Fibonacci Heaps

https://cs.pomona.edu/classes/cs140/

Outline

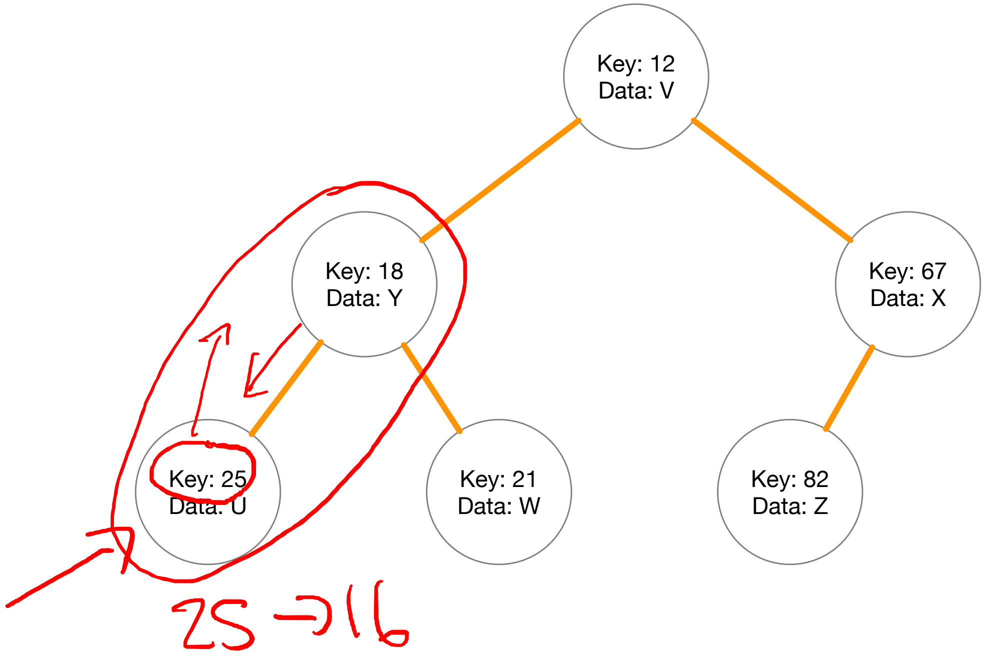
Topics and Learning Objectives

- Discuss Fibonacci Heaps
- Understand the benefits of Fibonacci Heaps
- Analyze the amortized running time of Fibonacci Heaps

Exercise

- Fibonacci Heap practice

Dijkstra's Reminders



During each iteration we need to:

Extract Min

1. Find the vertex v that
   - Is reachable from the start vertex using the vertices found so far
   - Has the minimal path length from the start vertex among all options

Decrease Key

2. Update the possible paths lengths of all vertices connected to v

Binary Heap Priority Queue

- An almost-full binary tree
- Satisfies the heap property

<u>Insert</u>

- Add to the end and bubble up, O(lg n)

<u>Extract-Min</u>

- Replace root with last node and bubble down, O(lg n)

<u>Decrease-Key</u>

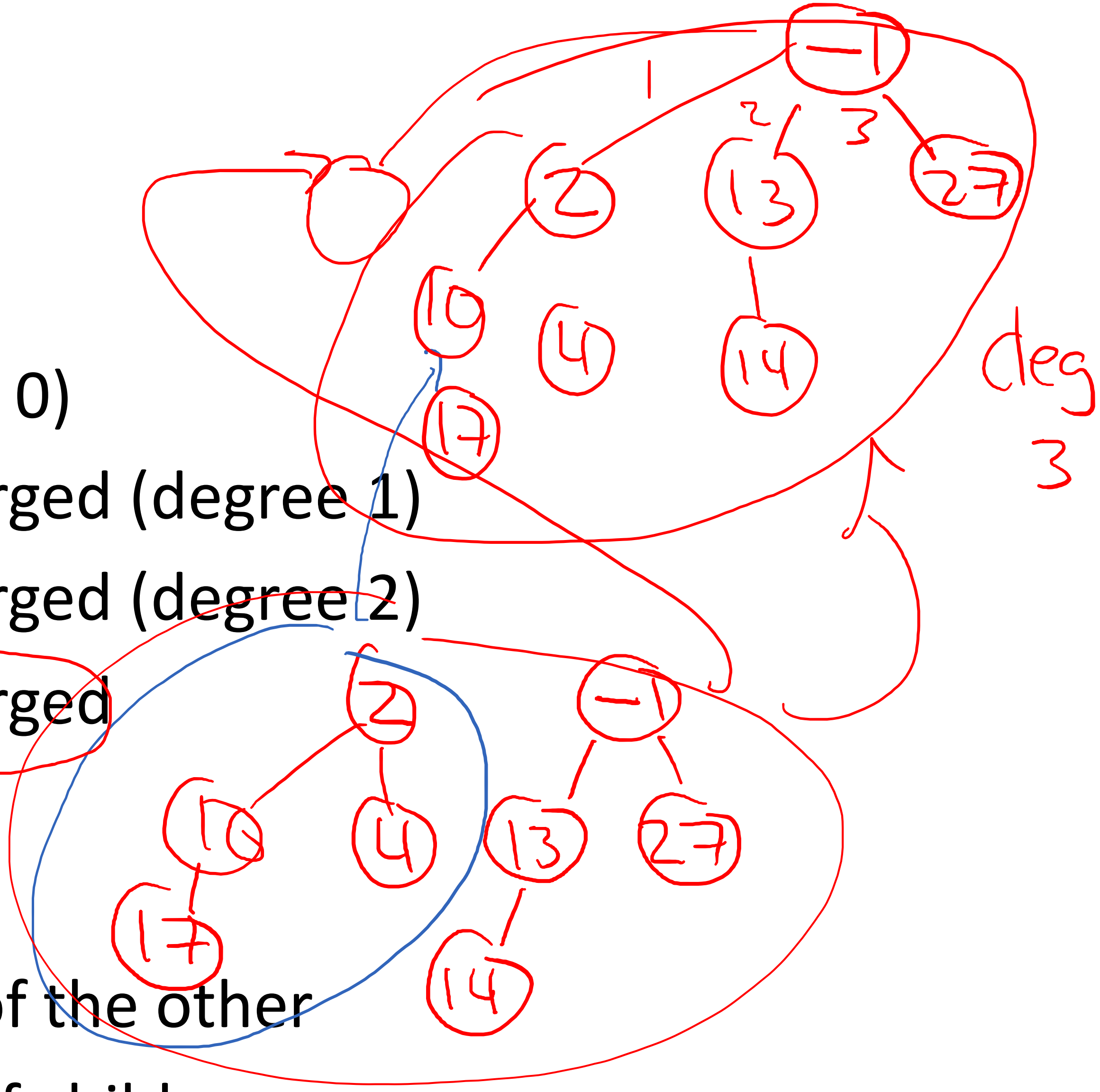
- Change key and bubble up, O(lg n)

Binomial Heap Priority Queue

- Uses a forest of binomial trees

Binomial Trees Can Be…

1. A single node (a tree with degree 0)
2. Two trees of degree 0 can be merged (degree 1)
3. Two trees of degree 1 can be merged (degree 2)
4. Two trees of degree 2 can be merged
5. …



