Advanced Data Structures

Programming Assignment 1

Name: Heng Yao

UFID: 0679-4920

UF Email: hengyao1993@ufl.edu

Nov. 17, 2016

1. Program Structure

I have two code files, FibonacciHeap.h and FibonacciHeap.cpp. The .cpp file mainly deals with input and output processing. First, I extract hashtag string and hashtag count from input line and construct new Fibonacci node based on the input. If this is a new node in hash map then insert it to Fibonacci heap as a new node, if it's already exists in hash map then increase the key of corresponding node in heap by count. Last I insert hashtag string and the pointer to the node in heap to hash map. The flow chart of FibonacciHeap.cpp is as shown in figure 1.

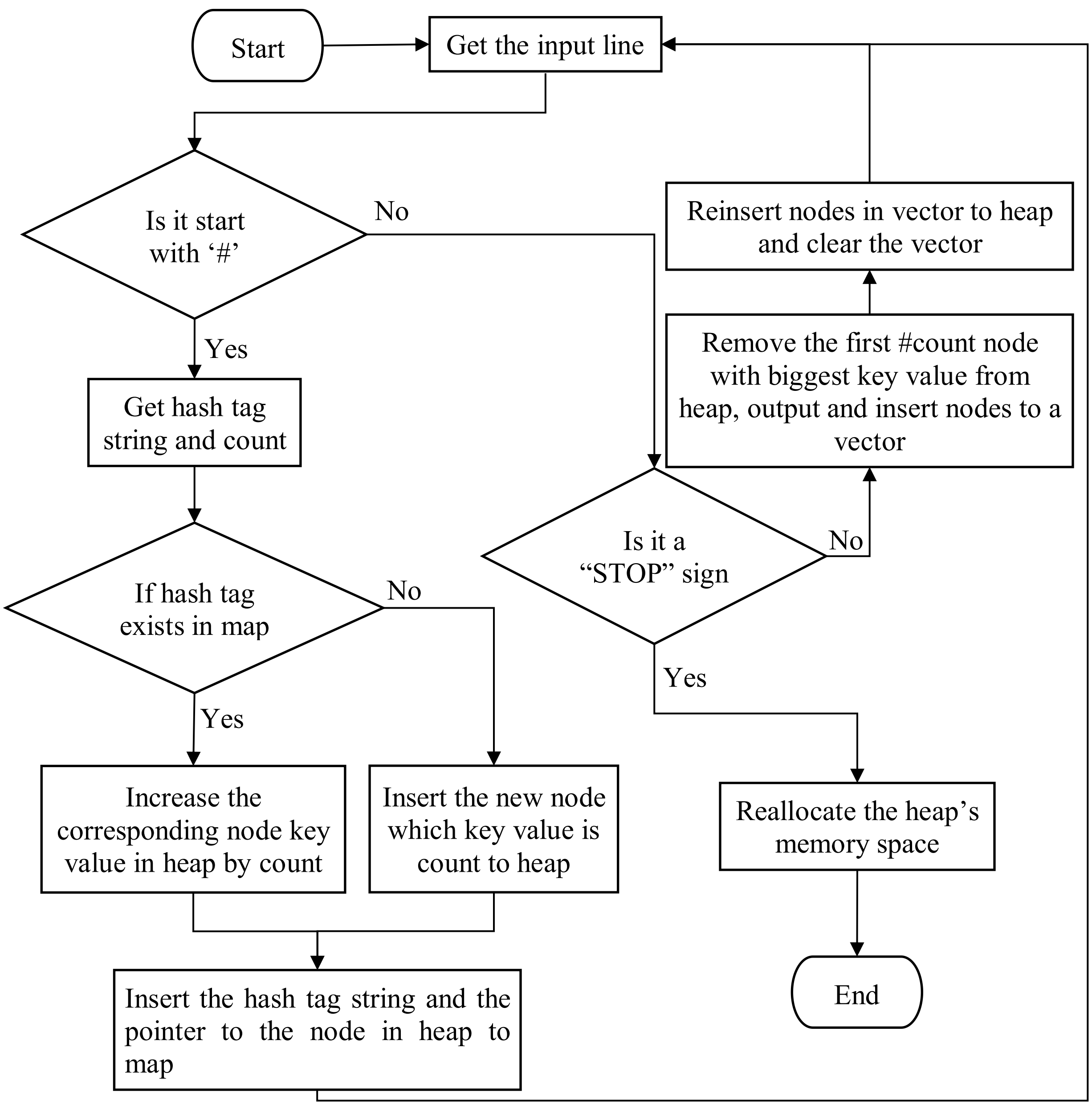


Figure 1. Flow chart of FibonacciHeap.cpp

The .h file implement the Fibonacci heap. There are two classes in this header file, FibonNode and FibonHeap class. FibonNode class defines node structure in fibonacci heap, FibonHeap class defines fibonacci heap member variables and member functions. The class diagram of FibonNode class is as shown in figure 2.

```
FibonNode
---
+key : T
+hashString : string
+degree : int = 0
+child : FibonNode< T >* = NULL
+parent : FibonNode< T >* = NULL
+leftSibling : FibonNode< T >* = this
+rightSibling : FibonNode< T >* = this
+childCut: bool = false
