Fibonacci heap and its application in Dijkstra's algorithm and Prim's algorithm

By

Rushitkumar M. Jasani	**(2018201034)**
Priyendu D. Mori	**(2018201103)**

A project submitted as a
part of the course

Advanced Problem Solving

Guide
Dr. Venkatesh Choppella

Teaching Assistant
Sunchit Sharma

1. Introduction

1.1. Fibonacci heap

A Fibonacci heap is a *collection of rooted trees* that are min-heap ordered. That is, each tree obeys the min-heap property: the key of a node is greater than or equal to the key of its parent.

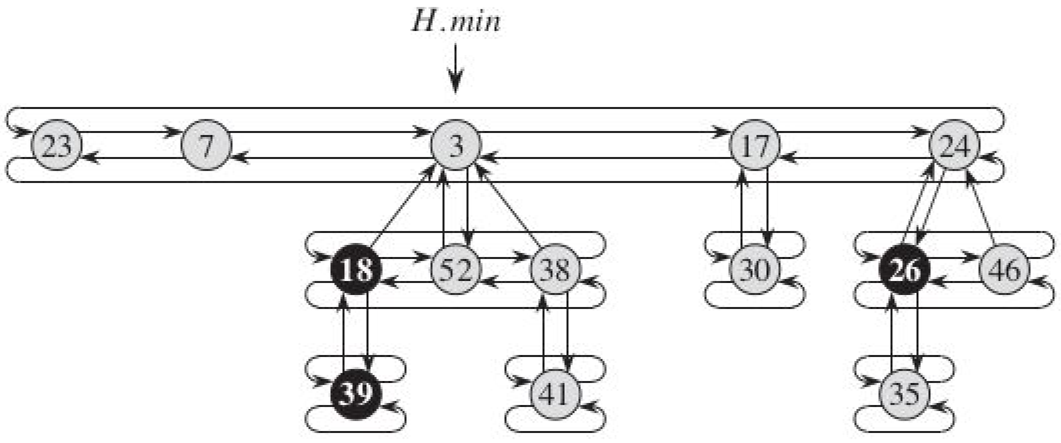
From a theoretical standpoint, Fibonacci heaps are especially desirable when the number of EXTRACT-MIN and DELETE operations is small relative to the number of other operations performed. This situation arises in many applications. For example, some algorithms for graph problems may call DECREASE-KEY once per edge. For *dense graphs*, which have many edges, the $\Theta(1)$ amortized time of each call of DECREASE-KEY adds up to a big improvement over the $\Theta(\lg n)$ worst-case time of binary heaps.

From a practical point of view, however, the constant factors and programming complexity of Fibonacci heaps make them less desirable than ordinary binary (or k-ary) heaps for most applications, except for certain applications that manage large amounts of data. Thus, Fibonacci heaps are predominantly of theoretical interest. If a much simpler data structure with the same amortized time bounds as Fibonacci heaps were developed, it would be of practical use as well.

2. Implementation

2.1. Node Structure

As shown in Figure below, each node x contains a pointer x.*p* to its parent and a pointer x.*child* to any one of its children. The children of x are linked together in a circular, doubly linked list, which we call the child list of x. Each child y in a child list has pointers y.*left* and y.*right* that point to y's left and right siblings, respectively. If node y is an only child, then y.*left* = y.*right* = y. Siblings may appear in a child list in any order.



2.2. Operations

1. insert
2. get_min
3. merge_heap
4. decrease_key
5. extract_min

2.2.1. insert

Insertion simply involves creation of a new node with the required key and adding it to the root list. No attempt is made to combine the resulting list in any manner thus resulting in a constant time operation

2.2.2. get_min

It simply returns the minimum node pointer maintained with the heap. It is a constant time operation.

2.2.3. merge_heap

To merge two Fibonacci Heaps, we simply need to combine one root list with the another, update node count and minimum node pointer appropriately. It is a constant time operation.

2.2.4. decrease_key

In Fibonacci Heaps, the decrease_key operation is performed by simply cutting the node from its parent if the heap property is violated by the decreased node. However if cut trees arbitrarily, the exponential relation between the outdegree and the number of nodes may no longer be maintained. To avoid this, we follow a marking scheme, which ensures the exponential relationship.