- Merge by making one tree a child of the other
- Degree denotes a node's number of children

Binomial Heap Priority Queue

You don't need to understand the details, we just want to compare with a Fibonacci Heap

- Uses a forest of binomial trees, each satisfies the heap property
- At most one tree of each degree

Insert

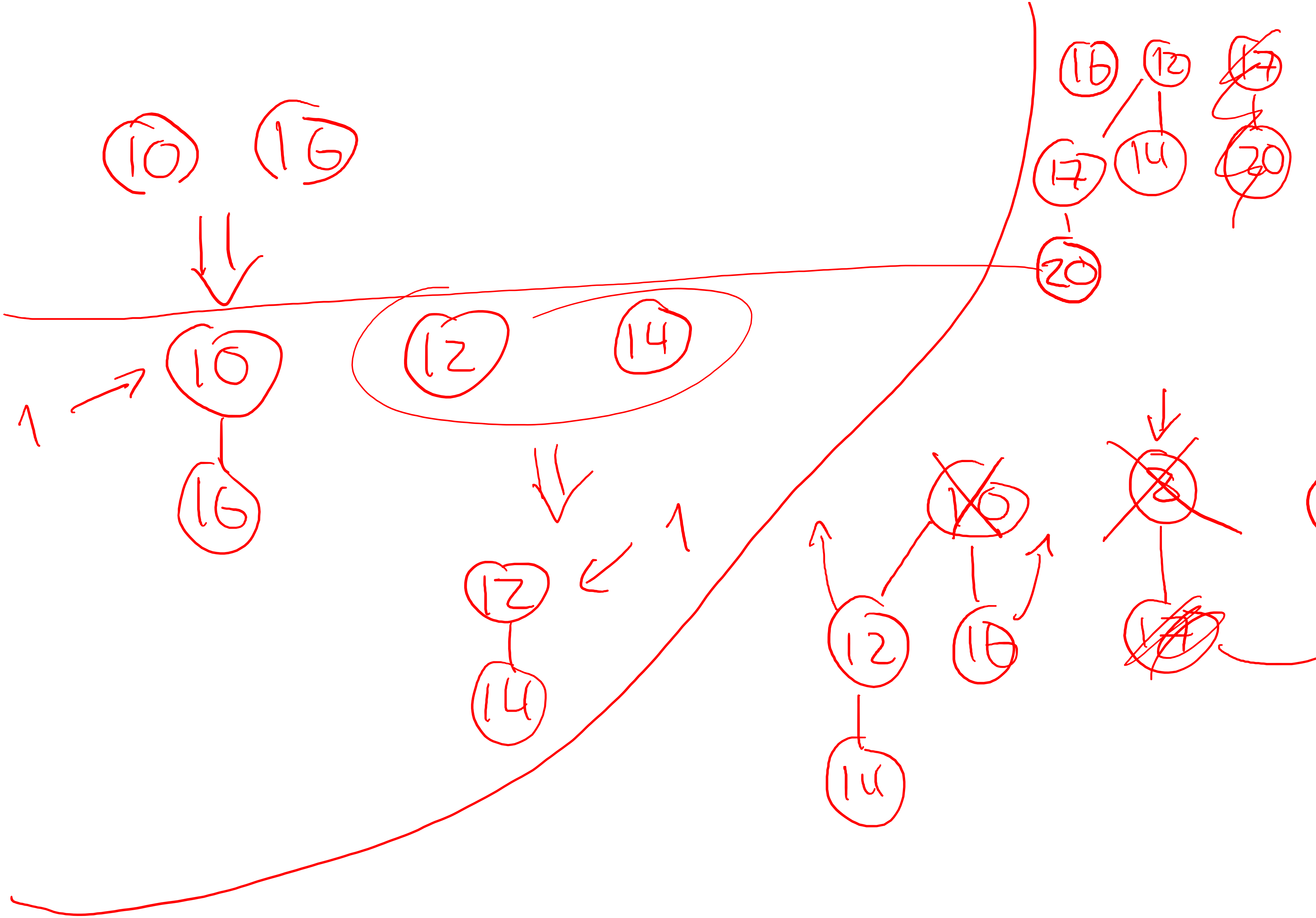
- Create a new, single-node tree and merge as needed, O(1)$_{amortized}$

Extract-Min

- Remove min root, promote its children, and merge as needed, O(lg n)

Decrease-Key

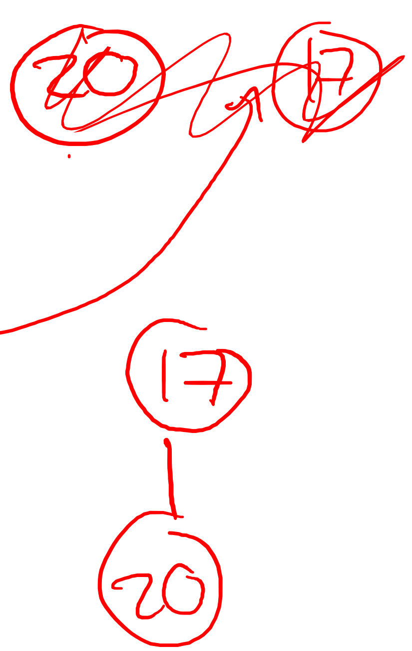
- Change key and bubble up, O(lg n)



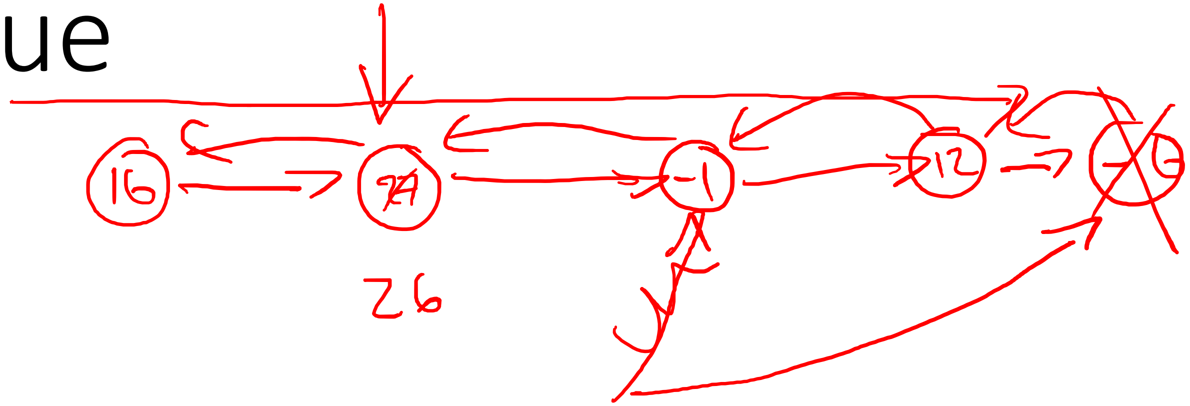
Example Binomial Heap

Operations:

- Insert 10
- Insert 16
- Insert 12
- Insert 14
- Insert 8
- Insert 17
- Insert 20
- Extract-Min
- Extract-Min