---
+FibonNode( key : T, hashString : string, degree : int,
child : FibonNode< T >*, leftSibling : FibonNode< T >*,
rightSibling : FibonNode< T >*, parent : FibonNode< T >*,
childCut : bool )
```

Figure 2. FibonNode class diagram

The class diagram of FibonHeap class is as shown in figure 3.

```
FibonHeap
---
-maxNode : FibonNode< T >* = NULL
-nodeNum : int = 0
-treeTable : FibonNode< T >** = NULL
+nodesVec : vector< FibonNode< T >* >
---
+insertNode( node : FibonNode< T >* ) : void
+addRootList( node : FibonNode< T >*, root : FibonNode< T >* : void
+removeFromLinkedList( node : FibonNode< T >* ) : void
+increaseKey( node : FibonNode< T >*, value : T) : void
+cascadingCut( node : FibonNode< T >*) : void
+cut( parent : FibonNode< T >*, node : FibonNode< T >* ) : void
+extractMaxtree() : FibonNode< T >*
+removeMax() : FibonNode< T >*
+link( root : FibonNode< T >*, node : FibonNode< T >*) : FibonNode< T >*
+pairwiseCombine() : void
+reinsertNodes( vec : vector< FibonNode< T >* > ) : void
+deleteHeap() : void
+releaseNodes( max : FibonNode< T >*) : void
```

Figure 3. FibonHeap class diagram

The tree structure of a max Fibonacci heap is as shown in figure 4.

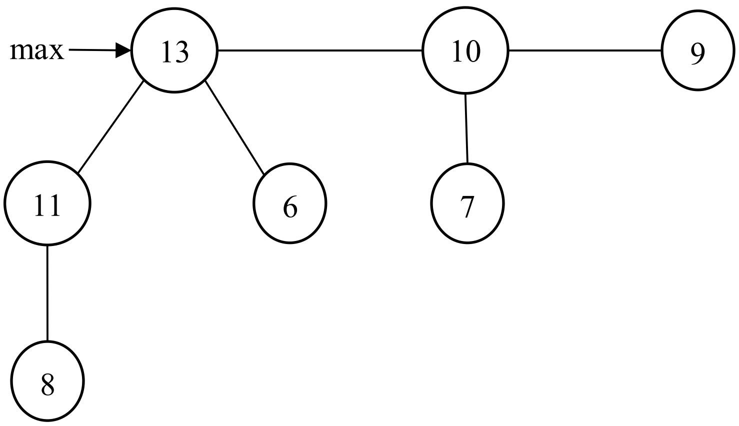


Figure 4. Tree structure of a Fibonacci heap

The corresponding memory structure is as shown in figure 5:

