COP 5536 Fall 2018 Project Report

Most Popular Keywords using Fibonacci Heap and Hash table

Akash Shingte
UFID: 4874-1966
UF Email: a.shingte@ufl.edu

November 15, 2018

1 Introduction

The project was based on the idea to count most popular keywords at any point of time for a search engine. The task was to be accomplished using a priority queue which will keep track of all the frequencies of a keyword and a hash table which will keep track of the node in the priority queue belonging to a keyword.

1.1 Fibonacci Heap

A Fibonacci heap is a data structure which has a application as a priority queue. It consists of heap ordered trees where each sub tree maintains the heap priority. Fibonacci heaps are known for their constant amortized time for operations like insert, meld and decrease(increase) key and logarithmic amortized time for remove(max or arbitrary) element.

	Actual Time	Amortized Time
Insert	O(1)	O(1)
Extract max(min)	O(n)	O(lg n)
Increase(decrease) key	O(n)	O(1)

1.2 Hash tables

Hash table is an associative array data structure in which data is store in key-value pairs. Hash tables uses a hash functions which decide which bucket an element is stored or looked up. Hash tables have a property that they give constant time search and insert in average case.

	Average	Worst
Search	O(1)	O(n)
Insert	O(1)	O(n)

2 Compilation

The project can be compiled using the make build automation tool. The makefile is included with the project which compiles the project and creates executable file.

```
$ make
```

The created class files can be cleaned with

```
$ make clean
```

In order to create a jar file we can run

```
$ make jar
```

3 Execution

If the project has been compiled just using the make functionality it can be executed using the command -

```
$ java keywordcounter [input file]
```

If a jar file was created using make jar, it can be executed using

```
$ java -jar keywordcounter [input file]
```

4 Assumptions

As per the requirements, following assumptions have been considered for execution of the project.

- Input file is in text(.txt) format.
- Input keyword can be any arbitrary string including alphanumeric characters and special symbols.
- The count/frequency can only be a positive number greater than 0.
- No spaces in the keywords.
- For two keywords to be same, whole word should match. youtube and youtube_music are two different keywords. Also, uppercase and lowercase are treated differently.

- One query has only one integer.
- If there are more than one keyword with similar frequencies, they may appear in the output in any order.
- Query integer is less than or equal to 20
- Input file may consist of more than 1 million keywords. Maximum limit for the number of keywords was not specified. Hence, assuming a input file of more than 1 billion keywords.
- Input file format

```
$[keyword] [frequency]
.....
.....
.....
[query integer]
.....
.....
.....
stop
```

The keyword and query integer can appear any number of times and at any location in the file.

5 Expected Output

The output file will consists of keywords for every query integer. If the query integer is greater than the number of keywords appeared then all the keywords are output to the file. If the query integer appears before any of the keyword appears then a blank line appears in the output.

6 Project Structure

In this section, we will describe the various classes and methods of the project.

6.1 HeapNode Class

This class represent a particular node in a Fibonacci Heap. The fields can be summarized as follows -

```
- fields
    parent :: HeapNode :: Pointer to the parent or null
    child :: HeapNode :: Pointer to the child or null
```

```
    degree :: Integer :: Number of children of the node
    leftSibling :: HeapNode :: Pointer to the left sibling
    rightSibling :: HeapNode :: Pointer to right sibling
    mark :: Boolean :: Field to track the child lost by node
    frequency :: Integer :: The frequency of the keyword
- public functions
    HeapNode :: HeapNode :: constructor
```

6.1.1 Fields

1. **parent**
   Pointer to the parent of the node. If the node is at the root level list, the parent pointer is null.
2. **child**
   Pointer to the child of the node. If the node has no children, the child pointer is null. If the node has multiple children, the child pointer points to the first child though which rest of the children can be accessed.
3. **degree**
   Indicates how many children a node has. Can take values from 0 to Integer.MAX_INT
4. **leftSibling**
   Pointer to the left sibling of the node in the circular doubly linked list. If the node is the only node in the list the pointer will point to the node itself.
5. **rightSibling**
   Pointer to the right sibling of the node in the circular doubly linked list. If the node is the only node in the list the pointer will point to the node itself.
6. **mark**
   A Boolean field to indicate whether the node has lost a child or not. Initially, when the node is inserted in the root level list the field is false. If the node is at root level list, this field is not of much use. If the node is in a list below root level list, the mark field indicates whether the node is to be cut from it's parent and inserted into the root level list. The field takes true value when it itself loses one of it's child. If it loses another child, the node will be inserted in the root level list.
7. **frequency**
   The frequency on the keyword on the basis of which we maintain the max heap property of the Fibonacci heap.

