Project Description

Here we implement a system find the n most popular hashtags. Max Fibonacci Heap is implemented to store the frequency of hashtags. We also are storing the hashtag as the key and the value as the pointer to the node which will improve the complexity of search and updation of arbitrary node in the heap. We take the file with the hash tag frequencies as input to the program and store them in the heap and the hash map. In case we read a numeral ‘n’ which does not belong to a hashtag we do ‘n’ removals or find the top ‘n’ popular tags and write them to an output file. We repeat the above process of updating the structures and finding out the popular tags until we detect a stop in the input file

Working Environment

HARDWARE REQUIREMENT

Hard Disk space: 1 GB minimum or depends on the input file size
Memory: 512 MB
CPU: x86
OPERATING SYSTEM
Windows 10
Also tested on Ubuntu 16.01
COMPILER
Javac

Compiling Instructions

The program has been coded in JAVA and JVM compiler is used to compile the code and tested on thunder.cise.ufl.edu, Windows 10 (Intellij) and Ubuntu 16.01 platforms

To compile the program in Linux environments you can run the below command

```
$ make
```

The above command will generate the class files and also generate a jar file for the program.

To clear out the class files and the jar file you can run the below command

```
$ make clean
```

To execute the program you can follow any of the below methods.

Method 1: Windows and Linux

```
$ java hashtagcounter file_name
```

Eg: `java hashtagcounter sample_input.txt`

Method 2: Windows

```
hashtagcounter.jar file_name
```

Eg: `hashtagcounter.jar sample_input.txt`

Method 3: Windows and Linux

```
Java –jar hashtagcounter.jar file_name
```

Eg: `Java –jar hashtagcounter.jar sample_input.txt`

All the above 3 commands are tested on the specified environments.

Structure of the program and function descriptions

There are 3 classes that I have used to implement the programming project. The classes used are CFibonacciHeapNode.java, CFibonacciHeap.java and hashtagcounter.java

Below is how the functions are implemented and a brief description of them

CFibonacciHeapNode.java

The class defines the structure of the node that is used in the Fibonacci Heap.

The class variables used are

`hashtagname:`

This variable of type String contains the name of the hashtag being store on the Fibonacci Heap.

`Hashtagcount:`

This variable of type Int is used to store the frequency or number of occurrences of the hashtag.

`Degree:`

This variable of type Int specifies the number of nodes or immediate children in the next level. This will be 0 if there are no children.

`Is_cut:`

This is a Boolean variable which specifies if a child of the node was already removed. The value falso implies that no child of the node had been ever removed. By default this will be false for every node during instantiation.

`Parent:`

This is of the type CFibonacciHeapNode and is used by a node as a pointer to its parent. This will be a null pointer if the node is a root node of the tree.

`Child:`

This is of the type CFibonacciHeapNode and is used by a node to point to its first child. This will be a null pointer if the node doesn't have any children

`Right_sibling and left_sibling:`

This is also of type CFibonacciHeapNode is used to point to its right and left neighbor at the same level. This is used to implement circular doubly linked list at any level. A node have these pointers point to itself if there are no neighbors on the right and left side.

The constructor below is used to instantiate the node and set the properties above to default

`CFibonacciHeapNode(String hashtagname,int hashtagcount)`

Parameters: **`String hashtagname, int hashtagcount`**

This will be used to set the hashtagnmae and hashtagcount of the node. All other properties are set to default values and will be changed accordingly depending on future operations.

CFibonacciHeap.java

The class variable used are

`Max:`

This is of the type CFibonacciHeapNode is used by the heap as the pointer to the max node or highest frequency hashtag. This will be null if the Fibonacci Heap is empty.

`Name2node:`

This is of type hash map and is used to store the hash tag name as key and the node address of the corresponding hash tag as the value.

`Delnodestack:`

This is of the type Stack and is used to store the deleted CFibonacciHeapNode's as a result of the remove max operation. This will be used later to reinsert the nodes when all the remove max operations are completed.

This is the class where all the main methods of Fibonacci heap are defined. The operations in this class and its methods are performed on instances of CFibonacciHeapNode.

The methods defined in this class are:

`CFibonacciHeap()`

Parameters: **`Null.`**

This is the constructor used to initialize the empty Fibonacci Heap by setting max pointer to null and instantiating name2node and delnodestack variables.