Linked List Priority Queue



- A normal, doubly linked-list
- Really, nothing special

<u>Insert</u>

- Add to the end and update min pointer if needed, O(1)

<u>Extract-Min</u>

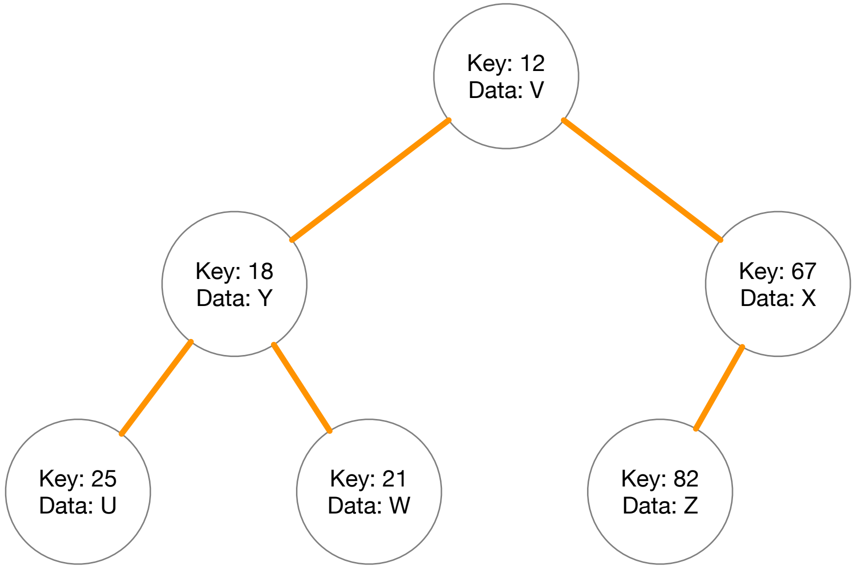
- Remove the min node, then find the new min node, O(n)

<u>Decrease-Key</u>

- Change key and update min pointer if needed, O(1)

Priority Queue Comparison

	Find Min	Extract Min	Insert	Decrease Key
Binary Heap	O(1)	O(lg n)	O(lg n)	O(lg n)
Binomial Heap	O(1)	O(lg n)	O(1) amortized	O(lg n)
Linked List	O(1)	O(n)	O(1)	O(1)
Fibonacci Heap	O(1)	O(lg n) amortized	O(1)	O(1) amortized



Originally created to improve Dijkstra's Single Source Shortest Path Algorithm

O(m + n lg n)

Time to call "Decrease Key" for each edge.

Time to call "Extract Min" on each vertex.

Quick Note on Amortized Analysis

- We skipped this lecture, but we might be able to fit it back in later

- Here's the important part
  - If we perform an operation k times, then

$$\text{Total true cost} = O(\text{Amortized cost})$$

$$\text{Total true cost} \leq c\ (\text{Amortized cost}) \text{ for all } n \geq n_0$$

We might do a lot of work in one call, but this work will benefit later calls

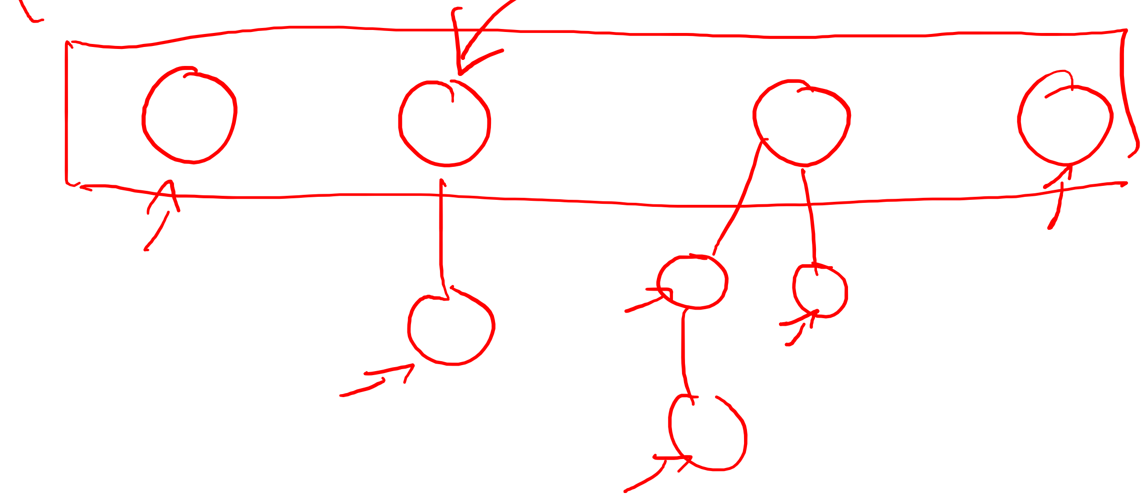
Fibonacci Heap, Basic Idea

- Maintain a set of Heaps (not necessarily binomial trees)
- Maintain a pointer to the minimum element
  - The minimum element will be the root of one of the heaps
- Maintain a set of “marked” nodes
- Lazily add nodes
- Cleanup in batches (more efficient this way)

Fibonacci Heap Details

```
STRUCT HeapNode<T>
    value: T
    key: Comparable
    degree: Integer = 0
    isLoser: Boolean = FALSE
    parent: HeapNode<T> = NONE
    children: List[HeapNode<T>] = []
```

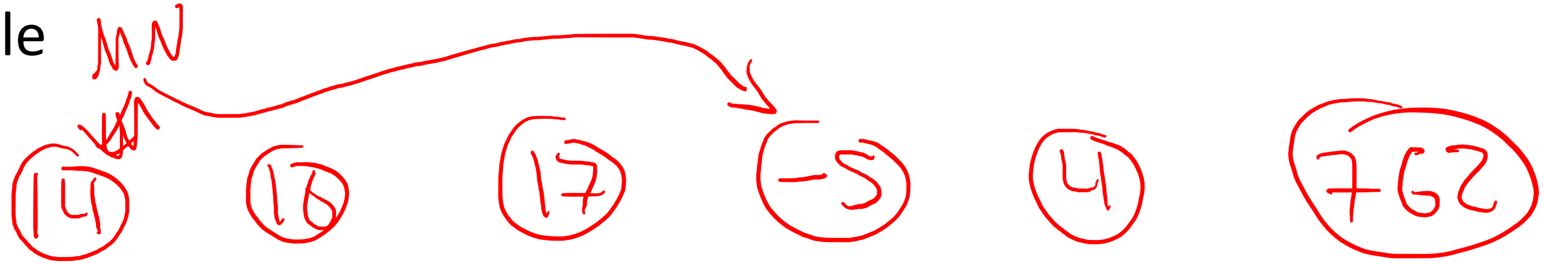
```
STRUCT PQ<T>
    heaps: Set[HeapNode<T>] = []
    minNode: HeapNode<T> = NONE
    lookupTable: Dict[T, HeapNode<T>] = {}
```



```
FUNCTION FibPQInsert(pq, value, key)
    newNode = HeapNode(value, key)
    pq.heaps.append(newNode)
    pq.lookupTable[value] = newNode
    IF newNode.key < pq.minNode.key THEN pq.minNode = newNode
```

Running Time?