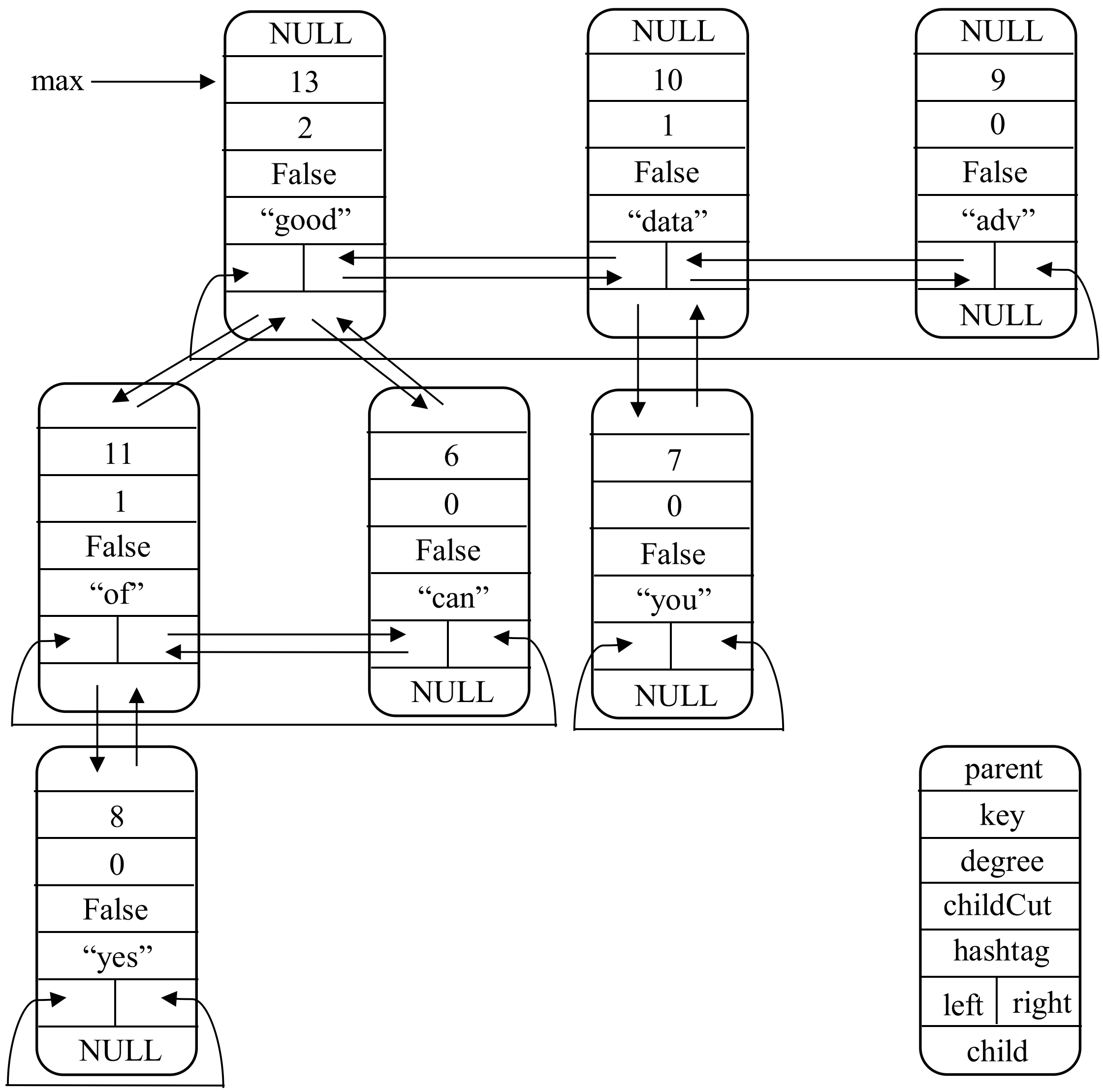


Figure 5. Memory structure of Fibonacci heap

2. Function Prototype

2.1 FibonNode Class

FibonNode class only has a constructor method, I initial the member variables here.

2.2 FibonHeap Class

(1) Insert new node to max Fibonacci heap

Insert operation diagram is as shown in figure 6.

