FIBONACCI HEAPS

Nirali Bandaru and Tilly Erwin

Dec 2019

MAT8650 – Data Structures

Prof. Timo Heister

Clemson University

INTRODUCTION

The Fibonacci heap is a data structure that performs priority queue operations using a collection of trees satisfying the min or max-heap property. It was originally developed to improve Dijkstra's algorithm. More efficient than its counterparts, the binomial heap or binary heap, most of the methods in the Fibonacci heap have a constant amortized time complexity.

In this investigation, a Fibonacci heap satisfying the min-heap property is implemented using dictionaries and pointers. Then, its time complexity of its operations is observed over various 'n' and compared with analogous operations of a standard priority queue.

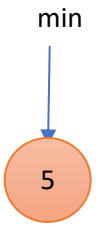
WORKING OF THE FIBONACCI HEAP

The Fibonacci heap performs the following functions:

1. Insert
2. Find Minimum
3. Delete Minimum
4. Decrease Priority
5. Merge

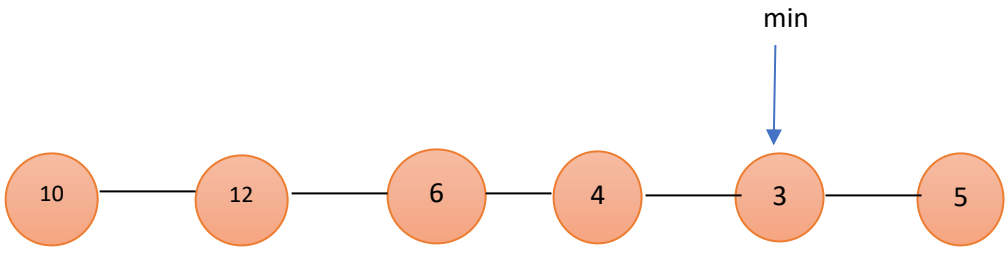
Visualizing the algorithm:

1. First node inserted:



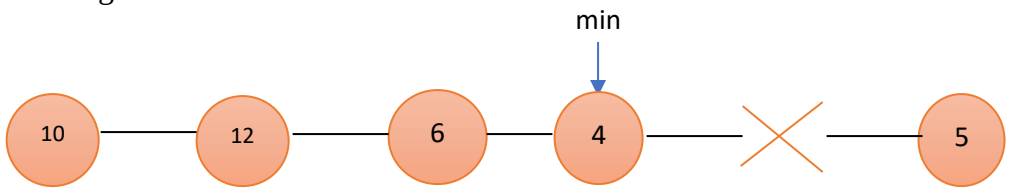
- The min pointer points to the first element that is inserted into the tree