Example



```
FUNCTION FibPQExtractMin(pq)
    # Remove the minimum heap node
    …


    # Promote children
    …


    # Continually merge heaps with the same degree
    …


    # Create new list of root heaps
    …


    # Set the new minimum
    …


    RETURN extractedNode.value
```

```
FUNCTION FibPQExtractMin(pq)
    # Remove the minimum heap node
    …

    # Promote children
    …

    # Continually merge heaps with the same degree
    …

    # Create new list of root heaps
    …

    # Set the new minimum
    …

    RETURN extractedNode.value
```

```
extractedNode = pq.minNode
pq.heaps.remove(extractedNode)
```

```
STRUCT PQ<T>
    heaps: Set[HeapNode<T>] = []
    minNode: HeapNode<T> = NONE
    lookupTable: Dict[T, HeapNode<T>] = {}
```

```
FUNCTION FibPQExtractMin(pq)
    # Remove the minimum heap node
    …

    # Promote children
    …

    # Continually merge heaps wit
    …

    # Create new list of root heaps
    …

    # Set the new minimum
    …

    RETURN extractedNode.value
```

```
extractedNode = pq.minNode
pq.heaps.remove(extractedNode)
```

```
FOR child IN minNode.children
    child.isLoser = FALSE
    pq.heaps.append(child)
```

```
STRUCT HeapNode<T>
    value: T
    key: Comparable
    degree: Integer = 0
    isLoser: Boolean = FALSE
    parent: HeapNode<T> = NONE
    children: List[HeapNode<T>] = []
```

```
FUNCTION FibPQExtractMin(pq)
    # Remove the minimum heap node
    …

    # Promote children
    …

    # Continually merge heaps w
    …

    # Create new list of root l
    …

    # Set the new minimum
    …

    RETURN extractedNode.value
```

Same process as for Binomial Heaps

```
# Continually merge heaps with the same degree
heapsByDegree = [NONE FOR _ IN pq.heaps]
FOR heap IN pq.heaps
    currentHeap = heap
    LOOP
        currentDegree = currentHeap.degree
        BREAK IF heapsByDegree[currentDegree] != NONE
        heapWithSameDegree = heapsByDegree[currentDegree]
        heapsByDegree[currentDegree] = NONE
        # Merge two trees
        IF currentHeap.key < heapWithSameDegree.key
            currentHeap.degree += 1
            currentHeap.children.append(heapWithSameDegree)
            heapWithSameDegree.parent = currentHeap
        ELSE
            heapWithSameDegree.degree += 1
            heapWithSameDegree.children.append(currentHeap)
            currentHeap.parent = heapWithSameDegree
    heapsByDegree[currentDegree] = currentHeap
```

```
FUNCTION FibPQExtractMin(pq)
    # Remove the minimum heap node
    …

    # Promote children
    …

    # Continually merge heaps wit
    …

    # Create new list of root heaps
    …

    # Set the new minimum
    …

    RETURN extractedNode.value
```

```
extractedNode = pq.minNode
pq.heaps.remove(extractedNode)
```

```
FOR child IN minNode.children
    child.isLoser = FALSE
    pq.heaps.append(child)
```

```
pq.heaps = [heap FOR heap IN heapsByDegree IF heap != NONE]
```

```
FUNCTION FibPQExtractMin(pq)
    # Remove the minimum heap node
    …

    # Promote children
    …

    # Continually merge heaps wit
    …

    # Create new list of root heaps
    …

    # Set the new minimum
    …

    RETURN extractedNode.value
```

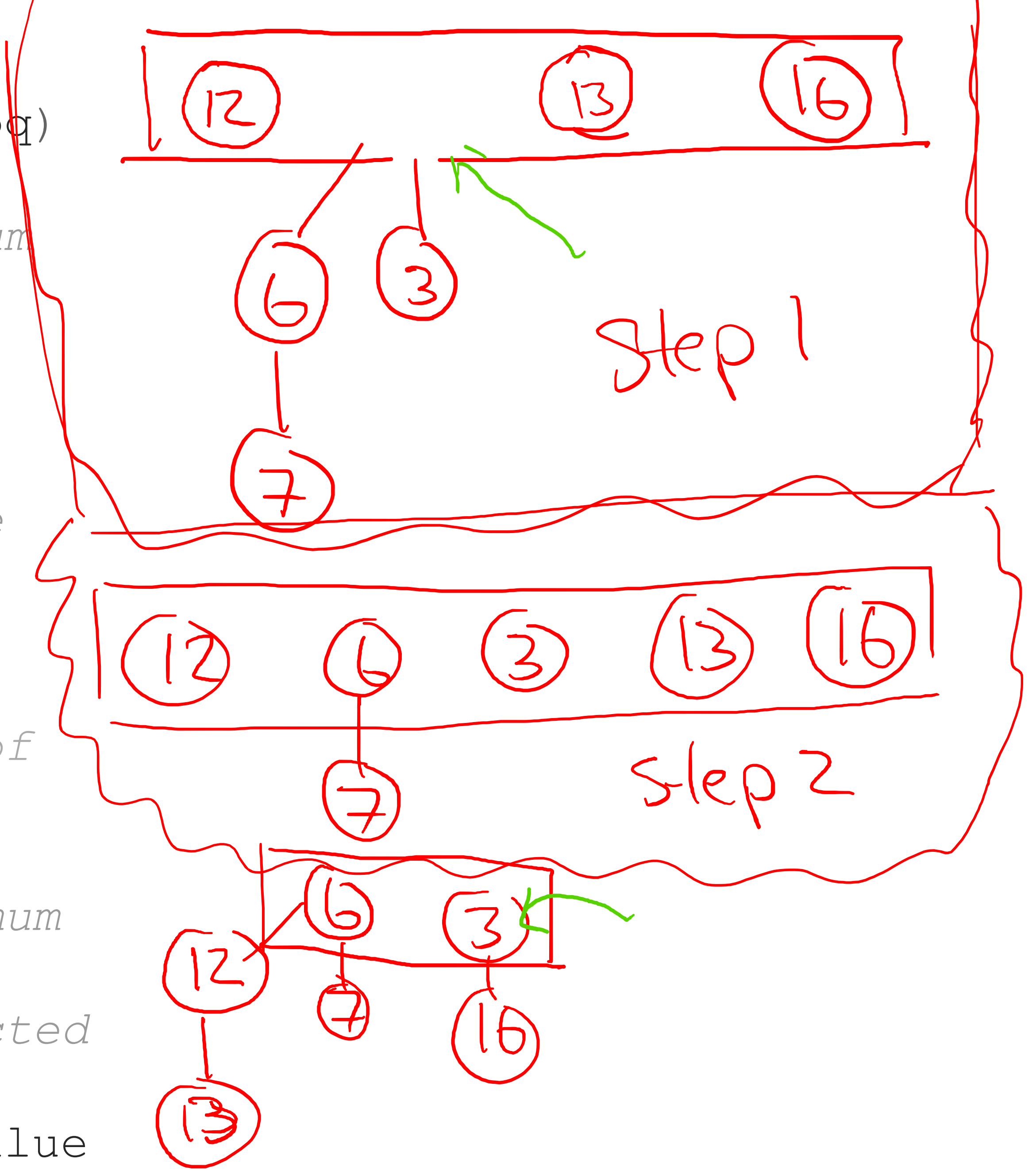
```
extractedNode = pq.minNode
pq.heaps.remove(extractedNode)
```