`public void insert(String hashtagname,int hashtagcount)`

Parameters: **`String hashtagname, int hashtagcount`**

Return: **`Null`**

This function takes in the hashtagname and the frequency of the hashtag from hashtagcount as parameters. It uses the Hash Map name2node to check if the hashtag is already present. If it is not present, it inserts the node into the Fibonacci Heap and sets the node as max node if the frequency of the new node is greater than the current max node. If the node is already present, the insert function calls the **`increasetagcount()`** method to update the frequency of the already available node and update the max node accordingly.

`private void heapmeld(CFibonacciHeapNode newnode, CFibonacciHeapNode max)`

Parameters: **`CFibonacciHeapNode newnode, CFibonacciHeapNode max`**

Return: **`Null`**

This method takes the newly inserted node and the max node as parameters. This operation updates the sibling pointers of the new node to fit the new node into the heap. The right sibling of the new node is updated to be the previous right sibling of max node and the new right sibling of the max node is set to be the previous right sibling of the new node

`public String removemax()`

Parameters: **`Null`**

Return: **`String or the Hashtag name with highest frequency`**

This function is used to remove the nodes with maximum frequency in the heap. If the max pointer is pointing to null then heap is empty and we will return null. If the max node is not null, we check if the max node has any children by checking the child pointer of max node. If the child pointer is not null, then traverse to all the children using the child pointer as start and set the parent pointer of all those nodes pointing to max node as null. Later meld all the previous children of the max node to the top level of the heap by calling **`heapmeld()`** method. Set the max pointer to point to a new arbitrary node or the next node to the previous max. If the old max node had neighbors, to remove the stale pointers and update the heap from having any pointers to the removed max node we call the **`detach_from_heap()`** function. Once the old max node is completely removed from the system we call the **`pairwise_combine()`** function to combine the trees in the heap with same degree and also update the new max pointer accordingly. In the end we insert the deleted old max node to the Delnodestack which will be used for reinsertion later.

`public void reinsert_delitems()`

Parameters: **`Null`**

Return: **`Null`**

This method is used to reinsert the deleted items in the stack. We pop each node one by one from the Delnodestack and then call the **`insert(string, int)`** method.

`private void detach_from_heap(CFibonacciHeapNode delnode)`

Parameters: **`CFibonacciHeapNode delnode`**

Return: **`Null`**

This method is used to detach the delnode being passed from the tree or the heap. In this method we update all the right_sibling and left_sibling pointers pointing to the delnode or the node to be deleted.

`private void pairwise_combine()`

Parameters: **`Null`**

Return: **`Null`**

This method is used to merge trees with same degree. To implement the mentioned operation we keep 2 pointers, start and current. Both these pointers will be pointing to the same max node in the beginning and we also maintain tree_table which is an array list and maintan the list of trees based on the degree of the tree. Later in a do while loop we traverse through the trees in the heap by incrementing the current pointer to move next. We check if there is a tree with the same degree as the current node in the degree_table. If there is no such node, we add the current node to the degree_table and move next. If there is such a tree whose degree is same as the current node, we compare the frequencies of both the nodes and check who should be made the child. If the current child is to be become the child in the new tree formed by merging, we set the current node to the left sibling of the previous current node. We also check if the node that is going to be the child is the same as start node. If yes we move the start node pointer the right sibling of the previous start node. After updating the start and current pointers we merge the two trees by calling **`attach_tree()`** method. In the end we update the degree table by removing the old node before merging and adding the new tree as a result merging back to the degree_table structrure. After the merge process completes, we parse through the degree_table and update the max node accordingly.

`private void attach_tree(CFibonacciHeapNode tobeparent, CFibonacciHeapNode tobechild)`

Parameters: **`CFibonacciHeapNode tobeparent, CFibonacciHeapNode tobechild`**

Return: **`Null`**

This method takes in root node of the two trees tobeparent and tobechild, out of which tobeparent will be the root of the new tree as well and tobechild is attached as the child of the tobeparent node. So first we set the parent pointer of the tobechild node to point to the tobeparent node irrespectively. Then we check if the tobeparent node has children already. If it doesn't have any children, we set the child pointer of tobeparent to point to tobechild. If the node already has children then we meld the tobechild node with the children of the tobeparent node by calling **`heapmeld(tobeparent.child,tobechild)`** method. We reset the is_cut property of tobechild to false.

