a storm and social networking websites like Facebook, Twitter have managed to do this by following the current trends. How does an application/machine know the trends? "Hashtags!". One of the most efficient ways to do this is, using Maximum Fibonacci Heap.

Fibonacci Heap is classified into two types:

1. Minimum Fibonacci Heap
2. Maximum Fibonacci Heap

In this project, Maximum Fibonacci Heap has been used for hashtag application where highest frequency hashtags must be written to a file when a query occurs. Also, “Maximum Fibonacci Heap is used because it has better theoretical bounds for *increase_key* operation”.

Class:

Fibonacci_Heap class is used for developing the Fibonacci Heap and also merging the hashtag counter application with the Fibonacci Heap.

Two access modifiers are used in this class which are Public and Private.

Member Variables:

1. ***map <string,hash_pair> fib_hashtable*:** This hash table is used for tracking nodes in the Fibonacci Heap. Hash table is efficient because it takes O(1) to retrieve the value. Standard Template Library(STL) has been used for the implementation of the hash table.
2. ***node* max_pointer*:** This member variable *max_pointer* points to the node which has the maximum frequency.
3. ***ofstream outfile*:** This variable opens the “output_file.txt” file and is used by *Write_Output* function for appending the output.
4. ***map <int, node*> fib_rank*:** This hash table tracks the ranks during the process of melding. STL is used for implementation of the hash table.

5. ***Node* root*:** This member variable points to the root of the Fibonacci Heap.
6. ***vector <vector_pair> removed_values*:** This member variable stores the removed nodes when *remove_max* is called and is used during reinsertion of removed nodes.

Member Functions and their description:

1. `node* Fibonacci_Heap::Create(string hashtag, int value)`

   Parameters: string hashtag, int value
   Return type: node*
   Complexity: O(1)

   Working: This function checks if the hashtag read is in the hash table of Fibonacci Heap or not.

   Two conditions are possible:

   1. **If the hashtag is not present,** then the hashtag serves as a key for the hash table and the value is a pair of frequency and node pointer. The node pointer points to a newly created node which has all the information about itself like data(frequency), left pointer, right pointer, child, parent, rank, child cut value. These values are initialized and later Insert function is called.
   2. **If the hashtag is already present**, *Increase_Key* function is called.

2. `void Fibonacci_Heap::Insert(node* new_node)`

   Parameters: node* new_node
   Return type: void
   Complexity: O(1)

Working: This is a very important function in Fibonacci Heap. If the root doesn't exist, then the new node becomes the root and the *max_pointer* points to this node; else the new node becomes part of the doubly linked list (root list) with the root. Later, the *max_pointer* is updated by comparing with the existing *max_pointer*. The *max_pointer* points to the node with maximum frequency.

3. `void Fibonacci_Heap::Increase_Key(node* thenode, int value)`

   Parameters: node* thenode, int value
   Return type: void
   Complexity: O(n)

   Working: This function increases the data(frequency) of the *thenode* pointer and checks if it satisfies the maximum Fibonacci Heap condition. If it satisfies the condition, then no change is made to the structure and the new maximum pointer is calculated if the *thenode* is in the root list. If it doesn't satisfy the Fibonacci Heap condition, then the node is placed in the root list by the *Remove_Node* function and Cascade Cut operation is performed by *Cascade_Cut* function.

4. `void Fibonacci_Heap::Remove_Node(node* thenode)`

   Parameters: node* thenode
   Return type: void
   Complexity: O(n)

   Working: This function removes the node from its position and re-inserts it into the root list along with its children. Initially, the node is removed from its current location and the Insert function is called with node as the argument. The Insert function then places the node in the root list.

5. `void Fibonacci_Heap::Cascade_Cut(node* thenode)`

   Parameters: node* thenode
   Return type: void

   Working: If the parent of the removed node exists, and if its child cut value is false, the child cut value is changed to true. But if it is true, then the parent of the removed node must be removed and added to the doubly linked list of the root. This operation continues until we reach the root list or the child cut value of the parent is false. The nature of this problem is recursive and hence the implementation is also made recursive.

6. `void Fibonacci_Heap::Remove_Max()`

   Parameters: None
   Return type: void
   Complexity: O(n)

   Working: This function is called when a query appears to display the top most hashtags. This first writes the *max_pointer* to the "output_file.txt" file and then removes the *max_pointer* from the root list. Later, the *max_pointer* is removed from the Fibonacci heap hash table and the node is deleted by calling the Delete function. If the max pointer has children, then its children are added to the root list. After this, Meld function is called.

7. `void Fibonacci_Heap::Delete(string hashtag)`

Parameters: string hashtag
Return type: void
Complexity: O(1)

Working: This function first deletes the node and then erases the entry from the hashtable whose key is the hashtag.

8. `void Fibonacci_Heap::Meld()`

Parameters: None
Return type: void
Complexity: O(1)

Working: This function melds two trees with the same rank. So, for easy and efficient tracking of rank, a hash table named *fib_rank* is used where rank is used as the key of the hash table. Now, we traverse from the root. The rank of every node is inserted in the hash table if the rank doesn't exist. But if the rank exists, then the two trees are melded by calling the Meld2 function. After melding, the hash table is updated with the new rank and the traverse starts again until no two ranks match.

