Analysis of Algorithms

Fibonacci Heaps

Andres Mendez-Vazquez

October 29, 2015

Outline

Outline

Some previous definitions

Free Tree

A free tree is a connected acyclic undirected graph.

Rooted Tree

A rooted tree is a free tree in which one of the nodes is a root.

Ordered Tree

An ordered tree is a rooted tree where the children are ordered.

Some previous definitions

Free Tree

A free tree is a connected acyclic undirected graph.

Rooted Tree

A rooted tree is a free tree in which one of the nodes is a root.

Ordered Tree

An ordered tree is a rooted tree where the children are ordered.

Some previous definitions

Free Tree

A free tree is a connected acyclic undirected graph.

Rooted Tree

A rooted tree is a free tree in which one of the nodes is a root.

Ordered Tree

An ordered tree is a rooted tree where the children are ordered.

Outline

Ordered Tree

Definition

An ordered tree is an oriented tree in which the children of a node are somehow "ordered."

Example

Thus

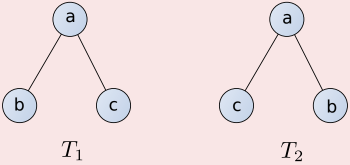
If T_1 and T_2 are ordered trees then $T_1 \neq T_2$ else $T_1 = T_2$.

Ordered Tree

Definition

An ordered tree is an oriented tree in which the children of a node are somehow "ordered."

Example



T_1 T_2

Thus

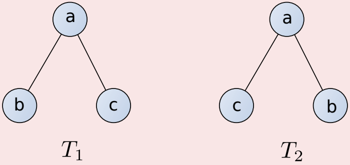
If T_1 and T_2 are ordered trees then $T_1 \neq T_2$ else $T_1 = T_2$.

Ordered Tree

Definition

An ordered tree is an oriented tree in which the children of a node are somehow "ordered."

Example



Thus

If T_1 and T_2 are ordered trees then $T_1 \neq T_2$ else $T_1 = T_2$.

Some previous definitions

Types of Ordered Trees

There are several types of ordered trees:

- k-ary tree
- Binomial tree
- Fibonacci tree

Some previous definitions

Types of Ordered Trees

There are several types of ordered trees:

- k-ary tree
- Binomial tree
- Fibonacci tree

Binomial Tree

A binomial tree is an ordered tree defined recursively.

Some previous definitions

Types of Ordered Trees

There are several types of ordered trees:

- k-ary tree
- Binomial tree
- Fibonacci tree

Binomial Tree

A binomial tree is an ordered tree defined recursively.

Some previous definitions

Types of Ordered Trees

There are several types of ordered trees:

- k-ary tree
- Binomial tree
- Fibonacci tree

Binomial Tree

A binomial tree is an ordered tree defined recursively.

Some previous definitions

Types of Ordered Trees

There are several types of ordered trees:

- k-ary tree
- Binomial tree
- Fibonacci tree

Binomial Tree

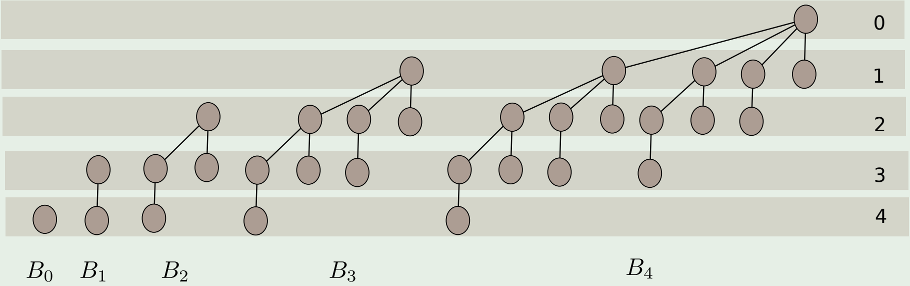
A binomial tree is an ordered tree defined recursively.

Outline

1. Introduction
   - Basic Definitions
   - Ordered Trees
2. Binomial Trees
   - Example
3. Fibonacci Heap
   - Operations
   - Fibonacci Heap
   - Why Fibonacci Heaps?
   - Node Structure
   - Fibonacci Heaps Operations
     - Mergeable-Heaps operations - Make Heap
     - Mergeable-Heaps operations - Insertion
     - Mergeable-Heaps operations - Minimum
     - Mergeable-Heaps operations - Union
   - Complexity Analysis
   - Consolidate Algorithm
   - Potential cost
   - Operation: Decreasing a Key
   - Why Fibonacci?
4. Exercises
   - Some Exercises that you can try

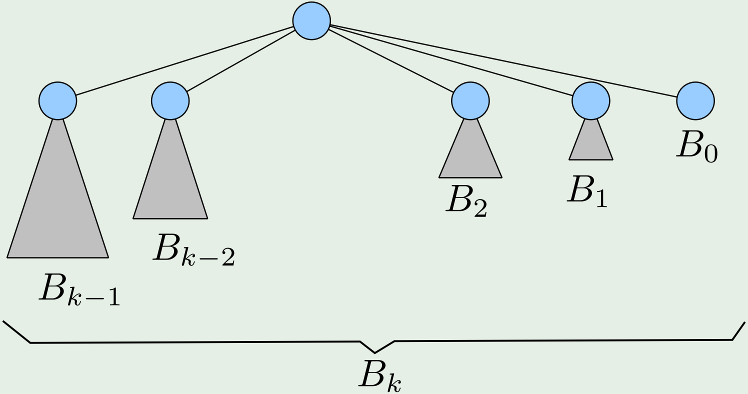
Examples

Recursive Structure



This can be seen too as

Recursive Structure



Properties of binomial trees

Lemma 19.1

For the binomial tree B_k:

1. There are 2^k nodes.
2. The height of the tree is k.
3. There are exactly $\binom{k}{i}$ nodes at depth i for $i = 0, 1, ..., k$.
4. The root has degree k, which is greater than that of any other node; moreover if the children of the root are numbered from left to right by $k-1, k-2, ..., 0$ child i is the root of a subtree B_i.

Properties of binomial trees

Lemma 19.1

For the binomial tree B_k:

1. There are 2^k nodes.
2. The height of the tree is k.
3. There are exactly $\binom{k}{i}$ nodes at depth i for $i = 0, 1, \ldots, k$.
4. The root has degree k, which is greater than that of any other node; moreover if the children of the root are numbered from left to right by $k-1, k-2, \ldots, 0$ child i is the root of a subtree B_i.

Proof!

Look at the white-board.

Properties of binomial trees

Lemma 19.1

For the binomial tree B_k:

1. There are 2^k nodes.
2. The height of the tree is k.
3. There are exactly $\binom{k}{i}$ nodes at depth i for $i = 0, 1, \ldots, k$.
4. The root has degree k, which is greater than that of any other node; moreover if the children of the root are numbered from left to right by $k-1, k-2, \ldots, 0$ child i is the root of a subtree B_i.

Proof!

Look at the white-board.

Properties of binomial trees

Lemma 19.1

For the binomial tree B_k:

1. There are 2^k nodes.
2. The height of the tree is k.
3. There are exactly $\binom{k}{i}$ nodes at depth i for $i = 0, 1, ..., k$.
4. The root has degree k, which is greater than that of any other node; moreover if the children of the root are numbered from left to right by $k-1, k-2, ..., 0$ child i is the root of a subtree B_i.

Proof!

Look at the white-board.

Properties of binomial trees

Lemma 19.1

For the binomial tree B_k:

1. There are 2^k nodes.
2. The height of the tree is k.
3. There are exactly $\binom{k}{i}$ nodes at depth i for $i = 0, 1, ..., k$.
4. The root has degree k, which is greater than that of any other node; moreover if the children of the root are numbered from left to right by $k-1, k-2, ..., 0$ child i is the root of a subtree B_i.

Proof!

Look at the white-board.

Properties of binomial trees

Lemma 19.1

For the binomial tree B_k:

1. There are 2^k nodes.
2. The height of the tree is k.
3. There are exactly $\binom{k}{i}$ nodes at depth i for $i = 0, 1, ..., k$.
4. The root has degree k, which is greater than that of any other node; moreover if the children of the root are numbered from left to right by $k-1, k-2, ..., 0$ child i is the root of a subtree B_i.

Proof!

Look at the white-board.

Outline

Mergeable Heap Operations

The Fibonacci heap data structure is used to support the operations "meargeable heap" operations

1. MAKE -HEAP() creates and returns a new heap containing no elements.
2. INSERT(H, x) inserts element x, whose key has already been filled in, into heap H.
3. MINIMUM(H) returns a pointer to the element in heap H whose key is minimum.
4. EXTRACT-MIN(H) deletes the element from heap H whose key is minimum, returning a pointer to the element.

Mergeable Heap Operations

The Fibonacci heap data structure is used to support the operations "meargeable heap" operations

1. MAKE -HEAP() creates and returns a new heap containing no elements.
2. INSERT(H, x) inserts element x, whose key has already been filled in, into heap H.
3. MINIMUM(H) returns a pointer to the element in heap H whose key is minimum.
4. EXTRACT-MIN(H) deletes the element from heap H whose key is minimum, returning a pointer to the element.

Mergeable Heap Operations

The Fibonacci heap data structure is used to support the operations "meargeable heap" operations

1. MAKE -HEAP() creates and returns a new heap containing no elements.
2. INSERT(H, x) inserts element x, whose key has already been filled in, into heap H.
3. MINIMUM(H) returns a pointer to the element in heap H whose key is minimum.
4. EXTRACT-MIN(H) deletes the element from heap H whose key is minimum, returning a pointer to the element.

Mergeable Heap Operations

The Fibonacci heap data structure is used to support the operations “meargeable heap” operations

1. MAKE -HEAP() creates and returns a new heap containing no elements.
2. INSERT(H, x) inserts element x, whose key has already been filled in, into heap H.
3. MINIMUM(H) returns a pointer to the element in heap H whose key is minimum.
4. EXTRACT-MIN(H) deletes the element from heap H whose key is minimum, returning a pointer to the element.

Mergeable Heap Operations

The Fibonacci heap data structure is used to support the operations “meargeable heap” operations

5. UNION(H_1, H_2) creates and returns a new heap that contains all the elements of heaps H_1 and H_2. Heaps are “destroyed” by this operation.
6. DECREASE-KEY(H, x, k) assigns to element x within heap H the new key value k.
7. DELETE(H, x) deletes element x from heap H.

Mergeable Heap Operations

The Fibonacci heap data structure is used to support the operations “meargeable heap” operations

5. UNION(H_1, H_2) creates and returns a new heap that contains all the elements of heaps H_1 and H_2. Heaps are “destroyed” by this operation.
6. DECREASE-KEY(H, x, k) assigns to element x within heap H the new key value k.
7. DELETE(H, x) deletes element x from heap H.

Mergeable Heap Operations

The Fibonacci heap data structure is used to support the operations "meargeable heap" operations

5. UNION(H_1, H_2) creates and returns a new heap that contains all the elements of heaps H_1 and H_2. Heaps are "destroyed" by this operation.
6. DECREASE-KEY(H, x, k) assigns to element x within heap H the new key value k.
7. DELETE(H, x) deletes element x from heap H.

Outline

1. Introduction
   - Basic Definitions
   - Ordered Trees
2. Binomial Trees
   - Example
3. Fibonacci Heap
   - Operations
   - **Fibonacci Heap**
   - Why Fibonacci Heaps?
   - Node Structure
   - Fibonacci Heaps Operations
     - Mergeable-Heaps operations - Make Heap
     - Mergeable-Heaps operations - Insertion
     - Mergeable-Heaps operations - Minimum
     - Mergeable-Heaps operations - Union
   - Complexity Analysis
   - Consolidate Algorithm
   - Potential cost
   - Operation: Decreasing a Key
   - Why Fibonacci?
4. Exercises
   - Some Exercises that you can try

Fibonacci Heap

Definition

A Fibonacci heap is a collection of rooted trees that are min-heap ordered.

Fibonacci Heap

Definition

A Fibonacci heap is a collection of rooted trees that are min-heap ordered.

Meaning

Each tree obeys the min-heap property:

- The key of a node is greater than or equal to the key of its parent.
- It is an **almost unordered binomial** tree is the same as a binomial tree except that the root of one tree is made any child of the root of the other.

Fibonacci Heap

Definition

A Fibonacci heap is a collection of rooted trees that are min-heap ordered.

Meaning

Each tree obeys the min-heap property:

- The key of a node is greater than or equal to the key of its parent.
- It is an **almost unordered binomial** tree is the same as a binomial tree except that the root of one tree is made any child of the root of the other.

Fibonacci Heap

Definition

A Fibonacci heap is a collection of rooted trees that are min-heap ordered.

