Programming Project Report:
Keyword Counter

Name: Jaemoon Lee

UF E-mail: j.lee1@ufl.edu

1. Max Fibonacci Heap Implementations

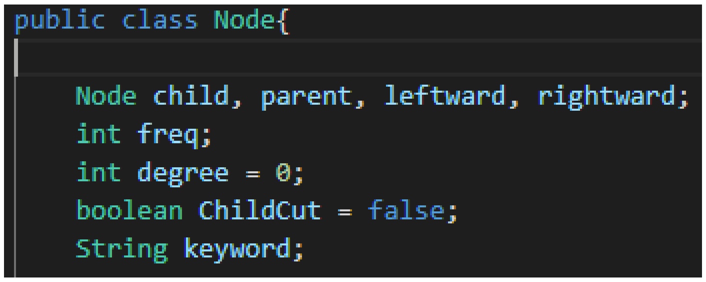
- Max Fibonacci Heap supports the following operations:

  · GetMax, Insert, RemoveMax, Meld

  · IncreaseKey, Arbitrary remove

(1) Node structure

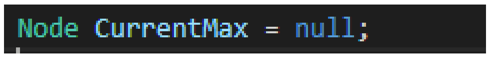
- Each node has degree, child, data, and childcut field. Here, keyword will be data.
- Each node has a pointer to parent node. And each parent node has a pointer to one of its children.
- Siblings are connected by a circular doubly linked list.



```
public class Node{

    Node child, parent, leftward, rightward;
    int freq;
    int degree = 0;
    boolean ChildCut = false;
    String keyword;
```

(2) Get Max Value

- In Max Fibonacci Heap, we should keep track of currently overall max value.

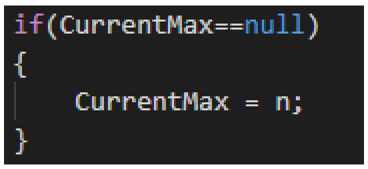


```
Node CurrentMax = null;
```

(3) Insert

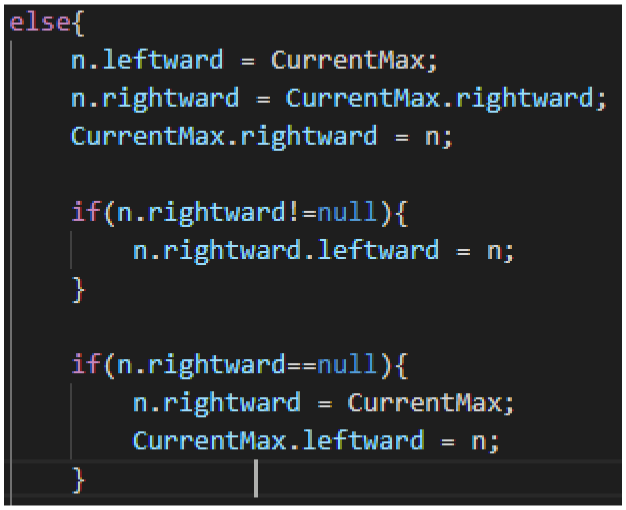
- Create a new max tree with a newly inserted element.

  → If there is no element in Fibonacci Heap, CurrntMax node will point to the newly created node.



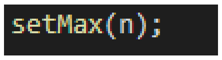
```
if(CurrentMax==null)
{
    CurrentMax = n;
}
```

→ If Max Fibonacci Heap is not empty, the newly inserted node will be added to doubly linked circular list in the root level. In this implementation, newly inserted node will be placed next to the currently max node.



```
else{
    n.leftward = CurrentMax;
    n.rightward = CurrentMax.rightward;
    CurrentMax.rightward = n;

    if(n.rightward!=null){
        n.rightward.leftward = n;
    }

    if(n.rightward==null){
        n.rightward = CurrentMax;
        CurrentMax.leftward = n;
    }
```

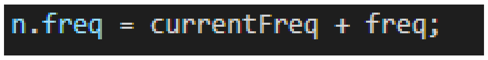
- After insertion, we would check if the newly inserted node has a higher frequency than the existing currently max value, then we will update the CurrentMax.



```
setMax(n);
```