9. `void Fibonacci_Heap::Meld2(node* current_node, node* table_node)`

Parameters: node* current_node, node* table_node
Return type: void

Working: This function melds two trees (current node (and its tree)), table node (and its tree). As this is a max tree, if the frequency of the current node is greater than the table node then the table node along with its tree will be appended to the child list of the current node and vice versa. Meld2 returns to the Meld function which performs the required operations again. These operations come to a halt when there are no nodes with same ranks.

10. `void Fibonacci_Heap::FindMax()`

    Parameters: None
    Return type: void

    Working: This function is called after all the melding is done. As the name indicates, the function finds out the node with max frequency in the root list and the *max_pointer* points to this node.

11. `Fibonacci_Heap()`

    Parameters: None
    Return type: void

    Working: The constructor sets the root to NULL, removes the "output_file.txt" file and opens(creates) a new one.

12. `void Fibonacci_Heap::Write_Output(string thestring)`

    Parameters: string thestring
    Return type: void

Working: This function appends the string hashtag to the "output_file.txt" file.

Main Function:

`int main(int argc, char *argv[])` **:**

In the main function the input file is being read until it reaches EOF (End of file).

In the process of reading, as per the instructions, three different inputs are possible.

1. Hashtag, Frequency
2. Query number
3. Stop

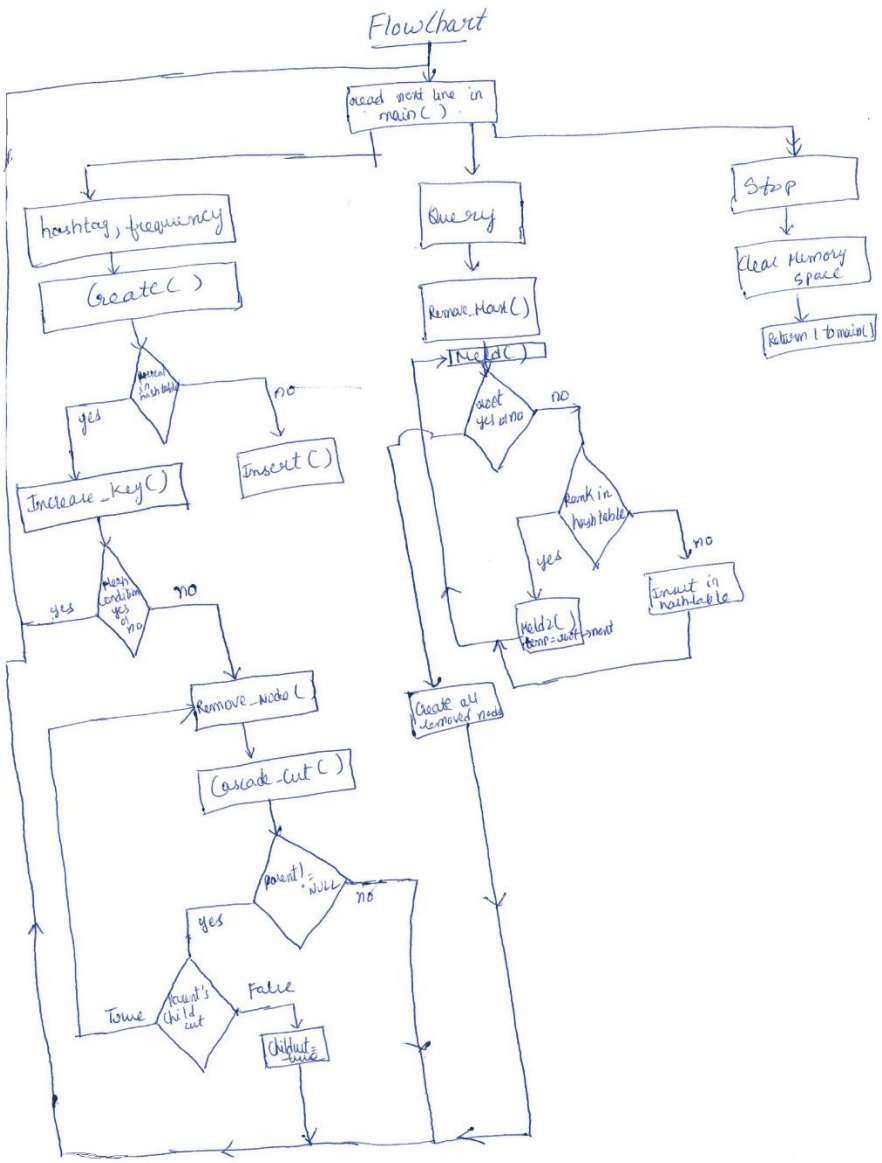
The (1) and (2) types of inputs are read continuously until Stop is read.

When the first kind of input is read, the hashtag and the frequency are stored and the Create function is called.

When the second kind of input is read, the *Remove_Max* function is called.

When the third kind of input is read, control is returned.

Structure of the program:



Conclusion:

The Hashtag counter with Fibonacci Heap was developed using C++ and it satisfies all the provided test cases with the required complexity.