Fibonacci Heaps

Advanced Algorithms and Data Structures - Lecture 8

Venanzio Capretta

Monday 18 November 2019

School of Computer Science, University of Nottingham

Complexity of Heap Operations

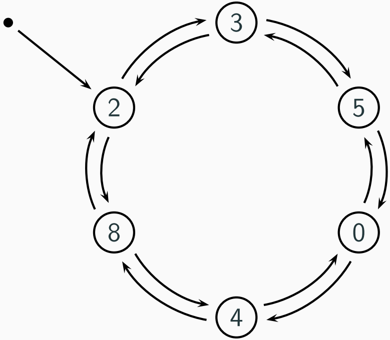
Summary of the complexity of basic heap operation
for several kinds of heaps:
(Fibonacci Heap extraction is amortized complexity)

	`insert`	`minimum`	`extract`	`union`
`Binary`	$O(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$
`Leftist`	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$
`Binomial`	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
`Fibonacci`	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$

Amortized Complexityi s a method of analyzing the running time
that takes into account a whole sequence of operations:
Some operation have a long running time, some have a short running time
Amortized complexity refers to the average time of the whole sequence

Wheels

As a first step towards the definition of Fibonacci Heaps
we study doubly-linked circular lists (wheels)

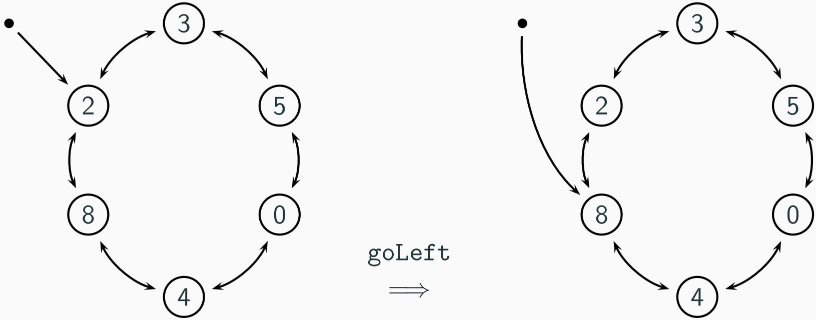


(We use the notation $\llbracket 2, 3, 5, 0, 4, 8 \rrbracket$)

A sequence of values linked in a circle with a head pointer
to one value (2 in the example)
and operations to move the pointer, insert and delete elements

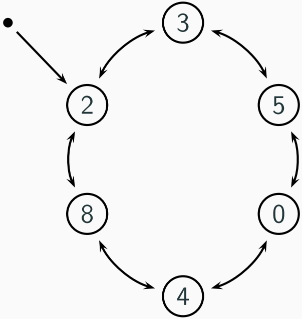
Wheel Operations: Move

`goRight` and `goLeft`
move the head pointer clockwise (to 3) or anti-clockwise (to 8):



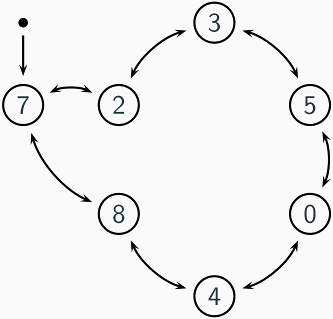
Wheel Operations: Insert

`insert` a new element just before the pointer:



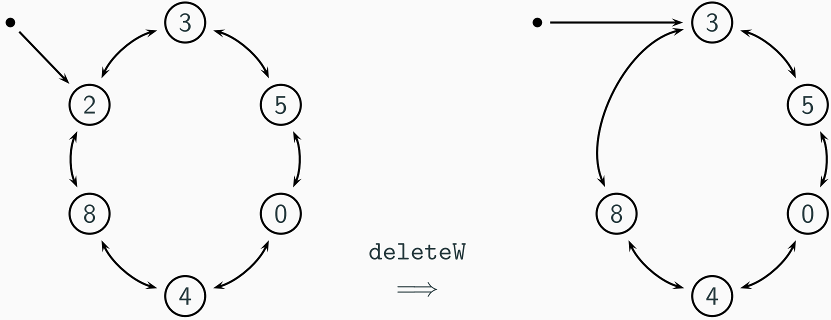
`insertW 7`

$\Longrightarrow$



Wheel Operations: Delete

`delete` the element pointed at:
(and move the pointer clockwise)



Implementation of Wheels

In imperative programming, you can implement wheels using pointers going back and forth between every pair of elements.
The complexity of each operation is $\Theta(1)$.

