COP 5536 - PROJECT REPORT

JAMES DIFFENDERFER
JDIFFEN1@UFL.EDU
UFID: 6191-4098

Introduction

Here we include all function prototypes for this implementation of a max Fibonacci heap. The two classes defined for this implementation of a max Fibonacci heap are a *Node* class and the *Fibonacci Heap* class. Both of these classes are defined in the header file `keywordcounter.h` located in the `Include` directory. Following our presentation of the function prototypes for these classes, we briefly hightlight some information about the main function which is defined in the file `keywordcounter.cpp` and is located in the `Source` directory.

Node Class

In this section, we provide the contents and functions used for the *Node* class which are used as nodes for the Fibonacci heap implementation. We first provide the contents of each node followed by the function prototypes in the *Node* class.

Class Contents:

Parameters	**Type**	**Description**
data	`long int`	Data in node (keyword frequency)
degree	`long int`	Degree of node (number of children)
keyword	`string`	Keyword at node
childcut	`bool`	Indicates whether node has lost child since becoming child of current parent
child	`Node *`	Child pointer
lsibling	`Node *`	Left sibling pointer
rsibling	`Node *`	Right sibling pointer
parent	`Node *`	Parent pointer

Function Prototypes:

`void merge(Node *nd)`

Description	Void function which makes tree rooted at given node a subtree of the current node.	
Parameters	nd	Pointer to root of tree which is to be made subtree of current node
Return value	No values are returned	

Date: November 15, 2018.

Fibonacci Heap Class

In this section, we provide the contents and functions used for the *Fibonacci Heap* class. We first provide the contents of each Fibonacci heap followed by the function prototypes in the *Fibonacci Heap* class.

Class Contents:

Parameters	**Type**	**Description**
n	`long int`	Number of nodes in fibonacci heap
max	`Node *`	Pointer to node containing the max key
hashmap	`unordered_map<string, Node*>`	Hash table for keywords mapped to node pointer

Function Prototypes:

`bool only_child(Node *nd)`

Description	Boolean function which checks if node has any siblings.	
Parameters	nd	Pointer to node to check if it has siblings.
Return value	Returns TRUE if given node is only child FALSE if given node has at least one sibling.	

`bool barren(Node *nd)`

Description	Boolean function which checks if node has any children.	
Parameters	nd	Pointer to node to check if it has children.
Return value	Returns TRUE if given node has no children and FALSE if given node has at least one child.	

`Node *orphan(Node *nd)`

Description	Function which checks if node has a parent.	
Parameters	nd	Pointer to node to check if it has a parent.
Return value	If the given node has no parent, it returns NULL. If given node has a parent, it returns the pointer `nd->parent->child`.	

`long int rank_bound()`

Description	Compute bound on rank based on current number of nodes.	
Parameters	–	No inputs.
Return value	Returns bound on rank for rank tree.	

`void meld(Node *nd)`

Description	Meld single node at root level.	
Parameters	nd	Pointer to node to be melded to root of Fibonacci heap.
Return value	No value is returned.	

`void PrintRootLtoR()`

Description	Print root level nodes of Fibonacci heap starting with max and moving to right.	
Parameters	–	No inputs.
Return value	No value is returned.	

`long int number_of_nodes()`

Description	Return number of nodes in Fibonacci heap.	
Parameters	–	No inputs.
Return value	Number of nodes in Fibonacci heap is returned.	

`Node *find_max()`

Description	Find and return pointer to node containing maximum value in heap.	
Parameters	–	No inputs.
Return value	Returns pointer to node containing maximum value in Fibonacci heap.	

`void insert(string keyword, long int frequency)`

Description	Insert node into heap. If node containing keyword is already in heap, the frequency in the Fibonacci heap is increased by the input frequency amount.	
Parameters	keyword	Keyword to store in heap.
	frequency	Frequency of keyword
Return value	No value is returned.	

`void remove_max()`

Description	Remove max node from heap.	
Parameters	–	No inputs.
Return value	No value is returned.	

`bool remove(Node *nd)`

Description	Remove node from heap given pointer to node. Updates childcut of input parent node (if necessary).	
Parameters	nd	Pointer to node to be removed from Fibonacci heap.
Return value	Returns FALSE if childcut of parent was not changed from TRUE and returns TRUE if childcut of parent node was TRUE.	

`void increase_key(string keyword, long int amount)`

Description	Given a node, increase its frequency value by a given amount.	
Parameters	keyword	Keyword for which frequency is increased.
	frequency	Amount to increase frequency of keyword.
Return value	No value is returned.	

Main Function and Other Information

The main function that is compiled and can be called as an executable is defined in the file `keywordcounter.cpp` and is located in the `Source` directory. The main function was fairly straightforward. For every line containing a pair (*keyword*, *frequency*), the pair was inserted into the max Fibonacci heap by calling the function `insert()`. Note that the `insert()` function defined above handles the case for when the *keyword* is already in the heap and internally triggers a call to `increase_key()` instead of inserting the pair as a new element of the heap. When a query is requested, say for n elements, the main function calls `remove_max()` n times in a row. After each `remove_max()` call, the removed pair (*keyword*, *frequency*) is pushed onto a FIFO queue and the *keyword* is printed to the file `output_file.txt`. This process is repeated until the n maximum elements have been removed from the max Fibonacci heap. At this time, the elements are popped off of the FIFO queue and reinserted into the max Fibonacci heap one at a time. When a stop command is received the main function frees all alocated memory and exits the program. Checks for memory leaks were performed using *Valgrind*. The file containing the solution, `output_file.txt`, can be found in the same directory as the executable.

There are several make options defined in the `Makefile`. The default executable can be compiled by typing `make` in the main directory. There are also compiler options set to compile for use with *Valgrind*, to print extra information while running the program (which was primarily used during the debugging phase), and to generate an ordered list of all keywords and frequencies at each query request using standard library functions (which was used to check if a solution for a problem was valid). These various modes can be compiled by typing `make use_valgrind`, `make debug`, and `make generate_solution`, respectively.