COP 5536 Fall 2016 Programming Project

YUVARAJ SRIPATHI

UFID – 14677313

ysripath@ufl.edu

Assignment Description

To implement a system to find the n most popular hashtags. The hashtags and corresponding frequencies are obtained from an input file.

Max Fibonacci heap structure is used for keeping track of the frequencies of the hashtags.

Programming Environment

Programming Language – C++

Compiler – g++

Executable obtained from running makefile – hashtagcounter

Input – command line input file

Execution instructions

Run the makefile using “make” command which generates the “hashtagcounter” executable.

“g++ -std=c++0x main.cpp StringRelated.cpp Node.cpp -o hashtagcounter”

For the input file “input.txt”, use the following command to get the output in an output file output_file.txt

$./hashtagcounter input.txt

Handling the input strings from the input text file

The input string can be either of the 3 types of string pattern

1. String that contains hashtag string and the frequency for the same. Example - #example 35
2. String that contains just the query number, the number of maximum hashtags to find. Example - 3
3. String that is equal to “stop” indicating end of the input file. Example – stop

An utility function (`utilStruct checkHashTagPresence(string dataValue) {…}`) is used for parsing the string and figuring out as to what are its contents and how the same should be manipulated. The function returns a structure whose members are initialized according to the input string parsed.

utilStruct structure

```
struct utilStruct
{
	bool hashValue; // To denote presence of #
	int frequencyValue; // Frequency value associated with the #tag string
	int queryNumber; // If no #, then the number in the input string represents
query number
	string data; // #tag string without #
};
```

The structure contains the fields – *hashValue* (true if ‘#’ is parsed else false), *frequencyValue* (the number of times the hashtag occurs as per the parsed input string, *queryNumber* (if there is no ‘#’ found, then it is a number denoting the number of max hashtags to find).

The utility function for parsing the input string and initializing the utilStruct structure accordingly,

```
utilStruct checkHashTagPresence(string dataValue)
{
	utilStruct InputData;

	char buf[1000] = { 0 };
	int length = dataValue.size();
	strncpy_s(buf, dataValue.c_str(), length); // copy the string into the char
array buf
	if (buf[0] == '#') // check for #
	{
		// # is present, create an array with the actual hashtag string without #
		char nBuf[1000] = { 0 };
		char fBuf[100] = { 0 };
		// update the hashValue member of the InputData structure as true
```

```
                InputData.hashValue = true;

                int j = 0;
                while (buf[j + 1] != ' ') // Delimeter is a space character
                {
                        nBuf[j] = buf[j + 1];
                        j++;
                }
                nBuf[j] = '\0';
                j += 2;

                // Obtain the frequency value from the hashtag string
                int k = 0;
                while (j != length)
                {
                        fBuf[k] = buf[j];
                        j++;
                        k++;
                }
                fBuf[k] = '\0';
                // Initialize the data member of InputData structure with the hastag
string without #
                InputData.data = string(nBuf);
                // Initialize the frequencyValue member of InputData structure with the
frequency value obtained from the hashtag string
                InputData.frequencyValue = atoi(string(fBuf).c_str());

                InputData.queryNumber = -1;

                return InputData;
        }
        else
        {
                // If no # found, then obtain the query number and update the same in
InputData structure
                InputData.hashValue = false;
                InputData.data = "";
                InputData.frequencyValue = -1;
                InputData.queryNumber = atoi(dataValue.c_str());
                return InputData;
        }
}
```

InputData is the utilStruct structure that is created and initialized in this function based on the parsed string and the same is returned to invoker of *checkHashTagPresence(…)*.

Representation of nodes in the Fibonacci heap

Each of the nodes in the heap is a *Node* class object. The class *Node* contains the members and constructor required for constructing and maintaining the Fibonacci heap.

Class *Node* object,

```
class Node
{
public:
	string pData; // #tag string
	int pRankValue; // Rank value of the node in the Fibonacci heap
	int pFrequency; // Frequency value of the #tag
	Node* pChild; // Pointer to child if any in the Fibonacci heap structure
	Node* pLeftSibling; // Pointer to the left sibling of the node
	Node* pRightSibling; // Pointer to the right sibling of the node
	Node* pParent; // Pointer to the parent of the node in the Fibonacci heap
	int pChildCutValue;  //0 FALSE, 1 TRUE, -1 UNDEFINED (ROOT LEVEL NODES)

	// Parameterized construtor to initialize the Node class object with #tag string
and its frequency values as the parameters
	Node(string dataValue,int frequencyValue);
	~Node();
};
```

Class *Node* contains the members – ***pData*** (hashtag string), ***pRankValue*** (degree/rank of the node in the Fibonacci heap), ***pFrequency*** (the number of times the hashtag occurs), ***pChild*** (pointer to the child node in the Fibonacci heap), ***pLeftSibling*** (pointer to the sibling towards the left side of the node), ***pRightSibling*** (pointer to the sibling towards the right side of the node), ***pParent*** (pointer to the parent of the node), ***pChildCutValue*** (child cut value of the node – used in cascade cut operations with respect to Fibonacci heap).