In functional programming, you can implement wheels by a pair of lists, similarly to what we have done for FIFO queues.
All operations can be programmed with amortized complexity $\Theta(1)$.

Hint: Since we can move both left and right, the *best* state of the structure is when both lists have the same length. This should be the state with the highest potential.

Fibonacci Heaps

A Fibonacci Heap is a kind of fractal wheel:

Fibonacci Heaps

A Fibonacci Heap is a kind of fractal wheel:
It's a wheel of values in which every element is in turn connected with another Fibonacci heap

The structure is similar to a Binary or Leftist Heap in which each node is replaced by a wheel

Fibonacci Heaps

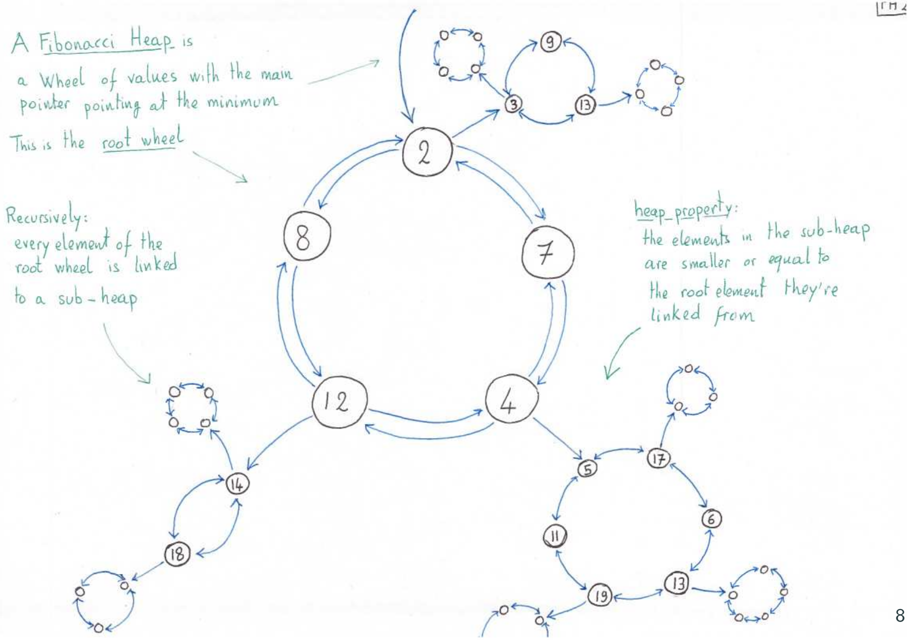
A Fibonacci Heap is a kind of fractal wheel:
It's a wheel of values in which every element is in turn connected with another Fibonacci heap

The structure is similar to a Binary or Leftist Heap in which each node is replaced by a wheel

Formally a Fibonacci Heap has:

- A root wheel of values, with the main pointer pointing to the smallest of them
  The values don't need to be ordered
- Each element of the root wheel is in turn connected to a sub-heap
  (The sub-heap could be empty)
- Heap Property: The elements of the sub-heap are larger or equal to the root element they're linked from

A Fibonacci Heap



Consolidation

With this structure, we can implement heap operations with very low amortized complexity

Consolidation

With this structure, we can implement heap operations with very low amortized complexity

Idea: We perform some operations (insertion and union) lazily, without worrying about the structure of the heap