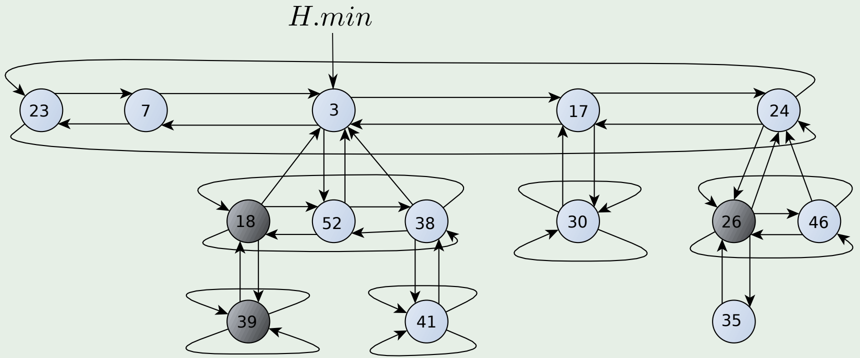
Meaning

Each tree obeys the min-heap property:

- The key of a node is greater than or equal to the key of its parent.
- It is an **almost unordered binomial** tree is the same as a binomial tree except that the root of one tree is made any child of the root of the other.

Fibonacci Structure

Example



Outline

Why Fibonacci Heaps?

Fibonacci Heaps facts

- Fibonacci heaps are especially desirable when the number of calls to Extract-Min and Delete is small.
- All other operations run in $O(1)$.

Applications

Fibonacci heaps may be used in many applications. Some graph problems, like minimum spanning tree and single-source-shortest-path.

Why Fibonacci Heaps?

Fibonacci Heaps facts

- Fibonacci heaps are especially desirable when the number of calls to Extract-Min and Delete is small.
- All other operations run in $O(1)$.

Applications

Fibonacci heaps may be used in many applications. Some graph problems, like minimum spanning tree and single-source-shortest-path.

It is more...

We have that

Procedure	Binary Heap (Worst Case)	Fibonacci Heap (Amortized)
Make-Heap	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\log n)$	$\Theta(1)$
Minimum	$\Theta(1)$	$\Theta(1)$
Extract-Min	$\Theta(\log n)$	$\Theta(\log n)$
Union	$\Theta(n)$	$\Theta(1)$
Decrease-Key	$\Theta(\log n)$	$\Theta(1)$
Delete	$\Theta(\log n)$	$\Theta(\log n)$

Outline

Fields in each node

The Classic Ones

Each node contains a $x.parent$ and $x.child$ field.

The ones for the doubled linked list

The children of each a node x are linked together in a circular double linked list:

- Each child y of x has a $y.left$ and $y.right$ to do this.

Fields in each node

The Classic Ones

Each node contains a $x.parent$ and $x.child$ field.

The ones for the doubled linked list

The children of each a node x are linked together in a circular double linked list:

- Each child y of x has a $y.left$ and $y.right$ to do this.

Thus, we have the following important labels

Field *degree*

- Did you notice that there in no way to find the number of children unless you have complex exploratory method?
- We store the number of children in the child list of node x in $x.degree$.

Thus, we have the following important labels

Field *degree*

- Did you notice that there in no way to find the number of children unless you have complex exploratory method?
- We store the number of children in the child list of node x in $x.degree$.

The Amortized Label

Each child has the field $mark$.

- Newly created nodes are unmarked (Boolean value FALSE), and a node becomes unmarked whenever it is made the child of another node.

Thus, we have the following important labels

Field *degree*

- Did you notice that there in no way to find the number of children unless you have complex exploratory method?
- We store the number of children in the child list of node x in $x.degree$.

The Amortized Label

Each child has the field $mark$.

IMPORTANT

1. The field $mark$ indicates whether a node has lost a child since the last time was made the child of another node.
2. Newly created nodes are unmarked (Boolean value $FALSE$), and a node becomes unmarked whenever it is made the child of another node.

Thus, we have the following important labels

Field $degree$

- Did you notice that there in no way to find the number of children unless you have complex exploratory method?
- We store the number of children in the child list of node x in $x.degree$.

The Amortized Label

Each child has the field $mark$.

IMPORTANT

1. The field $mark$ indicates whether a node has lost a child since the last time was made the child of another node.
2. Newly created nodes are unmarked (Boolean value $FALSE$), and a node becomes unmarked whenever it is made the child of another node.

Thus, we have the following important labels

Field *degree*

- Did you notice that there in no way to find the number of children unless you have complex exploratory method?
- We store the number of children in the child list of node x in $x.degree$.

The Amortized Label

Each child has the field $mark$.

IMPORTANT

1. The field $mark$ indicates whether a node has lost a child since the last time was made the child of another node.
2. Newly created nodes are unmarked (Boolean value $FALSE$), and a node becomes unmarked whenever it is made the child of another node.

The child list

Circular, doubly linked list have two advantages for use in Fibonacci heaps:

- First, we can remove a node from a circular, doubly linked list in $O(1)$ time.
- Second, given two such lists, we can concatenate them (or "splice" them together) into one circular, doubly linked list in $O(1)$ time.

The child list

Circular, doubly linked list have two advantages for use in Fibonacci heaps:

- First, we can remove a node from a circular, doubly linked list in $O(1)$ time.
- Second, given two such lists, we can concatenate them (or "splice" them together) into one circular, doubly linked list in $O(1)$ time.

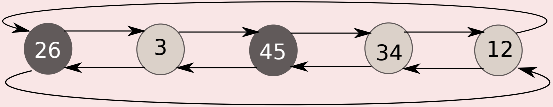
Example

The child list

Circular, doubly linked list have two advantages for use in Fibonacci heaps:

- First, we can remove a node from a circular, doubly linked list in $O(1)$ time.
- Second, given two such lists, we can concatenate them (or "splice" them together) into one circular, doubly linked list in $O(1)$ time.

Example



Additional

First

The roots of all the trees in a Fibonacci heap H are linked together using their left and right pointers into a circular, doubly linked list called the root list of the Fibonacci heap.

Additional

First

The roots of all the trees in a Fibonacci heap H are linked together using their left and right pointers into a circular, doubly linked list called the root list of the Fibonacci heap.

Second

- The pointer $H.min$ of the Fibonacci data structure thus points to the node in the root list whose key is minimum.
- Trees may appear in any order within a root list.

Additional

First

The roots of all the trees in a Fibonacci heap H are linked together using their left and right pointers into a circular, doubly linked list called the root list of the Fibonacci heap.

Second

- The pointer $H.min$ of the Fibonacci data structure thus points to the node in the root list whose key is minimum.
- Trees may appear in any order within a root list.

Third

The Fibonacci data structure has the field $H.n$ =**the number of nodes currently in the Fibonacci Heap** H.

Additional

First

The roots of all the trees in a Fibonacci heap H are linked together using their left and right pointers into a circular, doubly linked list called the root list of the Fibonacci heap.

Second

- The pointer $H.min$ of the Fibonacci data structure thus points to the node in the root list whose key is minimum.
- Trees may appear in any order within a root list.

Third

The Fibonacci data structure has the field $H.n =$**the number of nodes currently in the Fibonacci Heap** H**.**

Outline

Idea behind Fibonacci heaps

Main idea

Fibonacci heaps are called lazy data structures because they delay work as long as possible using the field $mark$!!!

Make Heap

Make a heap

You only need the following code:

MakeHeap()

1. $min[H] = NIL$
2. $n(H) = 0$

Complexity is as simple as

- Cost $O(1)$.

Make Heap

Make a heap

You only need the following code:

MakeHeap()

1. $min[H] = NIL$
2. $n(H) = 0$

Complexity is as simple as

- Cost $O(1)$.

Insertion

Code for Inserting a node

Fib-Heap-Insert(H, x)

```
$x.degree = 0$
$x.p = NIL$
$x.child = NIL$
$x.mark = FALSE$
if $H.min = NIL$
    Create a root list for $H$ containing just $x$
    $H.min = x$
else insert x into $H's$ root list
    if $x.key < H.min.key$
        $H.min = x$
$H.n = H.n + 1$
```

Insertion

Code for Inserting a node

Fib-Heap-Insert(H, x)

```
$x.degree = 0$
$x.p = NIL$
$x.child = NIL$
$x.mark = FALSE$
if $H.min = NIL$
    Create a root list for $H$ containing just $x$
    $H.min = x$
else insert x into $H's$ root list
    if $x.key < H.min.key$
        $H.min = x$
$H.n = H.n + 1$
```

Insertion

Code for Inserting a node

```
Fib-Heap-Insert$(H, x)$
    $x.degree = 0$
    $x.p = NIL$
    $x.child = NIL$
    $x.mark = FALSE$
    if $H.min = NIL$
        Create a root list for $H$ containing just $x$
        $H.min = x$
    else insert x into $H's$ root list
        if $x.key < H.min.key$
            $H.min = x$
    $H.n = H.n + 1$
```

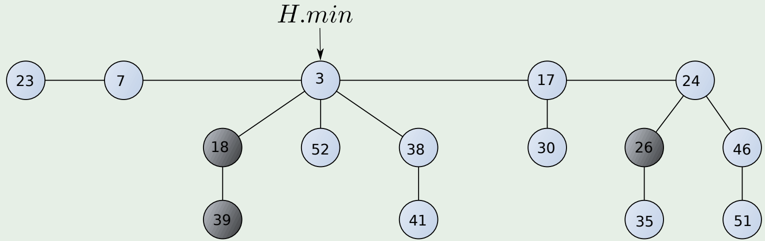
Insertion

Code for Inserting a node

```
Fib-Heap-Insert($H, x$)
    $x.degree = 0$
    $x.p = NIL$
    $x.child = NIL$
    $x.mark = FALSE$
    if $H.min = NIL$
        Create a root list for $H$ containing just $x$
        $H.min = x$
    else insert x into $H's$ root list
        if $x.key < H.min.key$
            $H.min = x$
    $H.n = H.n + 1$
```

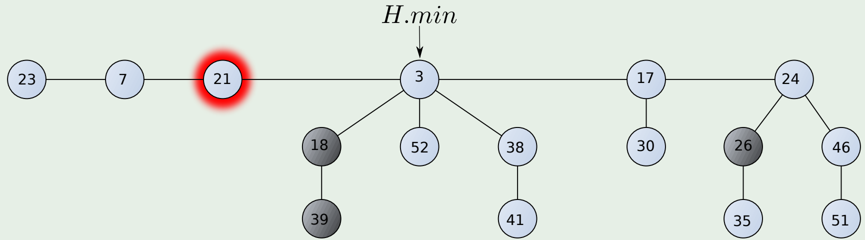
Inserting a node

Example



Inserting a node

Example



Minimum

Finding the Minimum

- Simply return the key of $min(H)$.
- The amortized cost is simply $O(1)$.
    - We will analyze this later on...

Minimum

Finding the Minimum

- Simply return the key of $min(H)$.
- The amortized cost is simply $O(1)$.
  - We will analyze this later on...

What about Union of two Heaps?

Code for Union of Heaps

```
Fib-Heap-Union($H_1, H_2$)
    $H$ =Make-Fib-Heap()
    $H.min = H_1.min$
    Concatenate the root list of $H_2$ with the root list of $H$
    if ($H_1.min == NIL$) or ($H.min \neq NIL$ and $H_2.min.key < H_1.min.key$)
        $H.min = H_2.min$
    $H.n = H_1.n + H_2.n$
    return $H$
```

What about Union of two Heaps?

Code for Union of Heaps

Fib-Heap-Union(H_1, H_2)

```
$H$ =Make-Fib-Heap()
$H.min = H_1.min$
Concatenate the root list of $H_2$ with the root list of $H$
If $(H_1.min == NIL)$ or $(H.min \neq NIL$ and $H_2.min.key < H_1.min.key)$
    $H.min = H_2.min$
$H.n = H_1.n + H_2.n$
return $H$
```

What about Union of two Heaps?

Code for Union of Heaps

```
Fib-Heap-Union($H_1, H_2$)
    $H$ =Make-Fib-Heap()
    $H.min = H_1.min$
    Concatenate the root list of $H_2$ with the root list of $H$
    **If** $(H_1.min == NIL)$ or $(H.min \neq NIL$ and $H_2.min.key < H_1.min.key)$
        $H.min = H_2.min$
    $H.n = H_1.n + H_2.n$
    **return** $H$
```

Outline

In order to analyze Union...

We introduce some ideas from...

Our old friend amortized analysis and potential method!!!

Amortized potential function

We have the following function

$$\Phi(H) = t(H) + 2m(H)$$

- Where:
    - $t(H)$ is the number of trees in the Fibonacci heap
    - $m(H)$ is the number of marked nodes in the tree.