The parameterized constructor for the *Node* class is invoked for creating the *Node* object. The parameters *dataValue* and *frequencyValue* are used for initializing the *pData* and *pFrequency* members of the *Node* object.

```
Node::Node(string dataValue, int frequencyValue)
{
	pData = dataValue; // #tag string
	pRankValue = 0; // Rank value of the node in the Fibonacci heap
	pFrequency = frequencyValue; // Frequency value of the #tag
	pChild = NULL; // Pointer to child if any in the Fibonacci heap structure
	pLeftSibling = NULL; // Pointer to the left sibling of the node
	pRightSibling = NULL; // Pointer to the right sibling of the node
	pParent = NULL; // Pointer to the parent of the node in the Fibonacci heap
	pChildCutValue = -1; // Child cut value of the node in Fibonacci heap,
initialized to -1 undefined (root level node)
}
```

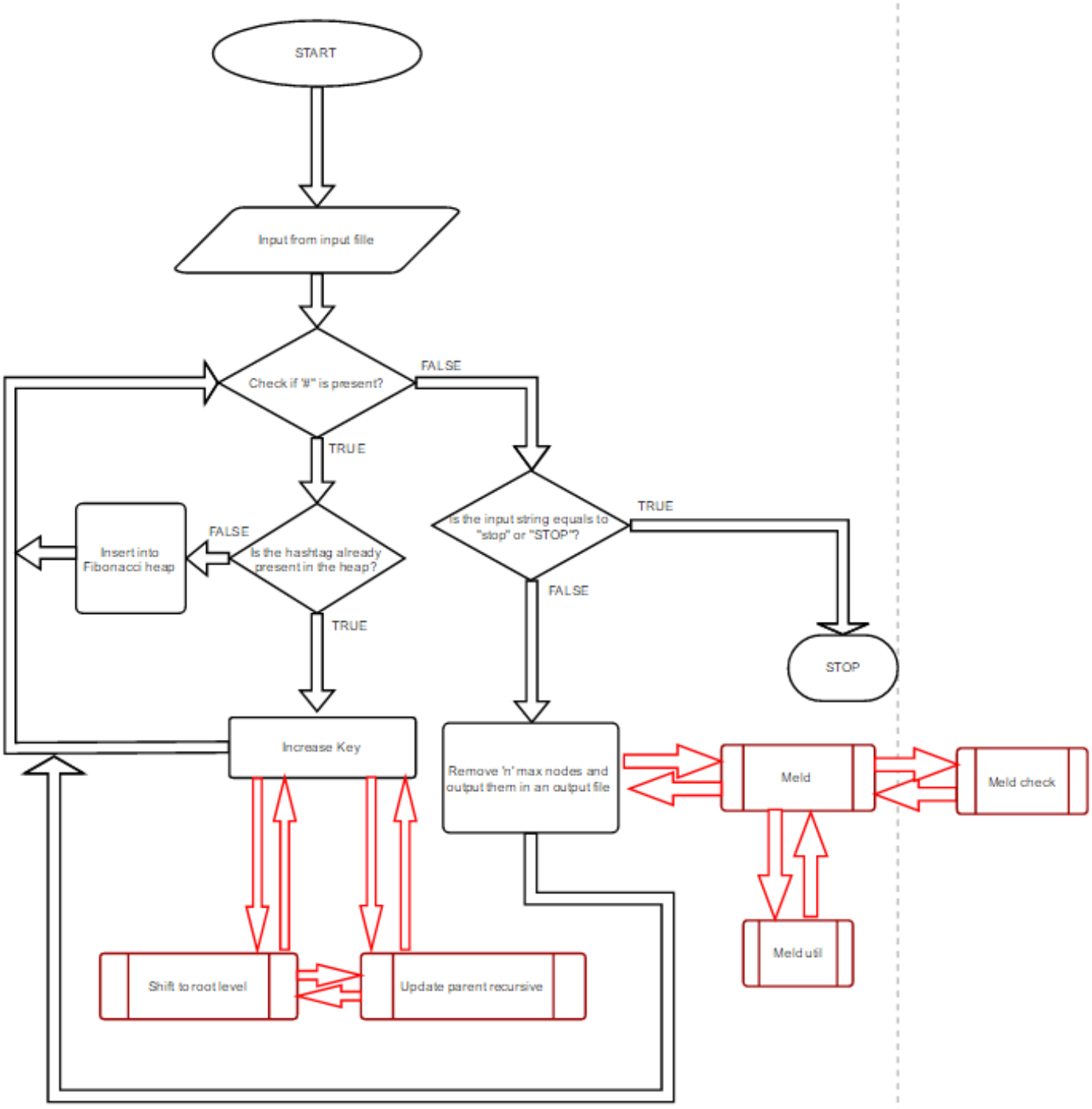
Following are the set of assumptions and rules followed for the creation and maintaining the Fibonacci heap,

