Advances Data Structures (COP 5536)

Fall 2016

Programming Project Report

Vinayak Deshpande

vsd335@ufl.edu

UF ID: 41029538

PROJECT DESCRIPTION

The goal of the project is to implement hashtag counter using Fibonacci Heaps. We are required to use Max Fibonacci heap structure to implement the hash tag counter.

The program reads an input text file containing hashtags with count values, and queries. The heap is initialized with the given hashtags and then based on query, the hashtags with maximum count values will be written to the output file.

Hash tables are used to keep track of all the hashtags (keys) and their corresponding nodes (values) in the max Fibonacci heap. We are to use some of the common operations performed on Fibonacci heaps such as Insert() to insert new nodes for the hashtags, IncreaseKey() to increase the count of already appeared hashtags. RemoveMax() will remove a hashtag with maximum count value. Hashtags with similar frequency are allowed to be written in any order.

PROGRAMMING ENVIRONMENT

Operating System: Windows 10
Language Used: Java
Development Environment: Eclipse Mars
Java Runtime Environment: Jre 1.8.0_111

COMPILING INSTRUCTIONS

The project is compiled and tested on **thunder.cise.ufl.edu server** and Eclipse Mars was used in windows to implement the code.

Instructions

Step1: Remote access: username@thunder.cise.ufl.edu

Step2: Navigate to the directory containing the project.

Step3: To generate executable (.class)

Command: make

Step4: To run the executable with input file

Command: java hashtagcounter file_name

Step5: Verify the generated output file

NODE STRUCTURE FOR THE FIBONACCI HEAP

The whole project is implemented in one single java class called '**hashtagcounter'** which has a subclass called '**HeapEntry'**, which defines the basic structure of a node in a Fibonacci heap.

The node structure is as shown below.

```
/**
 * This inner class holds the node structure for the Fibonacci heap with private fields
**/
public static final class HeapEntry {

    private int hDegree = 0;                // Number of children in the node

    public HeapEntry hLeftSib;              // Left Sibling element
    public HeapEntry hRightSib;             // Right Sibling element

    public HeapEntry hParent= null;         // Parent in the fibonacci heap (if any)
    public HeapEntry hChild = null;         // Child node (if any)

    public int  hElem;                      // Element stored in the node
    public boolean hChildCut = false;       // Whether a node has lost a child or not

    // Set the initial left and right sibling of a new node to itself
    public HeapEntry(int elem, String hHashTag) {
        hRightSib = hLeftSib = this;
        this.hHashTag = hHashTag;
        hElem = elem;
    }

}
```

Definition of each node field is as given below,

hDegree (Type: Integer): This field is used to store the degree (number of children) of node in the heap

hLeftSib (Type: HeapEntry): This field stores the pointer to the right sibling of a node. Initially set using constructor.

hRightSib (Type: HeapEntry): This field stores the pointer to the right sibling of a node. Initially set using constructor.

hParent (Type: HeapEntry): This field holds the pointer to the parent node, it shall be initially set to **null**. Also all nodes at the root will have their parent filed set to **null**.

hChild (Type: HeapEntry): This field indicates pointer to the child of a node. A node can have any number of children but it can access all its children using only this pointer. All the children of a node are doubly linked on either side.

hElem (Type: Integer): This holds the count value of a hashtag.

hElem (Type: Boolean): This fields indicates whether a node has previously lost a child or not. Initially this field is set to **false**, when the node loses a child due to IncreaseKey() operation, we set this to true. The next time, this node will be removed from its parent and inserted back to the node.

Apart from all the fields defined above, I have included few other fields, that are not part of node structure, important ones are as given below,

hMax (Type: HeapEntry): This is a pointer to the node with maximum count value. This is defined public so that it can be accessed in any of the methods defined in the main class (hashtagcounter). This will help us easily navigate through the linked list of nodes in the root and also to easily remove the node with max element when RemoveMax() is called.

hSize (Type: Integer): Every time a new node is inserted into the heap, this variable is increased by one. This will help us know the exact number of nodes currently in the heap.

static HashMap <String, HeapEntry> hm = new HashMap<> (): This hash map is declared globally and holds all the hashtags in the heap along with their node pointers.

HashMap<Integer, HeapEntry> degreeMap = new HashMap<> (): This hash map is used to store the degree of a node and its node pointer while pairwise combine operation.

METHODS USED FOR FIBONACCI HEAP OPERATIONS

The '**hashtagcounter'** further has methods implemented to perform various operations on the heap. All the methods, input and return parameters and the functionality they perform are explained in detail below.

public static void main (String [] args)

Return type: void
Parameters: arguments like input file_name etc.