Consolidation

With this structure, we can implement heap operations with very low amortized complexity

Idea: We perform some operations (insertion and union) lazily, without worrying about the structure of the heap

When we delete an element,
we must traverse the wheel to find the new minimum
We take advantage of this traversal to "consolidate" the heap

Consolidation

With this structure, we can implement heap operations with very low amortized complexity

Idea: We perform some operations (insertion and union) lazily, without worrying about the structure of the heap

When we delete an element,
we must traverse the wheel to find the new minimum
We take advantage of this traversal to "consolidate" the heap

The degree of an element is
the length of the main wheel of the sub-heap it is linked to

A heap is consolidated if all its root elements have different degrees

Consolidation

With this structure, we can implement heap operations with very low amortized complexity

Idea: We perform some operations (insertion and union) lazily, without worrying about the structure of the heap

When we delete an element,
we must traverse the wheel to find the new minimum
We take advantage of this traversal to "consolidate" the heap

The degree of an element is
the length of the main wheel of the sub-heap it is linked to

A heap is consolidated if all its root elements have different degrees

Q: If the heap and all the sub-heaps are consolidated,
What is the minimum number of elements the heap
as a function of the length of the root wheel?

Definition of the Data Structure

We formally implement the data structure

Each node will contain a value, the node degree, and the sub-heap

```
data FibHeap = FHeap (Wheel (Key, Int, FibHeap))
```

Definition of the Data Structure

We formally implement the data structure

Each node will contain a value, the node degree, and the sub-heap

```
data FibHeap = FHeap (Wheel (Key, Int, FibHeap))
```

The example in the previous page is written:

$$\texttt{FHeap} \Lbag (2, 3, \texttt{h}_1), (7, 0, \texttt{emptyW}), (4, 6, \texttt{h}_2), (12, 2, \texttt{h}_3), (8, 0, \texttt{emptyW}) \Rbag$$

Definition of the Data Structure

We formally implement the data structure

Each node will contain a value, the node degree, and the sub-heap

```
data FibHeap = FHeap (Wheel (Key, Int, FibHeap))
```

The example in the previous page is written:

$$\mathtt{FHeap} \Lbag (2, 3, \mathtt{h}_1), (7, 0, \mathtt{emptyW}), (4, 6, \mathtt{h}_2), (12, 2, \mathtt{h}_3), (8, 0, \mathtt{emptyW}) \Rbag$$

The head element is 2, it has degree 3, because its sub-heap h_1 has three elements in its root wheel

Definition of the Data Structure

We formally implement the data structure

Each node will contain a value, the node degree, and the sub-heap

```
data FibHeap = FHeap (Wheel (Key, Int, FibHeap))
```

The example in the previous page is written:

$$\mathtt{FHeap}\ \{\!\!\!\circ\ (2, 3, \mathtt{h}_1), (7, 0, \mathtt{emptyW}), (4, 6, \mathtt{h}_2), (12, 2, \mathtt{h}_3), (8, 0, \mathtt{emptyW})\ \circ\!\!\!\}$$

The head element is 2, it has degree 3, because its sub-heap h_1 has three elements in its root wheel

The element 7 has degree 0 because its sub-heap is empty

(This is not a consolidated heap: Two nodes with the same degree 0)

Heap Operations

- The empty heap simply has an empty wheel:

```
emptyH = FHeap [| |]
```

Heap Operations

- The empty heap simply has an empty wheel:

```
emptyH = FHeap [o o]
```

- The minimum is just the head of the wheel:

```
minimum (FHeap w) = first (headW w)
```

Heap Operations

- The empty heap simply has an empty wheel:

```
emptyH = FHeap ⦃ ⦄
```

- The minimum is just the head of the wheel:

```
minimum (FHeap w) = first (headW w)
```