Using amortised analysis, the running time of decrease_key operation comes out to be O(1).

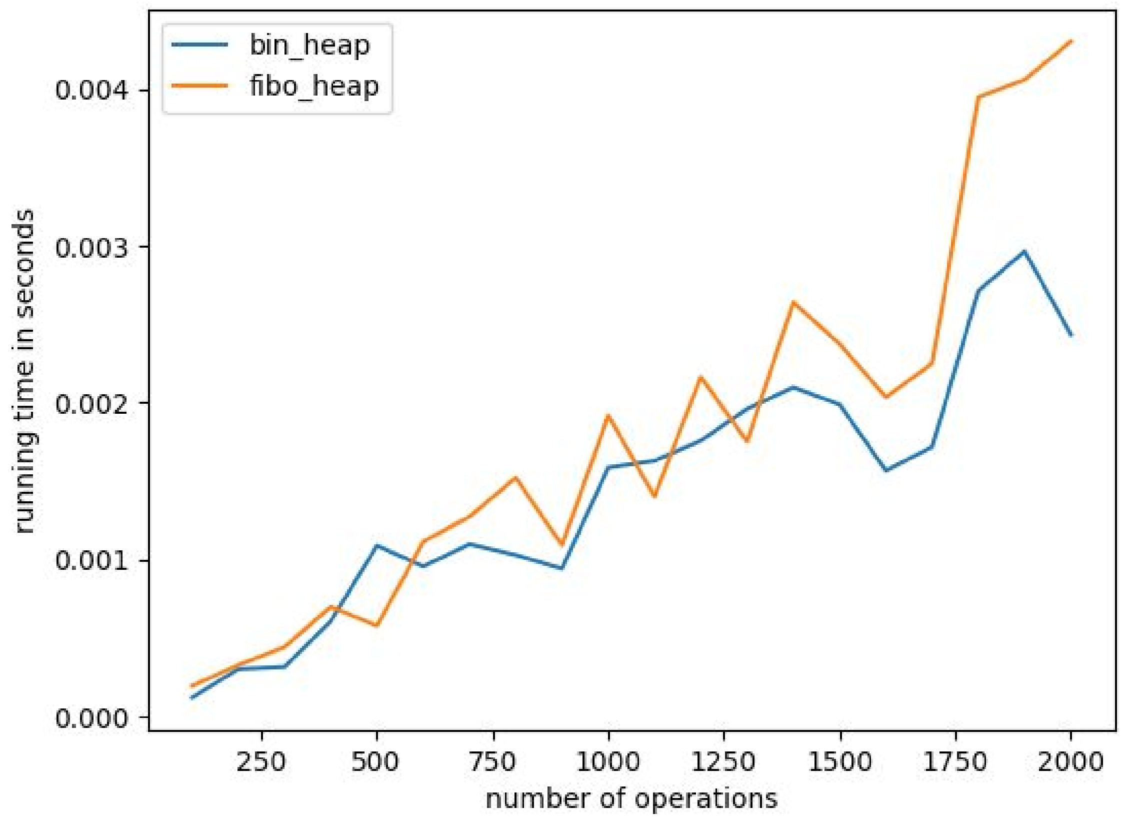
2.2.5. extract_min

The operation of deleting the minimum node is much more complicated than the previously defined operations. The task of combining all the trees in the root list to reduce their number is also done in the extract_min operation. Using amortised analysis, the running time of extract_min comes out be O(log n).

The extract_min operation combines the root list such that for a particular value of outdegree of the root, only one tree is present. This is done by making an array of trees with the outdegree of the tree as its index. Obviously the array can store only one tree with a particular outdegree. Now all the trees in the root list are fetched and added to this array according to their outdegree. If another tree is already present in the location , the two trees of equal outdegree are linked such that heap property is maintained resulting in a tree with outdegree increased by unity. This tree is again added to the next index location and the above process repeated until an empty index is found for the new tree resulting at each step. The maximum resulting outdegree by following the above process can shown to be O(log n).