2. Arbitrary number of nodes inserted:



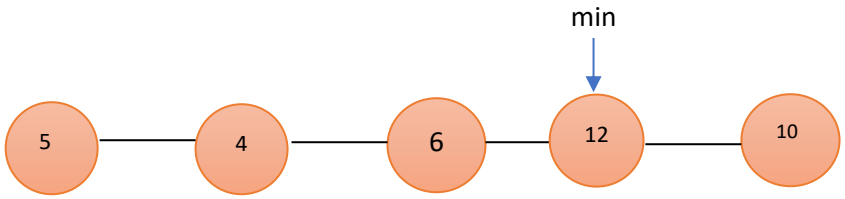
- Minimum element is updated

3. Deleting the minimum element:

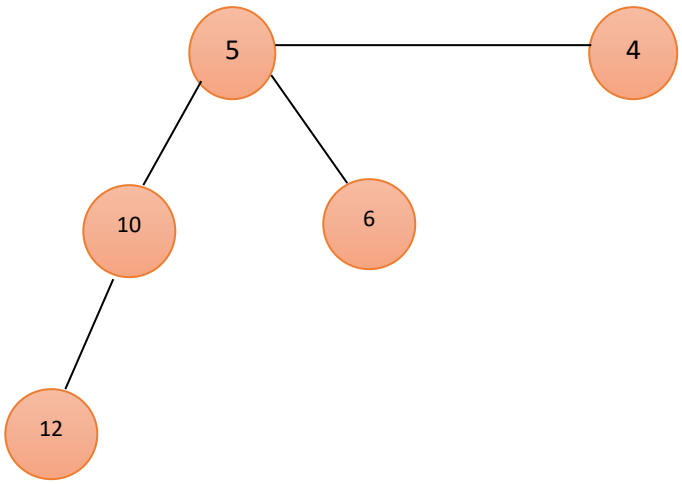


- Minimum element is updated

4. Merging happens after deleting the min element



Merged:




```
FH._adjMap {1: [], 2: [], 3: [], 4: [], 5: [], 6: []}
Parent Map:  {1: None, 2: None, 3: None, 4: None, 5: None, 6: None}
Root list:  [1, 2, 3, 4, 5, 6]
Value Dictionary:  {1: 5, 2: 3, 3: 4, 4: 6, 5: 12, 6: 10}

After deleting minimum element:  {1: [4, 6], 3: [], 4: [], 5: [], 6: [5]}
Root List:  [1, 3]
Parent map:  {1: None, 2: None, 3: None, 4: 1, 5: 6, 6: 1}
Degree Dictionary:  {1: 4, 3: 1, 4: 1, 5: 1, 6: 2}
```

IMPLEMENTATION

Dictionaries/Lists created in the implementation:

Dictionaries:

1. Adjacency Map: Generates the tree with respect to the ID's of the elements
2. Parent Map: Maps each child to its parent with respect to the ID's of the elements
3. Degree Dictionary: Maps the element ID to the respective degree
4. ID-to-Value: Each ID is mapped to its corresponding element
5. Value-to-ID: Each element value is mapped to its corresponding ID

Lists:

6. Root List: stores the ID's of the roots
7. Mark List: stores the nodes that have already had children cut from them

Methods created in the implementation:

1. Find min
   - Returns the minimum.
   - If the user wants the minimum value, an output results in this method returning the minimum value by accessing the ID to Value dictionary.
   - Otherwise (with an output of 1), the method returns its ID number. The minimum pointer is pointed to the ID number.
2. Delete min
   - The delete min method performs the search for the minimum value and updates it.
   - First, Minimum element's ID is accessed and deleted.
   - The ID of the minimum is deleted from the Root List.
   - If the min element has children, the IDs of the children are added to the Root List.
   - These children are marked to have no parent.
   - Min element is deleted from the Degree Dictionary.
   - Min element's value is deleted from the ID to Value dictionary.
   - Min element's ID is deleted from the Value to ID dictionary.
   - Then, the elements are merged into one tree when Merge All is called.
3. Insert
   - When an element is inserted, it is first assigned an ID using a counter that is initialized in the beginning.
   - This element is stored in Degree Dictionary, Root List, Value Dictionary and ID dictionary.
   - A new tree is made for this element in order to add children when necessary.
   - If there are elements present in the list already, then the new element is compared with the previous elements and minimum element is updated if necessary.

4. Decrease priority
    - When an element is assigned a higher priority (lower value), this method places the element with the new priority appropriately in order to satisfy the min-heap property.
    - First, the method checks if the new priority is a lower number than the old one.
    - The new element is given a new ID.
    - If the value list is empty:
        i. Then, the new priority is updated in the Value Dictionary.
        ii. Old value and old ID of the element are deleted.
    - Otherwise:
        i. Value and ID of the last added element are deleted.
    - For the changed priority, make a new tree and add it to the value dictionary
    - If the min-heap property is not violated:
        i. The old element is deleted.
    - If the min-heap property is violated:
        i. This means that the value is lower than the parent.
        ii. Element is cut from the parent, and parent is marked as having children cut from it.
        iii. If the parent had already been marked, the parent node is cut as well, and its parent is marked.
5. Cut Node
    - This method is used when the min-heap property is violated and the value of the child is lower than the value of the parent, which means the priority of the child is higher.
    - If an element is added to children of a node and that node is marked already, this method allows for the node to be cut from the tree and made its own tree so as to avoid future conflict.
6. Merge
    - Two roots are compared and merged.
    - Degree is updated.
7. Merge All
    - Keys are sorted based on their values.
    - A temporary degree is created for each element
    - If two elements are known to have the same degree, they are merged.
    - Merge () is called repeatedly until the tree is compact
    - Temporary degree is removed.