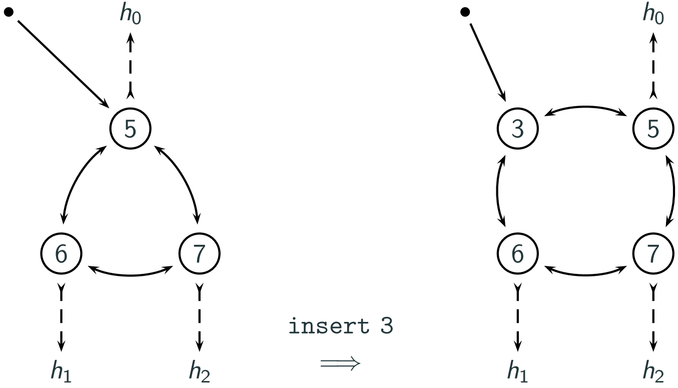
- Insertion adds the new element to the wheel with empty sub-heap, but must move the head right if the inserted element is bigger than the previous head:

```
insertH x h@(FHeap w) =
  if (isEmptyW w)
  then FHeap ⦃(x, 0, emptyH)⦄
  else if x ≤ minimum h
       then FHeap (insertW (x,0,emptyH) w)
       else FHeap (goRight (insertW (x,0,emptyH) w))
```

Insertion Example 1

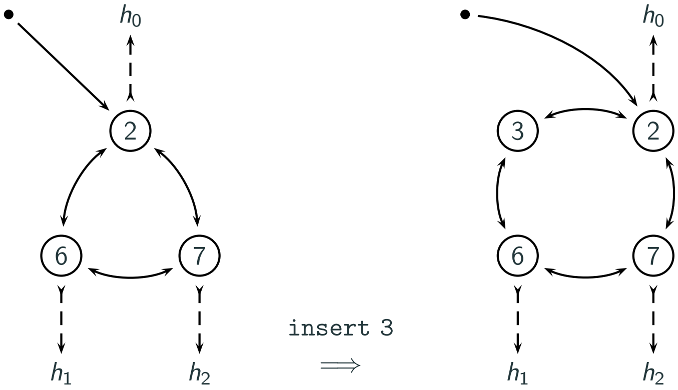
For example, if we insert 3 into a heap with minimum 5

3 becomes the new minimum:



Insertion Example 2

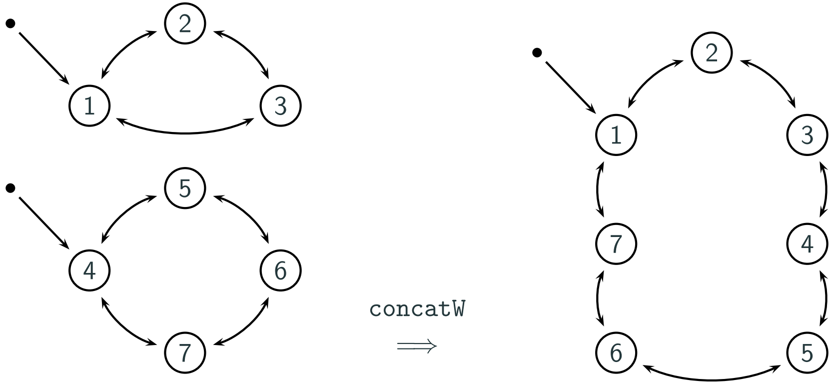
But if we insert 3 into a heap with minimum 2
2 remains the minimum:



Union 1

Union of two heaps is done by simply concatenating the corresponding wheels
We must make sure that the head pointer points to the minimum of the heads of the two heaps

Exercise: Implement the concatenation of two wheels:



Union 2

```
union h1@(FHeap w1) h2@(FHeap w2) =
  if isEmpty h1 then h2 else
  if isEmpty h2 then h1 else
  if minimum h1 ≤ minimum h2
    then FHeap (concatW w1 w2)
    else FHeap (concatW w2 w1)
```

We compare the minimums of the two heaps
and we concatenate the wheels so that
the smaller one becomes the new minimum