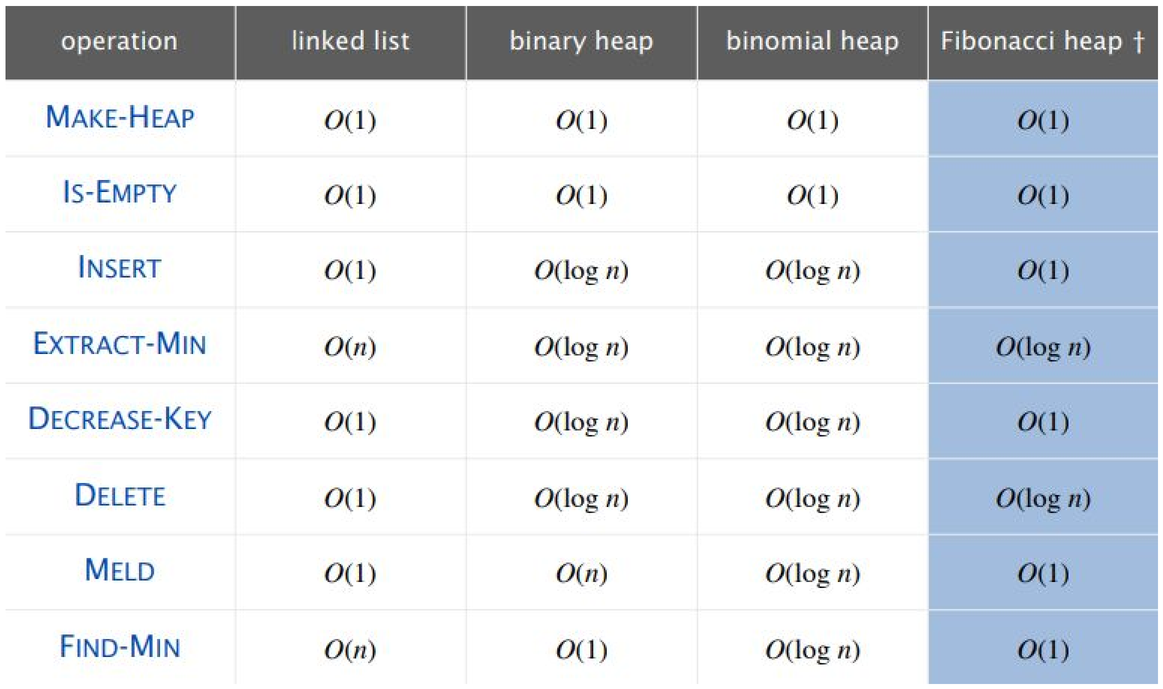
2.3. Testing of Fibonacci Heap

To test the implementation, we performed same operations on fibonacci heap as well as binary heap through a script and their outputs turned out to be the same with variation in running time as shown in the graph below :



As we can see from the graph, Fibonacci heap doesn't perform well in every case. As mentioned above, the internal constants of the fibonacci heap makes it slower which is evident in the graph.

Priority queues performance cost summary



operation	linked list	binary heap	binomial heap	Fibonacci heap †
MAKE-HEAP	$O(1)$	$O(1)$	$O(1)$	$O(1)$
IS-EMPTY	$O(1)$	$O(1)$	$O(1)$	$O(1)$
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
EXTRACT-MIN	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
DECREASE-KEY	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
DELETE	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
MELD	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$
FIND-MIN	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$

† *amortized*

As complexity of decrease key operation in fibonacci heap is better compared to the same in other priority queues, it does give a better performance in algorithms that use decrease key a lot like *Dijkstra's single source shortest path* algorithm and *Prim's minimum spanning tree* algorithm.