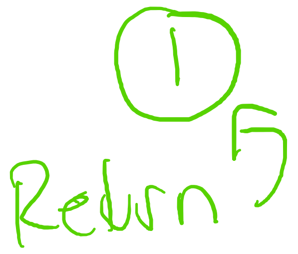
```
FOR child IN minNode.children
    child.isLoser = FALSE
    pq.heaps.append(child)
```

```
pq.heaps = [heap FOR heap IN heapsByDegree IF heap != NONE]
```

```
pq.minNode = pq.heaps[0]
FOR heap IN pq.heaps[1..]
    IF heap.key < pq.minNode.key THEN pq.minNode = heap
```

```
FUNCTION FibPQExtractMin(pq)
    # (1) Remove the minimum
    # heap node

    # (2) Promote children

    # (3) Continually merge
    # heaps with the same
    # degree

    # (4) Create new list of
    # root heaps

    # (5) Set the new minimum

    # (6) Return the extracted
    # node
    RETURN extractedNode.value
```



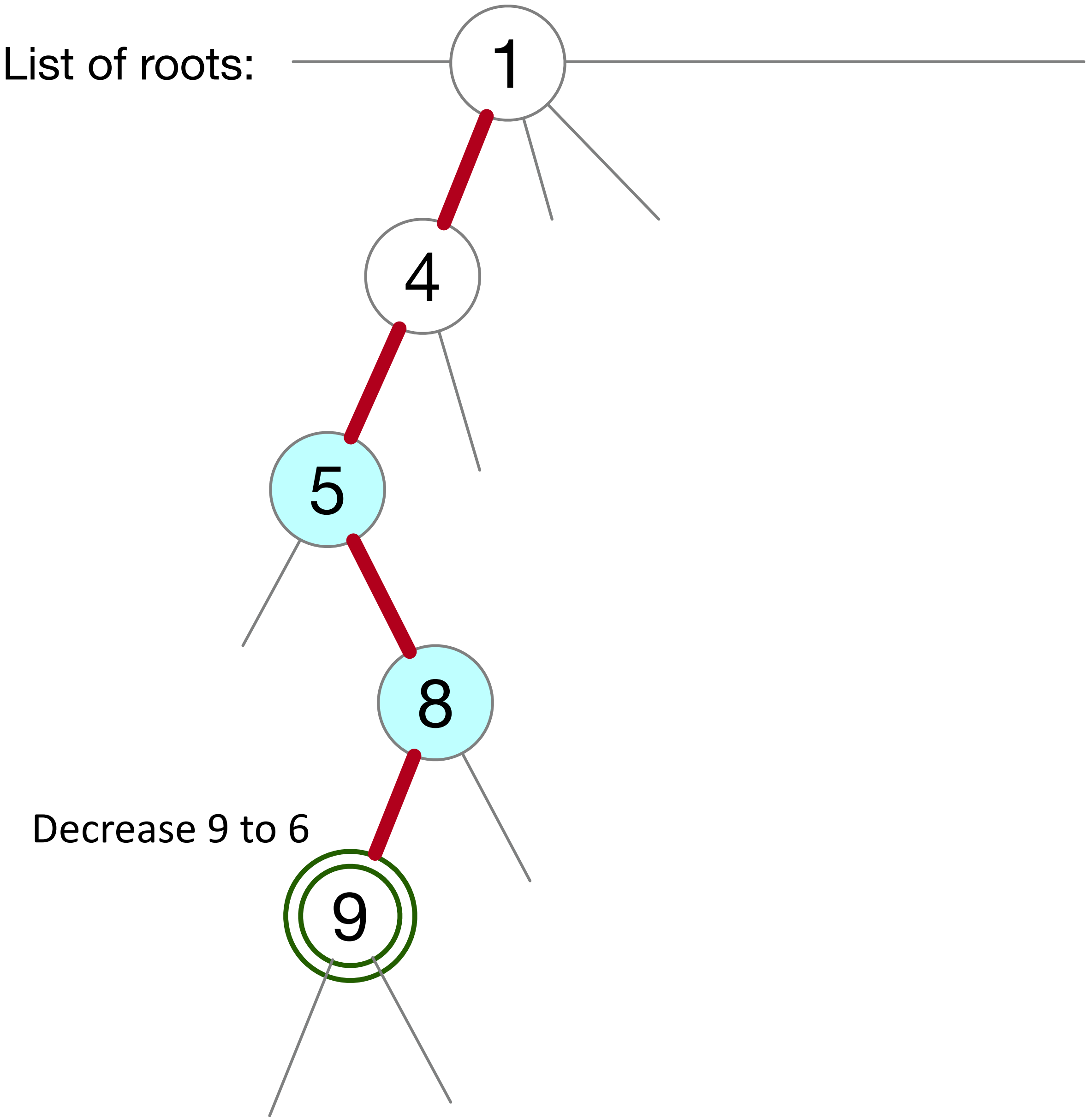


```
FUNCTION FibPQDecreaseKey(pq, value, newKey)
    node = pq.lookupTable[value]
    node.key = newKey
    parent = node.parent

    IF parent != NONE && node.key < parent.key
        LOOP
            parent.children.remove(node)
            pq.heaps.append(node)
            IF node.key < pq.minNode.key THEN pq.minNode = node
            node.isLoser = FALSE
            BREAK IF parent == NONE || parent.isLoser == FALSE
            node = parent
        IF parent != NONE
            parent.isLoser = TRUE
```