Amortized potential function

We have the following function

$$\Phi(H) = t(H) + 2m(H)$$

- Where:
  - $t(H)$ **is the number of trees in the Fibonacci heap**
  - $m(H)$ **is the number of marked nodes in the tree.**

Amortized analysis

The amortized analysis will depend on there being a known bound $D(n)$ on the **maximum degree** of any node in an n-node heap.

Amortized potential function

We have the following function

$$\Phi(H) = t(H) + 2m(H)$$

- Where:
  - $t(H)$ **is the number of trees in the Fibonacci heap**
  - $m(H)$ **is the number of marked nodes in the tree.**

Amortized analysis

The amortized analysis will depend on there being a known bound $D(n)$ on the **maximum degree** of any node in an n-node heap.

Amortized potential function

We have the following function

$$\Phi(H) = t(H) + 2m(H)$$

- Where:
  - $t(H)$ **is the number of trees in the Fibonacci heap**
  - $m(H)$ **is the number of marked nodes in the tree.**

Amortized analysis

The amortized analysis will depend on there being a known bound $D(n)$ on the **maximum degree** of any node in an n-node heap.

Amortized potential function

We have the following function

$$\Phi(H) = t(H) + 2m(H)$$

- Where:
  - $t(H)$ **is the number of trees in the Fibonacci heap**
  - $m(H)$ **is the number of marked nodes in the tree.**

Amortized analysis

The amortized analysis will depend on there being a known bound $D(n)$ on the **maximum degree** of any node in an n-node heap.

Observations about $D(n)$

About the known bound $D(n)$

- $D(n)$ is the maximum degree of any node in the binomial heap.
- It is more if the Fibonacci heap is a collection of unordered trees, then $D(n) = \log n$.
    - We will prove this latter!!!

Observations about $D(n)$

About the known bound $D(n)$

- $D(n)$ is the maximum degree of any node in the binomial heap.
- It is more if the Fibonacci heap is a collection of unordered trees, then $D(n) = \log n$.
    - We will prove this latter!!!

Observations about $D(n)$

About the known bound $D(n)$

- $D(n)$ is the maximum degree of any node in the binomial heap.
- It is more if the Fibonacci heap is a collection of unordered trees, then $D(n) = \log n$.
    - We will prove this latter!!!

Back to Insertion

First

If H' is the Fibonacci heap after inserting, and H before that:

$$t(H') = t(H) + 1$$

Second

$$m(H') = m(H)$$

Then the change of potential is

$\Phi(H') - \Phi(H) = 1$ then complexity analysis results in $O(1) + 1 = O$

Back to Insertion

First

If H' is the Fibonacci heap after inserting, and H before that:

$$t(H') = t(H) + 1$$

Second

$$m(H') = m(H)$$

Then the change of potential is

$\Phi(H') - \Phi(H) = 1$ then complexity analysis results in $O(1) + 1 = O$

Back to Insertion

First

If H' is the Fibonacci heap after inserting, and H before that:

$$t(H') = t(H) + 1$$

Second

$$m(H') = m(H)$$

Then the change of potential is

$\Phi(H') - \Phi(H) = 1$ then complexity analysis results in $O(1) + 1 = O$

Other operations: Find Min

It is possible to rephrase this in terms of potential cost

By using the pointer $min[H]$ potential cost is 0 then $O(1)$.

Other operations: Union

Union of two Fibonacci heaps

```
Fib-Heap-Union($H_1$, $H_2$)
    $H$ =Make-Fib-Heap()
    $H.min = H_1.min$
    Concatenate the root list of $H_2$ with the root list of $H$
    If ($H_1.min == NIL$) or ($H.min \neq NIL$ and $H_2.min.key < H_1.min.key$)
        $H.min = H_2.min$
    $H.n = H_1.n + H_2.n$
    return $H$
```

Other operations: Union

Union of two Fibonacci heaps

Fib-Heap-Union(H_1, H_2)

```
H = Make-Fib-Heap()
H.min = H_1.min
Concatenate the root list of H_2 with the root list of H
If (H_1.min == NIL) or (H.min ≠ NIL and H_2.min.key < H_1.min.key)
    H.min = H_2.min
H.n = H_1.n + H_2.n
return H
```

Cost of uniting two Fibonacci heaps

First

- $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$.

Second

- $c_i = O(1)$ this is because the number of steps to make the union operations is a constant.

Potential analysis

$$\widehat{c}_i = c_i + \Phi(H) - [\Phi(H_1) + \Phi(H_2)] = O(1) + 0 = O(1).$$

We have then a complexity of $O(1)$.

Cost of uniting two Fibonacci heaps

First

- $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$.

Second

- $c_i = O(1)$ this is because the number of steps to make the union operations is a constant.

Potential analysis

$$\widehat{c}_i = c_i + \Phi(H) - [\Phi(H_1) + \Phi(H_2)] = O(1) + 0 = O(1).$$

We have then a complexity of $O(1)$.

Cost of uniting two Fibonacci heaps

First

- $t(H) = t\left(H_1\right) + t\left(H_2\right)$ and $m(H) = m\left(H_1\right) + m\left(H_2\right)$.

Second

- $c_i = O(1)$ this is because the number of steps to make the union operations is a constant.

Potential analysis

$$\widehat{c_i} = c_i + \Phi\left(H\right) - \left[\Phi\left(H_1\right) + \Phi\left(H_2\right)\right] = O(1) + 0 = O(1).$$

We have then a complexity of $O(1)$.

Extract min

Extract min

Fib-Heap-Extract-Min(H)

```
$z = H.min$
if $z \neq NIL$
    for each child $x$ of $z$
        add $x$ to the root list of $H$
        $x.p = NIL$
    remove $z$ from the root list of $H$
    if $z == z.right$
        $H.min = NIL$
    else $H.min = z.right$
        Consolidate$(H)$
    $H.n = H.n - 1$
return $z$
```

Extract min

Extract min

Fib-Heap-Extract-Min(H)

```
$z = H.min$
if $z \neq NIL$
    for each child $x$ of $z$
        add $x$ to the root list of $H$
        $x.p = NIL$
    remove $z$ from the root list of $H$
    if $z == z.right$
        $H.min = NIL$
    else $H.min = z.right$
        Consolidate$(H)$
    $H.n = H.n - 1$
return $z$
```

Extract min

Extract min

```
Fib-Heap-Extract-Min($H$)
$z = H.min$
if $z \neq NIL$
    for each child $x$ of $z$
        add $x$ to the root list of $H$
        $x.p = NIL$
    remove $z$ from the root list of $H$
    if $z == z.right$
        $H.min = NIL$
    else $H.min = z.right$
        Consolidate($H$)
    $H.n = H.n - 1$
return $z$
```

Extract min

Extract min

```
Fib-Heap-Extract-Min($H$)
z = H.min
if $z \neq NIL$
    for each child $x$ of $z$
        add $x$ to the root list of $H$
        $x.p = NIL$
    remove $z$ from the root list of $H$
    if $z == z.right$
        $H.min = NIL$
    else $H.min = z.right$
        Consolidate($H$)
    $H.n = H.n - 1$
return $z$
```

Extract min

Extract min

Fib-Heap-Extract-Min(H)

```
$z = H.min$
if $z \neq NIL$
    for each child $x$ of $z$
        add $x$ to the root list of $H$
        $x.p = NIL$
    remove $z$ from the root list of $H$
    if $z == z.right$
        $H.min = NIL$
    else $H.min = z.right$
        Consolidate$(H)$
    $H.n = H.n - 1$
return $z$
```

Extract min

Extract min

```
Fib-Heap-Extract-Min($H$)
$z = H.min$
if $z \neq NIL$
    for each child $x$ of $z$
        add $x$ to the root list of $H$
        $x.p = NIL$
    remove $z$ from the root list of $H$
    if $z == z.right$
        $H.min = NIL$
    else $H.min = z.right$
        Consolidate($H$)
    $H.n = H.n - 1$
return $z$
```

What is happening here?

First

Here, the code in lines 3-6 remove the node z and adds the children of z to the root list of H.

What is happening here?

First

Here, the code in lines 3-6 remove the node z and adds the children of z to the root list of H.

Next

If the Fibonacci Heap is not empty a consolidation code is triggered.

What is happening here?

Thus

- The consolidate code is used to **eliminate** subtrees that have the same root degree by linking them.
- It repeatedly executes the following steps:
    1. Find two roots x and y in the root list with the same degree. Without loss of generality, let $x.key \leq y.key$.
    2. Link y to x: remove y from the root list, and make y a child of x by calling the FIB-HEAP-LINK procedure.
        - ★ This procedure increments the attribute $x.degree$ and clears the mark on y.

What is happening here?

Thus

- The consolidate code is used to **eliminate** subtrees that have the same root degree by linking them.
- It repeatedly executes the following steps:
  1. Find two roots x and y in the root list with the same degree. Without loss of generality, let $x.key \leq y.key$.
  2. Link y to x: remove y from the root list, and make y a child of x by calling the FIB-HEAP-LINK procedure.
     - ★ This procedure increments the attribute $x.degree$ and clears the mark on y.

What is happening here?

Thus

- The consolidate code is used to **eliminate** subtrees that have the same root degree by linking them.
- It repeatedly executes the following steps:
  1. Find two roots x and y in the root list with the same degree. Without loss of generality, let $x.key \leq y.key$.
  2. Link y to x: remove y from the root list, and make y a child of x by calling the FIB-HEAP-LINK procedure.
     - This procedure increments the attribute $x.degree$ and clears the mark on y.

What is happening here?

Thus

- The consolidate code is used to **eliminate** subtrees that have the same root degree by linking them.
- It repeatedly executes the following steps:
  1. Find two roots x and y in the root list with the same degree. Without loss of generality, let $x.key \leq y.key$.
  2. Link y to x: remove y from the root list, and make y a child of x by calling the FIB-HEAP-LINK procedure.
     - ★ This procedure increments the attribute $x.degree$ and clears the mark on y.

What is happening here?

Thus

- The consolidate code is used to **eliminate** subtrees that have the same root degree by linking them.
- It repeatedly executes the following steps:
  1. Find two roots x and y in the root list with the same degree. Without loss of generality, let $x.key \leq y.key$.
  2. Link y to x: remove y from the root list, and make y a child of x by calling the FIB-HEAP-LINK procedure.
     - ★ This procedure increments the attribute $x.degree$ and clears the mark on y.

Outline

Consolidate Code

Consolidate(H)

```
Let $A(0 \ldots D(H.n))$ be a new array
for $i = 0$ to $D(H.n)$
    $A[i] = NIL$
for each $w$ in the root list of $H$
    $x = w$
    $d = x.degree$
    while $A[d] \neq NIL$
        $y = A[d]$
        if $x.key > y.key$
            exchange $x$ with $y$
        Fib-Heap-Link($H$, $y$, $x$)
        $A[d] = NIL$
        $d = d + 1$
    $A[d] = x$
$H.min = NIL$
for $i = 0$ to $D(H.n)$
    if $A[i] \neq NIL$
        if $H.min == NIL$
            create a root list for $H$
            containing just $A[i]$
            $H.min = A[i]$
        else
            insert $A[i]$ into $H$'s
            root list
            if $A[i].key < H.min.key$
                $H.min = A[i]$
```

Consolidate Code

Consolidate(H)

```
Let $A(0...D(H.n))$ be a new array
for $i = 0$ to $D(H.n)$
    $A[i] = NIL$
for each $w$ in the root list of $H$
    $x = w$
    $d = x.degree$
    while $A[d] \neq NIL$
        $y = A[d]$
        if $x.key > y.key$
            exchange $x$ with $y$
        Fib-Heap-Link($H, y, x$)
        $A[d] = NIL$
        $d = d + 1$
    $A[d] = x$
$H.min = NIL$
for $i = 0$ to $D(H.n)$
    if $A[i] \neq NIL$
        if $H.min == NIL$
            create a root list for $H$
            containing just $A[i]$
            $H.min = A[i]$
        else
            insert $A[i]$ into $H$'s
            root list
            if $A[i].key < H.min.key$
                $H.min = A[i]$
```

Consolidate Code

Consolidate(H)

```
Let $A(0...D(H.n))$ be a new array
for $i = 0$ to $D(H.n)$
    $A[i] = NIL$
for each $w$ in the root list of $H$
    $x = w$
    $d = x.degree$
    while $A[d] \neq NIL$
        $y = A[d]$
        if $x.key > y.key$
            exchange $x$ with $y$
        Fib-Heap-Link($H, y, x$)
        $A[d] = NIL$
        $d = d + 1$
    $A[d] = x$
$H.min = NIL$
for $i = 0$ to $D(H.n)$
    if $A[i] \neq NIL$
        if $H.min == NIL$
            create a root list for $H$
            containing just $A[i]$
            $H.min = A[i]$
        else
            insert $A[i]$ into $H's$
            root list
            if $A[i].key < H.min.key$
                $H.min = A[i]$
```