3. Dijkstra’s SSSP Algorithm using fibonacci heap

3.1. Introduction

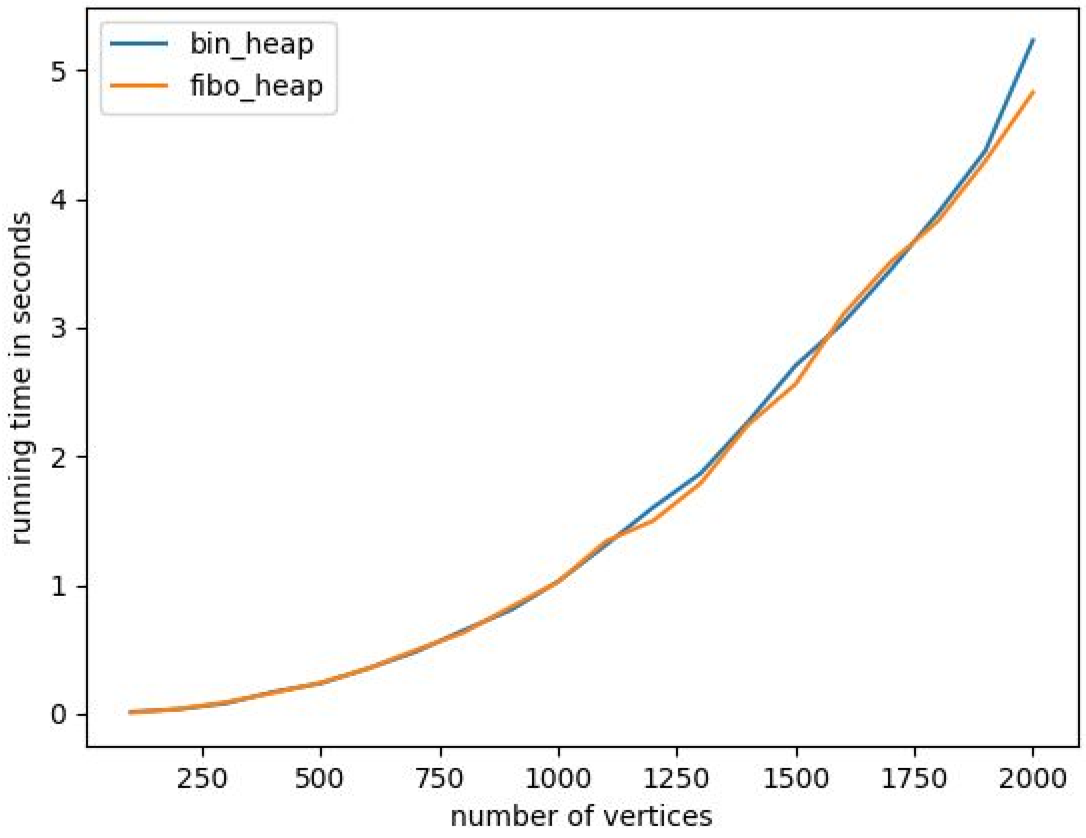
Dijkstra’s algorithm computes the shortest paths from a source vertex to every other vertex in a graph, the so-called single source shortest path problem. The implementations of Dijkstra’s algorithm vary in the data structure that is used for the algorithm. Here the Fibonacci heap and binary heap implementations of Dijkstra’s algorithm are implemented and compared.

3.2. Comparison

Using fibonacci heap time complexity of Dijkstra’s algorithm decreases to $O(E + V\log(V))$ from $O((E + V) \log(V))$ where E is number of edges and V is number of vertices present in the graph. So for a dense graph (i.e. $E = O(V^2)$) the implementation of Dijkstra’s algorithm using fibonacci heap performs better compared to binary heap.

We created a script that generates test data for complete graph of 100 to 2000 vertices and runs both the implementations for those test data which resulted in the following graph.

Following graph works as a proof that fibonacci heap are better from a theoretical standpoint but not practical as it didn’t give much of a performance boost as promised in theory.


bin_heap
fibo_heap
running time in seconds
number of vertices
0
1
2
3
4
5
250
500
750
1000
1250
1500
1750
2000