Exercise

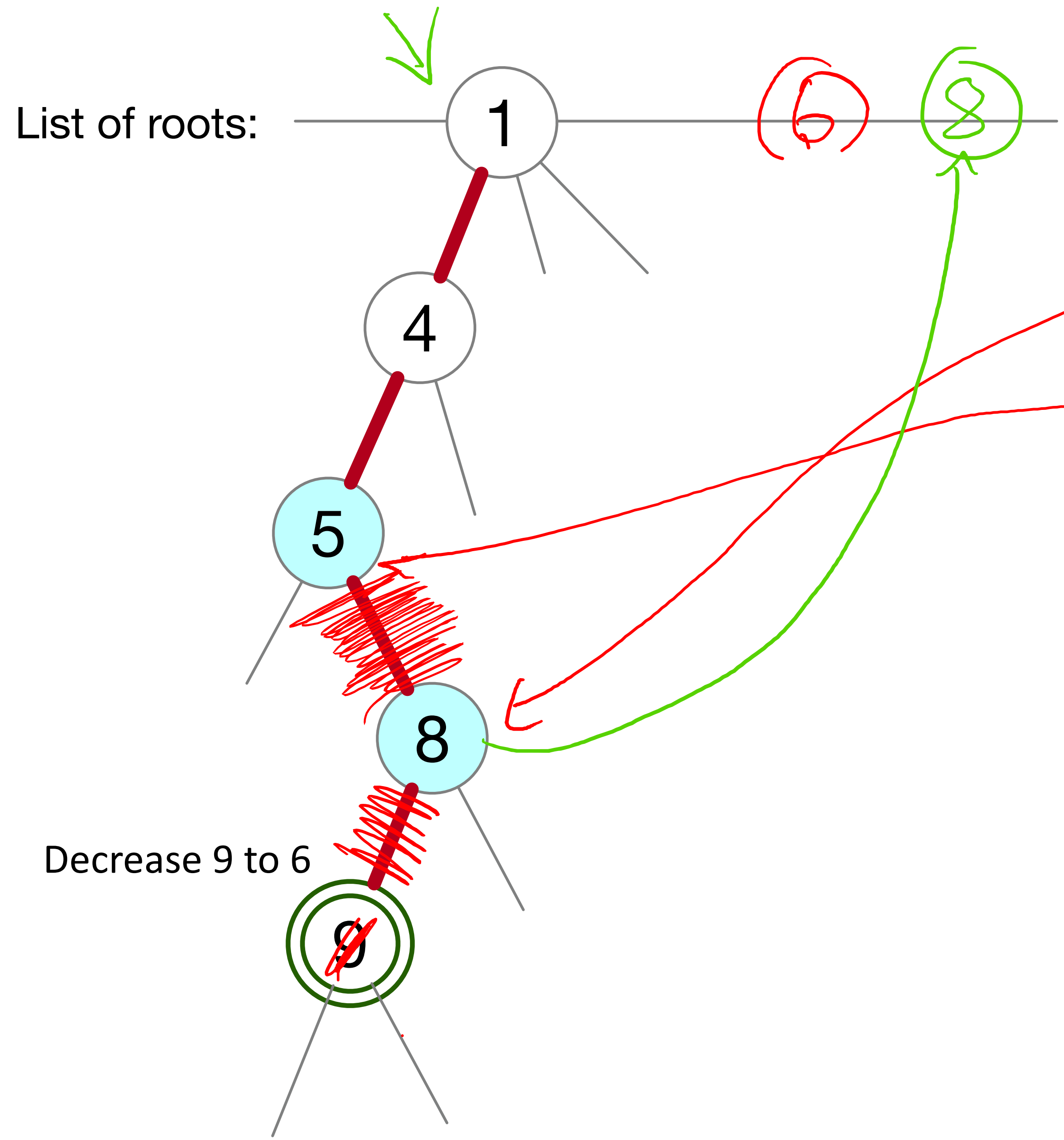
Initial Fibonacci Heap

Final Fibonacci Heap



Example from Damon Wischik at Cambridge University.

Initial Fibonacci Heap



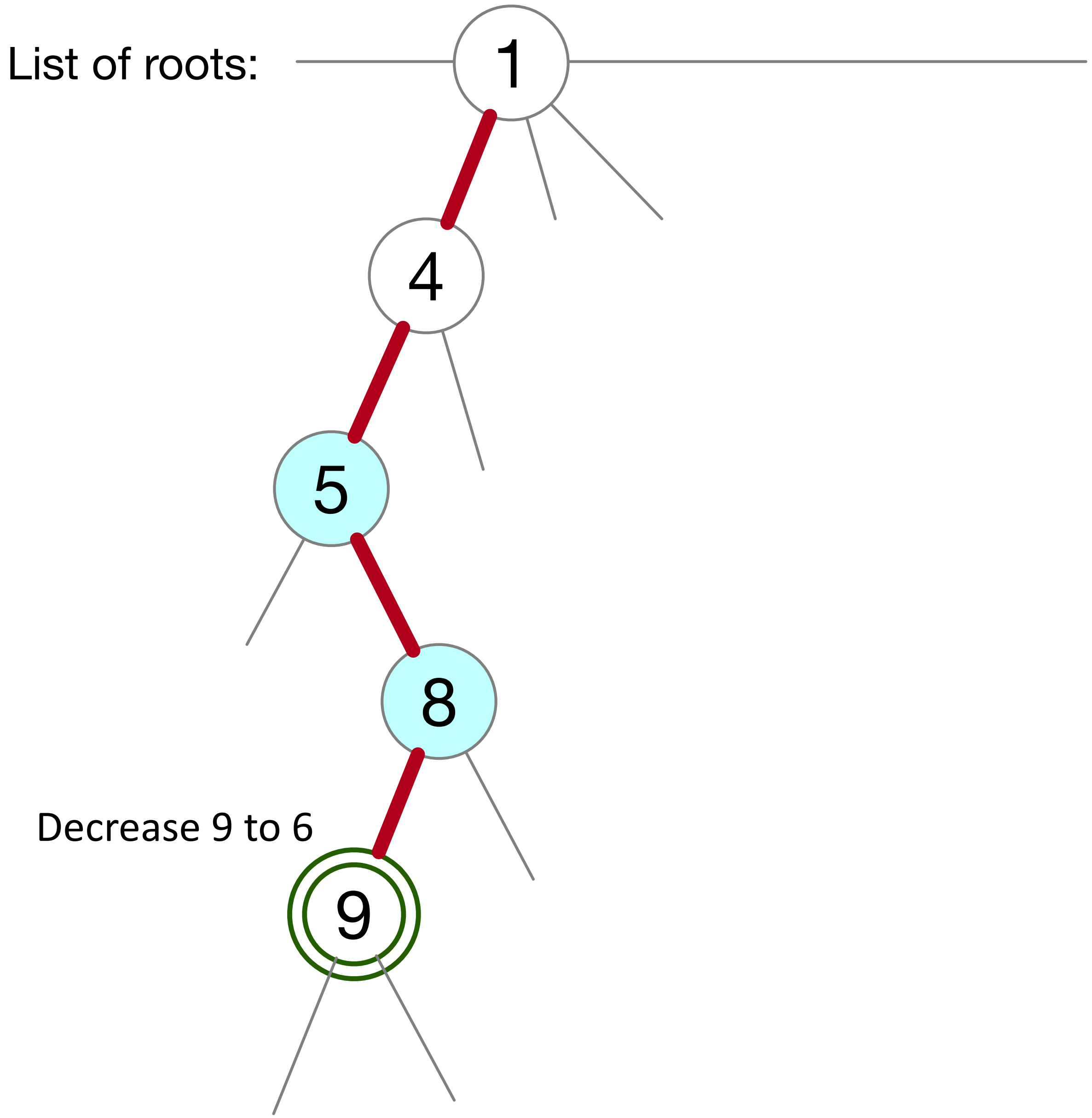
Final Fibonacci Heap

List of roots:

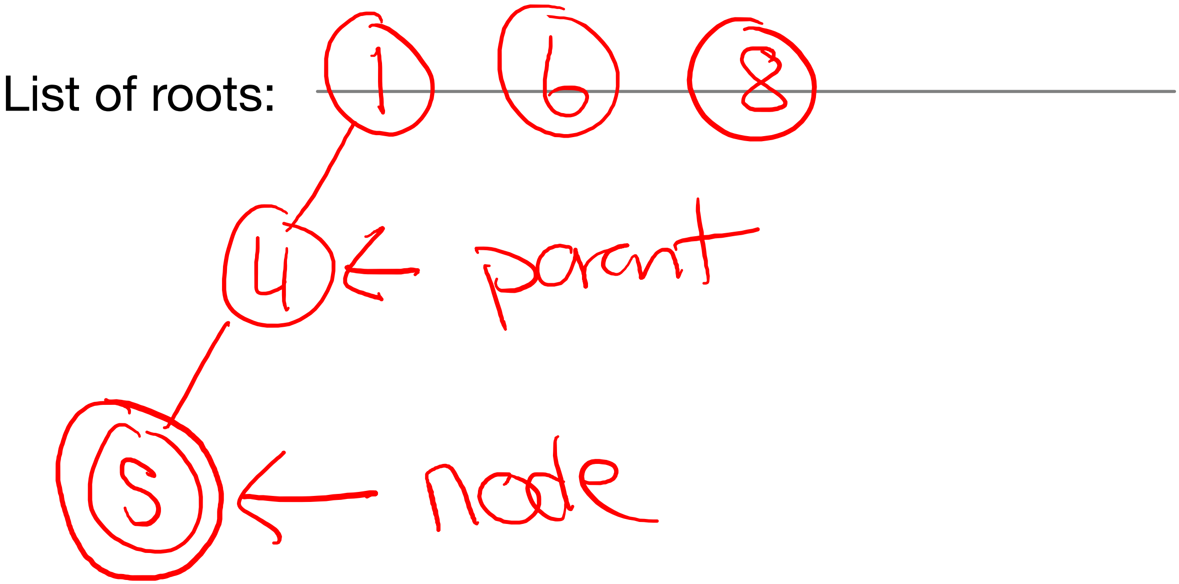
```
FUNCTION FibPQDecreaseKey(pq, value, newKey)
    node = pq.lookupTable[value]
    node.key = newKey
    parent = node.parent

    IF parent != NONE && node.key < parent.key
        LOOP
            parent.children.remove(node)
            pq.heaps.append(node)
            IF node.key < pq.minNode.key THEN pq.minNode = node
            node.isLoser = FALSE
            BREAK IF parent == NONE || parent.isLoser == FALSE
            node = parent
        IF parent != NONE
            parent.isLoser = TRUE
```

Example from Damon Wischik at Cambridge University.