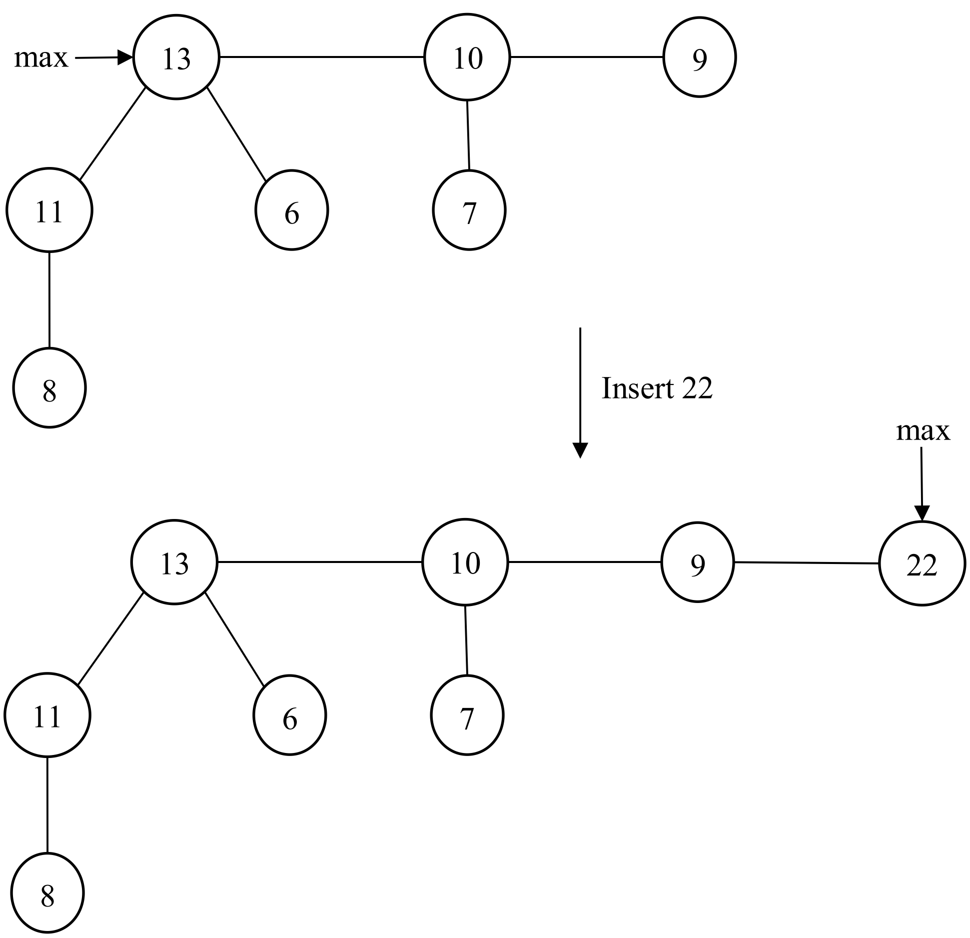


Figure 6. Insert function diagram

```
// Insert a new node to heap
template <class T> void insertNode( FibonNode<T> *node )
{
    if   heap is empty
                 heap's max node = node
        else
```

Insert the node to heap's top level doubly linked list
If node's key > heap's max node's key
heap's max node = node
Add the total node number in heap by 1
}

(2) Insert a node to another node's sibling list

I always insert the node between the another node and it's left sibling, if the another node is root, this process is as shown figure 7.

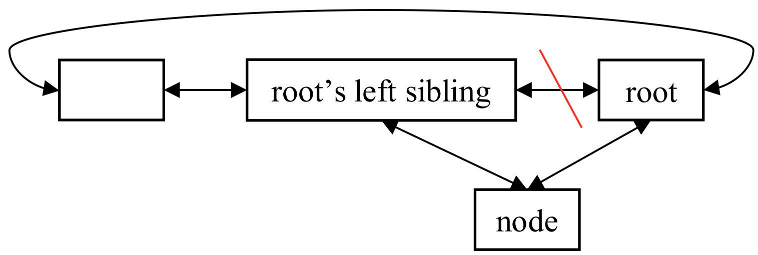


Figure 7. insert node to root's sibling list

```
// Add node to root's sibling list
template <class T> void addRootList( FibonNode<T> *node, FibonNode<T> *root )
{
        if node != NULL and root != NULL
                insert the node between the root node and it's left sibling
        else
                show alert message and return
}
```

(3) Remove the node from it's doubly linked sibling list

I modify right sibling pointer of node's left sibling to the address of node's right sibling, and I modify left sibling pointer of node's right sibling to the address of node's left sibling. This process is as shown in figure 8.