Consolidate Code

Consolidate(H)

```
Let $A(0...D(H.n))$ be a new array
for $i = 0$ to $D(H.n)$
    $A[i] = NIL$
for each $w$ in the root list of $H$
    $x = w$
    $d = x.degree$
    while $A[d] \neq NIL$
        $y = A[d]$
        if $x.key > y.key$
            exchange $x$ with $y$
        Fib-Heap-Link($H, y, x$)
        $A[d] = NIL$
        $d = d + 1$
    $A[d] = x$
$H.min = NIL$
for $i = 0$ to $D(H.n)$
    if $A[i] \neq NIL$
        if $H.min == NIL$
            create a root list for $H$
            containing just $A[i]$
            $H.min = A[i]$
        else
            insert $A[i]$ into $H's$
            root list
            if $A[i].key < H.min.key$
                $H.min = A[i]$
```

Consolidate Code

Consolidate(H)

```
Let $A(0 ... D(H.n))$ be a new array
for $i = 0$ to $D(H.n)$
    $A[i] = NIL$
for each $w$ in the root list of $H$
    $x = w$
    $d = x.degree$
    while $A[d] \neq NIL$
        $y = A[d]$
        if $x.key > y.key$
            exchange $x$ with $y$
        Fib-Heap-Link($H$, $y$, $x$)
        $A[d] = NIL$
        $d = d + 1$
    $A[d] = x$
$H.min = NIL$
for $i = 0$ to $D(H.n)$
    if $A[i] \neq NIL$
        if $H.min == NIL$
            create a root list for $H$
            containing just $A[i]$
            $H.min = A[i]$
        else
            insert $A[i]$ into $H$'s
            root list
            if $A[i].key < H.min.key$
                $H.min = A[i]$
```

Consolidate Code

Consolidate(H)

```
Let $A(0 \ldots D(H.n))$ be a new array
for $i = 0$ to $D(H.n)$
    $A[i] = NIL$
for each $w$ in the root list of $H$
    $x = w$
    $d = x.degree$
    while $A[d] \neq NIL$
        $y = A[d]$
        if $x.key > y.key$
            exchange $x$ with $y$
        Fib-Heap-Link$(H, y, x)$
        $A[d] = NIL$
        $d = d + 1$
    $A[d] = x$
$H.min = NIL$
for $i = 0$ to $D(H.n)$
    if $A[i] \neq NIL$
        if $H.min == NIL$
            create a root list for $H$
            containing just $A[i]$
            $H.min = A[i]$
        else
            insert $A[i]$ into $H$'s
            root list
            if $A[i].key < H.min.key$
                $H.min = A[i]$
```

Consolidate Code

Consolidate(H)

```
Let $A(0 \ldots D(H.n))$ be a new array
for $i = 0$ to $D(H.n)$
    $A[i] = NIL$
for each $w$ in the root list of $H$
    $x = w$
    $d = x.degree$
    while $A[d] \neq NIL$
        $y = A[d]$
        if $x.key > y.key$
            exchange $x$ with $y$
        Fib-Heap-Link($H, y, x$)
        $A[d] = NIL$
        $d = d + 1$
    $A[d] = x$
$H.min = NIL$
for $i = 0$ to $D(H.n)$
    if $A[i] \neq NIL$
        if $H.min == NIL$
            create a root list for $H$
            containing just $A[i]$
            $H.min = A[i]$
        else
            insert $A[i]$ into $H$'s
            root list
            if $A[i].key < H.min.key$
                $H.min = A[i]$
```

Consolidate Code

Consolidate(H)

```
Let $A(0 \ldots D(H.n))$ be a new array
for $i = 0$ to $D(H.n)$
    $A[i] = NIL$
for each $w$ in the root list of $H$
    $x = w$
    $d = x.degree$
    while $A[d] \neq NIL$
        $y = A[d]$
        if $x.key > y.key$
            exchange $x$ with $y$
        Fib-Heap-Link($H, y, x$)
        $A[d] = NIL$
        $d = d + 1$
    $A[d] = x$
$H.min = NIL$
for $i = 0$ to $D(H.n)$
    if $A[i] \neq NIL$
        if $H.min == NIL$
            create a root list for $H$
            containing just $A[i]$
            $H.min = A[i]$
        else
            insert $A[i]$ into $H's$
            root list
            if $A[i].key < H.min.key$
                $H.min = A[i]$
```

Consolidate Code

Consolidate(H)

```
Let $A(0 ... D(H.n))$ be a new array
for $i = 0$ to $D(H.n)$
    $A[i] = NIL$
for each $w$ in the root list of $H$
    $x = w$
    $d = x.degree$
    while $A[d] \neq NIL$
        $y = A[d]$
        if $x.key > y.key$
            exchange $x$ with $y$
        Fib-Heap-Link($H, y, x$)
        $A[d] = NIL$
        $d = d + 1$
    $A[d] = x$
$H.min = NIL$
for $i = 0$ to $D(H.n)$
    if $A[i] \neq NIL$
        if $H.min == NIL$
            create a root list for $H$ containing just $A[i]$
            $H.min = A[i]$
        else
            insert $A[i]$ into $H's$ root list
            if $A[i].key < H.min.key$
                $H.min = A[i]$
```

Consolidate Code

Consolidate(H)

```
Let $A(0 \dots D(H.n))$ be a new array
for $i = 0$ to $D(H.n)$
    $A[i] = NIL$
for each $w$ in the root list of $H$
    $x = w$
    $d = x.degree$
    while $A[d] \neq NIL$
        $y = A[d]$
        if $x.key > y.key$
            exchange $x$ with $y$
        Fib-Heap-Link($H, y, x$)
        $A[d] = NIL$
        $d = d + 1$
    $A[d] = x$
$H.min = NIL$
for $i = 0$ to $D(H.n)$
    if $A[i] \neq NIL$
        if $H.min == NIL$
            create a root list for $H$
            containing just $A[i]$
            $H.min = A[i]$
        else
            insert $A[i]$ into $H's$
            root list
            if $A[i].key < H.min.key$
                $H.min = A[i]$
```

Consolidate Code

Consolidate(H)

```
Let $A(0 ... D(H.n))$ be a new array
for $i = 0$ to $D(H.n)$
    $A[i] = NIL$
for each $w$ in the root list of $H$
    $x = w$
    $d = x.degree$
    while $A[d] \neq NIL$
        $y = A[d]$
        if $x.key > y.key$
            exchange $x$ with $y$
        Fib-Heap-Link($H, y, x$)
        $A[d] = NIL$
        $d = d + 1$
    $A[d] = x$
$H.min = NIL$
for $i = 0$ to $D(H.n)$
    if $A[i] \neq NIL$
        if $H.min == NIL$
            create a root list for $H$
            containing just $A[i]$
            $H.min = A[i]$
        else
            insert $A[i]$ into $H'$s
            root list
            if $A[i].key < H.min.key$
                $H.min = A[i]$
```

Consolidate Code

Consolidate(H)

```
Let $A(0 \ldots D(H.n))$ be a new array
for $i = 0$ to $D(H.n)$
    $A[i] = NIL$
for each $w$ in the root list of $H$
    $x = w$
    $d = x.degree$
    while $A[d] \neq NIL$
        $y = A[d]$
        if $x.key > y.key$
            exchange $x$ with $y$
        Fib-Heap-Link($H, y, x$)
        $A[d] = NIL$
        $d = d + 1$
    $A[d] = x$
$H.min = NIL$
for $i = 0$ to $D(H.n)$
    if $A[i] \neq NIL$
        if $H.min == NIL$
            create a root list for $H$
            containing just $A[i]$
            $H.min = A[i]$
        else
            insert $A[i]$ into $H's$
            root list
            if $A[i].key < H.min.key$
                $H.min = A[i]$
```

Consolidate Code

Consolidate(H)

```
Let $A(0 \dots D(H.n))$ be a new array
for $i = 0$ to $D(H.n)$
    $A[i] = NIL$
for each $w$ in the root list of $H$
    $x = w$
    $d = x.degree$
    while $A[d] \neq NIL$
        $y = A[d]$
        if $x.key > y.key$
            exchange $x$ with $y$
        Fib-Heap-Link($H, y, x$)
        $A[d] = NIL$
        $d = d + 1$
    $A[d] = x$
$H.min = NIL$
for $i = 0$ to $D(H.n)$
    if $A[i] \neq NIL$
        if $H.min == NIL$
            create a root list for $H$
            containing just $A[i]$
            $H.min = A[i]$
        else
            insert $A[i]$ into $H's$
            root list
            if $A[i].key < H.min.key$
                $H.min = A[i]$
```

Fib-Heap-Link Code

Fib-Heap-Link(H, y, x)

1. Remove y from the root list of H
2. Make y a child of x, incrementing $x.degree$
3. $y.mark = FALSE$

Fib-Heap-Link Code

Fib-Heap-Link(H, y, x)

1. Remove y from the root list of H
2. Make y a child of x, incrementing $x.degree$
3. $y.mark = FALSE$

Fib-Heap-Link Code

Fib-Heap-Link(H, y, x)

```
1. Remove $y$ from the root list of $H$
2. Make $y$ a child of $x$, incrementing $x.degree$
3. $y.mark = FALSE$
```

Auxiliary Array

The Consolidate uses

An auxiliary pointer array $A\left[0...D\left(H.n\right)\right]$

It keeps track of

The roots according the degree

Auxiliary Array

The Consolidate uses

An auxiliary pointer array $A\left[0 ... D\left(H.n\right)\right]$

It keeps track of

The roots according the degree

Code Process

A while loop inside of the for loop - lines 1-15

- Using a w variable to go through the root list
- This is used to fill the pointers in $A\left[0...D\left(H.n\right)\right]$
- Then you link both trees using who has a larger key
- Then you add a pointer to the new min-heap, with new degree, in A.

Code Process

A while loop inside of the for loop - lines 1-15

- Using a w variable to go through the root list
- This is used to fill the pointers in $A\left[0 \ldots D\left(H.n\right)\right]$
- Then you link both trees using who has a larger key
- Then you add a pointer to the new min-heap, with new degree, in A.

Then - lines 15-23

Lines 15-23 clean the original Fibonacci Heap, then using the pointers at the array A, each subtree is inserted into the root list of H.

Code Process

A while loop inside of the for loop - lines 1-15

- Using a w variable to go through the root list
- This is used to fill the pointers in $A\left[0 \ldots D\left(H.n\right)\right]$
- Then you link both trees using who has a larger key
- Then you add a pointer to the new min-heap, with new degree, in A.

Then - lines 15-23

Lines 15-23 clean the original Fibonacci Heap, then using the pointers at the array A, each subtree is inserted into the root list of H.

Code Process

A while loop inside of the for loop - lines 1-15

- Using a w variable to go through the root list
- This is used to fill the pointers in $A\left[0 \ldots D\left(H.n\right)\right]$
- Then you link both trees using who has a larger key
- Then you add a pointer to the new min-heap, with new degree, in A.

Then - lines 15-23

Lines 15-23 clean the original Fibonacci Heap, then using the pointers at the array A, each subtree is inserted into the root list of H.

Code Process

A while loop inside of the for loop - lines 1-15

- Using a w variable to go through the root list
- This is used to fill the pointers in $A\left[0...D\left(H.n\right)\right]$
- Then you link both trees using who has a larger key
- Then you add a pointer to the new min-heap, with new degree, in A.

Then - lines 15-23

Lines 15-23 clean the original Fibonacci Heap, then using the pointers at the array A, each subtree is inserted into the root list of H.

Loop Invariance

We have the following Loop Invariance

At the start of each iteration of the while loop, $d = x.degree$.

Init

Line 6 ensures that the loop invariant holds the first time we enter the loop.

Maintenance

We have two nodes x and y such that they have the same degree then

- We link them together and increase the d to $d + 1$ adding a new tree pointer to A with degree $d + 1$

Loop Invariance

We have the following Loop Invariance

At the start of each iteration of the while loop, $d = x.degree$.

Init

Line 6 ensures that the loop invariant holds the first time we enter the loop.

Maintenance

We have two nodes x and y such that they have the same degree then

- We link them together and increase the d to $d+1$ adding a new tree pointer to A with degree $d+1$

Loop Invariance

We have the following Loop Invariance

At the start of each iteration of the while loop, $d = x.degree$.

Init

Line 6 ensures that the loop invariant holds the first time we enter the loop.

