Generic Programming Project Report

Project Title: Generic Fibonacci Heap

Project ID: 37

Team Details:

Abdur Rahman Hatim - PES1201801503

Avinash Ratnam - PES1201800883

Harichandana Magapu - PES1201801041

Abstract:

A Fibonacci Heap is a collection of heap-ordered trees. It is a lazy version of a Binomial Heap. It is a data structure used to implement priority queue operations, as it has a much better amortised running time than all other heaps, including binary and binomial heaps. Their name comes from the Fibonacci numbers, which show up in the analysis due to the following constraints:

1. When two sub-heap trees of the same rank exist, they are merged. 2. When more than one node is removed from a sub-heap tree, the parent node is removed.

Due to these, each tree has a minimum number of nodes that corresponds to a Fibonacci number (a tree of rank r has at least F_{r+2} nodes), i.e. each tree is a Fibonacci tree.

Another key feature of Fibonacci Heaps is the reduction in the number of times we heapify the Fibonacci Heap. We only heapify and clean up the heap when we pop the minimum value from the Root List. Hence We then heapify, by comparing node values of the same degree and accordingly set the node as a parent node or a child node. Hence, a new Minimum node is found and is at the top of the root list.

Some operations supported by a Fibonacci Heap are:

- Find Min in **O(1)**
- Extract Min in **O(log n)**
- Insert in **O(1)**

- Decrease Key in **O(1)**
- Merge in **O(1)**

* All-time complexities mentioned are amortised (average performance, rather than worst-case)

Implementation Details:

We have implemented generic Fibonacci heaps in C++ using template classes and functions. This file will be included as a header file that the clients can use in their programs.

Class Definition and Structure:

Each node of the Fibonacci heap has been represented as a generic class, including the following attributes:

- Pointer to the previous (left) sibling of node
- Pointer to next (right) sibling of the node
- Pointer to the parent node
- Pointer to the child node
- A value of any type
- An integer to indicate the degree

The Node class has been made canonical by providing a constructor, a copy constructor, a copy assignment operator and a destructor.

Fibonacci heap has been represented as a generic class with a pointer to the minimum value of the heap and multiple member functions depicting the operations

The Fibonacci heap has been made canonical by providing the following:

- **Constructor:** This initialises the data members when the object is created
- **Copy Constructor:** This calls the copy_fheap function of the Node class where copying of the nodes is done.
- **Copy Assignment Operator:** First, the memory assigned to the Fibonacci heap on the left-hand side of the assignment operator is deallocated by calling the destructor if it is not self-assignment. Then the copy_fheap function of the Node class is called, where copying of the nodes is done

● **Destructor:** This calls the deleteAll function, which releases the allocated resources

Operations:

The following are some operations we have implemented which can be performed on a Fibonacci Heap:

- Creating Node:

The Node class is used to create a new node that will be inserted into the heap. Node value is taken as a parameter for the constructor, and Prev, Next, Parent and Child pointers are initialised to NULL.

- Merging Heaps:

When merging a new heap into the original heap, we have a pointer pointing to the minimum root node of both heaps. We then compare the values of the nodes, and the node with the smaller value becomes the minimum root for the merged heap.

- Deleting all Nodes:

To delete all the nodes in a Fibb Heap, we use a destructor. We first set a reference pointer and iterator pointer to the root node. Then, we iterate the latter through the heap in a Do While loop, deleting the following node and iterating to the next node until the current node is equal to the reference node again.

- Adding a Child Node:

First, we create the new child node, then pass the child node and a pointer to the parent node to the function. Thus, we first assign a parent node and increment the degree of the parent node. We then pass the child node, along with a pointer, to the parent node to the Merge() function where it is incorporated into the heap.

- Unmark and Unparent All:

This is a helper function for the PopMin function, which cuts off the root node of a tree and also unmarks and adds all the children to the root list.

- PopMin

This function returns the minimum root of the heap and consolidates the heap (such that no two trees have the same degree) using a buffer. This is the only time when Heapification of the entire Heap occurs. This is because heapification is expensive; hence when a node is cut it is added to the root list. When we remove the Minimum node, we heapify by comparing the values of nodes with the same degree; the smaller value becomes the parent node while the other, a child node.

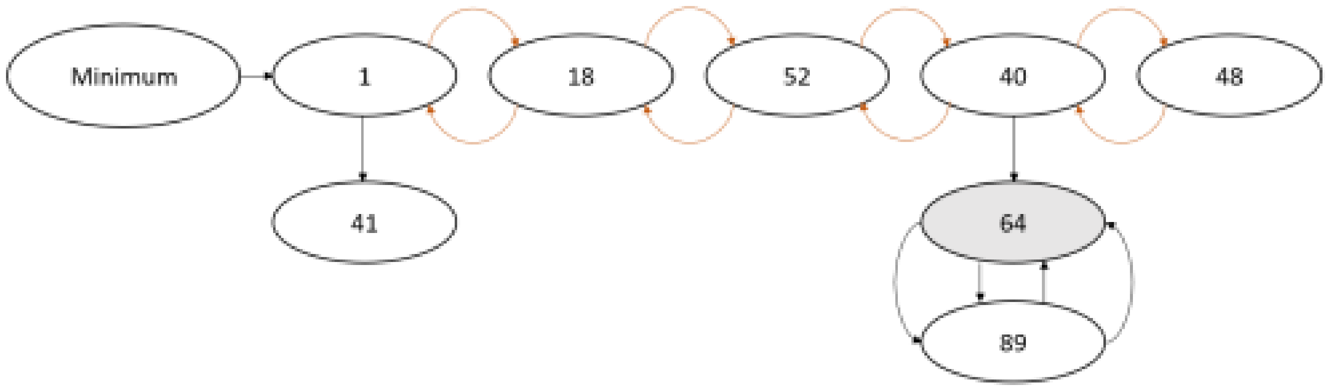
- Cut

This function cuts nodes from the heap according to our maximally damaged tree constraint (parent can lose at most one child, and if they lose more, they must be cut from the tree as well). A parent node that has already lost a child is marked with a flag. Cutting a child of a marked node triggers cascading cuts.

- Decrease Key

This operation decreases the value contained in a node. If this change in value violates the heap invariant property, a cut operation is done on the node. If there is a parent node, it is marked once the cut operation is done. If the cut node is smaller than the current heap minimum, the minimum is reassigned.

Sample Output Heap:



How to run the Software:

The implementation file consisting of all the implementation code and functions with respect to Fibonacci heaps has been provided as a header file that is included in the client code as follows:

```
#include “fibb_heap.h”
```

Once this is included, clients can create Fibonacci heaps and call functions to perform different operations as explained above or in the **readme** file associated with this document using the provided interface.

The client file is then compiled using the g++ utility.

```
g++ client.cpp -o exec
```

The generated executable is then loaded and executed.

```
./exec
```