ADVANCED DATA STRUCTURES (COP 5536)

POPULAR KEYWORDS USED IN A SEARCH ENGINE

November 15, 2018

Prudhvee Narasimha Sadha
UFID: 4689 9656
prudhvee.sadha@ufl.edu
Department of Computer & Information Science & Engineering
University of Florida - Fall 2018

PROBLEM STATEMENT:

A new search engine "DuckDuckGo" is implementing a system to count the most popular keywords used in their search engine. They want to know what the n most popular keywords are at any given time. We are required to undertake that implementation. Keywords will be given from an input file together with their frequencies. We need to use a max priority structure to find the most popular keywords.

IMPLEMENTATION:

The following data structures are used to implement the project.

1. **Max Priority Queue** - The max priority queue is used to keep track of the most popular keywords used in the search engine. We implement **Fibonacci Heap** in the project because of its optimal amortized complexities.

Fibonacci Heap Amortized & Actual Complexities		
Operation	Actual	Amortized
Insert	O(1)	O(1)
ExtractMax	O(n)	$O(\log n)$
Meld	O(1)	O(1)
Remove	O(n)	$O(\log n)$
IncreaseKey	O(n)	O(1)

2. **HashTable** – We use the Java utility HashTable to store the keywords and their corresponding frequency nodes of the Fibonacci Heap.

FILES USED IN THE PROJECT:

The following are the files implemented in the project.

1. **Node.java** - The class 'Node' contains the node structure of the Fibonacci Heap.

2. **FibonacciHeap.java -** This class defines the methods required to implement 'Fibonacci Heap'.

3. **keywordcounter.java** - This is the main class that reads the input, uses Fibonacci Heap to store the frequencies of the corresponding keywords in the Fibonacci Heap and a HashTable to map the keywords to the corresponding frequencies in the Fibonacci Heap.

METHODS AND VARIABLES DESCRIPTION & PROJECT FLOW:

1. **Node.java**

Node structure		
Variable	*Data Type*	*Description*
degree	Integer	The degree of the corresponding Node.
name	String	The keyword that the key in the node corresponds to.
key	Integer	The frequency of the keyword that the Node corresponds to.
parent	Node	The parent node of the corresponding node.
child	Node	The child node of the corresponding node.
left	Node	The left sibling of the corresponding node.
right	Node	The right sibling of the corresponding node.
mark	boolean	The child cut value of the corresponding node.

2. **FibonacciHeap.java**

Variables		
Variable	*Data Type*	*Description*
myMax	Node	Pointer to the Node that contains the largest key in the Heap.
myNumberOfNodes	Integer	Stores the number of nodes present in the Heap.

Methods		
void FibHeapInsert(Node x)		
Description	Insert the Node x into the Fibonacci Heap H	
Parameters	Node x	Node to be inserted into the Fibonacci Heap.
Return value	void	

void FibHeapIncreaseKey(Node x, Integer k)		
Description	Increase the key of a Node to the new value passed.	
Parameters	Node x Integer k	The Node. The new value that the Node key is to be increased to.
Return Value	Void	

Node FibHeapExtractMax()	
Description	Extract the Node with the largest key from the Fibonacci Heap.
Parameters	No parameters are passed to the method.
Return value	Node with the largest key in the Fibonacci Heap.

void consolidate()	
Description	Pairwise combine the nodes in the root list with same degree. myMax will now point to a new node with largest key.
Parameters	No parameters
Return Value	Void

void fibHeapLink(Node y, Node x)		
Description	Make Node y the child of Node x and remove Node y from the root list.	
Parameters	Node y	The Node that is to become the child.
	Node x	The Node that is to become the parent of the child.
Return Value	Void	

void cut(Node x, Node y)		
Description	Remove the Node x from the children list of the Node y. Add Node x to the root list.	
Parameters	Node y	The parent Node whose child is to be removed.
	Node x	The Node who is to be removed and added to the root list.
Return Value	Void	

void cascadingCut(Node y)		
Description	Initiate Cascading cut from the Node y. If mark of Node y is set to false, make it true. Else remove the Node y as child from its parent and add to the root list. Initiate cascading cut on the parent of the Node y.	
Parameters	Node y	The Node from where the cascading cut is to be initiated.
Return Value	Void	

3. **keywordcounter.java**

Variables		
Variable	*Data Type*	*Description*
startTime	long	The start time of the flow in milli seconds.
reader	BufferedReader	Reader to read input from the file using FileReader.
writer	BufferedWriter	Writer to write the output to a file using FileWriter.
hashTable	HashTable<String, Node>	The hashtable data structure to map the keywords to their frequencies.
H	FibonacciHeap	The Fibonacci Heap object that we use as max priority queue to store the frequencies of the keywords.
line	String	Reads each line of the input file to process.
endTime	long	The end time of the flow in milli seconds.
totalTime	long	The total time of the flow in milli seconds.

Control flow of the main method	
if block	Check if the line starts with $ If yes, then add the keyword to the hashtable if not present If present, then increase the frequency of the keyword by the new value.
else if block	Check if the line is a digit *n* If yes, extract the top *n* Nodes from the heap H and write them to the output file. Reinsert the extracted Nodes into the Heap H.
last *else if* block	If the line is *stop* without $ sign, then break.