4. Prim's Minimum Spanning Tree using Fibonacci Heap

4.1. Introduction

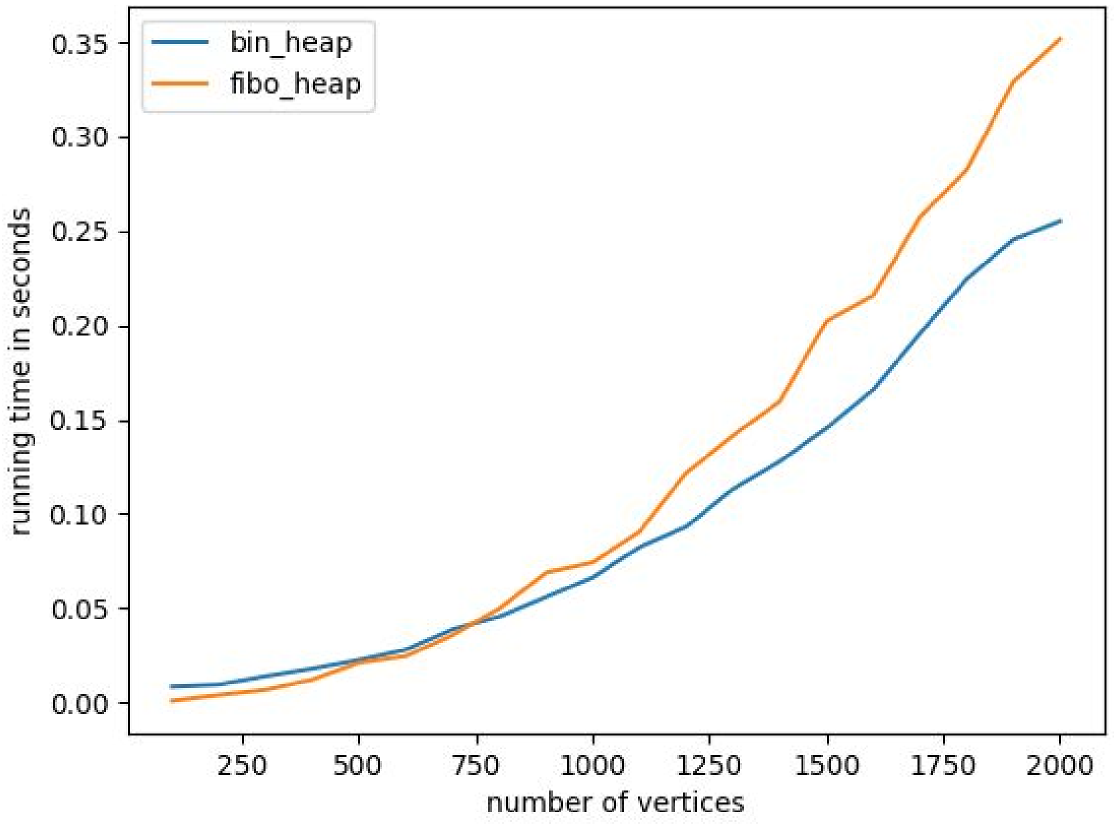
Prim's algorithm computes the minimum spanning tree in a graph. The implementations of Prim's algorithm vary in the data structure that is used for the algorithm. Here the Fibonacci heap and binary heap implementations of Prim's algorithm are implemented and compared.

4.2. Comparison

Using fibonacci heap time complexity of Prim's algorithm decreases to $O(E + V\log(V))$ from $O((E + V) \ \log(V))$ where E is number of edges and V is number of vertices present in the graph. So for a dense graph (i.e. $E = O(V^2))$ the implementation of Prim's algorithm using fibonacci heap performs better compared to binary heap.

We created a script that generates test data for complete graph of 100 to 2000 vertices and runs both the implementations for those test data which resulted in the following graph.

Following graph works as a proof that fibonacci heap are better from a theoretical standpoint but not practical as it didn't give much of a performance boost as promised in theory.


bin_heap
fibo_heap
running time in seconds
number of vertices
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
250
500
750
1000
1250
1500
1750
2000

5. User manual

5.1. Dependencies

Platform : Linux

Software requirements :

1) **python 2.7**
2) **SortedSet python module** : *pip install sortedcontainers --user*
3) **matplotlib python module** : *pip install matplotlib --user*
4) **tkinter python module** : *sudo apt-get install python-tk*
5) **gnu g++ compiler** : *sudo apt-get install g++*

5.2. Directory structure of project

```
.
├── binary_heap
│   ├── binary_heap.h
│   └── binary_heap_test.cpp
├── dijkstra_sssp_algorithm
│   ├── dijkstra_bin.cpp
│   └── dijkstra_fibo.cpp
├── fibonacci_heap
│   ├── fibonacci_heap.h
│   └── fibonacci_heap_test.cpp
├── prims_mst_algorithm
│   ├── prims_bin.cpp
│   └── prims_fibo.cpp
├── readMe.md
└── scripts
    ├── compare.py
    ├── dijkstra_tc.py
    ├── plot1.py
    ├── plot.py
    ├── prims_tc.py
    ├── testcase_generator.py
    ├── test_dij.sh
    ├── test_prim.sh
    ├── test.sh
    └── tmp
```