- Every new node is inserted in the Fibonacci heap at the root level.
- In every new node being created, the *pParent* pointer points to itself, the *pLeftSibling* and *pRightSibling* pointers again point to itself representing a circular doubly linked list.
- The *pChild* pointer is initialized to NULL.
- The *pChildCutValue* is initialized to -1 (indicating the node is present at the root level).
- MAX pointer - *maxPointer* always points to the node which has the maximum frequency value.
- The new nodes are always inserted to the right of then *maxPointer* in the Fibonacci heap.

Hash table in the form of unordered map is used for keeping record of all the nodes in the Fibonacci heap. In this unordered map implementation, key value is the hashtag string and the value contains the corresponding pointer for the *Node* object.

```
unordered_map<string, Node*> hashTable; // Container for all the nodes in the heap. Key
- #tag string, Value - Corresponding node pointer
```

Program flow structure



In the *main(…)* function, use input and output *filestream* to handle file input and file output,

```
ofstream outputFile; // filestream to handl output to a file

int main(int argc, char **argv)
{
	string fileName(argv[1]);  // fileName takes the name of the input file
specified in the command line
	Node* maxPointer = NULL;    // A pointer that will always point to the node with
maximum frequency
	ifstream inputFile; // Filestream for handling input
	string inputString;
	inputFile.open(fileName); // access the input file to start reading the inputs
	outputFile.open("output_file.txt"); // open the output file output.txt to record
the outputs
	if (inputFile.is_open() && outputFile.is_open())
	{
	   …
	}
	…
}
```

On reading the input strings from the input file, they are parsed to extract the necessary information and returned via the structure *InputData* of the type *utilStruct*.

Based on the data present in this structure, one of the following operation is performed with respect to the Fibonacci heap.

- If there is a hashtag string and its frequency value (Ex - #example 3), then first check if a node representing the same hashtag string is already present in the heap or not. If it's already present, update its frequency by increasing the same by calling the *increaseKey(..)* function else create a new node for the hashtag string and insert the same into the Fibonacci heap using the function *insertFibHeap(…)*.
- If there is no hashtag string, but only query number is found then, find the 'query' number of maximum nodes (hashtags with maximum frequency) from the Fibonacci heap and write the same on to a output file. For this the function *removeMax(…)* is called.
- If the input string is either "stop" or "STOP", it denotes the end of input file, the program is terminated.