Project Flow:

1. Read each line from the input file, until the file is completed or the line *stop* is encountered.
2. For each line, if the line encountered starts with a *$*, then if the keyword is already present in the Hashtable increase the frequency by the new value. If not present, then add the keyword to the Hashtable and the frequency to the Fibonacci Heap.
3. If the line encountered is a number *n*, then extract the top *n* keys from the Fibonacci Heap and write the corresponding keywords to the output file in a new line in the decreasing order of the frequencies.

Compiling and Running Instructions:

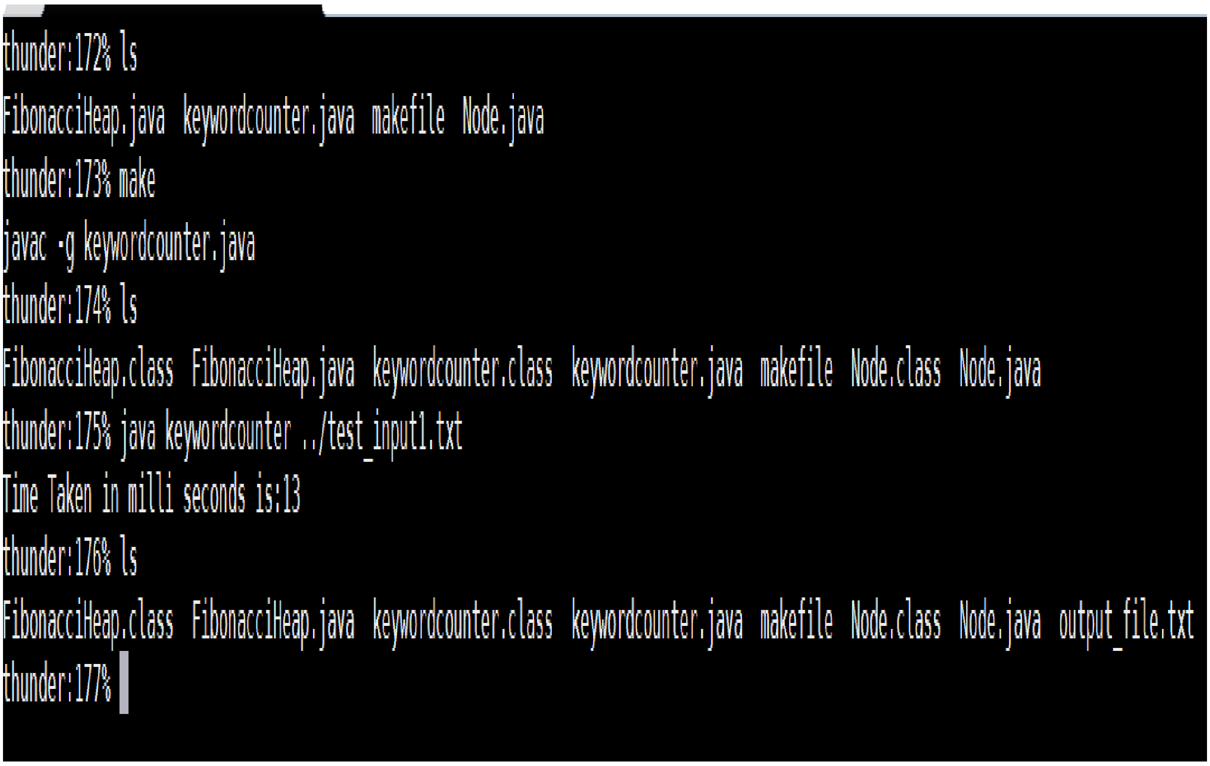
The program is compiled and tested in a local 64 bit machine of 8 GB RAM with java compiler *javac* in Java 8 JRE and also on *thunder.cise.ufl.edu* server using PuTTY.

Steps to execute the program:

- Extract the contents of the Zip file to a folder.
- Run ***make*** to compile the project. This creates the executable *keywordcounter*.
- Now, you can run the program using any input file. The syntax to run is :
  java keywordcounter /path/to/the/input/file.txt
- To run the program on an input file using make, run
  make run INPUT=/path/to/the/input/file.txt
- To run the program on the sample input file millionInput.txt that is in the folder, try ***make run***

RESULTS:

The code was run over an input file with a million queries and the results are generated as expected and the average total time to run the program was noted to be around 800 ms. The order of the keywords for each write query might be different as there can be multiple keywords with the same frequency and is completely depended upon the queries executed till the current query.



```
thunder:172% ls
FibonacciHeap.java  keywordcounter.java  makefile  Node.java
thunder:173% make
javac -g keywordcounter.java
thunder:174% ls
FibonacciHeap.class  FibonacciHeap.java  keywordcounter.class  keywordcounter.java  makefile  Node.class  Node.java
thunder:175% java keywordcounter ../test_input1.txt
Time Taken in milli seconds is:13
thunder:176% ls
FibonacciHeap.class  FibonacciHeap.java  keywordcounter.class  keywordcounter.java  makefile  Node.class  Node.java  output_file.txt
thunder:177%
```

CONCLUSION:

The problem statement of the project is implemented and the requirements are met by using Fibonacci Heap as the max priority queue data structure to find the top n keywords.