TESTING

- An automated test was written for each method to ensure that it worked for all possible inputs.
- These tests included a test for duplicates. The code proved to be able to handle duplicates due to the assigning of ID’s for each element that is inserted into the collection of trees.

METHOD TESTS:

- Insert:

```
INSERT FUNCTION TEST:
Adj Map: {1: [], 2: [], 3: [], 4: [], 5: [], 6: [], 7: [], 8: [], 9: [], 10: [], 11: [], 12: [], 13: [], 14: [], 15: [], 16:
[], 17: [], 18: [], 19: [], 20: []}
Parent Map: {1: None, 2: None, 3: None, 4: None, 5: None, 6: None, 7: None, 8: None, 9: None, 10: None, 11: None, 12: None, 13:
None, 14: None, 15: None, 16: None, 17: None, 18: None, 19: None, 20: None}
Root List: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
3 54
{1: 54, 2: 51, 3: 8, 4: 76, 5: 60, 6: 3, 7: 6, 8: 91, 9: 3, 10: 59, 11: 38, 12: 59, 13: 86, 14: 75, 15: 60, 16: 22, 17: 56, 18:
59, 19: 70, 20: 13}
```

- Find Min:

```
FIND MIN TEST:
{1: 94, 2: 68, 3: 73, 4: 97, 5: 97, 6: 96, 7: 65, 8: 52, 9: 52, 10: 81}
Minimum:  8
Insert 3:  {1: 94, 2: 68, 3: 73, 4: 97, 5: 97, 6: 96, 7: 65, 8: 52, 9: 52, 10: 81, 11: 3}
Minimum after adding 3:  11
```

- Delete Min:

```
DELETE MIN TEST:
{1: 75, 2: 28, 3: 81, 4: 21, 5: 80, 6: 99, 7: 26, 8: 24, 9: 74, 10: 72}
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 6, 0, 0, 0, 0, 0, 0, 0]
Root Comparison 3 6
[0, 0, 3, 0, 0, 0, 0, 0, 0]
[0, 5, 3, 0, 0, 0, 0, 0, 0]
Root Comparison 1 5
Root Comparison 1 3
[0, 0, 0, 0, 1, 0, 0, 0, 0]
[0, 9, 0, 0, 1, 0, 0, 0, 0]
Root Comparison 10 9
[0, 0, 10, 0, 1, 0, 0, 0, 0]
[0, 2, 10, 0, 1, 0, 0, 0, 0]
Root Comparison 7 2
Root Comparison 7 10
Root Comparison 7 1
[0, 0, 0, 0, 0, 0, 0, 0, 7]
After deleting min:  {1: 75, 2: 28, 3: 81, 5: 80, 6: 99, 7: 26, 8: 24, 9: 74, 10: 72}
```