Code snippet for handling the above-mentioned Fibonacci heap operations based on the input string read,

```
while (getline(inputFile, inputString)) // Loop as long as there exists input data in
the file (Until EOF)
		{
			utilStruct InputData = checkHashTagPresence(inputString); //
Utility function that fills the utilStruct structure from the input
			if (InputData.hashValue == true) // if there is #tag
```

```
                        {
                                auto itr = hashTable.find(InputData.data);
                                if (itr != hashTable.end())
                                {
                                        // Increase the frequency value if tag already
exists in the heap
                                        itr->second->pFrequency += InputData.frequencyValue;
                                        // Call increaseKey function for the same
                                        maxPointer = increaseKey(maxPointer, itr->second);
                                }
                                else // if the tag is not found in the heap
                                {

                                        // First entry to Fibonacci heap
                                        if (maxPointer == NULL)
                                        {
                                                maxPointer = new Node(InputData.data,
InputData.frequencyValue);
                                                maxPointer->pParent = maxPointer;
                                                // Child cut value for root level node is
undefined = -1
                                                maxPointer->pChildCutValue = -1;
                                                // Circular Doubly Linked Lista - sibling
nodes
                                                maxPointer->pLeftSibling = maxPointer;
                                                maxPointer->pRightSibling = maxPointer;
                                                maxPointer->pRankValue = 0;

                                                hashTable.insert(make_pair(InputData.data,
maxPointer)); // insert the new node into the hashTable map
                                        }
                                        else
                                        {
                                                maxPointer = insertFibHeap(maxPointer,
InputData.data, InputData.frequencyValue); // insert the new node into the Fibonacci
heap
                                        }

                                }
                        }
                        else if (InputData.data.compare("stop") == 0 ||
InputData.data.compare("STOP") == 0) // end of input on finding the string stop
                        {
                                break;
                        }
                        else // No #tag found. Hence its a query to obtain query number of
max nodes from the Fibonacci heap
                        {
                                maxPointer = removeMax(maxPointer, InputData.queryNumber);
// Removes InputData.queryNumber number of max nodes from the Fibonacci heap and
reinsert the same into the heap after successful meld
                                // Empty the contents of the buffer and initialize bufCount
to 0, so that the same can be used for next query
                                bufCount = 0;
                                for (int i = 0; i <= 25; i++)
                                {
                                        buffer[i] = NULL;
                                }
```

```
                }
        }
```

Functions for performing Fibonacci heap operations and corresponding utility functions used:

- ***insertFibHeap(…)*** – Function that creates new node from the parameters (hashtag string and frequency value), and inserts the same into the Fibonacci heap and the hash table *hashTable*. After inserting the new node in the Fibonacci heap, the *maxPointer* (that points to the node that has maximum frequency value) is updated accordingly and returns the same.

```
Node* insertFibHeap(Node* maxPointer, string data, int frequencyValue)
{
	Node* node = new Node(data, frequencyValue);
	// ChildCut value undefined for root level nodes
	node->pChildCutValue = -1;
	node->pParent = node;
	// Check if new node is greater than existing max in Fib heap
	//Insert in top level and Update maxPointer to the new node
	Node* temp = maxPointer;

	// Insert to immediate right of max pointer
	temp->pRightSibling->pLeftSibling = node;
	node->pRightSibling = temp->pRightSibling;
	temp->pRightSibling = node;
	node->pLeftSibling = temp;

	// Insert the new node in the hashTable map
	hashTable.insert(make_pair(data, node));
	// return the node with maximum frequency as the pointer to max node
	if (frequencyValue >= maxPointer->pFrequency)
	{
		return node;
	}
	else
	{
		return maxPointer;
	}
}
```

- ***increaseKey(…)*** – Function for increasing the frequency of the hashtag string whose node is already present in the Fibonacci heap. As per the implementation of this function, if the increased frequency node has frequency greater than or equal to its parent node then, the updated node is moved to the root level (rank value of the parent

node is decremented) and **cascade cut** operation is performed in a recursive manner for its parent node. If the frequency of the updated node is lesser than frequency value of its parent node then, do nothing. Cascade operation happens based on the following set of rules,