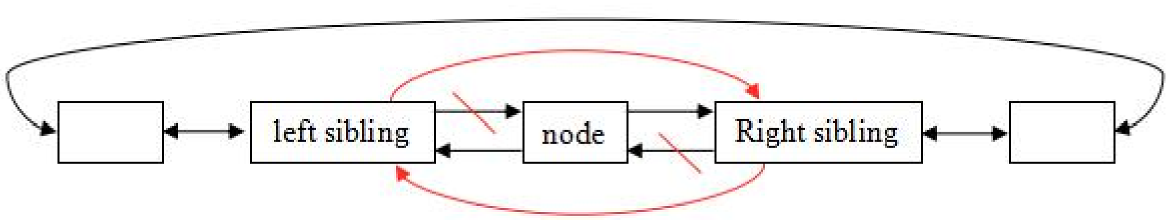


Figure 8. remove node from it's sibling list

```
// Remove the node from it's doubly linked sibling list
template <class T> void removeFromLinkedList( FibonNode<T> *node )
{
        node->leftSibling->rightSibling = node->rightSibling;
        node->rightSibling->leftSibling = node->leftSibling;
```

}

(4) Increase the existing node's key value by word count

If the hash tag is already exist in Fibonacci heap, I will increase the existing key value. If the node key is bigger than it's parent, cut and cascading cut will be executed. Increase key operation diagram is as shown in figure 9.

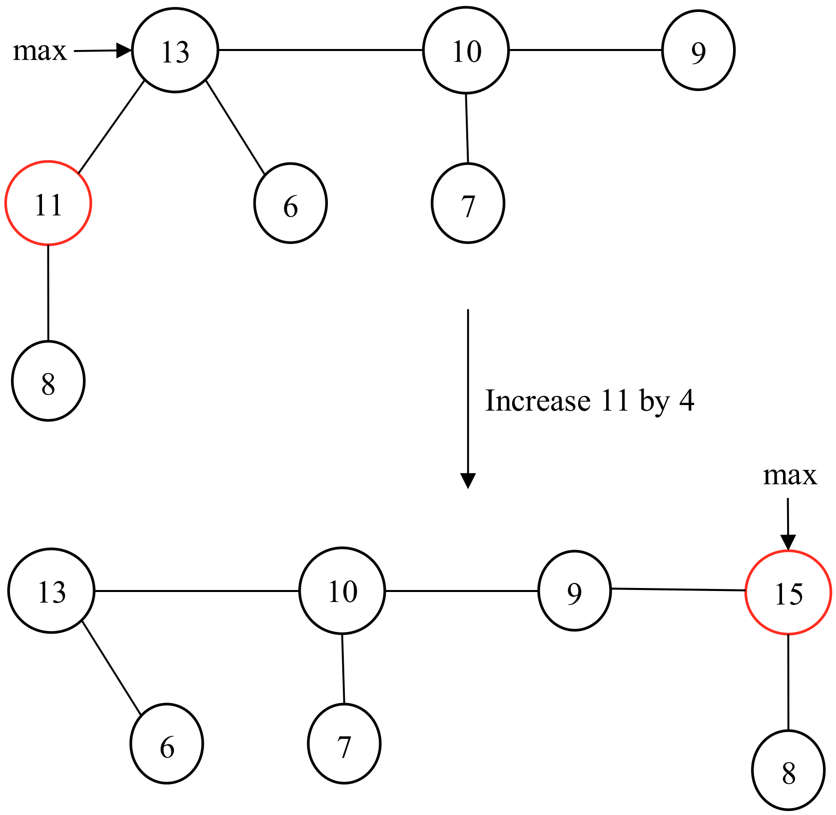


Figure 9. Increase key operation diagram

```
// Increase the node's key by value
template <class T> void increaseKey( FibonNode<T> *node, T value )
{
        increase node's key by value
        if the node has a parent and node's key is bigger than it's parent's key
                cut the node from it's parent
                perform cascading cut from node's parent
        if the node's key is bigger than heap's max node's key
                max node = node
}
```

(5) Cascading cut

```
// Cascading cut follow path from parent of the node to the root
template <class T> void cascadingCut( FibonNode<T> *node )
```

```
{
        if the childCut flag of this node is true
                remove the node from it's sibling list
                remove it from it's parent's child list
                do cascadingCut to the node’s parent
        else
                assign the node’s childCut flag to true
}
```