- Decrease Priority:

```
DECREASE PRIORITY TEST:
Adj Map: {1: [], 2: [], 3: [], 4: [], 5: [], 6: [], 7: [], 8: []}
Adj Map- decreased priority from 5 to 3: {1: [], 2: [], 3: [], 4: [], 5: [], 6: [], 7: [], 9: []}
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 2, 0, 0, 0, 0, 0, 0, 0]
Root Comparison 3 2
[0, 0, 3, 0, 0, 0, 0, 0, 0]
[0, 4, 3, 0, 0, 0, 0, 0, 0]
Root Comparison 8 4
Root Comparison 8 3
[0, 0, 0, 0, 8, 0, 0, 0, 0]
[0, 10, 0, 0, 8, 0, 0, 0, 0]
Root Comparison 5 10
[0, 0, 5, 0, 8, 0, 0, 0, 0]
[0, 7, 5, 0, 8, 0, 0, 0, 0]
Root Comparison 1 7
Root Comparison 1 5
Root Comparison 1 8
[0, 0, 0, 0, 0, 0, 0, 0, 1]
```

- Handling Duplicates:

```
Test Duplicate Insert
Test Duplicate decrease key
Test Duplicate Delete Min
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 13, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Root Comparison 12 13
[0, 0, 12, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 11, 12, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Root Comparison 10 11
Root Comparison 10 12
[0, 0, 0, 0, 10, 0, 0, 0, 0, 0, 0, 0]
[0, 9, 0, 0, 10, 0, 0, 0, 0, 0, 0, 0]
Root Comparison 8 9
[0, 0, 8, 0, 10, 0, 0, 0, 0, 0, 0, 0]
[0, 5, 8, 0, 10, 0, 0, 0, 0, 0, 0, 0]
Root Comparison 6 5
Root Comparison 6 8
Root Comparison 6 10
[0, 0, 0, 0, 0, 0, 0, 0, 6, 0, 0, 0]
[0, 3, 0, 0, 0, 0, 0, 0, 6, 0, 0, 0]
Root Comparison 4 3
[0, 0, 4, 0, 0, 0, 0, 0, 6, 0, 0, 0]
[0, 2, 4, 0, 0, 0, 0, 0, 6, 0, 0, 0]
Root Comparison 14 2
Root Comparison 14 4
2
11
```

- Merge:

```
MERGE TEST:
Adj Map: {1: [], 2: [], 3: [], 4: [], 5: [], 6: [], 7: [], 8: [], 9: [], 10: []}
Root Comparison 9 10
After merging:  {1: [], 2: [], 3: [], 4: [], 5: [], 6: [], 7: [], 8: [], 9: [10], 10: [], 11: [], 12: []}
```

- Merge All:

```
MERGE ALL TEST:
Adj Map: {1: [], 2: [], 3: [], 4: [], 5: [], 6: [], 7: [], 8: [], 9: [], 10: []}
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 2, 0, 0, 0, 0, 0, 0, 0, 0]
Root Comparison 3 2
[0, 0, 3, 0, 0, 0, 0, 0, 0, 0]
[0, 8, 3, 0, 0, 0, 0, 0, 0, 0]
Root Comparison 6 8
Root Comparison 6 3
[0, 0, 0, 0, 6, 0, 0, 0, 0, 0]
[0, 5, 0, 0, 6, 0, 0, 0, 0, 0]
Root Comparison 9 5
[0, 0, 9, 0, 6, 0, 0, 0, 0, 0]
[0, 7, 9, 0, 6, 0, 0, 0, 0, 0]
Root Comparison 1 7
Root Comparison 1 9
Root Comparison 1 6
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
[0, 10, 0, 0, 0, 0, 0, 0, 1, 0]
Root Comparison 4 10
After merging: {1: [7, 9, 6], 2: [], 3: [2], 4: [10], 5: [], 6: [8, 3], 7: [], 8: [], 9: [5], 10: []}
Final root list: [1, 4]
```

- Below is the error-free output of the method tests and test for duplicates:

```
----------------------------------------------------------------------
Ran 7 tests in 0.016s

OK
<unittest.runner.TextTestResult run=7 errors=0 failures=0>
```

COMPLEXITY

Complexity of the Fibonacci Heap:	
Method	**Complexity**
Find min	O (1)
Delete min	O (log n)
Insert	O (1)
Decrease Priority	O (1)
Merge	O (1)
Merge All	O (log n)