- If the child cut value of the parent node is -1, signifies that it's a root level node hence just decrement the rank of the parent node and move the updated node to root level and update the *maxPointer* accordingly.
- If the child cut value of the parent is 0 (FALSE) then, decrement the parent's rank value and move the updated node to the root level, update the child cut value of the parent node to 1 (TRUE) and update the *maxPointer* accordingly.
- If the child cut value of the parent is 1 (TRUE) then, move the updated node to root level and then recursively check for child cut values for parent's parent and move all the parent nodes to the root level whose child cut value is 1 (TRUE) and correspondingly update their parent's child cut value and decrement the rank values. The base condition of the recursion is to find a parent node whose child cut value is 0 (FALSE) or the parent node should be at the root level with undefined child cut value -1.

```
Node* increaseKey(Node* maxPointer, Node* updatedNode)
{
	Node* tempParent = updatedNode->pParent;
	// Frequency value is already increased prior to this function call, just
check for the balance in the heap
	// if the increased key node is at root level, just check for max value
	if (updatedNode->pChildCutValue == -1)
	{
		// Child cut value = -1, undefined - Root level
		// Update maxPointer
		if (updatedNode->pFrequency >= maxPointer->pFrequency)
		{
			return updatedNode;
		}
		return maxPointer;
	}

	// Check if the updatedNode's frequency value is greater than or equal to
the parent value if parent exists
	if ((updatedNode->pParent == updatedNode) || (updatedNode->pParent-
>pFrequency > updatedNode->pFrequency))
	{
		// Node is in the root level. So ignore
		return maxPointer;
	}
	else
	{

		// Before removing the node, check if it has left and right
siblings and update accordingly.
		// Also update the child pointer of the parent node.

		// updatedNode is the only child has no siblings
```

```
		if ((updatedNode->pLeftSibling == updatedNode) && (updatedNode->pRightSibling == updatedNode))
		{
			// Update parent node's child pointer
			updatedNode->pParent->pChild = NULL;
			updatedNode->pParent->pRankValue--;
			updatedNode->pParent = updatedNode;

			maxPointer = shiftToRootLevel(maxPointer, updatedNode);
		}
		// updatedNode has siblings
		else
		{
			Node* temp = updatedNode->pLeftSibling;
			temp->pRightSibling = updatedNode->pRightSibling;
			updatedNode->pRightSibling->pLeftSibling = temp;


			// update parent's child pointer to the immediate left sibling of updatedNode only if the parent node's child pointer points to updatedNode
			if (updatedNode->pParent->pChild == updatedNode)
			{
				updatedNode->pParent->pChild = updatedNode->pLeftSibling;
			}
			updatedNode->pParent->pRankValue--;
			updatedNode->pParent = updatedNode;
			updatedNode->pLeftSibling = updatedNode;
			updatedNode->pRightSibling = updatedNode;
			maxPointer = shiftToRootLevel(maxPointer, updatedNode);
		}

		// if parent node is at root level, child cut value -1, undefined
		if (tempParent->pChildCutValue == -1)
		{
			// Do nothing
			return maxPointer;
		}
		//if parent node child cut value is 0, false
		else if (tempParent->pChildCutValue == 0)
		{
			// Update parent child cut value from False to True, 0 to 1
			tempParent->pChildCutValue = 1;
			return maxPointer;
		}
		//if parent node child cut value is 1, true
		else
		{
			// Remove the parent-children tree and move them to root level
			maxPointer = updateParentRecursive(maxPointer, tempParent);
			return maxPointer;
		}

	}
}
```

- ***shiftToRootLevel(…)*** – Utility function used in *increaseKey(…)* function for moving the required node to the root level and correspondingly updating the *maxPointer*.

```
Node* shiftToRootLevel(Node* maxPointer, Node* updatedNode)
{
	// Move the node to immediate right of max pointer and update the
corresponding fields

	if (maxPointer == NULL)
	{
		// There are no nodes in Fibonacci heap
		updatedNode->pParent = updatedNode;
		updatedNode->pChildCutValue = -1;
		updatedNode->pLeftSibling = updatedNode;
		updatedNode->pRightSibling = updatedNode;
		return updatedNode;
	}

	// If there are nodes already present in the Fibonacci heap
	// Inseert the updatedNode to the right of maxPointer and update
accordingly
	updatedNode->pRightSibling = maxPointer->pRightSibling;
	maxPointer->pRightSibling->pLeftSibling = updatedNode;
	maxPointer->pRightSibling = updatedNode;
	updatedNode->pLeftSibling = maxPointer;
	updatedNode->pChildCutValue = -1;
	updatedNode->pParent = updatedNode;


	// return the node with maximum frequency as the pointer to max node
	if (updatedNode->pFrequency >= maxPointer->pFrequency)
	{
		return updatedNode;
	}
	else
		return maxPointer;
}
```

