Jinal Mashruwala jmashruw@ufl.edu UFID: 6498-3393

Fibonacci Heap

Index

1. Project Description

We are required to implement a system to find the n most popular hashtags appeared on social media such as Facebook or Twitter.
Basic idea for the implementation is to use a max priority structure to find out the most popular hashtags. We are using following structures for the implementation:

- Max Fibonacci heap: use to keep track of the frequencies of hashtags
- Hash table: Key for the hash table is hashtag and value is pointer to the corresponding node in the Fibonacci heap

2. Working Environment

Hardware requirement

- HARD DISK SPACE: 4 GB minimum
- MEMORY: 512 MB
- CPU: Intel Core i5

Operating system

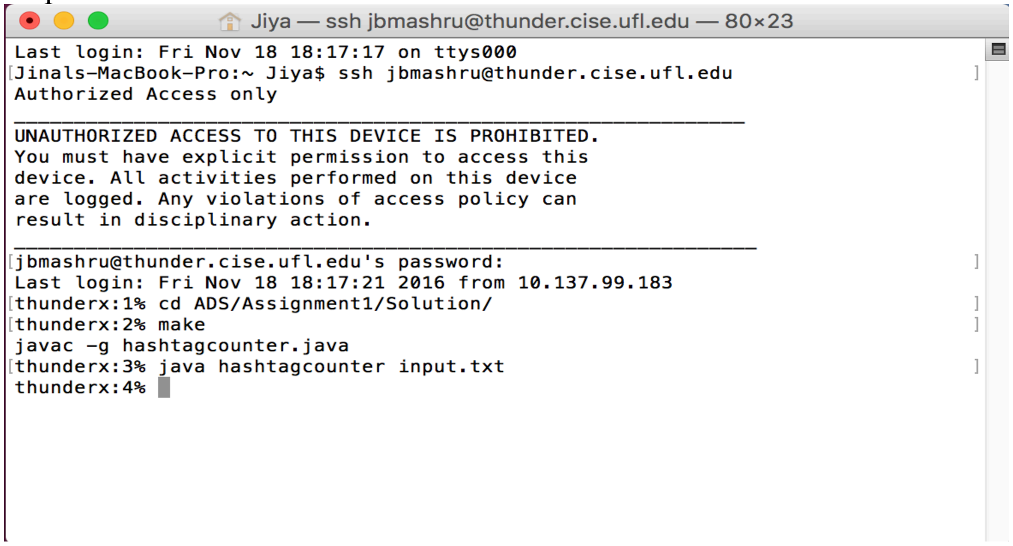
mac OS X 10

Compiler

Javac

3. Compiling Instructions

- The project has been compiled and tested on thunder.cise.ufl.edu and javac compiler
- To execute the program, you can remotely access the server using:
  ssh username@thunder.cise.ufl.edu
- Steps to execute are:

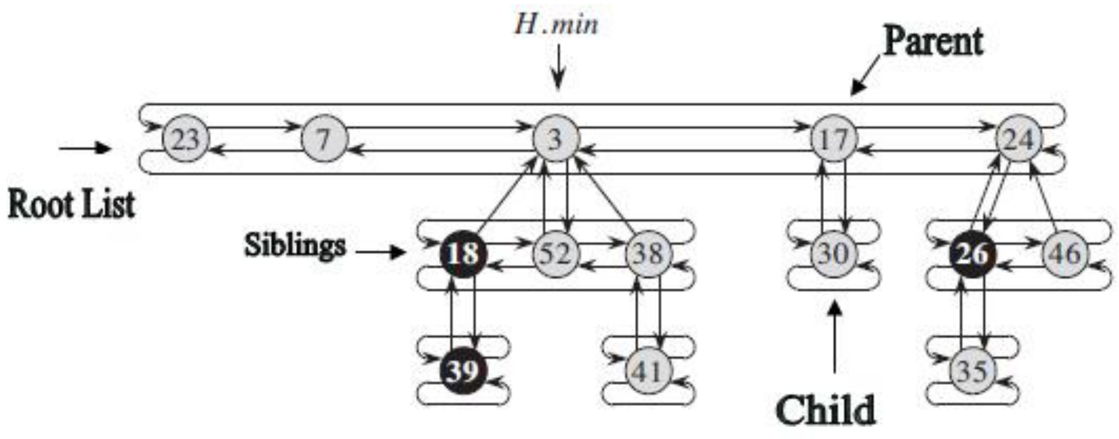


```
Last login: Fri Nov 18 18:17:17 on ttys000
Jinals-MacBook-Pro:~ Jiya$ ssh jbmashru@thunder.cise.ufl.edu
Authorized Access only
____________________________________________________________
UNAUTHORIZED ACCESS TO THIS DEVICE IS PROHIBITED.
You must have explicit permission to access this
device. All activities performed on this device
are logged. Any violations of access policy can
result in disciplinary action.
_____________________________________________________________
jbmashru@thunder.cise.ufl.edu's password:
Last login: Fri Nov 18 18:17:21 2016 from 10.137.99.183
thunderx:1% cd ADS/Assignment1/Solution/
thunderx:2% make
javac -g hashtagcounter.java
thunderx:3% java hashtagcounter input.txt
thunderx:4%
```