Maintenance

We have two nodes x and y such that they have the same degree then

- We link them together and increase the d to $d+1$ adding a new tree pointer to A with degree $d+1$

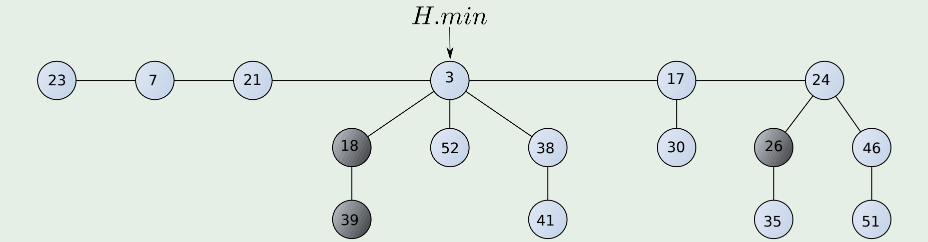
Loop Invariance

Termination

We repeat the while loop until $A\left[d\right] = NIL$, in which case there is no other root with the same degree as x.

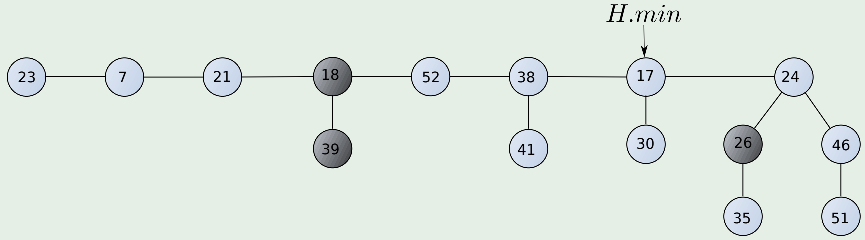
Example of consolidation

We remove $H.min == 3$



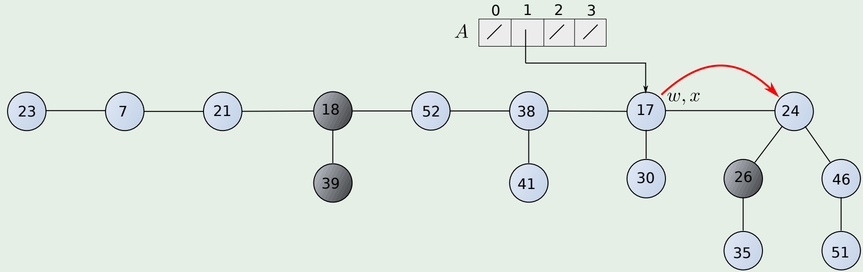
Example of consolidation

The children are moved to the root's list



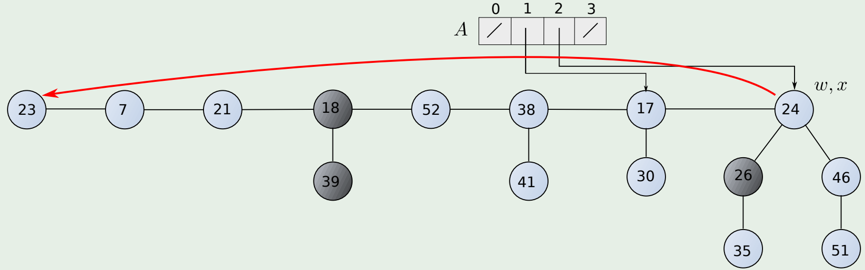
Example of consolidation

Now, you get Consolidation running beginning $A[1] \rightarrow 17$



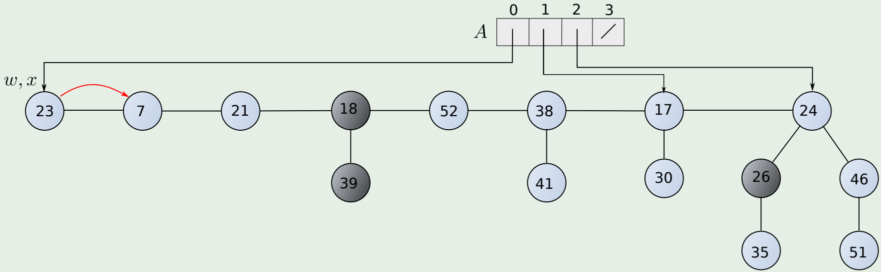
Example of consolidation

Now, $A[2] \rightarrow 24$



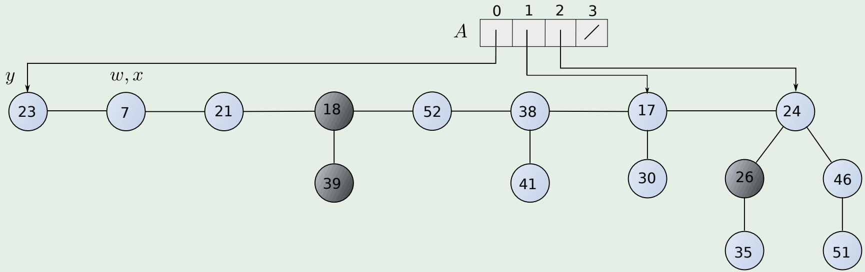
Example of consolidation

We have a pointer to a node with $degree = 0$



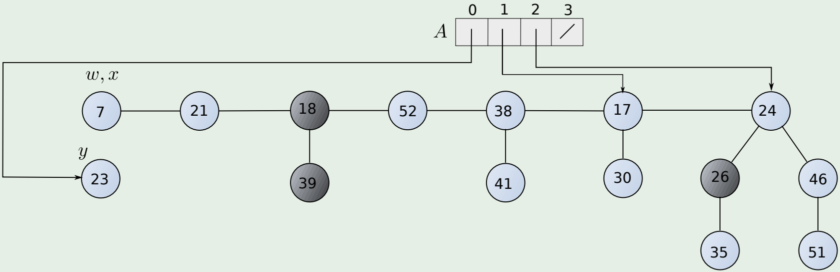
Example of consolidation

We don't do an exchange



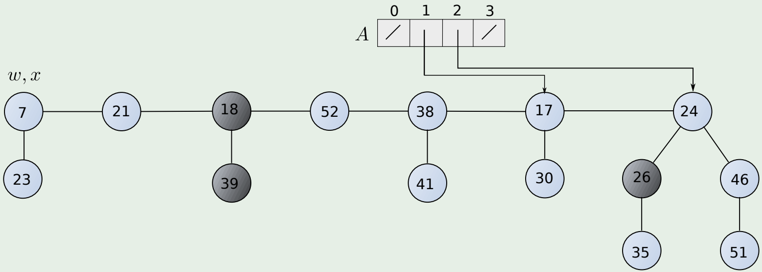
Example of consolidation

Remove y from the root's list



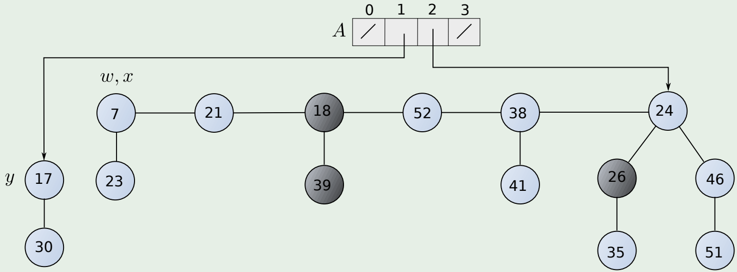
Example of consolidation

Make y a child of x



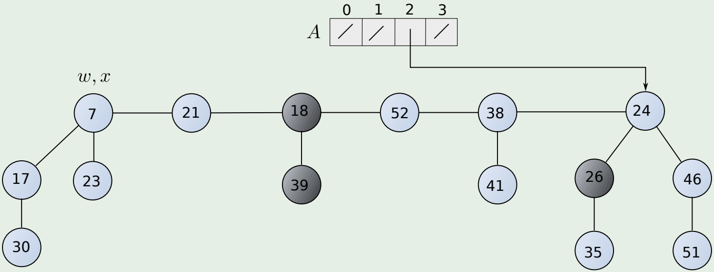
Example of consolidation

Remove y from the root list



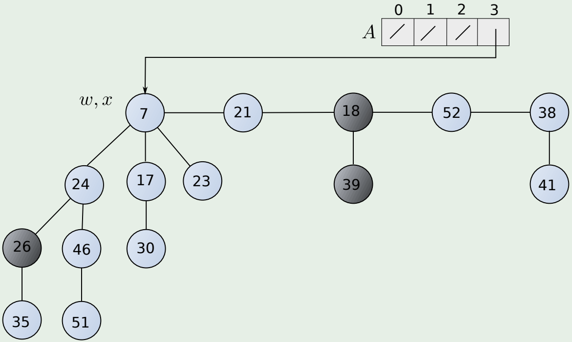
Example of consolidation

Make y a child of x



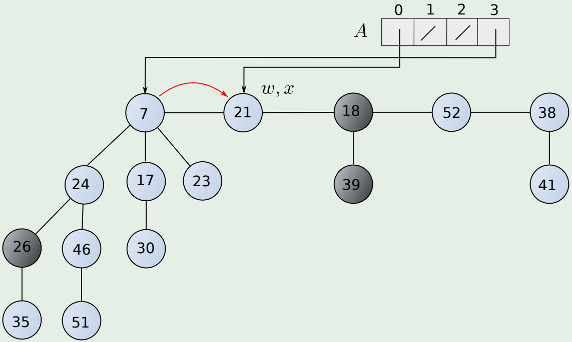
Example of consolidation

and we point to the the element with $degree = 3$



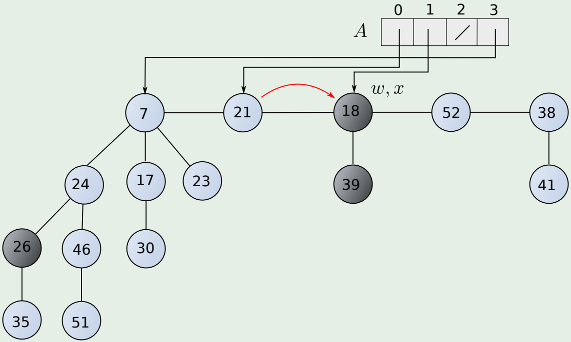
Example of consolidation

We move to the next root's child and point to it from A



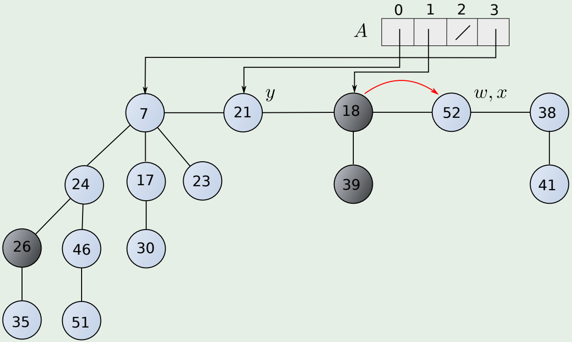
Example of consolidation

We move to the next root's child and point to it from A



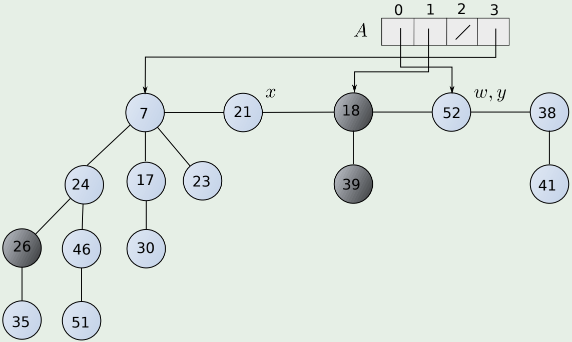
Example of consolidation

We move to the next root's child and point to it from A



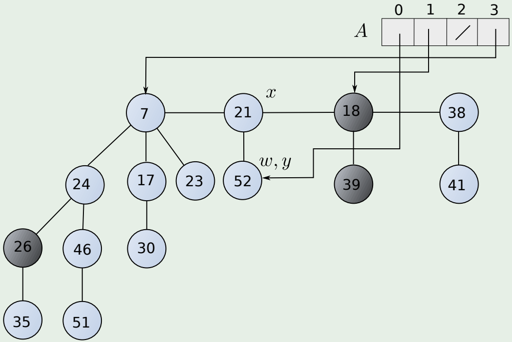
Example of consolidation

We point $y = A[0]$ then do the exchange between $x \longleftrightarrow y$



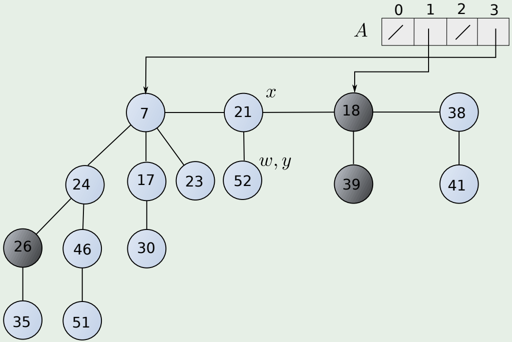
Example of consolidation

Link x and y



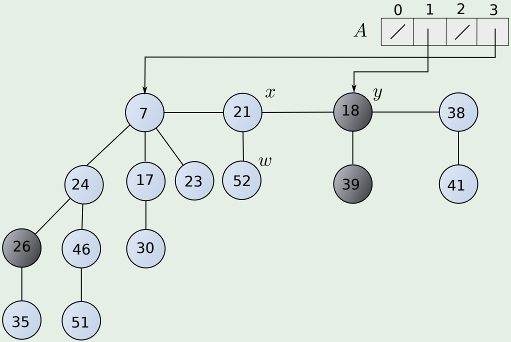
Example of consolidation

Make $A[d] = NIL$



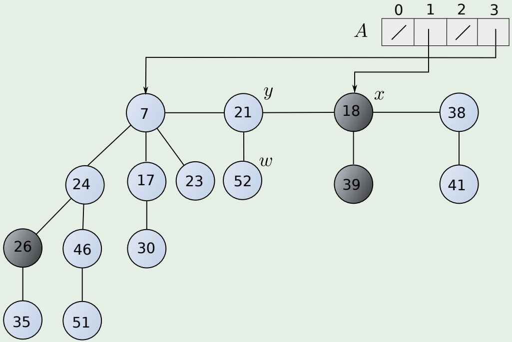
Example of consolidation

We make $y = A[1]$



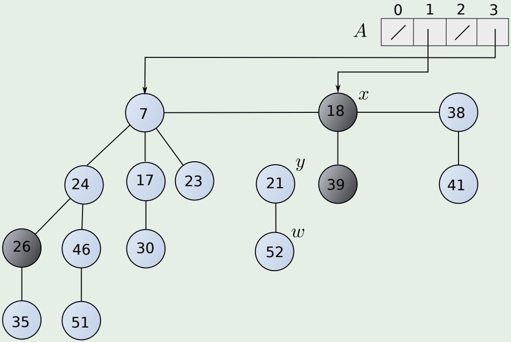
Example of consolidation

Do an exchange between $x \longleftrightarrow y$



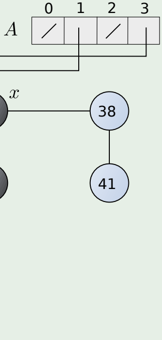
Example of consolidation

Remove y from the root's list



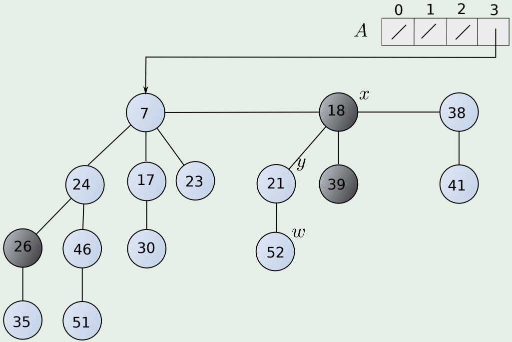
Example of consolidation

Make y a child of x



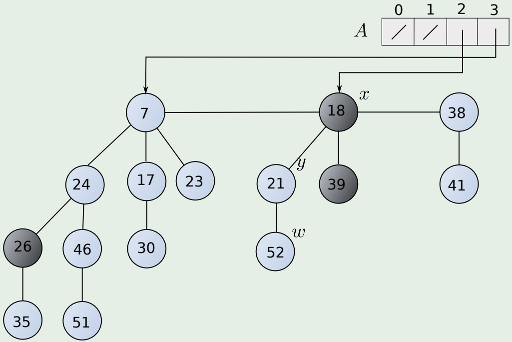
Example of consolidation

Make $A[1] = NIL$, then we make $d = d + 1$



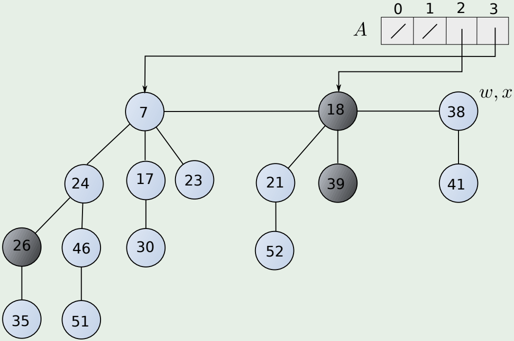
Example of consolidation

Because $A[2] = NIL$, then $A[2] = x$



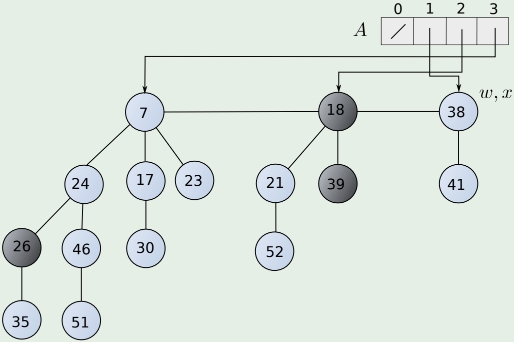
Example of consolidation

We move to the next w and make $x = w$



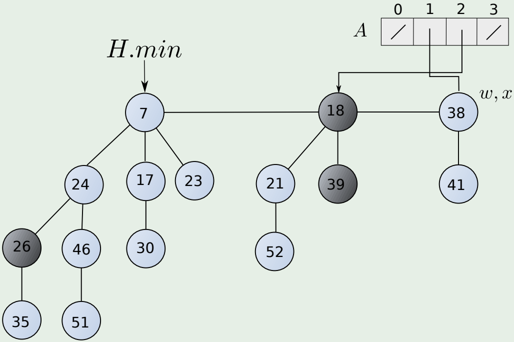
Example of consolidation

Because $A[1] = NIL$, then jump over the while loop and make $A[1] = x$



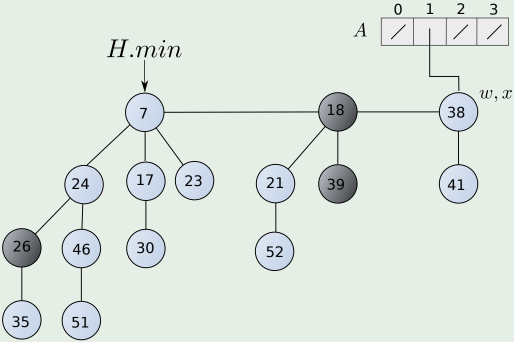
Example of consolidation

Because $A[1] \neq NIL$ insert into the root's list and because $H.min = NIL$, it is the first node in it



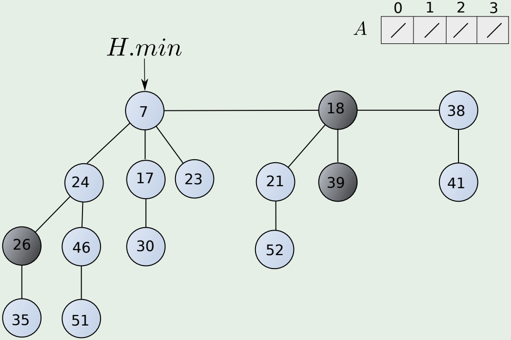
Example of consolidation

Because $A\left[2\right] \neq NIL$ insert into the root's list no exchange of min because $A\left[2\right].key > H.min.key$



Example of consolidation

Because $A[3] \neq NIL$ insert into the root's list no exchange of min because $A[3].key > H.min.key$



Outline

Cost of Extract-min

Amortized analysis observations

The cost of FIB-EXTRACT-MIN contributes at most $O\left(D\left(n\right)\right)$ because

1. The for loop at lines 3 to 5 in the code FIB-EXTRACT-MIN.
2. for loop at lines 2-3 and 16-23 of CONSOLIDATE.

Cost of Extract-min

Amortized analysis observations

The cost of FIB-EXTRACT-MIN contributes at most $O\left(D\left(n\right)\right)$ because