- ***updateParentRecursive(…)*** – Utility function for performing **cascade cut** operation on the Fibonacci heap starting with the parent node of the node updated by *increaseKey(…)* function and returns the *maxPointer* accordingly.

```
Node* updateParentRecursive(Node* maxPointer, Node* updatedParentNode)
{
	// updatedParentNode Child cut value is True, hence remove the sub tree
and move it to root level
	// Check and update the parent of updateParentNode
	if (updatedParentNode->pChildCutValue == -1 || updatedParentNode-
>pChildCutValue == 0)
	{
		return maxPointer;
	}
	Node* tempParent = updatedParentNode->pParent;
```

```
	// check if updatedParentNode has siblings
	if ((updatedParentNode->pLeftSibling == updatedParentNode) &&
(updatedParentNode->pRightSibling == updatedParentNode))
	{
		tempParent->pChild = NULL;
		updatedParentNode->pLeftSibling = updatedParentNode;
		updatedParentNode->pRightSibling = updatedParentNode;
		updatedParentNode->pParent = updatedParentNode;

		// Decrement the rank of parent of updatedParent node
		tempParent->pRankValue--;

		// Update the child cut value of the parent of updatedParent node
accordingly
		if (tempParent->pChildCutValue == -1)
		{
			// Child cut value undefined - root level
			return (shiftToRootLevel(maxPointer, updatedParentNode));
		}
		else if (tempParent->pChildCutValue == 0)
		{
			// Child cut value 0 - False
			// Update the parent's child cut value to 1 - True
			tempParent->pChildCutValue = 1;
			return (shiftToRootLevel(maxPointer, updatedParentNode));
		}
		else
		{
			// Child cut value is 1 - True
			maxPointer = shiftToRootLevel(maxPointer,
updatedParentNode);
			// Shift the parent of updatedParent node also to root
level
			return (updateParentRecursive(maxPointer, tempParent));
		}
	}
	else // Siblings exist for updatedParentNode
	{
		Node* temp = updatedParentNode->pLeftSibling;
		updatedParentNode->pRightSibling->pLeftSibling = temp;
		temp->pRightSibling = updatedParentNode->pRightSibling;

		// check if updatedParentNode is the child of its parent and
update accordingly
		if (tempParent->pChild == updatedParentNode)
		{
			tempParent->pChild = updatedParentNode->pLeftSibling;
		}
		updatedParentNode->pLeftSibling = updatedParentNode;
		updatedParentNode->pRightSibling = updatedParentNode;
		updatedParentNode->pParent = updatedParentNode;

		// Decrement the rank of parent of updatedParent node
		tempParent->pRankValue--;
		if (tempParent->pChildCutValue == -1)
		{
			// Child cut value undefined - root level
			return (shiftToRootLevel(maxPointer, updatedParentNode));
```

```
		}
		else if (tempParent->pChildCutValue == 0)
		{
			// Child cut value 0 - False
			// Update the parent's child cut value to 1 - True
			tempParent->pChildCutValue = 1;
			return (shiftToRootLevel(maxPointer, updatedParentNode));
		}
		else
		{
			// Child cut value is 1 - True
			maxPointer = shiftToRootLevel(maxPointer,
updatedParentNode);
			// Shift the parent of updatedParent node also to root
level
			return (updateParentRecursive(maxPointer, tempParent));
		}
	}
}
```