Initial Fibonacci Heap

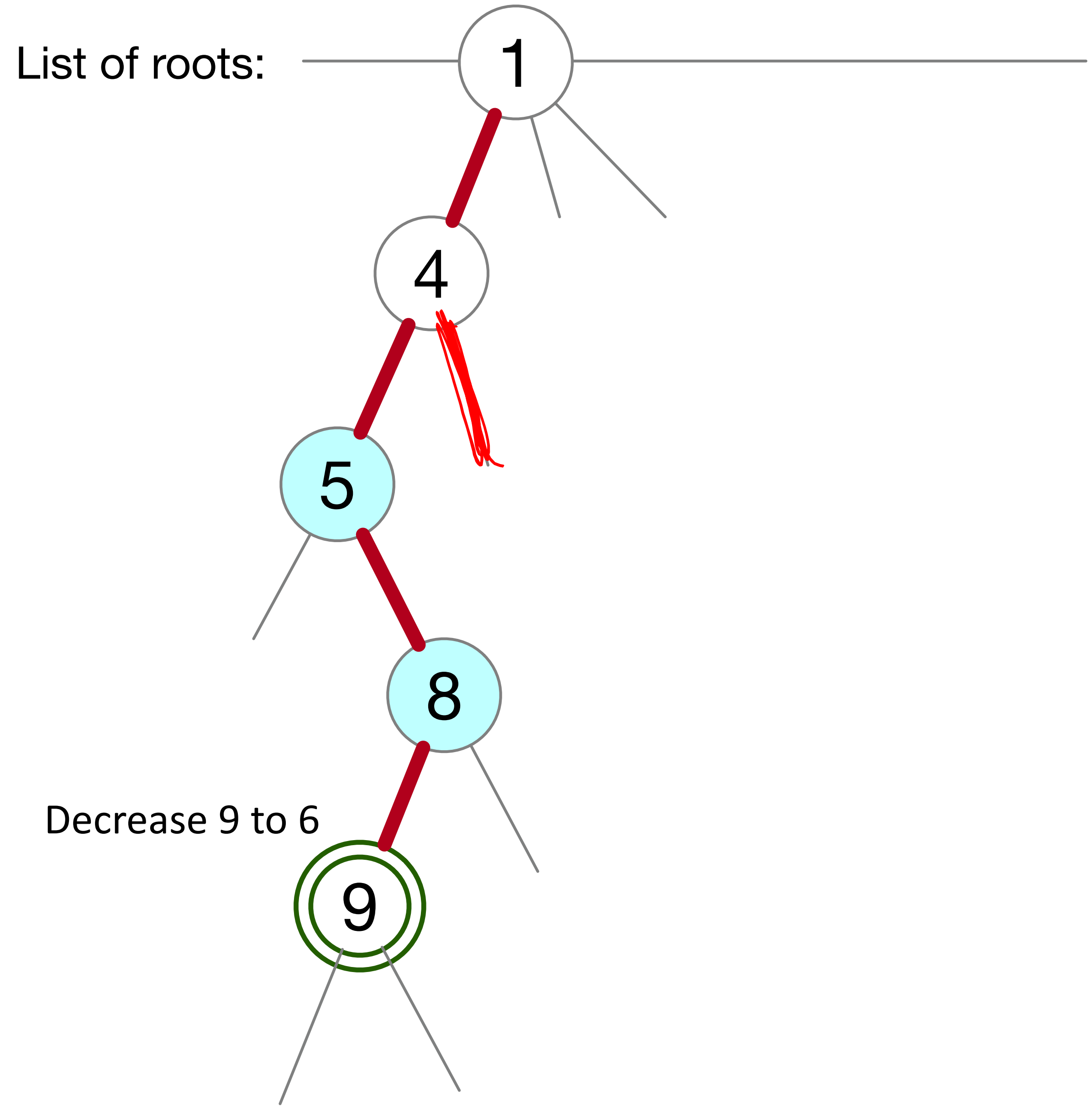


Final Fibonacci Heap

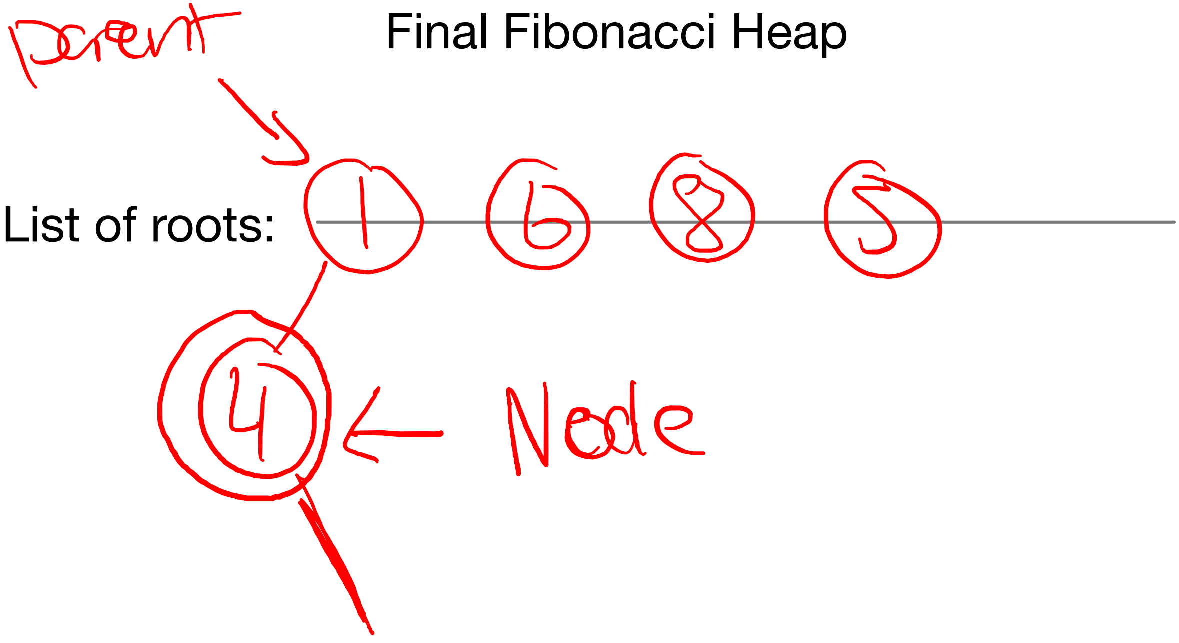


Example from Damon Wischik at Cambridge University.

Initial Fibonacci Heap



Final Fibonacci Heap



Example from Damon Wischik at Cambridge University.

Fibonacci Heaps Insert Running Time

Insert

- All we do is add a single node to the list of heaps and then check to see if it is the new minimum node

Fibonacci Heaps Extract-Min Running Time

Extract-Min

1. Remove the minimum heap node
2. Promote children
3. Continually merge heaps with the same degree
4. Create new list of root heaps
5. Set the new minimum
6. Return the extracted node

Which of these are the easiest?

Fibonacci Heaps Extract-Min Running Time

Extract-Min

1. Remove the minimum heap node, O(1)
2. Promote children
3. Continually merge heaps with the same degree
4. Create new list of root heaps
5. Set the new minimum
6. Return the extracted node, O(1)

Fibonacci Heaps Extract-Min Running Time

Extract-Min

1. Remove the minimum heap node, O(1)
2. Promote children
3. Continually merge heaps with the same degree
4. Create new list of root heaps
5. Set the new minimum
6. Return the extracted node, O(1)

Promote children

- What is the maximum number of children for the minimum?

- It depends on the number of nodes in the Fibonacci heap

- For now, let's call this d_{max}
  - This is the maximum degree of any node
  - Remember that degree denotes the number of direct children of a node
  - We'll figure how an upper bound on d_{max} later

- Promotion then takes $O(d_{max})$

Fibonacci Heaps Extract-Min Running Time

Extract-Min

1. Remove the minimum heap node, O(1)
2. Promote children, $O(d_{max})$
3. Continually merge heaps with the same degree
4. Create new list of root heaps
5. Set the new minimum
6. Return the extracted node, O(1)

Continually merge heaps with the same degree

- With n nodes in the Fibonacci heap, what is the maximum number of merges we can perform?
- O(n)

For example, if we have a bunch of singleton heaps.

- This seems like we will do O(n) work to perform the Extract-Min operation!
- However, **we very rarely perform O(n) merges**
- An amortized analysis tells us that the aggregate cost of this operation is actually O(lg n)

I've provided a video for those interested, but since we skipped this lecture, I am going to skip the analysis here. I have provided some resources on the course website.

Fibonacci Heaps Extract-Min Running Time

Extract-Min

1. Remove the minimum heap node, O(1)
2. Promote children, $O(d_{max})$
3. Continually merge heaps with the same degree, O(lg n)
4. Create new list of root heaps
5. Set the new minimum
6. Return the extracted node, O(1)

```
FUNCTION FibPQExtractMin(pq)
    # Remove the minimum heap node
    …

    # Promote children
    …

    # Continually merge heaps with the same degree
    …

    # Create new list of root heaps
    …

    # Set the new minimum
    …

    RETURN extractedNode.value
```

```
pq.heaps = [heap FOR heap IN heapsByDegree IF heap != NONE]
```

```
pq.minNode = pq.heaps[0]
FOR heap IN pq.heaps[1..]
    IF heap.key < pq.minNode.key THEN pq.minNode = heap
```

Fibonacci Heaps Extract-Min Running Time

Extract-Min

1. Remove the minimum heap node, O(1)
2. Promote children, $O(d_{max})$
3. Continually merge heaps with the same degree, $O(\lg n)_{amortized}$
4. Create new list of root heaps, $O(d_{max})$
5. Set the new minimum, $O(d_{max})$
6. Return the extracted node, O(1)

We'll come back to d_{max} in a bit!

Fibonacci Heaps Decrease-Key Running Time

Decrease-Key

1. Change key in constant time
2. Two cases
    1. If there is no heap violation, then we are done
    2. If there is a heap violation, then we recursively
        1. Promote the node
        2. Check if the parent is a double loser
            1. If the parent is not a loser, then we mark it as a loser and we are done
            2. Otherwise, we continue to "promote the node" with parent as the current node

Fibonacci Heaps Decrease-Key Running Time

Decrease-Key

What is the running time of this path?

1. Change key in constant time
2. Two cases
   1. If there is no heap violation, then we are done
   2. If there is a heap violation, then we recursively
      1. Promote the node
      2. Check if the parent is a double loser
         1. If the parent is not a loser, then we mark it as a loser and we are done
         2. Otherwise, we continue to “promote the node” with parent as the current node

Fibonacci Heaps Decrease-Key Running Time

Decrease-Key

1. Change key in constant time
2. Two cases
    1. If there is no heap violation, then we are done
    2. If there is a heap violation, then we recursively
        1. Promote the node
        2. Check if the parent is a double loser
            1. If the parent is not a loser, then we mark it as a loser and we are done
            2. Otherwise, we continue to "promote the node" with parent as the current node

What is the running time of this path?

It appears to be O(lg n)

An amortized analysis will give us a running time of $O(1)_{amortized}$

Losers, d_{max} , and Naming Rights

- We only merge trees with the same degree

- Looking at a single tree with degree d, you'll see that
  - The leftmost child has degree d-1
  - The second from the left has degree d-2
  - The third from the left has degree d-3
  - And so on
  - The rightmost child has degree 0

- If a node loses one child, then we have the same basic structure
- If a node loses two children, then it is kicked out of the tree

Losers, d_{max} , and Naming Rights

Summary

- Fibonacci Heaps are based on the idea of lazy cleanup
- We don't fix the binomial trees until we can fix a bunch at the same time
- We need amortized analysis to show a more useful running time (instead of a worst-case running time)

	Find Min	**Extract Min**	**Insert**	**Decrease Key**
Binary Heap	O(1)	O(lg n)	O(lg n)	O(lg n)
Binomial Heap	O(1)	O(lg n)	O(1) amortized	O(lg n)
Linked List	O(1)	O(n)	O(1)	O(1)
Fibonacci Heap	O(1)	O(lg n) amortized	O(1)	O(1) amortized