`private void increasetagcount(CFibonacciHeapNode Node2update, int additionalcount)`

Parameters: **`CFibonacciHeapNode Node2update, int additionalcount`**

Return: **`Null`**

This method takes in the node address of the node whose frequency is to be updated Node2update and also the additional count that should be added to the frequency of the node. After we increase the hashtagcount of the Node2update node, we check if the node has a parent and if it is now greater than its parent. If it is not greater than its parent, we stop at that point. If it is greater than its parent, we cut the node from the tree by calling the method **`cutfromtree(node,parent)`** and also recursively call **`cascadingcut(parent)`** to perform the cut operation on the way up to the root based on the is_cut value of the parent and its grandparent. In the end we also check if the updated node's frequency or hashtagcount is greater than the max node and if so we update the max node to point to the update node Node2update.

`private void cutfromtree(CFibonacciHeapNode child, CFibonacciHeapNode parent)`

Parameters: **`CFibonacciHeapNode child, CFibonacciHeapNode parent`**

Return: **`Null`**

This method is used to remove a node from tree and update the pointers accordingly. This method takes in 2 parameters CFibonacciHeapNode child i.e the child to be cut from the tree and CFibonacciHeapNode parent i.e the parent of the node to be removed. We check if the child pointer of the parent I pointing directly to the node that is to be cut and if so we update the pointer to the right sibling of the node to be removed. We also check if that is the only child of the parent and if the sibling pointers are pointing to itself then we set the child pointer of the parent to null. Then we set the parent pointer of the child to be removed to null. After updating the parent and child pointer to detach the node and update the sibling pointers we call **`detach_from_heap(child)`**. After the successful detach we meld the detached node to the top level nodes of the list by calling **`heapmeld(child,max)`**. We also update the is_cut value of the removed node to false.

`private void cascadingcut(CFibonacciHeapNode node)`

Parameters: **`CFibonacciHeapNode node`**

Return: **`Null`**

This method is used to recursively move up the tree and check if the parent nodes on the way up has to be detached from the tree depending on the is_cut values. This takes in the node address for whom we examine if we have to remove it. We first check if the node address we have got is not root. If it is a root node i.e parent pointer of the node is null we don't do anything. If it is not a root node, we check the value of the is_cut field and if it is false which means that this was the first time a child node was removed from this node structure. We set the is_cut value to true and return the control back to the caller. If the is_cut value is true which means that there was an earlier instance of the child removal and the present removal is the second one then we remove the node by calling **`cutfromtree(node,parent)`**. And we call **`cascadingcut(parent)`** recursively to traverse up till the root of the root and perform the same operation above.

<u>hashtagcounter.java</u>

This is the class where we implement the main() function and instantiate the heap.

`public static void main(String[] args)`

<u>Parameters:</u> **`Filename of the file to be read in args[0]`**

<u>Return:</u> **`Null`**

In this main method we take the file name as the input argument, instantiate the heap by calling **`CFibonacciHeap hashtags=new CFibonacciHeap()`** and use Scanner to read from the file obtained in the argument.