1. The for loop at lines 3 to 5 in the code FIB-EXTRACT-MIN.
2. for loop at lines 2-3 and 16-23 of CONSOLIDATE.

Cost of Extract-min

Amortized analysis observations

The cost of FIB-EXTRACT-MIN contributes at most $O(D(n))$ because

1. The for loop at lines 3 to 5 in the code FIB-EXTRACT-MIN.
2. for loop at lines 2-3 and 16-23 of CONSOLIDATE.

Cost of Extract-min

Amortized analysis observations

The cost of FIB-EXTRACT-MIN contributes at most $O\left(D\left(n\right)\right)$ because

1. The for loop at lines 3 to 5 in the code FIB-EXTRACT-MIN.
2. for loop at lines 2-3 and 16-23 of CONSOLIDATE.

Next

We have that

The size of the root list when calling Consolidate is at most

$$D(n) + t(H) - 1 \tag{1}$$

because the min root was extracted an it has at most $D(n)$ children.

Next

We have that

The size of the root list when calling Consolidate is at most

$$D(n) + t(H) - 1 \tag{1}$$

because the min root was extracted an it has at most $D(n)$ children.

Next

We have that

The size of the root list when calling Consolidate is at most

$$D(n) + t(H) - 1 \tag{1}$$

because the min root was extracted an it has at most $D(n)$ children.

Next

Then

At lines 4 to 14 in the CONSOLIDATE code:

- The amount of work done by the **for** and the **while** loop is proportional to $D(n) + t(H)$ because each time we go through an element in the root list (for loop).
- The while loop consolidate the tree pointed by the pointer to a tree x with same degree.

Next

Then

At lines 4 to 14 in the CONSOLIDATE code:

- The amount of work done by the **for** and the **while** loop is proportional to $D(n) + t(H)$ because each time we go through an element in the root list (for loop).
- The while loop consolidate the tree pointed by the pointer to a tree x with same degree.

The Actual Cost is

Then, the actual cost is $c_i = O\left(D\left(n\right) + t\left(H\right)\right)$.

Next

Then

At lines 4 to 14 in the CONSOLIDATE code:

- The amount of work done by the **for** and the **while** loop is proportional to $D(n) + t(H)$ because each time we go through an element in the root list (for loop).
- The while loop consolidate the tree pointed by the pointer to a tree x with same degree.

The Actual Cost is

Then, the actual cost is $c_i = O\left(D\left(n\right) + t\left(H\right)\right)$.

Next

Then

At lines 4 to 14 in the CONSOLIDATE code:

- The amount of work done by the **for** and the **while** loop is proportional to $D(n) + t(H)$ because each time we go through an element in the root list (for loop).
- The while loop consolidate the tree pointed by the pointer to a tree x with same degree.

The Actual Cost is

Then, the actual cost is $c_i = O\left(D\left(n\right) + t\left(H\right)\right)$.

Potential cost

Thus, assuming that H' is the new heap and H is the old one

- $\Phi\left(H\right) = t\left(H\right) + 2 \cdot m\left(H\right)$.
- $\Phi\left(H'\right) = D\left(n\right) + 1 + 2 \cdot m\left(H\right)$ because
  - H' has at most $D(n) + 1$ elements after consolidation
  - No node is marked in the process.

Potential cost

Thus, assuming that H' is the new heap and H is the old one

- $\Phi\left(H\right)=t\left(H\right)+2\cdot m\left(H\right)$.
- $\Phi\left(H'\right)=D\left(n\right)+1+2\cdot m\left(H\right)$ because
  - H' has at most $D(n)+1$ elements after consolidation
  - No node is marked in the process.

Potential cost

Thus, assuming that H' is the new heap and H is the old one

- $\Phi\left(H\right) = t\left(H\right) + 2 \cdot m\left(H\right)$.
- $\Phi\left(H'\right) = D\left(n\right) + 1 + 2 \cdot m\left(H\right)$ because
  - H' has at most $D(n) + 1$ elements after consolidation
  - No node is marked in the process.

Potential cost

Thus, assuming that H' is the new heap and H is the old one

- $\Phi(H) = t(H) + 2 \cdot m(H)$.
- $\Phi(H') = D(n) + 1 + 2 \cdot m(H)$ because
  - H' has at most $D(n) + 1$ elements after consolidation
  - No node is marked in the process.