Description: This function basically reads the input file containing hashtags and count values. It also creates new nodes for each different hash tags and puts them in hash map called **hm**. It is also responsible to call methods like InsertNode and IncreaseKey() based on the input. Whenever a query appears, it calls RemoveMax() function and the returned hashtag is written into the output file. It also ensures that all the file handles are closed at the end. Another hash map is called **dict** is used to store the hashtag and count value (key value) of each removed node after a query is received. This list is later used to insert back all the removed nodes.

public void InsertNode (HeapEntry newNode)

Return type: void
Parameters: HeapEntry newNode

Description: Whenever a new hashtag appears, a **newNode** is created in the main function and is sent as input parameter to this method. This method sets its parent field to **null** and calls another method called 'InsertIntoRootList' and passes to it the max node in the heap and the new node. In this program, the new node is always inserted next to the max node. This method does not return anything. It also increases the heap size by one every time a new node is inserted. This method is also invoked by NodeCut method when a node's key after IncreaseKey() happens to be larger than that of its parent. A node which has lost more than one child will can also be inserted back into the root using this method.

public static HeapEntry InsertIntoRootList (HeapEntry oldNode, HeapEntry newNode)

Return type: HeapEntry
Parameters: HeapEntry oldNode, HeapEntry newNode

Description: This method receives two nodes as input parameters. The **newNode** will be inserted into the root next to **oldNode**. Before inserting, we need to make sure few important conditions like whether the nodes are null or not. If one of the nodes is null, the function will take care of it and manages to inserts the newNode. Upon successful completion, this method returns the node with maximum count value to the calling function. One of the important function of this method is to change the right and left siblings appropriately when a new node is inserted into the root list, especially when the root contains more than one node.

public void IncreaseKey (HeapEntry node, int newVal)

Return type: void
Parameters: HeapEntry node, int newVal

Description: Whenever the same hashtag re-appears in the input file, we need to make sure that, it not inserted as a new node, rather we only required to increase its key by accessing the node stored against it in the hash map. To this method, we pass the **node** and the **newValue** which is to be used to increase the previous count value. The other responsibility of this method is to verify if the increased value exceeds the count value of its parent in which case this node needs to be cut from its parent and inserted back into the root. This involves two method calls: **NodeCut** and **NodeCascade** each of these nodes perform different operations as discussed below.

public void NodeCut (HeapEntry childNode, HeapEntry parentNode)

Return type: void
Parameters: HeapEntry childNode, HeapEntry parentNode

Description: This method removes the **childNode** from its **parentNode.** The method checks for various conditions before removing the node from its parent and inserting back into the root. The conditions are as follows:
1. The childNode and the child registered with the parent are the same, if so, we need to make sure if the childNode has any other siblings. In case if the child has siblings, the right sibling is made a child of the parentNode and the childNode is inserted into the root.
2. If the childNode is the only child of parentNode or if the degree of parentNode is 1, we can go ahead and remove the childNode and insert it to root. Set the parent's child field to null.
3. In case if the childNode and the parentNode's child are different, we have to remove the childNode and safely exchange its left and right sibling pointers and then insert the childNode to root.

In all the three possible cases, decrease the degree of the parentNode by one.

public void NodeCascade (HeapEntry parentNode)

Return type: void
Parameters: HeapEntry parentNode

Description: This method takes as input the parentNode which has just lost a child due to IncreaseKey() operation. If this node has already lost before (hChildCut = true), it also needs to be removed from its parent and inserted back into the root. If this node has lost a child for the first time, we need to set its hChildCut field to **true.**

public HeapEntry RemoveMax ()

Return type: HeapEntry
Parameters: null

Description: Based on the input query, the main function invokes this method to perform the RemoveMax() operation, where the node containing the maximum count value will be

removed from the heap. The removed node will be returned back to calling function. The RemoveMax() operation has access to the current max node and can easily remove it from the root. But it has to make sure the children (if any) of the max node are inserted back to root node list. Once the children are added to root, it iterates through the root nodes to determine the new max node. This step is necessary as it helps for pairwise combining in the next steps. This method invokes two important methods namely, 'AddChildren2Root()' and RecursiveMerge().

public void AddChildren2Root (HeapEntry firstChild)

Return type: void
Parameters: HeapEntry firstChild

Description: When a max node is removed from the root, we have to make sure that, its children are added to the root. The **firstChild** is the received as input is the child of the removed max node. This pointer can be used to access all the children of the removed node as they are linked together. The simplest of cases is when the firstChild is null, which means the removed node had no children. Other cases include, the degree of the removed node is either one or more than one. This method handles all these three cases.

public void RecursiveMerge (HeapEntry pairNode1)

Return type: void
Parameters: HeapEntry pairNode1

Description: This is one of the major methods that performs pairwise combining of root nodes of same degree until no two root nodes have same degree after every RemoveMax() operation. This action is required because, we want minimize the complexity by minimizing the access time of node. If we don't do this operation, the root list will be very long and it makes it complex to access a node in log(n) time.

So, to combine nodes in pairs until all the root nodes have different degree, we need to invoke this method recursively. Every time two nodes with same degree are combined, node with maximum count value becomes the parent node and it stays at the root, whereas the smaller one becomes its child. So we have to remove the child node from the node and exchange its sibling nodes appropriately so that the root list is intact.

To keep track of degree of each node, I have used another hash map called **degreeMap**, that holds the degree and corresponding node pointer. If the next node has the same degree, we can remove the already stored node and combine the two.

public HeapEntry PairwiseCombine (HeapEntry pairNode1, HeapEntry pairNode2)

Return type: HeapEntry
Parameters: HeapEntry pairNode1, HeapEntry pairNode2

Description: This method receives two nodes as parameters namely pairNode1 and pairNode2 and returns the node with maximum count value as parentNode to the calling method. This method can be described in two parts. In the first part, we check which of the two nodes has maximum count value and based on that, we decide which one will become child and parent node.

In the second part, the node that becomes child will be removed from the root list. We need to increase degree of the node that becomes parent and also update its children's linked list with this new node.