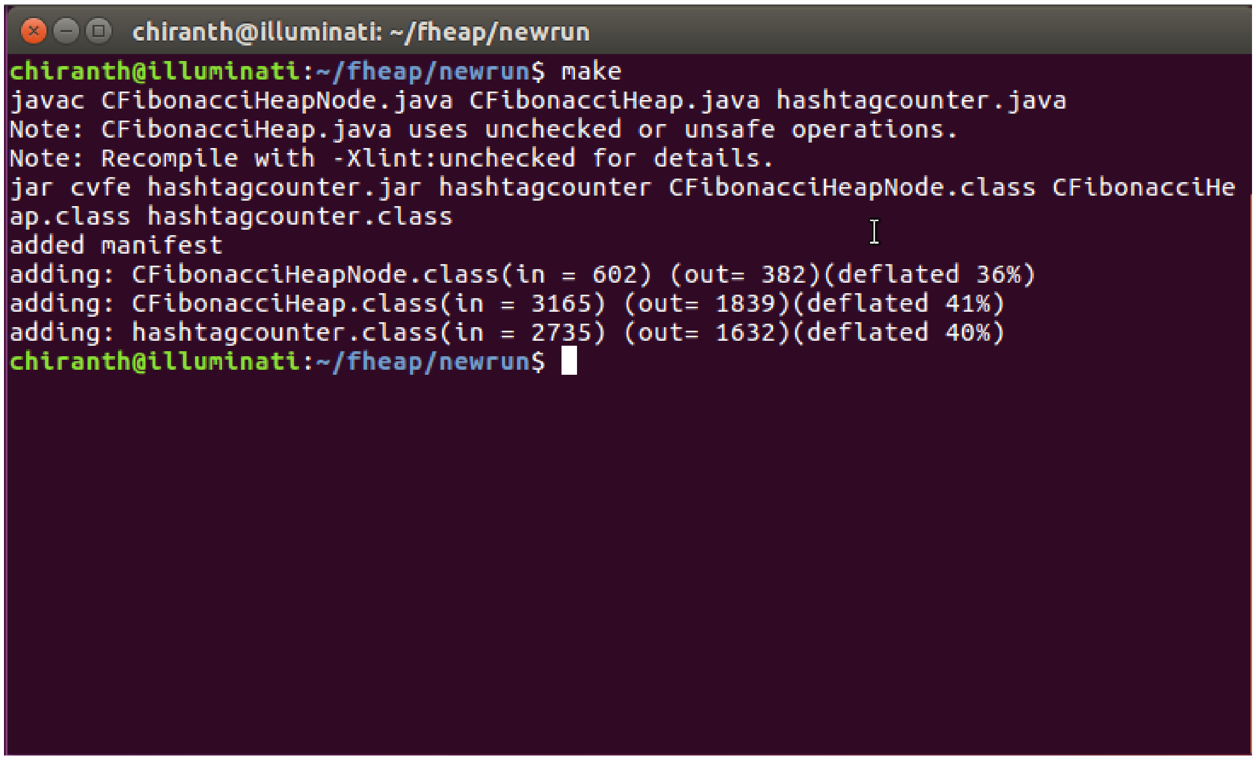
If the line read contains '#' then we call the **`hashtags.insert(parts[0], Integer.parseInt(parts[1].trim()))`** to insert the hashtagname and the frequency of the hashtag to the heap.

If the line read contains 'STOP', we stop the processing and exit from the program.

If the line read contains an integer 'numremoval', then we call **`hashtags.removemax()`** 'numremoval' times and write the hashtagname of the removed node into output_file.txt using a file writer in the same order as removed. Once we are done with all the removals, we call **`hashtags.reinsert_delitems()`** to reinsert all the deleted nodes back to the Fibonacci Heap.