- ***removeMax(…)*** – Function for finding the node with maximum frequency value. The following set of operations are performed once the node with maximum frequency value is found (pointer *maxPointer* points to the node with maximum frequency value).
  - Remove the maximum frequency node from the Fibonacci heap and store the same in a temporary container (*buffer*) and update the *maxPointer* accordingly.
  - Perform meld/merge operation on the Fibonacci heap i.e. merge all the nodes at the root level that are having equal rank/degree values (The node with smaller frequency value becomes the child of the node with bigger frequency value). Continue until there are no nodes in the root level that are having the same rank value. Update the *maxPointer* accordingly.
  - If more than one maximum frequency value is required then, repeat the process.
  - Once all the required set of maximum nodes are removed and corresponding meld/merge is performed then, reinsert all the removed nodes that are store in the temporary container (*buffer)* into the Fibonacci heap and clear the contents of the temporary container.

```
Node* buffer[26];
int bufCount = 0;

Node* removeMax(Node* maxPointer, int num)
{
	bool flag = false;
	while (num > 0)
	{
		Node* temp = maxPointer->pChild;
		// Check for siblings at the root level
		// if only maxPointer is present at the root level
		if ((maxPointer->pLeftSibling == maxPointer) && (maxPointer-
>pRightSibling == maxPointer))
		{
			if (!flag)
```

```
                        {
                                outputFile << maxPointer->pData << maxPointer-
>pFrequency; // output to file
                                flag = true;
                        }
                        else
                        {
                                outputFile << "," << maxPointer->pData <<
maxPointer->pFrequency; // output to file
                        }

                        maxPointer->pLeftSibling = maxPointer;
                        maxPointer->pRightSibling = maxPointer;
                        maxPointer->pChild = NULL;
                        maxPointer->pRankValue = 0;

                        //insert the max node into the buffer array to reinsert
after meld
                        buffer[bufCount++] = maxPointer;
                        maxPointer = NULL;

                        // check for children and update its level
                        if (temp != NULL)
                        {
                                // check for siblings of child
                                if (temp->pRightSibling == temp && temp-
>pLeftSibling == temp)
                                {
                                        // insert the child of maxPointer to root
level in Fibonacci heap
                                        maxPointer = insertFibHeapUtil(maxPointer,
temp);
                                }
                                else
                                {
                                        // insert all the siblings of the child of
maxPointer and the child also at the root level of the Fibonacci heap
                                        Node* temp2 = temp;
                                        while (temp2->pRightSibling != temp2)
                                        {
                                                temp = temp2->pRightSibling;
                                                maxPointer =
insertFibHeapUtil(maxPointer, temp2);
                                                temp2 = temp;
                                        }
                                        // insert the last of the siblings at root
level
                                        maxPointer = insertFibHeapUtil(maxPointer,
temp2);
                                }
                        }
                        else
                        {
                                // if all the nodes are removed from the
Fibonacci heap, set the maxPointer to NULL
                                maxPointer = NULL;
                                break;
                        }
```

```
		}
		else
		{
			if (!flag)
			{
				outputFile << maxPointer->pData<<maxPointer-
>pFrequency; // File output
				flag = true;
			}
			else
			{
				outputFile << "," << maxPointer->pData <<
maxPointer->pFrequency; // File output
			}

			// update the siblings' left and right link before
removing the max node
			Node* temp4 = maxPointer->pLeftSibling;
			temp4->pRightSibling = maxPointer->pRightSibling;
			maxPointer->pRightSibling->pLeftSibling = temp4;

			maxPointer->pLeftSibling = maxPointer;
			maxPointer->pRightSibling = maxPointer;
			maxPointer->pChild = NULL;
			maxPointer->pRankValue = 0;

			//insert the max node into the buffer array to reinsert
after meld
			buffer[bufCount++] = maxPointer;
			maxPointer = NULL;

			// update maxPointer at root level by checking the
frequency of each of the nodes at the root level
			int max = -1;
			Node* temp5 = temp4;

			while (temp4->pRightSibling != temp5)
			{
				if (temp4->pFrequency >= max)
				{
					max = temp4->pFrequency;
					maxPointer = temp4;
				}
				temp4 = temp4->pRightSibling;
			}
			if (temp4->pFrequency >= max)
				maxPointer = temp4;
			if (maxPointer == NULL)
				maxPointer = temp5;


			// check for children of the max node and update its
level to root
			if (temp != NULL)
			{
				// check for siblings of child
```

```
                        if (temp->pRightSibling == temp && temp->pLeftSibling == temp)
                        {
                                // No siblings for child
                                maxPointer = insertFibHeapUtil(maxPointer, temp);
                        }
                        else
                        {
                                // siblings are present for child
                                Node* temp2 = temp;
                                while (temp2->pRightSibling != temp2)
                                {
                                        temp = temp2->pRightSibling;
                                        maxPointer = insertFibHeapUtil(maxPointer, temp2);
                                        temp2 = temp;
                                }
                                maxPointer = insertFibHeapUtil(maxPointer, temp2);
                        }
                }
                else
                {
                        //maxPointer = NULL;
                }
        }
        num--;
        // Invoke meld function after every max node removal from the Fibonacci heap
        maxPointer = meld(maxPointer);
    }
    outputFile << endl; // File output


    // Reinsert all the max nodes removed back into the Fibonacci heap
    for (int i = 0; i < bufCount; i++)
    {
        maxPointer = insertFibHeapUtil(maxPointer, buffer[i]);
    }
    return maxPointer;
}
```