(6) Cut node from parent’s child list

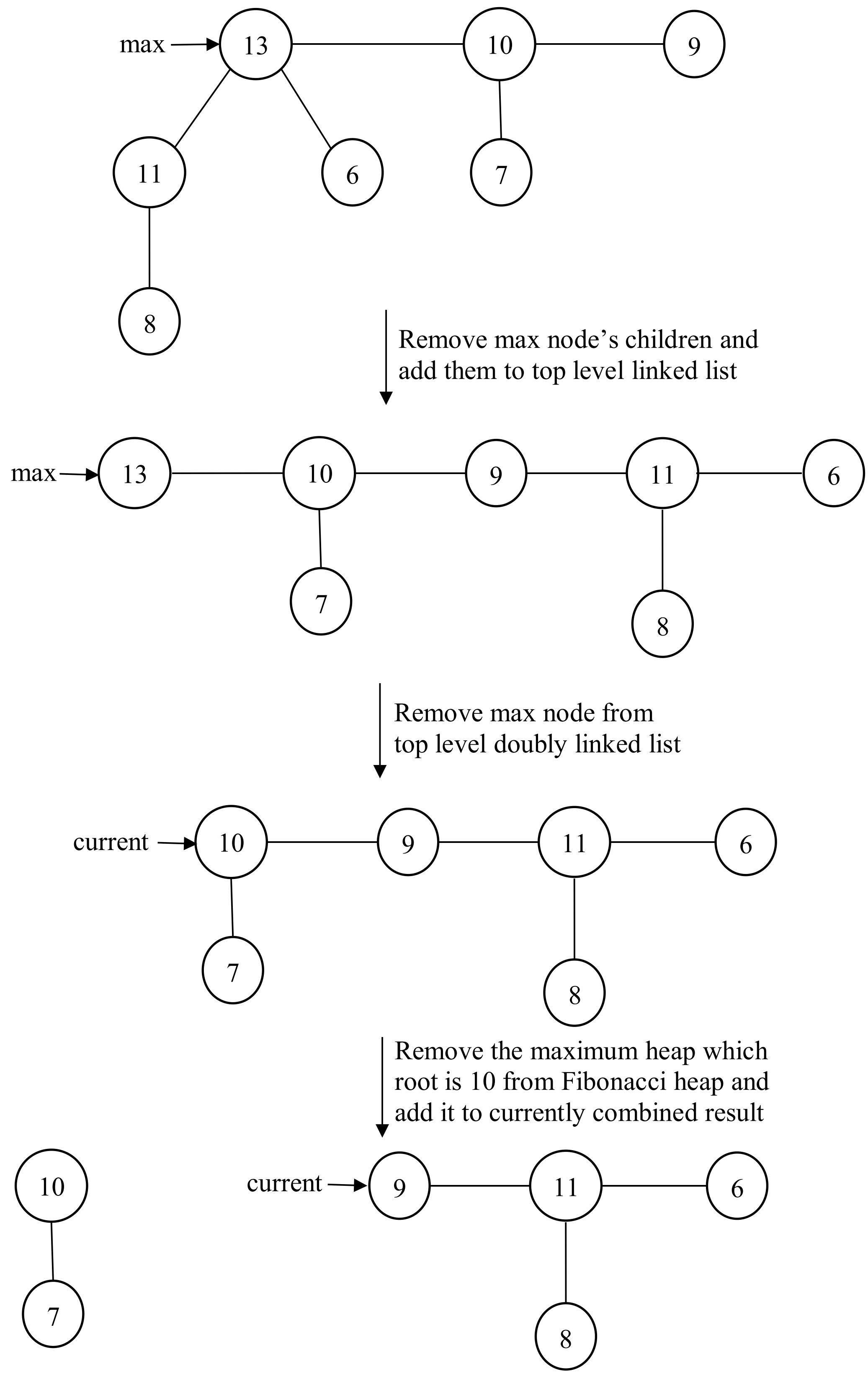
```
// Cut the node from it's sibling list and parent's child list
template <class T> void cut(FibonNode<T> *parent, FibonNode<T> *node)
{
        remove the node from it's sibling list
        update the node's parent's child pointer and degree
        update node's status and add it to heap’s top level doubly linked list
}
```

(7) Get the max node and it’s children

```
// Return the whole tree which root is the max node
template <class T> FibonNode<T>* extractMaxtree()
{
        if heap is empty
                return NULL
        remove the max node from it’s doubly linked sibling list
        update the max node of heap
        update deleted max node’s status and return it.
}
```