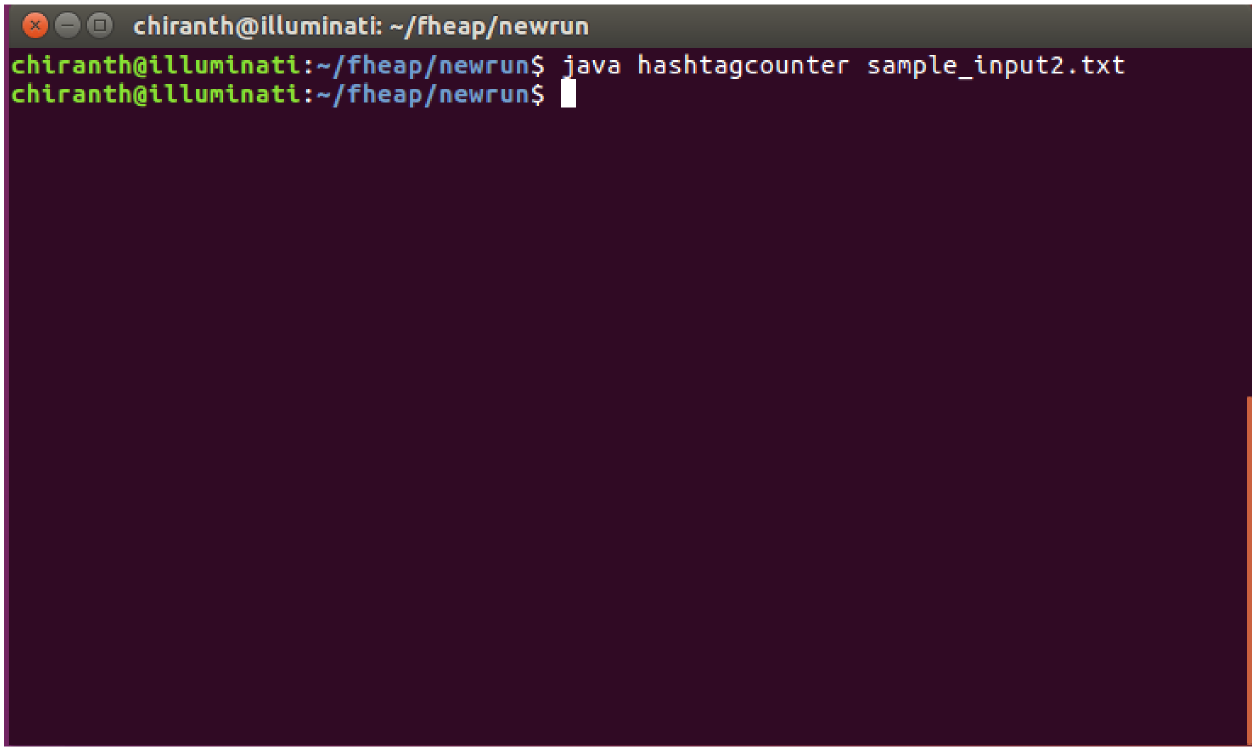
Running the program

Compiling the files and generating the jar executable



```
chiranth@illuminati:~/fheap/newrun$ make
javac CFibonacciHeapNode.java CFibonacciHeap.java hashtagcounter.java
Note: CFibonacciHeap.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
jar cvfe hashtagcounter.jar hashtagcounter CFibonacciHeapNode.class CFibonacciHe
ap.class hashtagcounter.class
added manifest
adding: CFibonacciHeapNode.class(in = 602) (out= 382)(deflated 36%)
adding: CFibonacciHeap.class(in = 3165) (out= 1839)(deflated 41%)
adding: hashtagcounter.class(in = 2735) (out= 1632)(deflated 40%)
chiranth@illuminati:~/fheap/newrun$
```