- ***meld(…)*** – Function for performing the meld/merger operation after removing the node with maximum frequency value (node pointed by *maxPointer*) from the Fibonacci heap. In this function, local hash table in the form of unordered_map is used to check for nodes that are having the equal degree/rank value and combine the same using an utility function *meldUtil(…)*. The process is repeated until all the nodes at the root level are having different rank/degree values and update the *maxPointer* accordingly. An utility function *melddCheck(…)* is used to confirm this.

```
Node* meld(Node* maxPointer)
{
    // Check if there are redundant rank values at root level
```

```
	if (!meldCheck(maxPointer))
	{
		return maxPointer;
	}

	// check if maxPointer in the only node at the root level
	if ((maxPointer->pRightSibling == maxPointer) && (maxPointer-
>pLeftSibling == maxPointer))
	{
		return maxPointer;
	}
	// There are more than one node at the root level
	bool flag = true;
	Node* begin = maxPointer;
	unordered_map<int, bool> degreeMap; // map for storing bool values of the
rank value of nodes at root level. Key - rank, value - true/false
	unordered_map<int, Node*> nodeMap; // map for storing nodes as per their
rank value. Key - rank, value - pointer to corresponding node
	//unordered_map<int, bool> checkMap;

	while (flag)
	{
		auto itr = degreeMap.find(begin->pRankValue); // Check if same
rank value nodes are found in the degreeMap
		if (itr == degreeMap.end())
		{
			// if there is no node with same rank in the map, insert
the same into the map and move to next node to the righ at root level
			degreeMap.insert(make_pair(begin->pRankValue, true));
			nodeMap.insert(make_pair(begin->pRankValue, begin));
			begin = begin->pRightSibling;
			continue;
		}
		else
		{
			// Node with same rank found in the map
			auto itr2 = nodeMap.find(begin->pRankValue);
			if (itr2->second == begin) // If both nodes are same, meld
is over
			{
				flag = false;
				break;
			}
			// Make the smaller node child of bigger node (smaller and
bigger in terms of their corresponding frequency)
			if (begin->pFrequency >= itr2->second->pFrequency)
			{
				begin = meldUtil(begin, itr2->second);
			}
			else
			{
				begin = meldUtil(itr2->second, begin);
			}
			itr->second = false;
			nodeMap.erase(itr->first);
			// clear the contents of the map for next iteration
			nodeMap.clear();
			degreeMap.clear();
```

```
                        // update maxPointer
                        if (begin->pFrequency >= maxPointer->pFrequency)
                                maxPointer = begin;
                        else
                                begin = maxPointer;
                }
        }

        return maxPointer;
}
```

- ***meldCheck(…)*** – Utility function to check if there exists nodes with same rank/degree value at the root level.

```
bool meldCheck(Node* maxPointer)
{
        if (maxPointer == NULL)
                return false;
        unordered_map<int, bool> boolMap;
        Node* temp = maxPointer;
        while (temp->pRightSibling != maxPointer)
        {
                auto itr = boolMap.find(temp->pRankValue);
                if (itr == boolMap.end())
                {
                        boolMap.insert(make_pair(temp->pRankValue, true));
                }
                else
                {
                        return true;   // Return true if there are redundant rank
value nodes are present at the root level
                }
        }
        return false; // Return false if there is no redundant rank value nodes
}
```

- ***meldUtil(…)*** – Utility function that takes two nodes as parameters and appends the node with lesser frequency value as the child of the node with larger frequency value and update the corresponding rank values.

```
Node* meldUtil(Node* bigger, Node* smaller)
{
        // update the smaller childCutValue to false. (false = 0)
        smaller->pChildCutValue = 0;
        // update smaller's parent to bigger node
        smaller->pParent = bigger;

        //update smaller's left and right at root level
        Node* temp = smaller->pRightSibling;
        smaller->pLeftSibling->pRightSibling = temp;
        temp->pLeftSibling = smaller->pLeftSibling;

        // check if bigger's child exist or not
```

```
	if (bigger->pChild == NULL) // if no child for the bigger, smaller
becomes the child
	{
		bigger->pChild = smaller;
		smaller->pLeftSibling = smaller;
		smaller->pRightSibling = smaller;
	}
	else // there exists a child for the bigger, link the smaller to the
right of existing child
	{
		Node* temp = bigger->pChild;
		smaller->pRightSibling = temp->pRightSibling;
		smaller->pLeftSibling = temp;
		temp->pRightSibling->pLeftSibling = smaller;
		temp->pRightSibling = smaller;
	}
	bigger->pRankValue++; // increment the rank/degree of the parent after
appending a new child
	return bigger;
}
```

Input File content

```
#saturday 5
#sunday 3
#saturday 10
#monday 2
#reading 4
#playing_games 2
#saturday 6
#sunday 8
#friday 2
#tuesday 2
#saturday 12
#sunday 11
#monday 6
3
#saturday 12
#monday 2
#stop 3
#playing 4
#reading 15
#drawing 3
#friday 12
#school 6
#class 5
5
Stop
```

Output File content

```
saturday,sunday,Monday
saturday,sunday,reading,friday,Monday
```

CONCLUSION

Fibonacci heap structure is successfully implemented to store the hashtag strings and corresponding number of frequency of occurrence. The implemented Fibonacci heap structure provides for finding a given number of hashtag strings (*n* maximum with respect to the frequency value of the hashtags) and the same is printed in the output file “output_file.txt”.