Potential cost

The Final Potential Cost is

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi\left(H'\right) - \Phi\left(H\right)\\ &= O\left(D\left(n\right) + t\left(H\right)\right) + D\left(n\right) + 1 + 2\cdot m\left(H\right) - t\left(H\right) - 2\cdot m\left(H\right)\\ &= O\left(D\left(n\right) + t\left(H\right)\right) - t\left(H\right)\\ &= O\left(D\left(n\right)\right).\end{aligned}$$

Potential cost

The Final Potential Cost is

$$
\begin{aligned}
\widehat{c}_i &= c_i + \Phi\left(H'\right) - \Phi\left(H\right)\\
&= O\left(D\left(n\right) + t\left(H\right)\right) + D\left(n\right) + 1 + 2\cdot m\left(H\right) - t\left(H\right) - 2\cdot m\left(H\right)\\
&= O\left(D\left(n\right) + t\left(H\right)\right) - t\left(H\right)\\
&= O\left(D\left(n\right)\right).
\end{aligned}
$$

We shall see that

$D(n) = O(\log n)$

Potential cost

The Final Potential Cost is

$$\begin{aligned}\widehat{c}_i &= c_i + \Phi\left(H'\right) - \Phi\left(H\right)\\ &= O\left(D\left(n\right) + t\left(H\right)\right) + D\left(n\right) + 1 + 2\cdot m\left(H\right) - t\left(H\right) - 2\cdot m\left(H\right)\\ &= O\left(D\left(n\right) + t\left(H\right)\right) - t\left(H\right)\\ &= O\left(D\left(n\right)\right).\end{aligned}$$

We shall see that

$D(n) = O(\log n)$

Potential cost

The Final Potential Cost is

$$\begin{aligned}
\widehat{c}_i &= c_i + \Phi\left(H'\right) - \Phi\left(H\right) \\
&= O\left(D\left(n\right) + t\left(H\right)\right) + D\left(n\right) + 1 + 2 \cdot m\left(H\right) - t\left(H\right) - 2 \cdot m\left(H\right) \\
&= O\left(D\left(n\right) + t\left(H\right)\right) - t\left(H\right) \\
&= O\left(D\left(n\right)\right),
\end{aligned}$$

We shall see that

$D(n) = O(\log n)$

Outline

Decreasing a key

Fib-Heap-Decrease-Key(H, x, k)

```
if $k > x.key$
        error "new key is greater than current key"
$x.key = k$
$y = x.p$
if $y \neq NIL$ and $x.key < y.key$
        CUT$(H, x, y)$
        Cascading-Cut$(H, y)$
if $x.key < H.min.key$
        $H.min = x$
```

Decreasing a key

Fib-Heap-Decrease-Key(H, x, k)

```
if $k > x.key$
        error "new key is greater than current key"
$x.key = k$
$y = x.p$
if $y \neq NIL$ and $x.key < y.key$
        CUT$(H, x, y)$
        Cascading-Cut$(H, y)$
if $x.key < H.min.key$
        $H.min = x$
```

Decreasing a key

Fib-Heap-Decrease-Key(H, x, k)

```
if $k > x.key$
        error "new key is greater than current key"
$x.key = k$
$y = x.p$
if $y \neq NIL$ and $x.key < y.key$
        CUT$(H, x, y)$
        Cascading-Cut$(H, y)$
if $x.key < H.min.key$
        $H.min = x$
```

Decreasing a key

Fib-Heap-Decrease-Key(H, x, k)

```
if $k > x.key$
        error "new key is greater than current key"
$x.key = k$
$y = x.p$
if $y \neq NIL$ and $x.key < y.key$
        CUT$(H, x, y)$
        Cascading-Cut$(H, y)$
if $x.key < H.min.key$
        $H.min = x$
```

Explanation

First

1. Lines 1–3 ensure that the new key is no greater than the current key of x and then assign the new key to x.
2. If x is not a root and if $x.key \leq y.key$, where y is x's parent, then CUT and CASCADING-CUT are triggered.

Explanation

First

1. Lines 1–3 ensure that the new key is no greater than the current key of x and then assign the new key to x.
2. If x is not a root and if $x.key \leq y.key$, where y is x's parent, then CUT and CASCADING-CUT are triggered.

Decreasing a key (continuation - cascade cutting)

Be lazy to remove keys!

```
Cut($H, x, y$)
    Remove $x$ from the child list of $y$, decreasing $y.degree$
    Add $x$ to the root list of $H$
    $x.p = NIL$
    $x.mark = FALSE$
```

```
Cascading-Cut($H, y$)
    $z = y.p$
    if $z \neq NIL$
        if y.mark == FALSE
            y.mark=TRUE
        else
            Cut($H, y, z$)
            Cascading-Cut($H, y$)($H, z$)
```

Decreasing a key (continuation - cascade cutting)

Be lazy to remove keys!

```
Cut($H, x, y$)
  1. Remove $x$ from the child list of $y$, decreasing $y.degree$
  2. Add $x$ to the root list of $H$
  3. $x.p = NIL$
  4. $x.mark = FALSE$
```

```
Cascading-Cut($H, y$)
  1. $z = y.p$
  2. if $z \neq NIL$
  3.     if y.mark == FALSE
  4.         y.mark=TRUE
  5.     else
  6.         Cut($H, y, z$)
  7.         Cascading-Cut($H, y$)($H, z$)
```

Explanation

Second

Then CUT simply removes x from the child-list of y.

Thus

The CASCADING-CUT uses the mark attributes to obtain the desired time bounds.

Explanation

Second

Then CUT simply removes x from the child-list of y.

Thus

The CASCADING-CUT uses the mark attributes to obtain the desired time bounds.

Explanation

The mark label records the following events that happened to y:

1. At some time, y was converted into an element of the root list.
2. Then, y was linked to (made the child of) another node.
3. Then, two children of y were removed by cuts.

Explanation

The mark label records the following events that happened to y:

1. At some time, y was converted into an element of the root list.
2. Then, y was linked to (made the child of) another node.
3. Then, two children of y were removed by cuts.

As soon as the second child has been lost, we cut y from its parent, making it a new root.

- The attribute $y.mark$ is TRUE if steps 1 and 2 have occurred and one child of y has been cut.
- The CUT procedure, therefore, clears $y.mark$ in line 4, since it performs step 1.
- We can now see why line 3 of FIB-HEAP-LINK clears $y.mark$.

Explanation

The mark label records the following events that happened to y:

1. At some time, y was converted into an element of the root list.
2. Then, y was linked to (made the child of) another node.
3. Then, two children of y were removed by cuts.

As soon as the second child has been lost, we cut y from its parent, making it a new root.

- The attribute $y.mark$ is TRUE if steps 1 and 2 have occurred and one child of y has been cut.
- The CUT procedure, therefore, clears $y.mark$ in line 4, since it performs step 1.
- We can now see why line 3 of FIB-HEAP-LINK clears $y.mark$.

Explanation

The mark label records the following events that happened to y:

1. At some time, y was converted into an element of the root list.
2. Then, y was linked to (made the child of) another node.
3. Then, two children of y were removed by cuts.

As soon as the second child has been lost, we cut y from its parent, making it a new root.

- The attribute $y.mark$ is TRUE if steps 1 and 2 have occurred and one child of y has been cut.
- The CUT procedure, therefore, clears $y.mark$ in line 4, since it performs step 1.
- We can now see why line 3 of FIB-HEAP-LINK clears $y.mark$.

Explanation

The mark label records the following events that happened to y:

1. At some time, y was converted into an element of the root list.
2. Then, y was linked to (made the child of) another node.
3. Then, two children of y were removed by cuts.

As soon as the second child has been lost, we cut y from its parent, making it a new root.

- The attribute $y.mark$ is TRUE if steps 1 and 2 have occurred and one child of y has been cut.
- The CUT procedure, therefore, clears $y.mark$ in line 4, since it performs step 1.
- We can now see why line 3 of FIB-HEAP-LINK clears $y.mark$.

Explanation

The mark label records the following events that happened to y:

1. At some time, y was converted into an element of the root list.
2. Then, y was linked to (made the child of) another node.
3. Then, two children of y were removed by cuts.

As soon as the second child has been lost, we cut y from its parent, making it a new root.

- The attribute $y.mark$ is TRUE if steps 1 and 2 have occurred and one child of y has been cut.
- The CUT procedure, therefore, clears $y.mark$ in line 4, since it performs step 1.
- We can now see why line 3 of FIB-HEAP-LINK clears $y.mark$.

Explanation

Now we have a new problem

- x might be the second child cut from its parent y to another node.
    - Therefore, in line 7 of FIB-HEAP-DECREASE attempts to perform a cascading-cut on y.

Explanation

Now we have a new problem

- x might be the second child cut from its parent y to another node.
  - Therefore, in line 7 of FIB-HEAP-DECREASE attempts to perform a cascading-cut on y.

We have three cases:

1. If y is a root return.
2. if y is not a root and it is unmarked then y is marked.
3. If y is not a root and it is marked, then y is CUT and a cascading cut is performed in its parent z.

Explanation

Now we have a new problem

- x might be the second child cut from its parent y to another node.
    - Therefore, in line 7 of FIB-HEAP-DECREASE attempts to perform a cascading-cut on y.

We have three cases:

1. If y is a root return.
2. if y is not a root and it is unmarked then y is marked.
3. If y is not a root and it is marked, then y is CUT and a cascading cut is performed in its parent z.

Once all the cascading cuts are done

the $H.min$ is updated if necessary

Explanation

Now we have a new problem

- x might be the second child cut from its parent y to another node.
    - Therefore, in line 7 of FIB-HEAP-DECREASE attempts to perform a cascading-cut on y.

We have three cases:

1. If y is a root return.
2. if y is not a root and it is unmarked then y is marked.
3. If y is not a root and it is marked, then y is CUT and a cascading cut is performed in its parent z.

Once all the cascading cuts are done

the $H.min$ is updated if necessary

Explanation

Now we have a new problem

- x might be the second child cut from its parent y to another node.
  - Therefore, in line 7 of FIB-HEAP-DECREASE attempts to perform a cascading-cut on y.

We have three cases:

1. If y is a root return.
2. if y is not a root and it is unmarked then y is marked.
3. If y is not a root and it is marked, then y is CUT and a cascading cut is performed in its parent z.

Once all the cascading cuts are done

the $H.min$ is updated if necessary

Explanation

Now we have a new problem

- x might be the second child cut from its parent y to another node.
    - Therefore, in line 7 of FIB-HEAP-DECREASE attempts to perform a cascading-cut on y.

We have three cases:

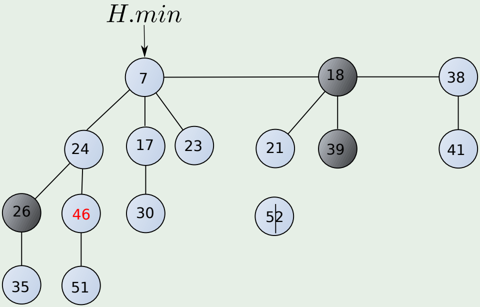
1. If y is a root return.
2. if y is not a root and it is unmarked then y is marked.
3. If y is not a root and it is marked, then y is CUT and a cascading cut is performed in its parent z.