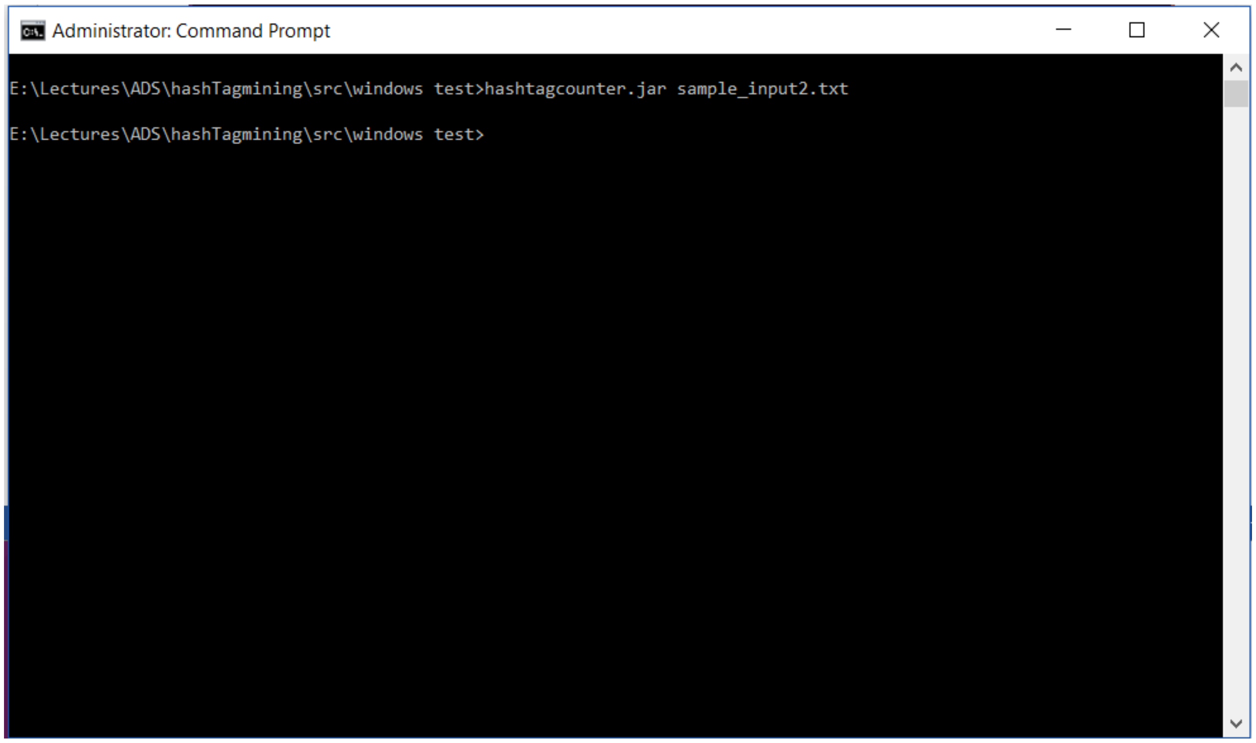
Executing the program: Method 1



chiranth@illuminati: ~/fheap/newrun

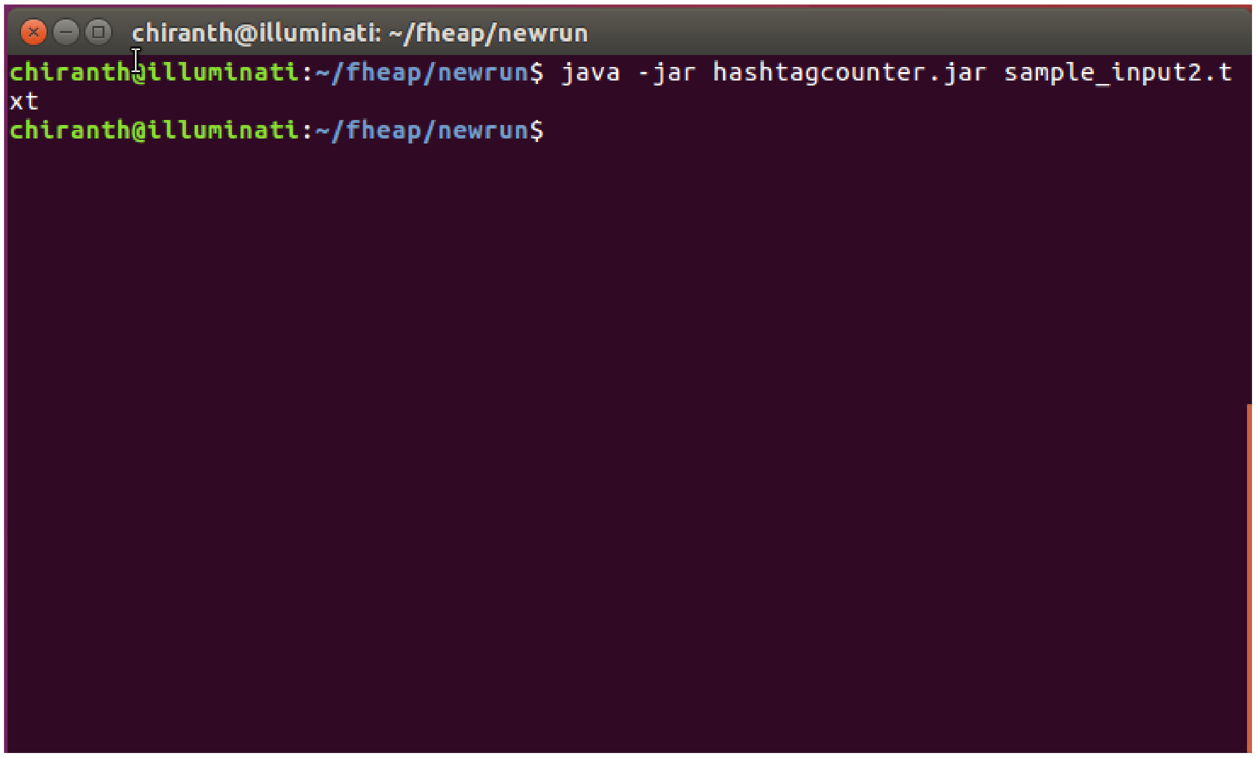
```
chiranth@illuminati:~/fheap/newrun$ java hashtagcounter sample_input2.txt
chiranth@illuminati:~/fheap/newrun$ 
```