(8) Remove the max node of Fibonacci heap

The remove max node operation diagram is as shown in figure 10.


max
13
10
9
11
6
7
8
Remove max node's children and
add them to top level linked list
max
13
10
9
11
6
7
8
Remove max node from
top level doubly linked list
current
10
9
11
6
7
8
Remove the maximum heap which
root is 10 from Fibonacci heap and
add it to currently combined result
10
7
current
9
11
6
8

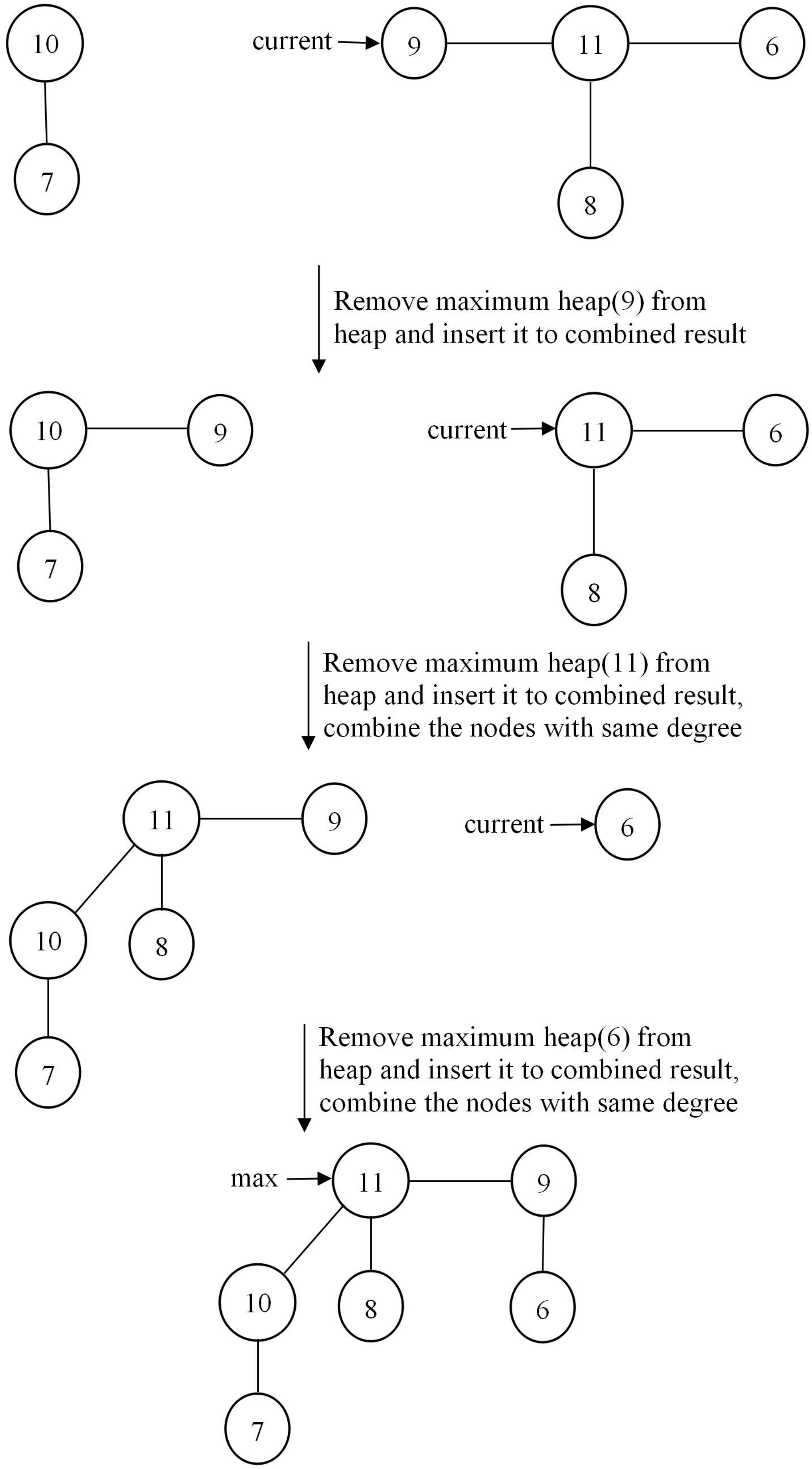


Figure 10. Remove max node operation diagram

```
// Remove the max node of fibonacci heap
template <class T> FibonNode<T>* removeMax()
{
	if heap is empty
		return NULL
	remove every child of max node from it's sibling list and add it to root list
	remove max node from it's sibling list and update heap's max node
	pairwise combine the remaining nodes in root list
	push the deleted max node to the node vector
}
```