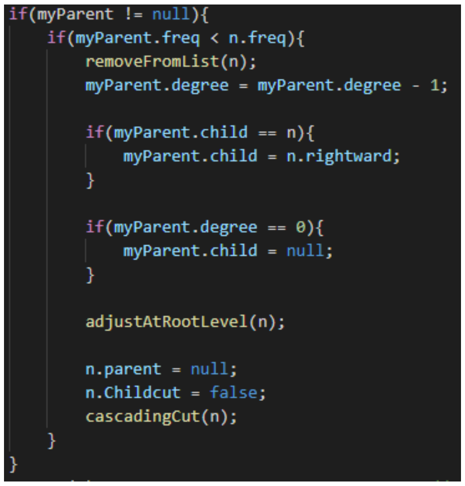
(4) IncreaseKey

- Take a specific node, and increase its frequency.



```
n.freq = currentFreq + freq;
```

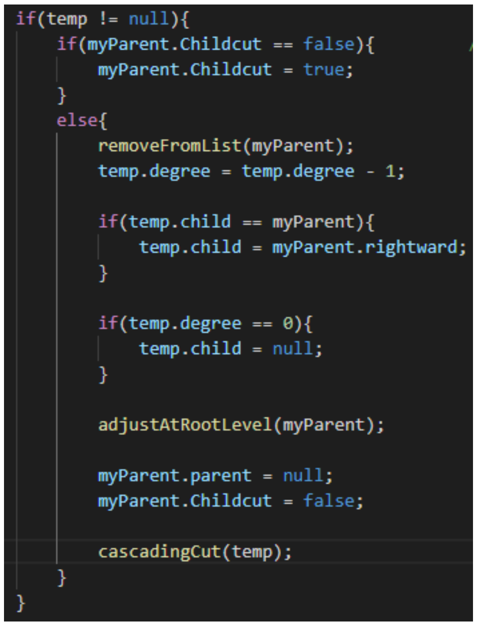
→ After increasing key operation, the newly increased node might greater than the key of its parent. Then, the subtree rooted the newly increased node should be removed from its parent and be added to the root level doubly linked circular list. Also, the parent pointer of the newly increased node should be set to null and ChildCut field will be set to false since ChildCut field is not defined in the root level. As for the previous parent, the degree field should be reduced by 1 and the child pointer might need to be updated. Then, the cascading cut operation will be performed from the previous parent.



```
if(myParent != null){
    if(myParent.freq < n.freq){
        removeFromList(n);
        myParent.degree = myParent.degree - 1;

        if(myParent.child == n){
            myParent.child = n.rightward;
        }

        if(myParent.degree == 0){
            myParent.child = null;
        }

        adjustAtRootLevel(n);

        n.parent = null;
        n.Childcut = false;
        cascadingCut(n);
    }
}
```

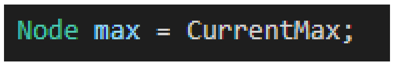
(5) CascadingCut

- When a node is removed from its parent, the cascading cut operation should be performed by following path from its parent to the root. This operation will be ended after reaching a node whose ChildCut field is ‘false’ and set the Childcut field to ‘true’. If a node’s ChildCut field is ‘true’, it will be removed from its list and be added to the root level. This operation will be perform recursively.

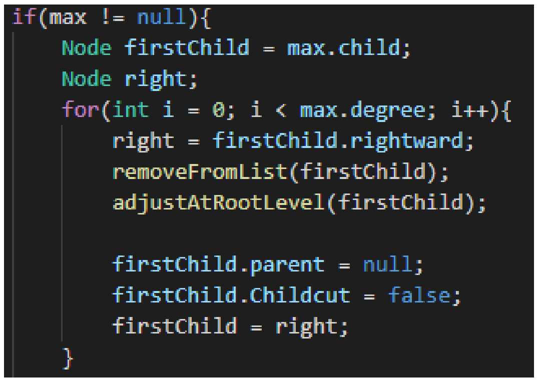

```
if(temp != null){
    if(myParent.Childcut == false){
        myParent.Childcut = true;
    }
    else{
        removeFromList(myParent);
        temp.degree = temp.degree - 1;

        if(temp.child == myParent){
            temp.child = myParent.rightward;
        }

        if(temp.degree == 0){
            temp.child = null;
        }

        adjustAtRootLevel(myParent);

        myParent.parent = null;
        myParent.Childcut = false;

        cascadingCut(temp);
    }
}
```

(6) RemoveMax

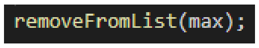
- Since we are keeping track of current max value in CurrentMax node, the result of RemoveMax operation will be simply returned by copying the CurrentMax node.


```
Node max = CurrentMax;
```

- After returning the result, children of the previous max node should become of roots of new trees. This means that they should be added to the root level doubly circular linked list. Set the parent pointer and ChildCut field of the children to null and ‘false’ respectively.

