

## Project Description

The project implements a system that gives the most popular keywords used in a search engine at any given time. Keywords are fed to the system from an input file that has the corresponding frequencies of the keywords. The desired output is what the  $n$  most popular keywords are at any given time.

The implementation is done using a Max Fibonacci Heap. The program reads the input file and adds the keywords and corresponding frequencies to the Fibonacci heap. If keywords are repeated in the input file, the program increases the value associated to that keyword which is already present in the heap. And when the user wants to get the top  $n$  keywords, the program removes the maximum of the heap  $n$  times, writes the output to a file and re-inserts the removed nodes back into the heap.

To achieve the requirements, the program uses a HashTable (java.util) to maintain the keywords and their corresponding Node pointers in the heap.

The program is written in Java. Compiled using javac. Except for the HashTable, the program doesn't use any other built-in data types.

<b><u>Fibonacci Max Heap</u></b>	
Amortized Complexity	
Insert	$O(1)$
Remove Max	$O(\log n)$
Increase Key	$O(1)$
Meld	$O(1)$

To achieve the amortized complexity, increaseKey() operations are followed by cascading cuts.

## How to run the program:

1. Logon to thunder.cise.ufl.edu
2. Extract the zip so the folder contents are present on disk
3. Run the makefile with 'make' on command line which compiles the java files
4. Run 'java keywordcounter *input\_file*' on command line where *input\_file* is the path to the input file.
5. There is now a file named as 'output\_file.txt' in the current directory with the desired result as the content.

## Structure of the program

The program is made of 2 java files:

- **keywordcounter.java** – Contains the main method which handles the file read and write operations. Uses an object of the FibonacciHeap class to add the content it reads from the file into the heap.
- **FibonacciHeap.java** – Has the core implementation of the max Fibonacci heap and all the supporting functions. It has the Node structure as an inner class.

The working of the program is as follows:

1. **keywordcounter.class** takes the path to the input file as an argument. It then parses the file line by line.
2. If the line reader reads two words, separated by spaces it then calls the 'addToHeap' function of the **FibonacciHeap.class**. This function internally checks if the keyword needs to be added as a new node ['add'] or to increase the value ['increaseValue'] of an existing node by taking help of a HashTable.
3. Whenever the 'increaseValue' function is called, it is followed by a series of cascading cuts performed so as to achieve the amortized complexity.
4. Going back to reading the file input, whenever there is a query with only a number ['n'] mentioned in a line, the program calls the 'getTopX' function which internally calls the 'removeMax' function of the heap 'n' times.
5. The program then writes the data present in these max Nodes to an output file. And finally re-inserts these removed nodes back into the heap to maintain the heap as it initially was.
6. 'removeMax' internally does pair wise merging of the nodes present in the top/root list according to their corresponding degrees.
7. The program also stops execution when there is a 'stop' keyword encountered in the input file.

## Class Structure

### 1. keywordcounter.class

Variable Name	Data Type	Description
inputFilePath	String	Path to the input file from commandLine argument
reader	BufferedReader	To read data from input file
writer	BufferedWriter	To write data into output file
fHeap	FibonacciHeap	Object of FibonacciHeap class
resultBuilder	StringBuilder	To store the program output during each query and finally write to output file

### 2. Node.class

Member Variables	Data Type	Description
keyword	String	Keywords mentioned in the input file
value	Integer	Frequency value of the corresponding keyword
nextSibling	Node	Store address of the sibling to the right/next of this node in the circular linked list
prevSibling	Node	Store address of the sibling to the left/previous of this node in the circular linked list
degree	Integer	To store degree [number of children] of a node
child	Node	Store address of child to this node
parent	Node	Store address of parent of this node
childCut	boolean	ChildCut value of this node [true/false]

### 3. FibonacciHeap.class

Member Variables	Data Type	Description
heapMax	Node	Points to the node with the max value of heap
numberOfNodes	Integer	Maintains the total number of nodes present in the heap
hashTable	HashTable	Hash table to store keywords and corresponding Node addresses.

## Method Definitions

### 1. public void addToHeap (int val, String keyname)

Description	If keyword is already present in the heap, looks up the node from hashTable and then calls 'increaseValue()' function. If not present, creates a node and calls 'add(newNode)' function	
Parameters	keyname	Keyword to be checked
	val	Frequency of the keyword
Return value	void	

### 2. public void add (Node newNode)

Description	Inserts newNode into the top circular linked list of Fibonacci heap and updates heapMax if required.	
Parameters	newNode	Node to be inserted
Return value	void	

### 3. public void increaseValue (int val, String keyname, Node matchedNode)

Description	Increases the already present 'matchedNode' value by 'val'. Then, calls the 'makeCut()' and 'makeCascadingCut()' functions when 'matchedNode.value' is greater than that of its parent's value. Updates heapMax position if required.	
Parameters	val	Value by which frequency of the node needs to be increased
	keyname	Keyword of the node to be increased
	matchedNode	Node present in the heap that matches the input keyword
Return value	void	