Union 2

```
union h1@(FHeap w1) h2@(FHeap w2) =
  if isEmpty h1 then h2 else
  if isEmpty h2 then h1 else
  if minimum h1 ≤ minimum h2
    then FHeap (concatW w1 w2)
    else FHeap (concatW w2 w1)
```

We compare the minimums of the two heaps
and we concatenate the wheels so that
the smaller one becomes the new minimum

We implemented most heap operations
in a naive way, without worrying about the structure of the heap

The only operation that rearranges the heap is extraction

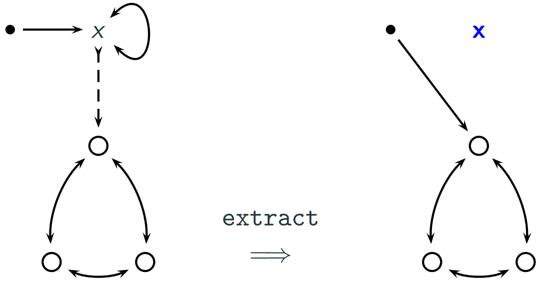
Extraction 1

When we extract the minimum from a heap:

Extraction 1

When we extract the minimum from a heap:

- If it was the only element of the wheel, the new heap is its sub-heap



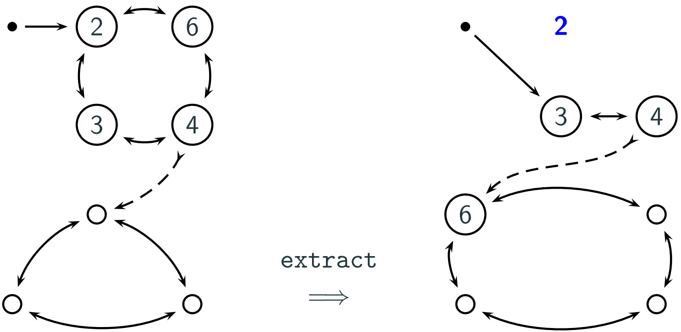
Extraction 2

- If there are other elements in the root wheel, we must
  - Concatenate the sub-heap of the extracted element with the remaining root wheel
  - Traverse the root wheel to find the new minimum
  - Take advantage of this traversal to restructure (consolidate) the heap

Extraction 2

- If there are other elements in the root wheel, we must
  - Concatenate the sub-heap of the extracted element with the remaining root wheel
  - Traverse the root wheel to find the new minimum
  - Take advantage of this traversal to restructure (consolidate) the heap

It can happen that an element that was originally on the root wheel is moved into some of the sub-heaps:



Extraction 3

So extraction removes the minimum, concatenates its sub-heap with the root wheel, and then consolidates:

```
extract :: FibHeap → (Key,FibHeap)
extract (FHeap w) =
  let ((x,FHeap wx), w’) = extractW w
  in  (x, consolidate (FHeap (concatenateW wx w’)))
```

(The code is sketchy, to make it work you must add a couple of details)

Consolidation consists in reorganizing the structure of the heap while at the same time finding the new minimum.

Consolidation 1

We use an array A in which we place
the nodes/sub-heaps from the root wheel

$A[d]$ will contain either nothing or a single node/sub-heap with degree d

Consolidation 1

We use an array A in which we place
the nodes/sub-heaps from the root wheel

$A[d]$ will contain either nothing or a single node/sub-heap with degree d

If we try to add a new node/heap to $A[d]$, but the place is already taken we link the two nodes/heaps into a single one with degree $d+1$ and try to put it in $A[d+1]$ (if that place is free)

Consolidation 1

We use an array A in which we place
the nodes/sub-heaps from the root wheel

$A[d]$ will contain either nothing or a single node/sub-heap with degree d

If we try to add a new node/heap to $A[d]$, but the place is already taken we link the two nodes/heaps into a single one with degree $d + 1$ and try to put it in $A[d + 1]$ (if that place is free)

Define a type of nodes/sub-heap that explicitly contains the degree:

```
type Node = (Key,Int,FibHeap)
```