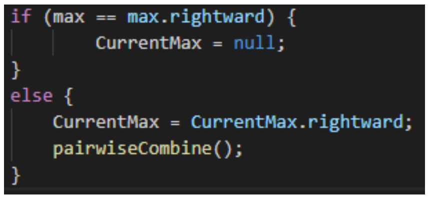

```
if(max != null){
    Node firstChild = max.child;
    Node right;
    for(int i = 0; i < max.degree; i++){
        right = firstChild.rightward;
        removeFromList(firstChild);
        adjustAtRootLevel(firstChild);

        firstChild.parent = null;
        firstChild.Childcut = false;
        firstChild = right;
    }
```

- After adding children of max node to the root level list, we will remove currently max node.



```
removeFromList(max);
```

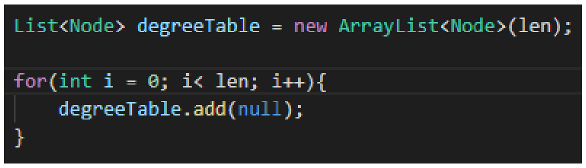
- Set right node of removed max node as new CurrnetMax temporarily and call pairwise combine operation



```
if (max == max.rightward) {
        CurrentMax = null;
}
else {
    CurrentMax = CurrentMax.rightward;
    pairwiseCombine();
}
```

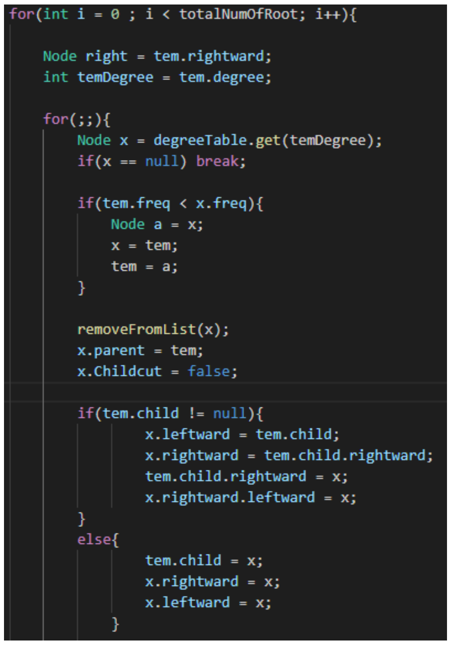
(7) PairwiseCombine

- Firstly, initiate degree table for pairwise combine operation



```
List<Node> degreeTable = new ArrayList<Node>(len);

for(int i = 0; i< len; i++){
    degreeTable.add(null);
}
```

- Do pairwise combine operation using a degree of a node. If there is no same entry in the degree table with a degree of a node, proceed to the next node. Or if there is same entry in the degree table with a degree of a node, then combine these trees by inserting a node with smaller frequency to a node with larger frequency. Then, set ChildCut field of a node with smaller frequency to 'false' since it became a child by pairwise combine operation. Also, child pointer of a node with larger frequency will be adjusted.



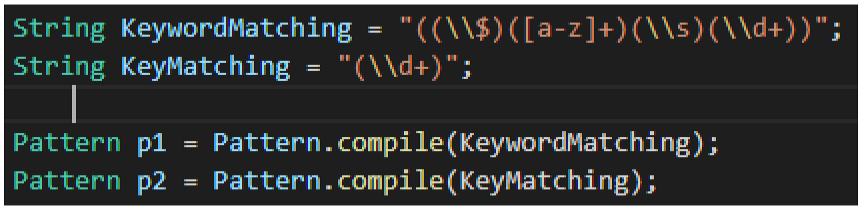
```
for(int i = 0 ; i < totalNumOfRoot; i++){

    Node right = tem.rightward;
    int temDegree = tem.degree;

    for(;;){
        Node x = degreeTable.get(temDegree);
        if(x == null) break;

        if(tem.freq < x.freq){
            Node a = x;
            x = tem;
            tem = a;
        }

        removeFromList(x);
        x.parent = tem;
        x.Childcut = false;

        if(tem.child != null){
                x.leftward = tem.child;
                x.rightward = tem.child.rightward;
                tem.child.rightward = x;
                x.rightward.leftward = x;
        }
        else{
                tem.child = x;
                x.rightward = x;
                x.leftward = x;
            }
```