6.1.2 Function Prototypes

1. **HeapNode constructor**

public HeapNode(int frequency)		
Description	Initialize the attributes of the HeapNode object	
Parameters	frequency	The frequency of the keyword
Return	HeapNode object with variables initialized	

6.2 FibonacciHeap Class

This class represents the Fibonacci heap data structures. The structure of this class can be summarized as follows -

```
- fields
    max :: HeapNode :: Points to the max element of the heap
    nodes :: Integer :: The number of nodes in the heap
    MAX_DEGREE :: Integer :: The maximum degree to be expected in consolidate

- public functions
    FibonacciHeap :: FibonacciHeap :: constructor
    insert(int frequency) :: HeapNode :: insert a new node in heap
    insert(HeapNode node) :: HeapNode :: insert a already created node back into heap
    increaseKey(HeapNode node, int amount) :: Void :: Increase key of the node by amount
    extractMax() :: HeapNode :: Removes the max element and returns it

- private functions
    insertInList(HeapNode head) :: HeapNode ::
    Insert a node into the circular doubly linked list pointed by head

    removeFromList(HeapNode head, HeapNode node) :: HeapNode ::
    Remove from the circular doubly linked list pointed by head

    countNodesInList(HeapNode head) :: Integer ::
    counts the nodes in a circular doubly linked list

    separateNodesInList(HeapNode head):: HeapNode[] ::
    Separates the nodes in circular doubly linked list into array

    cut(HeapNode node, HeapNode parent) :: Void ::
    Remove the node from parent & insert into root level list

    cascadingCut(HeapNode node) :: Void ::
    Recursive function to cut nodes along the path from node to root

    consolidate() :: Void :: Pairwise combine the nodes in root level list

    heapLink(HeapNode small, HeapNode big) :: Void ::
    Makes small the child of big in root level lists
```

6.2.1 Fields

1. **max** The pointer to the Max Fibonacci Heap. It always points to the maximum element in the Heap. If there are no elements in the Heap the max is null.

2. **nodes**
   The number of nodes in the Fibonacci Heap.

3. **MAX_DEGREE**
   The maximum degree of a node in the Fibonacci Heap. It has been proven that for Fibonacci heap containing n nodes the maximum degree of a node has to be O(log n).

6.2.2 Public Function Prototypes

1. FibonacciHeap Constructor

public FibonacciHeap()	
Description	Initialize the attributes of the FibonacciHeap object
Return	FibonacciHeap object with variables initialized

2. insert(int frequency)

public HeapNode insert(int frequency)		
Description	Insert a frequency node in the max heap. The method creates a new node with specified frequency and inserts, the node in the top level list of the fibonacci heap. A node corresponding to a keyword is created just once,If a keyword appears again in the input, the same node's frequency will be increased and the structure of the, fibonacci heap will be altered. The complexity of this function is O(1).	
Parameters	frequency	The frequency of the keyword which appeared in the input file
Return	The reference to the node which is inserted into the FibonacciHeap	

3. insert(HeapNode node)

public HeapNode insert(HeapNode node)		
Description	Insert a node without creating a new node. This method is useful when an already extracted node is to be inserted back into the heap. The node will be inserted in the root level list and takes O(1) time.	
Parameters	node	The node which was previously created
Return	The reference to the node which is inserted into the FibonacciHeap	

4. increaseKey(HeapNode node, int amount)

public void increaseKey(HeapNode node, int amount)		
Description	An operation to increase the frequency of the node to a new number. A pointer to the node exists from the external worldIncreasing a frequency of the node may,violate the max heap property hence the node is severed from it's parent and inserted in a top level list,In order to get a better amortized complexity a mark field is maintained on each node to track its history. If a node loses one child the mark field is set to true otherwise it stays false. If a node loses a second child the node is severed from it's parent as well and a cascading cut will take place in this way.	
Parameters	node	The node whose frequency is to be increased
	amount	Amount by which the frequency is to be increased
Return	The reference to the node which is inserted into the FibonacciHeap	

5. extractMax()

public HeapNode extractMax()	
Description	This method removes the maximum node of the Fibonacci heap and returns that node. When the node is removed, the children of the node are added to the root level list. The nodes in the top level list are then all pairwise combined in consolidate() method.
Return	The maximum node of the Fibonacci Heap which is now removed from the heap

6.2.3 Private function prototypes

1. insertInList(HeapNode head)

private HeapNode insertInList(HeapNode head, HeapNode newNode)		
Description	A private function to insert a node in a circular doubly linked list. It takes O(1) constant time to insert in a circular doubly linked list. It returns the inserted node.	
Parameters	head	The pointer to the the circular doubly linked list
	newNode	The node to be inserted in the circular doubly linked list
Return	The newly inserted node in the circular doubly linked list	