Note: then a Fibonacci Heap can be defined just as a wheel of nodes:

```
data FibHeap = FHeap (Wheel Node)
```

Consolidation 2

Linking two nodes:

insert the larger one as child of the smaller

```
link :: Node → Node → Node
link x@(kx,dx,hx) y@(ky,dy,hy) =
  if kx ≤ ky
  then (kx, dx+1, FHeap (insertN y hx))
  else (ky, dx+1, FHeap (insertN x hy))
```

(`insertN` should insert the node with its subheap,
making sure to still point at the minimum)

Consolidation 2

Linking two nodes:

insert the larger one as child of the smaller

```
link :: Node → Node → Node
link x@(kx,dx,hx) y@(ky,dy,hy) =
  if kx ≤ ky
  then (kx, dx+1, FHeap (insertN y hx))
  else (ky, dx+1, FHeap (insertN x hy))
```

(`insertN` should insert the node with its subheap,
making sure to still point at the minimum)

Now we use an array A of nodes

(In functional programming we can use Finite Maps)

Let us call `NArray` its type

Consolidation 2

Linking two nodes:

insert the larger one as child of the smaller

```
link :: Node → Node → Node
link x@(kx,dx,hx) y@(ky,dy,hy) =
  if kx ≤ ky
  then (kx, dx+1, FHeap (insertN y hx))
  else (ky, dx+1, FHeap (insertN x hy))
```

(`insertN` should insert the node with its subheap,
making sure to still point at the minimum)

Now we use an array A of nodes

(In functional programming we can use Finite Maps)

Let us call `NArray` its type

$A[d]$ is either empty or contains a node of degree d

Let us denote by $A[d \mapsto x]$

the array A where the entry $A[d]$ has been changed to x

Consolidation 3

Inserting a new node into the array will require checking if its degree is already taken:

```
insNA :: Node → NArray → NArray
insNA x@(kx,dx,hx) A =
  if A[dx] is undefined
    then A[dx ↦ x]
    else insNA (link x A[dx]) A
```

Note that if the degree `dx` in `A` is already occupied
We link `x` with the occupier `A[dx]` before inserting
We know this generates a new node of degree `dx+1`

Consolidation 3

We now transform a wheel of nodes into an array
by extracting and inserting them one by one

```
makeNA :: (Wheel Node) → NArray
makeNA w =
  if (isEmpty w)
    then emptyArray
    else let (x,w') = extractW w
         in  insNA x (makeNA w')
```

Consolidation 4

Once we have an array of nodes, stored by degree
we put them back together into a wheel

```
wheelNA :: NArray → (Wheel Node)
```

This works by starting from and empty wheel
and adding the elements from the array one by one
inserting them into the wheel with the following function
(Details depends on implementation,
with Haskell's finite maps we can use `foldr`)

Consolidation 4

Once we have an array of nodes, stored by degree
we put them back together into a wheel

```
wheelNA :: NArray → (Wheel Node)
```

This works by starting from and empty wheel
and adding the elements from the array one by one
inserting them into the wheel with the following function
(Details depends on implementation,
with Haskell's finite maps we can use `foldr`)

```
insNode x w =
  if (isEmpty w) or (x ≤ head w)
    then (insertW x w)
    else (goRight (insertW x w))
```

The last line guarantees that we are still pointing at the minimum

Consolidation 5

Finally we can put all the steps together:

```
consolidate :: FibHeap → FibHeap
consolidate (FHeap w) = wheelNA (makeNA w)
```

Consolidation 5

Finally we can put all the steps together:

```
consolidate :: FibHeap → FibHeap
consolidate (FHeap w) = wheelNA (makeNA w)
```

Complexity

All operations except `extract` are trivially $\Theta(1)$

We can show that `extract` runs in $O(\log n)$ amortized time
This depends on the relation between:

- the number elements of the heap
- the length of the root wheel

in a consolidated heap (see earlier exercise)

See IA for the definition of the potential function and the proof