Executing the program: Method 2



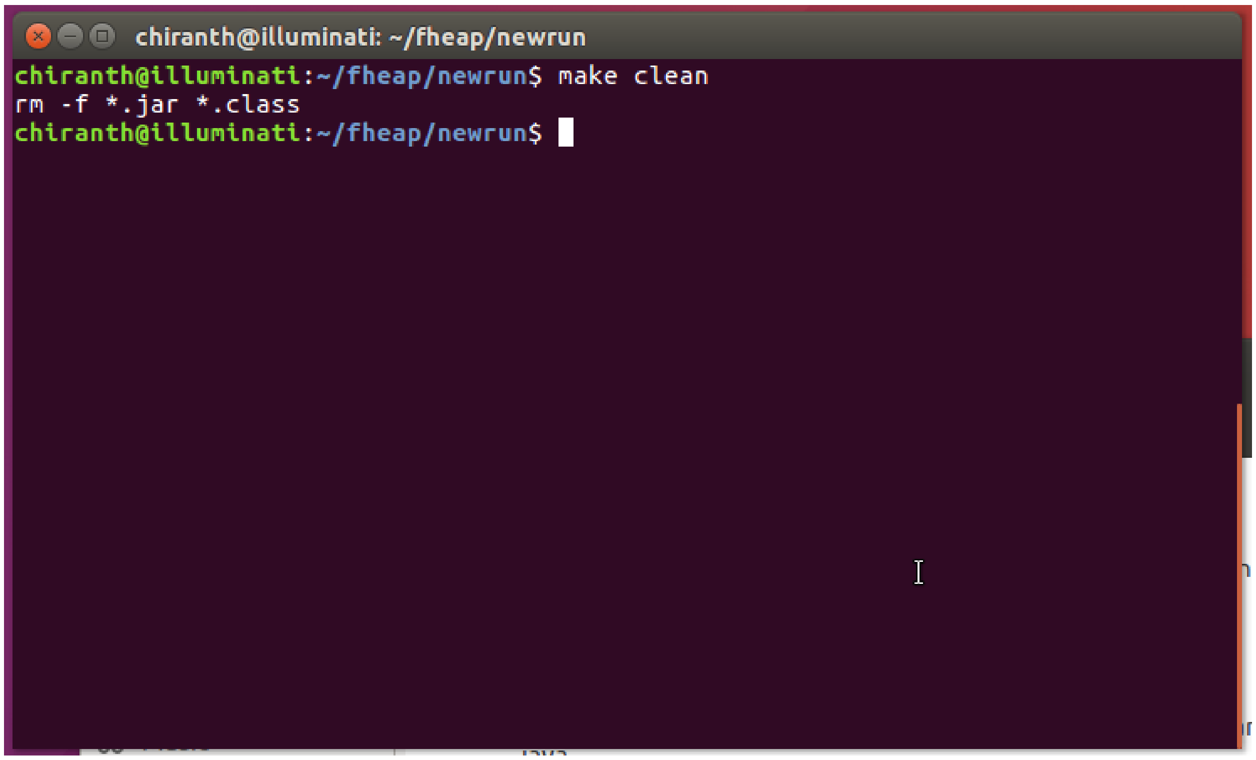
```
E:\Lectures\ADS\hashTagmining\src\windows test>hashtagcounter.jar sample_input2.txt

E:\Lectures\ADS\hashTagmining\src\windows test>
```

Executing the program: Method 3



```
chiranth@illuminati:~/fheap/newrun$ java -jar hashtagcounter.jar sample_input2.t
xt
chiranth@illuminati:~/fheap/newrun$
```

Cleaning the class files and the executables:



chiranth@illuminati: ~/fheap/newrun

```
chiranth@illuminati:~/fheap/newrun$ make clean
rm -f *.jar *.class
chiranth@illuminati:~/fheap/newrun$ 
```