2. removeFromList(HeapNode head, HeapNode node)

private HeapNode removeFromList(HeapNode head, HeapNode node)		
Description	A private function to remove a node from a circular doubly linked list. Returns the newly modified list. If there are no more,nodes remaining it will return null. It takes O(1) time to remove a node from a circular doubly linked list	
Parameters	head	The pointer to the the circular doubly linked list
	node	The node to be inserted in the circular doubly linked list
Return	The node removed from the circular doubly linked list	

3. countNodesInList(HeapNode head)

private HeapNode countNodesInList(HeapNode head)		
Description	Counts the nodes in a circular doubly linked list. Will take time proportional to the number elements in a circular doubly linked list which in the worst case is the number of nodes in the heap i.e. O(n)	
Parameters	head	The pointer to the the circular doubly linked list
Return	The number of nodes in the circular doubly linked list	

4. separateNodesInList(HeapNode head)

private HeapNode[] separateNodesInList(HeapNode head)		
Description	Separates the nodes in a circular doubly linked list into an array. This method is useful when the list is being modified and it makes it infeasible to iterate through the list in conventional manner.	
Parameters	head	The pointer to the the circular doubly linked list
Return	An array containing nodes in the circular doubly linked list	

5. cut(HeapNode node, HeapNode parent)

private void cut(HeapNode node, HeapNode parent)		
Description	A private function to cut the node from it's parent and insert it into root level list	
Parameters	node	The node to be severed from its parent
	parent	The parent of node from which it is to be severed

6. cascadingCut(HeapNode parent)

private void cascadingCut(HeapNode parent)		
Description	A private function which initiates a cascading cut and makes changes in a node's mark if they have already lost a child. This functions is called on a parent of the node. If the parent has lost it's first child the mark of the parent will be set to true and no further cascading cut will take place. If the parent has lost it's second child the node will be cut from the parent and a recursive call to cascading cut will be called using the node's parent.	
Parameters	parent	The parent of the node which had undergone the cut and intiated a cascadingCu call to its parent for better amortized time

7. heapLink(HeapNode small, HeapNode big)

private void heapLink(HeapNode small, HeapNode big)		
Description	Links a node whose frequency is smaller to a node whose frequency is greater. The small node becomes the child of the big node.	
Parameters	small	The node with frequency less than big node
	big	The node with frequency greater than small node

8. consolidate()

private void consolidate()	
Description	This method is responsible of pairwise combining of nodes based on degree field. It uses a degree table of MAX_DEGREE size whose upper bound is O(lg n) in the number of nodes expected in the heap. The consolidate() helps us to achieve the required amortized time for Fibonacci heap operations.

6.3 Runner class

This class is responsible for parsing the file, extracting keywords, creating heap and answering the query for most popular keywords when they appear in the input file. The class can be summarized as follows -

```
- static public function
    run :: Void :: String[] args

- static private functions
    parseLine :: Integer ::
    Parses the line in the input to decide
    a course of action
```

```
isValid :: Boolean ::
Checks whether the line in the input conforms to
the given requirements
```

6.3.1 Public function prototypes

1. run(String[] args)

public static void run(String args[])		
Description	This functions is responsible for parsing the file and taking appropriate action. The file may contain a keyword with a frequency in which case if the keyword is new, it is added to the heap and the reference of the node in the heap is stored in the hash table. A reverse hash table stores the key as node and value as keyword for better retrieval of keyword. If there is a query for most popular keywords, it will extract maximum from the heap for ‘query‘ number of times and insert it back again in the heap. If the file encounters a ‘stop‘ keyword it will stop the execution of the program.	
Parameters	args	Arguments passed to the main program which includes the input file

6.3.2 Private function prototypes

1. parseLine(String line)

private static int parseLine(String line) throws IOException		
Description	Parses the line and returns an integer specifying which type of line it encountered. Checks the validity of the line and throws exception if it is not valid.	
Parameters	line	The line to be parsed
Returns	The integer specifying action to be taken. If it encounters a keyword in the format $[keyword] [frequency], it will return 1. If it encounters a query integer it will return 2. If it encounters a stop message it will return 3.	

2. isValid(String line)

private static int isValid(String line)		
Description	Checks the line and determined whether it is in correct format. Checks the validity of the line by comparing it with regular expressions.	
Parameters	line	The line to be checked for its validity
Returns	True, if the line is valid and matches the regular expression Else, False	

6.4 keywordcounter

A class to pass the command line arguments to runner class. This class is required in order to satisfy the requirements of the project for a executable with this name. The naming conventions of Java are ignored in this case.

7 Conclusion

In this project, a method to get most popular keywords for an application using a Fibonacci Heap Priority queue was explored. The keywords will only have instance in heap and an external pointer to the node through which an increase key operation is done. The most popular keywords at any instance of time is obtained by extracting max from the heap and inserting it back again in the heap. The amortized time is obtained by doing ‘consolidate()‘ and ‘cascadingCut‘ operations.