### 4. private void makeCut (Node matchedNode, Node parentNode)

Description	Cuts the matchedNode from its parentNode and melds it into the top circular list by calling 'meldToTopList()' function. Updates the parent and child pointers of both the nodes and decreases the degree of the parentNode by 1.	
Parameters	matchedNode	Node that needs to be cut from its parent and melded into top circular list
	parentNode	Node which is the current parent to the 'matchedNode'
Return value	void	

### 5. private void makeCascadingCut (Node node)

Description	Cuts the node from its parent when the parent's childCut is true. Iteratively calls itself and stops when a parent's childCut value is false.	
Parameters	node	Node on which cascading cut is done
Return value	void	

#### 6. public Node removeMax ()

Description	Removes the heapMax node and then melds its children if any to the top circular list by calling 'meldToTopList()' function. Removes the keyword corresponding to heapMax from the Hash Table. Finally, calls the 'pairwiseCombine()' function
Parameters	None
Return value	Node which is the heapMax

#### 7. private void pairwiseCombine ()

Description	Creates a degree Table and uses this to combine nodes in the top circular list that have same degree. Finally, re-creates the heap with updated nodes and updates the heapMax. Utilizes 'makeChild()' and 'meldToTopList()' functions internally
Parameters	None
Return value	void

#### 8. private void makeChild (Node parentNode, Node childToBe)

Description	Makes the 'childToBe' node as a child to the 'parentNode' while updating the pointers and degrees of both the nodes		
Parameters	parentNode	Parent to be for the 'childToBe' node	
	childToBe	Node that needs to be made a child to 'parentNode'	
Return value	void		

#### 9. private void meldToTopList (Node nodeToMeld)

Description	Melds the 'nodeToMeld' node to the top circular list of the heap. This node could also be a head to a circular list and the function melds it to the top list accordingly.		
Parameters	nodeToMeld	Single node or pointer to a circular list that needs to be meld to the top list of heap.	

Return value	void
--------------	------

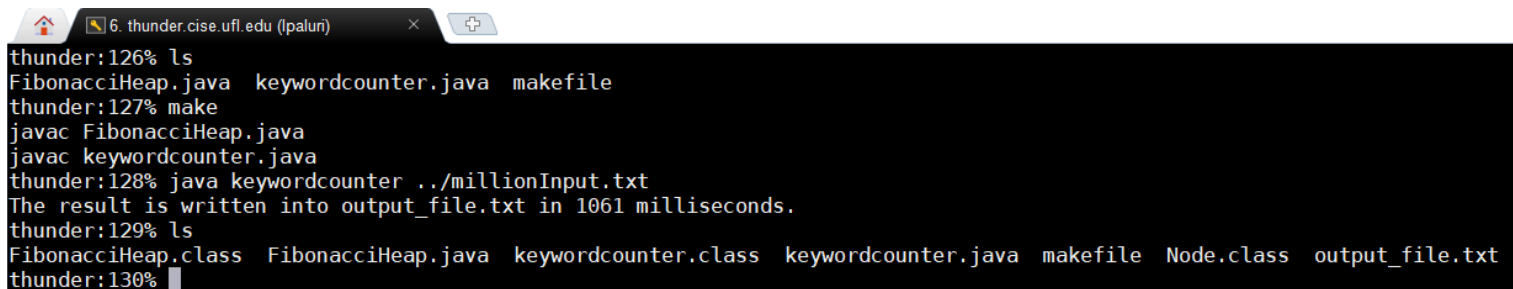
#### 10. public Node[] getTopX (int topX)

Description	Extracts the 'topX' maximum values of the heap one by one and returns them as an ArrayList. It then re-inserts these nodes back into the heap.		
Parameters	topX	Number of maximums to be read from the heap	
Return value	Node[] Contains array of 'topX' maximum values of the heap		

## Results

The program was run for 1 million inputs and the output time was found to be around 900ms. It was also run on the sample input given, and the desired output was generated in around 8ms.

The order of the keywords in the output can be however different when there are several different keywords with same frequencies.



```

thunder:126% ls
FibonacciHeap.java keywordcounter.java makefile
thunder:127% make
javac FibonacciHeap.java
javac keywordcounter.java
thunder:128% java keywordcounter ../millionInput.txt
The result is written into output_file.txt in 1061 milliseconds.
thunder:129% ls
FibonacciHeap.class FibonacciHeap.java keywordcounter.class keywordcounter.java makefile Node.class output_file.txt
thunder:130%

```

## Conclusions

The program successfully implements a max Fibonacci Heap to operate on several keywords and frequencies. The project achieves the amortized complexity and offers increased performance.