1. Find-min:
    a. No computation involved.
    b. Method only returns value from dictionary, or returns value of pointer, and therefore is O (1)
2. Delete min:
    a. Due to the calling of merge All method, this method's complexity becomes O (log n).
3. Insert:
    a. Insert does not involve computation.
    b. Element that is inserted is stored in respective lists and dictionaries.
    c. Therefore, has a complexity of O (1).
    d. This method also allows find-min to be O (1) because it points the minimum pointer to the minimum value and updates the minimum value each time an element is inserted.
4. Decrease Priority:
    a. Works with comparisons, no computation involved.
    b. Therefore, has a complexity of O (1).
5. Cut Node:
    a. Does not involve computation. Only comparisons. So, has complexity of O (1).
6. Merge:
    a. Does not involve computation. Only comparisons. So, has complexity of O (1).
7. Merge All:
    a. This method sorts based on values, and merges O (log n) times, and there has complexity of O (log n).

In comparison,

Complexity of a Priority Queue:

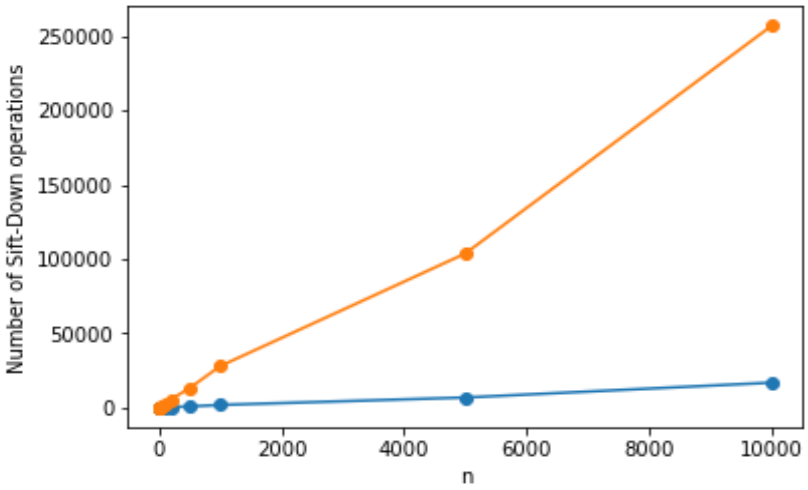
Method	Complexity
Find min	O (1)
Delete min	O (log n)
Insert	O (log n)
Decrease Priority	O (log n)

1. Find min returns minimum value with no computation, so is O (1).
2. Delete min is compared to sift down method and has a complexity of O (log n).
3. Insert is compared to bubble up method and has a complexity of O (log n).
4. Decrease priority complexity usually involves a bubble up and therefore has a complexity of O (log n)

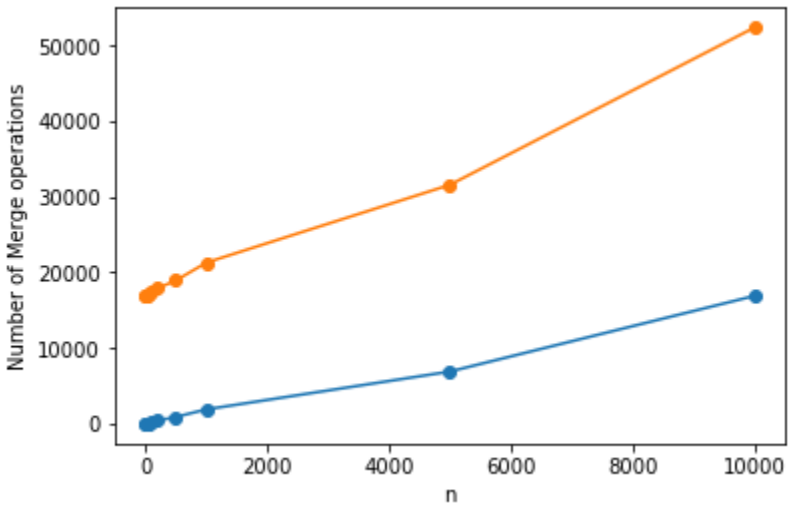
Comparison of Complexity between Priority Queue and Fibonacci Heap:

In order to test complexity, for various sizes of n, the number of operations of bubble up in a Priority Queue was compared with the number of operations of insert in a Fibonacci Heap. Likewise, the number of operations of merge in Fibonacci Heap were compared with the number of operations of sift down in Priority Queue in order to compare the complexity delete min and sift down methods. In Fibonacci Heap, merge operations were counted to demonstrate the complexity of delete min because merge All () method is called in delete min () and merge All depends on merge ().

Priority Queue:



Fibonacci Heap:



Legend-

Orange: Sift Down/ Merge

Blue: Bubble Up/ Insert

As demonstrated, the operations for Priority Queue far exceed the operations performed in Fibonacci Heap.

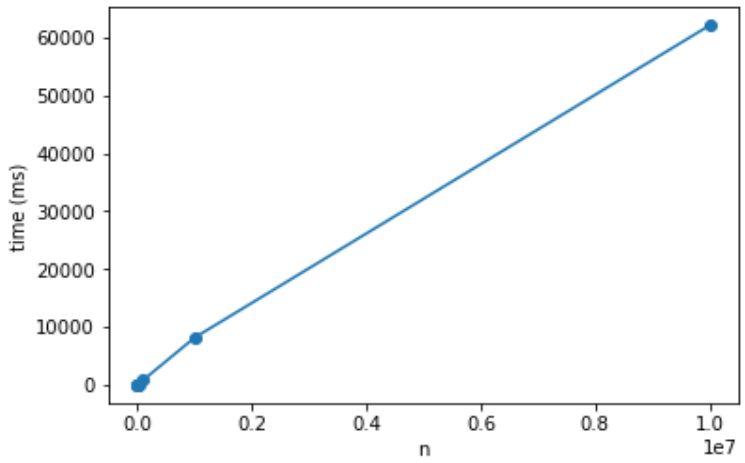
OBSTACLES FACED DURING IMPLEMENTATION:

1. Keeping track of the degree
    a. Various methods were tried in order to keep track of the degree. First thought to be simple, this caused the implementation code to be redone entirely in order to not cause a break.
    b. A degree dictionary was created in order to keep track of degrees.
2. Handling duplicates
    a. Although the original code was working perfectly, it had failed to handle duplicates. Originally, there was a dictionary that mapped the key to its value. However, if the same key was to be inserted a second time, the code would break since keys are to be unique.
    b. This was done by creating an ID number for each element that is inserted. This allowed for two elements of the same priority to have unique corresponding ID numbers, resolving the conflict. Then, there were separate dictionaries for each pair of ID and Value, Value and ID, Degree and Value.

3. Timing methods to demonstrate complexity
    a. An attempt was made to time the methods in the Fibonacci Heap in order to show a complexity of O (log n). However, this was not possible because to show O (log n), n needed to be large and the computer could not handle computations with n as large as 10,000 and above.
    b. However, priority queue was able to be timed and has produced the following graphs:

Bubble Up

```
Bubble Up:

[<matplotlib.lines.Line2D at 0x14a3e4633710>]
```



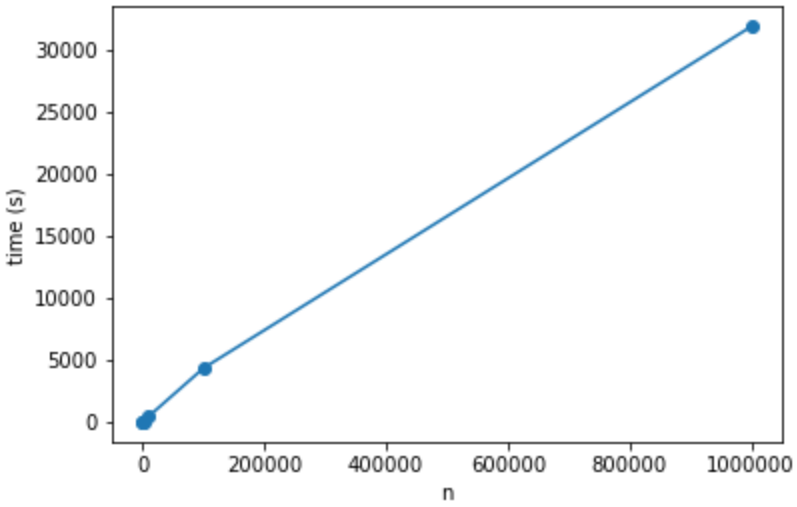
Sift Down

In [*]:

```
import random
import timeit

def siftdown_complexity(n):
        testArray = []
        for item in range(n-1):
            PQ.push(random.randint(1,1000))
            PQ.pop()

sizes = [10, 100, 1000, 10000, 100000, 1000000, 10000000]
print("Siftdown Complexity : ")
for n in sizes:
    print("Testing for size : ",n)
    %timeit siftdown_complexity(n)
```

```
Siftdown Complexity :
Testing for size :  10
411 µs ± 1.65 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
Testing for size :  100
4.41 ms ± 38.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
Testing for size :  1000
44.1 ms ± 131 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
Testing for size :  10000
441 ms ± 5.27 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
Testing for size :  100000
4.33 s ± 31.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
Testing for size :  1000000
39.1 s ± 868 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
Testing for size :  10000000
```

```
Siftdown:

[<matplotlib.lines.Line2D at 0x14a3be5ecd30>]
```



FUTURE WORK

A future work in this investigation may include the study of Dijkstra's algorithm, a major application of the Fibonacci Heap and Priority Queue, used in finding the shortest path. A comparison study of the effectiveness of each of the two data structures in implementing the Dijkstra's algorithm could be done and the complexity of the algorithm may be observed.

CONTRIBUTIONS

Nirali:

- Initial find min (), delete min () functions
- Method tests for delete min (), merge (), merge All (), and find min ()
- Complexity tests and plot functions to time the functions in the Fibonacci heap and Priority Queue
- Complexity tests and plot functions to measure count of operations for both Fibonacci Heap and Priority Queue
- Debugging
- Detecting errors in the code
- Project Report

Tilly:

- Initially, merge (), merge All (), insert (), and decrease priority ()
- Redone find min ()
- Made major changes to delete min () to suit degree dictionary and handle duplicates
- Completely re-worked code multiple times
- Initial tests, and method tests for decrease priority (), duplicates and insert ()
- Adjusted method tests for merge (), merge All () to account for changes in the code.
- Debugging
- Detecting errors in the code
- Project Presentation

REFERENCES

1. https://www.geeksforgeeks.org/fibonacci-heap-set-1-introduction/
2. http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap21.htm
3. https://www.cs.usfca.edu/~galles/visualization/FibonacciHeap.html
4. https://en.wikipedia.org/wiki/Fibonacci_heap
5. https://www.geeksforgeeks.org/fibonacci-heap-set-1-introduction/
6. https://www.cs.princeton.edu/~wayne/teaching/fibonacci-heap.pdf
7. https://brilliant.org/wiki/fibonacci-heap/
8. https://www.geeksforgeeks.org/fibonacci-heap-deletion-extract-min-and-decrease-key/
9. https://www.cs.princeton.edu/~wayne/cs423/lectures/fibonacci-4up.pdf
10. https://www.gatevidyalay.com/tag/degree-of-node-in-tree/
11. https://stackoverflow.com/questions/4527942/comparing-two-dictionaries-and-checking-how-many-key-value-pairs-are-equal
12. https://docs.python.org/2/library/unittest.html