4. Definition

- Fibonacci Heap is a data structure for priority queues, consisting a collection of trees with min-heap or max-heap property
- A min Fibonacci heap is a collection of min trees; a max Fibonacci heap is a collection of max trees
- Binomial heap is a special case of Fibonacci heaps
- Fibonacci heap supports the following four operation of Binomial heap:
    - GetMax / GetMin
    - Insert
    - DeleteMax / DeleteMin
    - Meld
- Fibonacci heaps supports following additional operations:
    - IncreaseKey / DecreaseKey
    - Delete (Arbitrary vertex)

5. **Fibonacci VertexStructure:**
    - Fibonacci Heap maintains a circular linked list for vertices present at each level
    - Each vertex has following pointers:
        - Pointer to left sibling
        - Pointer to right sibling
        - Pointer to parent vertex
        - Pointer to its first child
    - Each vertex also has two more attributes:
        - Degree: Counts the number of children
        - lostChild: Indicating whether a vertex has lost a child since the last time it was made the child of another vertex. Newly created vertices have lostChild set to false and a vertex becomes true if a vertex is losing its first child.
    - Fibonacci Heap maintains a min or max pointer to support fast find max or min
    - Please find the below image for your reference:



6. Fibonacci Heap Operations Implementations:

[Note: We will be considering only Max Fibonacci Heap]

- **GetMax:**
  - Throughout the code we have maintained one object that always holds current max value.

```
Vertex currMax = null;
```

- **Insert:**
  - Operation insert works by creating a new Max tree with one element.
  - If it’s the first tree in Fibonacci heap, then we current max vertex should point to this vertex

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

  - If it’s not the first tree, we will add it to root level doubly circular list and adjust pointers of adjacent vertex. Newly inserted vertex will be placed next to current max vertex

```
else{
          n.back = currMax;
          n.front = currMax.front;
          currMax.front = n;
          if(n.front!=null){

               n.front.back = n;
          }
          if(n.front==null){
               n.front = currMax;
               currMax.back = n;
          }
```

  - We would then check if the new inserted Max tree has a frequency higher than existing current max value, then we might want to update it:

```
setMax(n);
```

- **IncreaseKey:**
  - Increase key operation takes vertex, increases its frequency

```
n.freq = currFreq + freq;
```

  - If heap property becomes violated i.e after increasing key, the vertex is greater than the key of its parent, then vertex is cut from its parent. We will reduce parent's degree by 1. If parent had only one child, we will reset its child pointer to null. If it had other children, we will set new child as the adjacent vertex of this vertex. Then this vertex must be added to root level doubly circular linked list. Set its parent pointer to null and lostChild attribute will be set to 'false'. We will now cascading cut method to determine if we need to refactor any of our ancestors.

```
if(myParent.freq < n.freq){
            removeFromList(n);
            myParent.degree = myParent.degree - 1;
            if(myParent.child == n){
                  myParent.child = n.front;
            }
            if(myParent.degree == 0){
                  myParent.child = null;
            }
            adjustAtRootLevel(n);
            n.parent = null;
            n.lostChild = false;
            cascadingCut(n);
            }
```

- **CascadingCut:**
    - When a vertex is being cut from its parent, if a parent is not a root, it's lostChild attribute is set to 'true'. If it's already set to 'true', it is cut as well from its parent and parent's lostChild attribute is set to 'true'. We will recursively perform this operation upwards until we reach either the root or we
      or an vertex lostChild attribute is set to 'false'

```
Vertex temp = myParent.parent;
if(myParent.lostChild == false){
      myParent.lostChild = true;
}
else{
      removeFromList(myParent);
      temp.degree = temp.degree - 1;
      if(temp.child == myParent){
            temp.child = myParent.front;
      }
      if(temp.degree == 0){
            temp.child = null;
      }
      adjustAtRootLevel(myParent);
      myParent.parent = null;
      myParent.lostChild = false;
      cascadingCut(temp);
      }
```

- **DeleteMax:**
    - First we must store current Max vertex in some temporary variable so as to process on it on a local copy

```
Vertex max = currMax;
```

    - On its removal, its children should become roots of new trees. They must be freed from their existing position adjusting pointers of adjacent vertices and then they must be added to root level doubly circular linked list. Set its parent pointer to null and lostChild attribute will be set to 'false'

```
if(max != null){
        Vertex firstChild = max.child;
        Vertex right;
        for(int i = 0; i < max.degree; i++){
              right = firstChild.front;

              removeFromList(firstChild);

              adjustAtRootLevel(firstChild);

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

- Once, all its children are moved to root level, we can now safely remove current maximum vertex