2. Analysis of Input File

- In input file, keywords appear one per each line and start with $ sign. Frequencies are followed by keyword with a space between them. Queries are just integer number without $ sign. An example of input is like below:

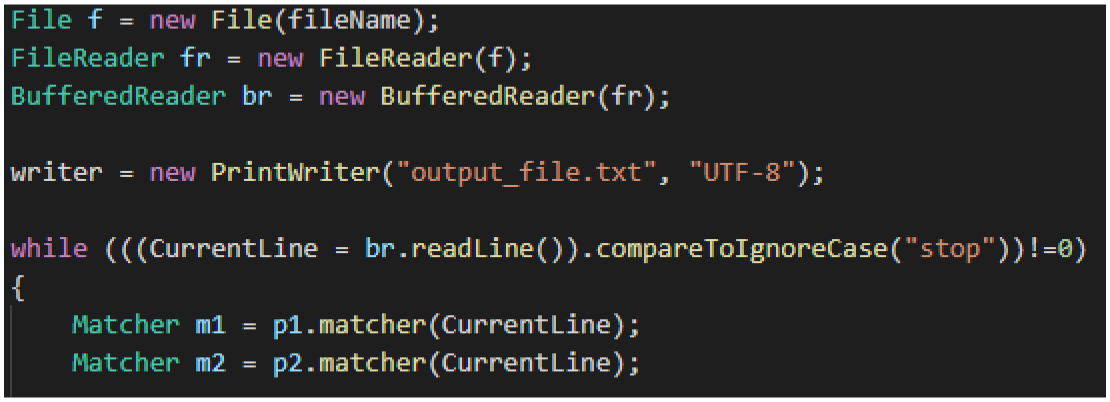
```
$ebay 12
$netflix 6
$cnn 5
5
stop
```

- In order to distinguish keywords with frequency and queries which are integer number, pattern string is used.



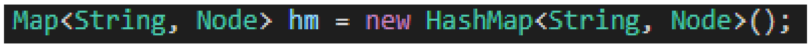
```java
String KeywordMatching = "((\\$)([a-z]+)(\\s)(\\d+))";
String KeyMatching = "(\\d+)";

Pattern p1 = Pattern.compile(KeywordMatching);
Pattern p2 = Pattern.compile(KeyMatching);
```

- After reading input file, matcher will distinguish keywords and queries unless “stop” appears.



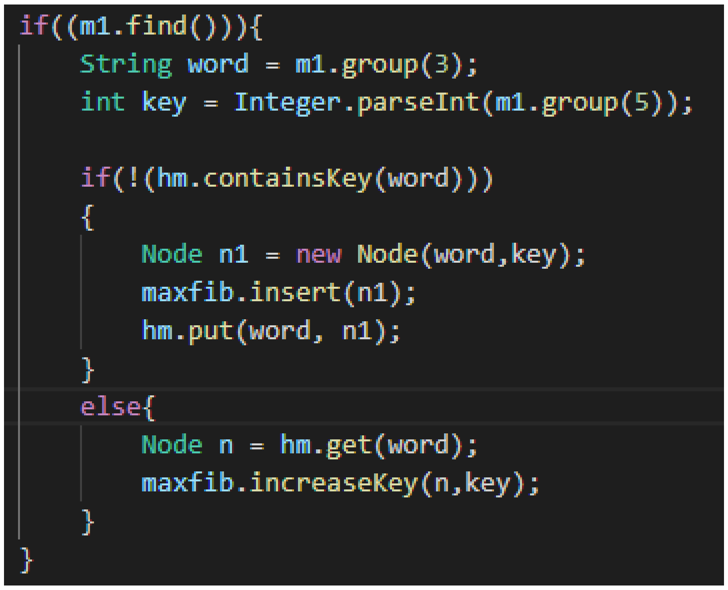
```java
File f = new File(fileName);
FileReader fr = new FileReader(f);
BufferedReader br = new BufferedReader(fr);

writer = new PrintWriter("output_file.txt", "UTF-8");

while (((CurrentLine = br.readLine()).compareToIgnoreCase("stop"))!=0)
{
    Matcher m1 = p1.matcher(CurrentLine);
    Matcher m2 = p2.matcher(CurrentLine);
```

- Hash table will be used in order to check that a keyword is already inserted to the Max Fibonacci Heap.