5.3. To run heaps manually

5.3.1. Fibonacci heap

run following command from root

1) cd fibonacci_heap
2) g++ fibonacci_heap_test.cpp
3) ./a.out

input format:

1) first line contains number of operations N.
2) next N lines contains one of the following operations:
   a) 0 x :- to insert x into heap
   b) 1 :- to get minimum from heap
   c) 2 :- to extract minimum from heap
   d) 3 x y :- to decrease x to y in heap

5.3.2. Binary heap

run following command from root

1) cd binary_heap
2) g++ binary_heap_test.cpp
3) ./a.out

input format same as above

5.4. To run and compare both heaps on random inputs

run following commands from root

1) cd scripts
2) ./test.sh

test.sh :

It first compiles both the heaps then generate random input in file ***tmp/input*** by running ***testcase_generator.py*** for operations ranging from 100 to 3000 in gaps of 150 and runs both the heaps on this input files and stores their output (***tmp/fibo_out*** and ***tmp/bin_out***) and running times(***tmp/fibo_time*** and ***tmp/bin_time***) in their respective files, compare both outputs by running ***compare.py*** and plot their times using ***plot1.py***.

5.5. To run Dijkstra’s algorithm

5.5.1. Implementation using fibonacci heap

run following command from root

1) cd dijkstra_sssp_algorithm
2) g++ dijkstra_fibo.cpp
3) ./a.out

input format:

1) first line contains number of test cases T.
2) first line of each test case contains number of vertices V and edges E in the undirected graph.
3) next E lines contains u,v and w :: source, destination and weight of edge.
4) last line of each test case contains s which is source vertex for single source shortest path algorithm.

5.5.2. Implementation using binary heap

run following command from root

1) cd dijkstra_sssp_algorithm
2) g++ dijkstra_bin.cpp
3) ./a.out

input format same as above

5.6. To run and compare Dijkstra’s algorithm on random inputs

run following commands from root

1) cd scripts
2) ./test_dij.sh

test_dij.sh :

It first compiles both the implementations of Dijkstra’s algorithm then generate random input in file ***tmp/input*** by running ***dijkstra_tc.py*** for operations ranging from 100 to 2000 in gaps of 100 and runs both the implementations on this input files and stores their output (***tmp/fibo_out*** and ***tmp/bin_out***) and running times (***tmp/fibo_time*** and ***tmp/bin_time***) in their respective files, compare both outputs by running ***compare.py*** and plot their times using ***plot.py***.

5.7. To run Prim's algorithm

5.7.1. Implementation using fibonacci heap

run following command from root

1) cd prims_mst_algorithm
2) g++ prims_fibo.cpp
3) ./a.out

input format:

1) first line contains number of vertices V and edges E in the undirected graph.
2) next E lines contains u,v and w :: source, destination and weight of edge.
3) last line of each test case contains s which is starting vertex for minimum spanning tree algorithm.

5.7.2. Implementation using binary heap

run following command from root

1) cd prims_mst_algorithm
2) g++ prims_bin.cpp
3) ./a.out

input format same as above

5.8. To run and compare Prim's algorithm on random inputs

run following commands from root

1) cd scripts
2) ./test_prim.sh

test_prim.sh :

It first compiles both the implementations of Prim's algorithm then generate random input in file ***tmp/input*** by running ***prims_tc.py*** for operations ranging from 100 to 2000 in gaps of 100 and runs both the implementations on this input files and stores their output (***tmp/fibo_out*** and ***tmp/bin_out***) and running times (***tmp/fibo_time*** and ***tmp/bin_time***) in their respective files, compare both outputs by running ***compare.py*** and plot their times using ***plot.py***.

6. References

- Introduction to Algorithms - Book by Charles E. Leiserson, Clifford Stein, Ronald Rivest, and Thomas H. Cormen
- https://en.wikipedia.org/wiki/Fibonacci_heap
- https://www.growingwiththeweb.com/data-structures/fibonacci-heap/overview
- https://web.stanford.edu/class/archive/cs/cs166/cs166.1146/lectures/07/Slides07.pdf
- https://www.youtube.com/playlist?list=PL6c3bOl0t1DEmWKmJ81xLnCB9wh0WggBg
- https://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/FibonacciHeaps.pdf