```
removeFromList(max);
```

- For the time being, set right vertex of recently removed maximum vertex as new current max and call pairwise merge

```
     if (max == max.front) {
               currMax = null;
      }
      else {
           currMax = currMax.front;
           pairwiseCombine();

      }
```

- **PairwiseMerge:**
  - First step is to set degree table to null for estimated maximum degree amongst all collections of max trees

```
List<Vertex> degreeTable = new ArrayList<Vertex>(len);
for(int i = 0; i< len; i++){
      degreeTable.add(null);
}
```

  - For each root vertex in root level doubly circular linked list, determine its degree. If for that degree, entry in degree table is set to null, it indicates we encountered a tree with this degree for the first time, and so continue with next vertex

```
for(int i = 0 ; i <totalNoOfRoot; i++){
    Vertex right = t.front;
    int tDegree = t.degree;
    for(;;){
           Vertex x = degreeTable.get(tDegree);
           if(x == null) break;
```

  - Else, if we already have an entry in degree table, we will combine these trees by making tree with smaller frequency root child of another.

```
    if(t.freq < x.freq){
                Vertex a = x;
                x = t;
                t = a;
           }
    removeFromList(x);
    x.parent = t;
```

  - If the maximum vertex has other childrens then we will adjust new child. Or else simply set it as its first child

```
if(t.child != null){
     x.back = t.child;
     x.front = t.child.front;
     t.child.front = x;
     x.front.back = x;
}
else{
    t.child = x;
    x.front = x;
    x.back = x;
    }
```

- Now increment new parent vertex degree by 1 and since vertex with smaller frequency will be made child, set its lostChild attribute to ‘false’. Set degree table entry to null and set new entry in degree table with incremented degree to point to this newly created tree

7. Time Complexity:

Operation	Binomial	Fibonacci
FindMax	$\Theta(\log n)$	$\Theta(1)$
DeleteMax	$\Theta(\log n)$	$\Theta(\log n)$
Insert	$\Theta(1)$	$\Theta(1)$
IncreaseKey	$\Theta(\log n)$	$\Theta(1)$
PairwiseMerge	$\Theta(\log n)$	$\Theta(1)$

8. Parsing Input File:

- We are given an input file with following data :

#saturday 2
#sunday 4
#saturday 5
1

- So let's make a pattern string to distinguish between which one are hashtags and which ones are frequency

```
String myHashTagPattern = "((#)([a-z]+)(\\s)(\\d+))";
String mykeyPattern = "(\\d+)";
Pattern p1 = Pattern.compile(myHashTagPattern);
Pattern p2 = Pattern.compile(mykeyPattern);
Matcher m1 = p1.matcher(sCurrentLine);
Matcher m2 = p2.matcher(sCurrentLine);
```

- If we find a hit for pattern "#saturday 4". We need to check if this entry is already existing in our hashmap. If it does, then we need to call increasekey operation of Fibonacci Heap. Otherwise, we will simply insert it into the Fibonacci Heap.

```
if((m1.find())) {
      String value = m1.group(3);
      int key =Integer.parseInt(m1.group(5));
      if(!(hm.containsKey(value)))
      {
            Vertex n1 = new Vertex(value,key);
            fib.insert(n1);
            hm.put(value, n1);
      }
      else{
            Vertex n = hm.get(value);
            fib.increaseFreq(n,key);
            }
    }
```

- If we find a hit for pattern "4". We need to call removeMax operation of Fibonacci heap for the specified count (say '4' in this case). We will store each extracted max vertex in queue and at the same time write it to output file. Once all the required number of vertices are extracted, we will insert it back to heap

```
else if(!m1.find() && m2.find()){
    int count = Integer.parseInt(m2.group(1));
    Queue<Vertex> q= new LinkedList<Vertex>();
    for(int i = 0; i < count; i++)
    {
     Vertex maximum = fib.extractMax();
     hm.remove(maximum.hashTag);
     Vertex n1 = new Vertex(maximum.hashTag, maximum.freq);
     q.add(n1);
     if(i==count-1)
     {
       writer.print(n1.hashTag);
     }
     else {
       writer.print(n1.hashTag+", ");
     }
    }
    writer.println();
    while(q.peek()!=null){
     Vertex n = q.poll();
     fib.insert(n);
     hm.put(n.hashTag,n);
    }
}
```

9. Conclusion & References:

Conclusion:

- Fibonacci heaps are more efficient in determine use cases like finding most recent hashtag or frequently searched key due to its better complexities when compared to its later derivatives, including quake heaps, violation heaps, strict Fibonacci heaps, rank pairing heaps

References:

- Wikipedia : https://en.wikipedia.org/wiki/Fibonacci_heap
- Chapter 19 – Fibonacci Heap of Introduction to Algorithms by Thomas H. Cormen