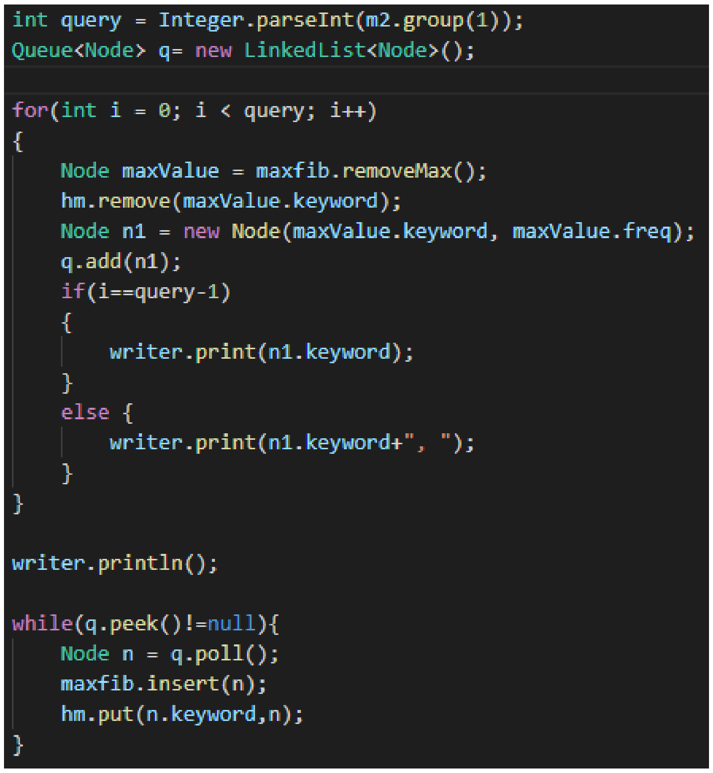
```java
Map<String, Node> hm = new HashMap<String, Node>();
```

- Suppose matcher finds a keyword. If there is same keyword in the hash table, we will use increaseKey operation. Otherwise, the keyword is inserted to the Max Fibonacci Heap.



```java
if((m1.find())){
    String word = m1.group(3);
    int key = Integer.parseInt(m1.group(5));

    if(!(hm.containsKey(word)))
    {
        Node n1 = new Node(word,key);
        maxfib.insert(n1);
        hm.put(word, n1);
    }
    else{
        Node n = hm.get(word);
        maxfib.increaseKey(n,key);
    }
}
```

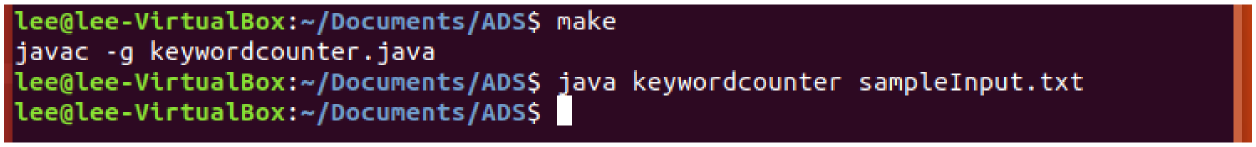
- Suppose matcher finds a query which is n. Then, n most frequently inserted keywords should be written in the output file. Firstly, removeMax operation is used for extracting n keywords. After that, a queue will store the keywords and the keywords will be written in the output file at the same time. After finishing writing n keywords, the keywords in the queue will be inserted back to the Max Fibonacci Heap.



```java
int query = Integer.parseInt(m2.group(1));
Queue<Node> q= new LinkedList<Node>();

for(int i = 0; i < query; i++)
{
    Node maxValue = maxfib.removeMax();
    hm.remove(maxValue.keyword);
    Node n1 = new Node(maxValue.keyword, maxValue.freq);
    q.add(n1);
    if(i==query-1)
    {
        writer.print(n1.keyword);
    }
    else {
        writer.print(n1.keyword+", ");
    }
}

writer.println();

while(q.peek()!=null){
    Node n = q.poll();
    maxfib.insert(n);
    hm.put(n.keyword,n);
}
```

3. Compiling Instructions

- Steps to execute is shown as below:



```
lee@lee-VirtualBox:~/Documents/ADS$ make
javac -g keywordcounter.java
lee@lee-VirtualBox:~/Documents/ADS$ java keywordcounter sampleInput.txt
lee@lee-VirtualBox:~/Documents/ADS$ 
```