(9) Link node to root’s child list

```
// Link the node to the root's child list
template <class T> FibonNode<T>* link(FibonNode<T> *root, FibonNode<T> *node)
{
	remove node from it's former sibling list
	add node to root's child list
	update the status of root
}
```

(10) Pairwise combine the nodes in top level doubly linked list with same degree

```
// Pairwise combine the remaining nodes in root list
template <class T> void pairwiseCombine()
{
	alloc the memory space to the table to keep track of trees by degree and initial it
	extract the whole tree which root is max node and then combine this tree with current combined result
	combine trees with the same degree till there are no equal degree trees in current combined result
	insert the pairwise combine result to a empty Fabionacci heap
}
```

(11) Reinsert the deleted nodes to Fibonacci heap

```
// Insert nodes in node vector to fibonacci heap
template <class T> void reinsertNodes(std::vector<FibonNode<T>*> vec)
{
	insert all nodes in vector to Fabionacci heap
}
```

(12) Deallocate the memory block point to Fibonacci heap

```
// Deallocate the memory block pointed to fibonacci heap
template <class T> void deleteHeap()
{
	deallocate every node's memory block recursively
	deallocate treeTable’s memory space
}
```

3. Result Analysis

I have tested my program by using the sample input. My result is completely consistent with the sample output. Considering there may be millions input lines so I use the unordered map in C++, when I using unordered map and testing my program by simple_input1.txt the total running time is 0.002246 seconds. While when I using map in C++ and testing with the same input, the total running time is 0.002872. Then I copying the sample_input1.txt for ten times and test my program again, when I using unordered_map the total running time is 0.017160s, when I using map, the total running time is 0.025042s. The result shows that unordered_map is more suitable when there is a large amount of input hash tags. The running result is as shown in figure 11.

	sample_input1.txt	Copying sample_input1.txt for 10 times

map	`Test case: sample_input1.txt`
	`Using map`
	`Running Time: 0.002872 seconds`
	`Program ended with exit code: 0`
	All Output ◇ Filter
		`Test case: Copying sample_input1.txt for 10 times`
		`Using map`
		`Running Time: 0.025042 seconds`
		`Program ended with exit code: 0`
		All Output ◇ Filter

unordered_map	`Test case: sample_input1.txt`
	`Using unordered map`
	`Running Time: 0.002246 seconds`
	`Program ended with exit code: 0`
	All Output ◇ Filter
		`Test case: Copying sample_input1.txt for 10 times`
		`Using unordered map`
		`Running Time: 0.017160 seconds`
		`Program ended with exit code: 0`
		All Output ◇ Filter

Figure 11. The result comparison when using map and unordered_map

4. Further Improvement

When doing the pairwise combination, we need to define a tree table to keep track of trees by degree. So we need to know the table's size in advance, the most obvious answer is the size must smaller than total node number of heap minus 1. It can be presented by equation (1).

$$table\_size < nodeNum - 1 \qquad (1)$$

However, when there is too many input hash tags, the total node number will be very large, so it will take too much memory space. So I tried to make the table size smaller. Assume that there are N nodes in Fibonacci heap, since the table is used to keep track of trees by degree, so I try to get as many trees with different root's degree as possible within N nodes. Then the number of trees with different root's degree plus 1 is maximum table size.

To maximum the number of trees with different root's degree constructed by N nodes, the node number of each tree should be like the following array: 1, 2, 3, 4, 5, ..., n. Now we have:

$$1+2+3+...+n=N \tag{2}$$

Based on equation (2), we can conclude that:

$$n=\sqrt{2N+\frac{1}{4}}-\frac{1}{2} \tag{3}$$

Therefore, if the total node number of Fibonacci heap is N, the maximum tree table size is:

$$\max\_table\_size=ceil(n)+1 \tag{4}$$

We will save much memory space if there is a large amount of input hash tags by using this equation.