Once all the cascading cuts are done

the $H.min$ is updated if necessary

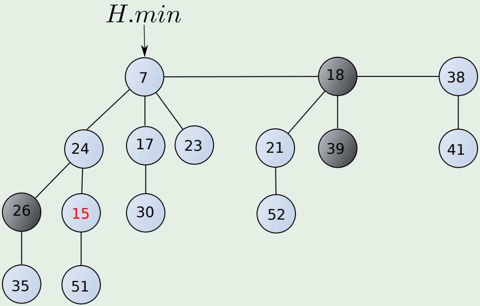
Example

46 is decreased to 15



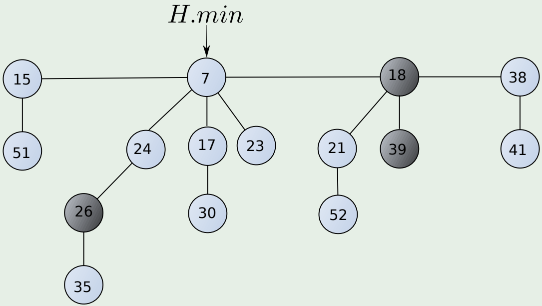
Example

46 is decreased to 15



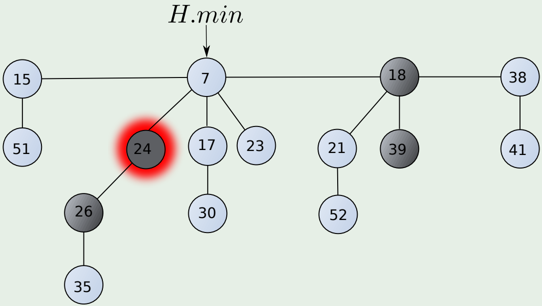
Example

Cut 15 to the root's list



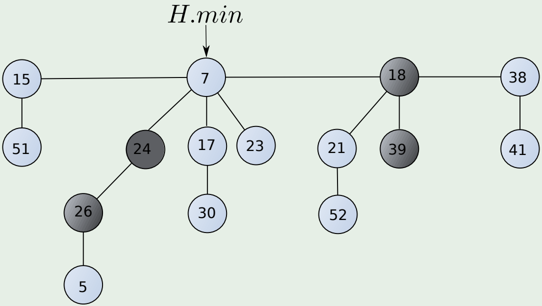
Example

Mark 24 to *True*



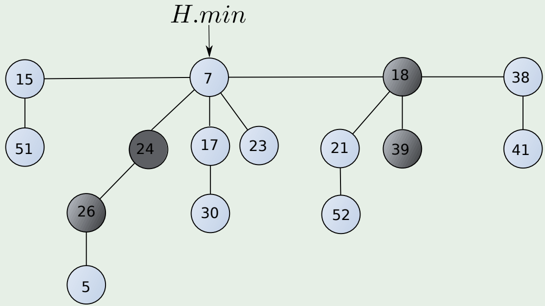
Example

Then 35 is decreased to 5



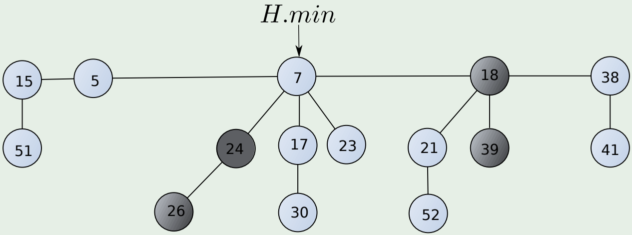
Example

Cut 15 to the root's list



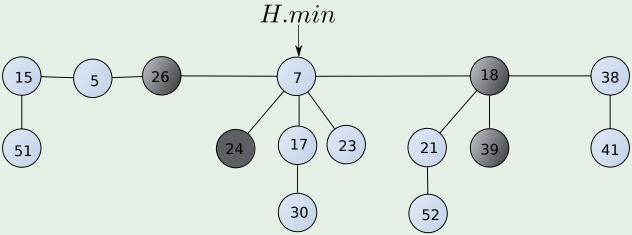
Example

Initiate cascade cutting



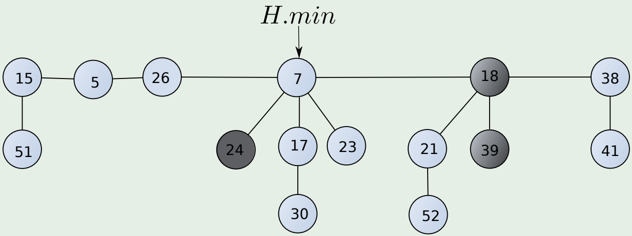
Example

Initiate cascade cutting moving 26 to the root's list



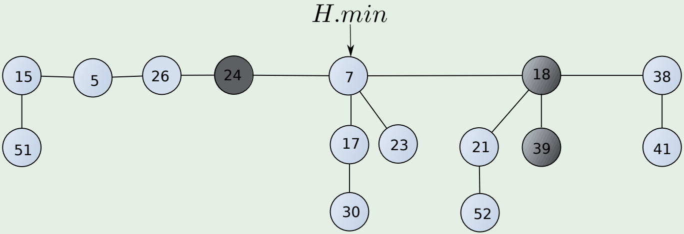
Example

Mark 26 to false



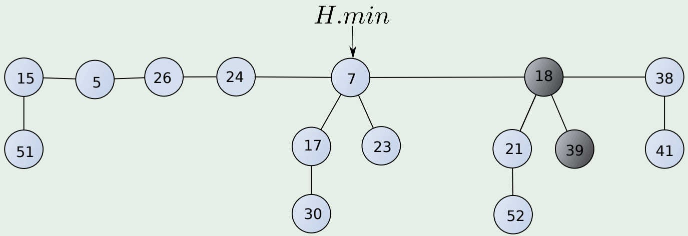
Example

Move 24 to root's list



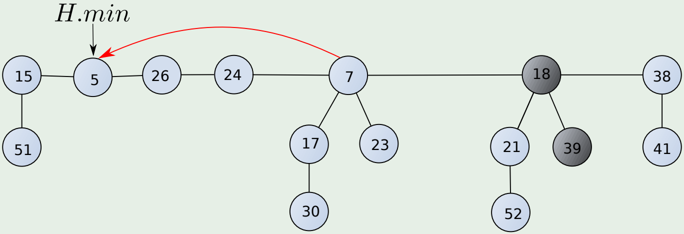
Example

Mark 26 to false



Example

Change the $H.min$



Potential cost

The procedure Decrease-Key takes

- $c_i = O(1)+$the cascading-cuts

Potential cost

The procedure Decrease-Key takes

- $c_i = O(1)+$the cascading-cuts

Assume that you require c calls to cascade CASCADING-CUT

- One for the line 6 at the code FIB-HEAP-DECREASE-KEY followed by $c-1$ others
  - The cost of it will take $c_i = O(c)$.

Potential cost

The procedure Decrease-Key takes

- $c_i = O(1)$+the cascading-cuts

Assume that you require c calls to cascade CASCADING-CUT

- One for the line 6 at the code FIB-HEAP-DECREASE-KEY followed by $c-1$ others
  - The cost of it will take $c_i = O(c)$.

Potential cost

Finally, assuming H' is the new Fibonacci Heap and H the old one

$$\Phi\left(H'\right) = \left(t\left(H\right) + c\right) + 2\left(m\left(H\right) - \left(c-1\right) + 1\right) \tag{2}$$

Where:

- $t\left(H\right) + c$
    - The # original trees + the ones created by the c calls.
- $m\left(H\right) - \left(c-1\right) + 1$
    - The original marks - $\left(c-1\right)$ cleared marks by Cut + the branch to $y.mark == FALSE$ true.

Potential cost

Finally, assuming H' is the new Fibonacci Heap and H the old one

$$\Phi\left(H'\right) = \left(t\left(H\right) + c\right) + 2\left(m\left(H\right) - \left(c-1\right) + 1\right) \tag{2}$$

Where:

- $t\left(H\right) + c$
    - The # original trees + the ones created by the c calls.
- $m\left(H\right) - \left(c-1\right) + 1$
    - The original marks - $\left(c-1\right)$ cleared marks by Cut + the branch to $y.mark == FALSE$ true.

Potential cost

Finally, assuming H' is the new Fibonacci Heap and H the old one

$$\Phi\left(H'\right) = \left(t\left(H\right) + c\right) + 2\left(m\left(H\right) - \left(c - 1\right) + 1\right) \tag{2}$$

Where:

- $t\left(H\right) + c$
  - The # original trees + the ones created by the c calls.
- $m\left(H\right) - \left(c - 1\right) + 1$
  - The original marks - $(c - 1)$ cleared marks by Cut + the branch to $y.mark == FALSE$ true.

Potential cost

Finally, assuming H' is the new Fibonacci Heap and H the old one

$$\Phi\left(H'\right) = \left(t\left(H\right) + c\right) + 2\left(m\left(H\right) - \left(c - 1\right) + 1\right) \tag{2}$$

Where:

- $t\left(H\right) + c$
  - The # original trees + the ones created by the c calls.
- $m\left(H\right) - \left(c - 1\right) + 1$
  - The original marks - $\left(c - 1\right)$ cleared marks by Cut + the branch to $y.mark == FALSE$ true.

Final change in potential

Thus

$$\Phi\left(H'\right) = t\left(H\right) + c + 2(m(H) - c + 2) = t\left(H\right) + 2m\left(H\right) - c + 4 \quad (3)$$

Final change in potential

Thus

$$\Phi\left(H'\right) = t\left(H\right) + c + 2(m(H) - c + 2) = t\left(H\right) + 2m\left(H\right) - c + 4 \quad (3)$$

Then, we have that the amortized cost is

$$\begin{aligned} \widehat{c}_i &= c_i + t\left(H\right) + 2m\left(H\right) - c + 4 - t\left(H\right) - 2m\left(H\right) \\ &= c_i + 4 - c = O(c) + 4 - c = O(1) \end{aligned}$$

Now we can see why the term $2m\left(H\right)$:

- One unit to pay for the cut and clearing the marking.
- One unit for making of a node a root.

Final change in potential

Thus

$$\Phi\left(H'\right) = t\left(H\right) + c + 2(m(H) - c + 2) = t\left(H\right) + 2m\left(H\right) - c + 4 \quad (3)$$

Then, we have that the amortized cost is

$$\begin{aligned} \widehat{c}_i &= c_i + t\left(H\right) + 2m\left(H\right) - c + 4 - t\left(H\right) - 2m\left(H\right) \\ &= c_i + 4 - c = O(c) + 4 - c = O(1) \end{aligned}$$

Observation

Now we can see why the term $2m(H)$:

- One unit to pay for the cut and clearing the marking.
- One unit for making of a node a root.

Final change in potential

Thus

$$\Phi\left(H'\right) = t\left(H\right) + c + 2(m(H) - c + 2) = t\left(H\right) + 2m\left(H\right) - c + 4 \tag{3}$$

Then, we have that the amortized cost is

$$\begin{aligned} \widehat{c}_i &= c_i + t\left(H\right) + 2m\left(H\right) - c + 4 - t\left(H\right) - 2m\left(H\right) \\ &= c_i + 4 - c = O(c) + 4 - c = O(1) \end{aligned}$$

Observation

Now we can see why the term $2m(H)$:

- One unit to pay for the cut and clearing the marking.
- One unit for making of a node a root.

Final change in potential

Thus

$$\Phi\left(H'\right) = t\left(H\right) + c + 2(m(H) - c + 2) = t\left(H\right) + 2m\left(H\right) - c + 4 \quad (3)$$

Then, we have that the amortized cost is

$$\begin{aligned} \widehat{c}_i &= c_i + t\left(H\right) + 2m\left(H\right) - c + 4 - t\left(H\right) - 2m\left(H\right) \\ &= c_i + 4 - c = O(c) + 4 - c = O(1) \end{aligned}$$

Observation

Now we can see why the term $2m(H)$:

- One unit to pay for the cut and clearing the marking.
- One unit for making of a node a root.

Final change in potential

Thus

$$\Phi\left(H'\right) = t\left(H\right) + c + 2(m(H) - c + 2) = t\left(H\right) + 2m\left(H\right) - c + 4 \quad (3)$$

Then, we have that the amortized cost is

$$\begin{aligned}\widehat{c}_i &= c_i + t\left(H\right) + 2m\left(H\right) - c + 4 - t\left(H\right) - 2m\left(H\right)\\ &= c_i + 4 - c = O(c) + 4 - c = O(1)\end{aligned}$$

Observation

Now we can see why the term $2m(H)$:

- One unit to pay for the cut and clearing the marking.
- One unit for making of a node a root.

Delete a node

It is easy to delete a node in the Fibonacci heap following the next code

```
Fib-Heap-Delete($H, x$)
  1 Fib-Heap-Decrease-Key($H, x, -\infty$)
  2 Fib-Heap-Extract-Min($H$)
```

Again, the cost is $O(D(n))$

Proving the $D(n)$ bound!!!

Let's define the following

We define a quantity $size(x)$= the number of nodes at subtree rooted at x, x itself.

- $x.degree$ is always maintained as an accurate count of the degree of

Proving the $D(n)$ bound!!!

Let's define the following

We define a quantity $size(x)$= the number of nodes at subtree rooted at x, x itself.

The key to the analysis is as follows:

- We shall show that $size(x)$ is exponential in $x.degree$.
- $x.degree$ is always maintained as an accurate count of the degree of x.

Proving the $D(n)$ bound!!!

Let's define the following

We define a quantity $size(x)$= the number of nodes at subtree rooted at x, x itself.

The key to the analysis is as follows:

- We shall show that $size(x)$ is exponential in $x.degree$.
- $x.degree$ is always maintained as an accurate count of the degree of x.

Now...

Lemma 19.1

Let x be any node in a Fibonacci heap, and suppose that $x.degree = k$. Let $y_1, y_2, ..., y_k$ denote the children of x in the order in which they were linked to x, from the earliest to the latest. Then $y_1.degree \geq 0$ and $y_i.degree \geq i - 2$ for $i = 2, 3, ..., k$.

Proof

1. Obviously, $y_1.degree \geq 0$.

Now...

Lemma 19.1

Let x be any node in a Fibonacci heap, and suppose that $x.degree = k$. Let $y_1, y_2, ..., y_k$ denote the children of x in the order in which they were linked to x, from the earliest to the latest. Then $y_1.degree \geq 0$ and $y_i.degree \geq i - 2$ for $i = 2, 3, ..., k$.

Proof

1. Obviously, $y_1.degree \geq 0$.
2. For $i \geq 2$, y_i was linked to x, all of $y_1, y_2, ..., y_{i-1}$were children of x, so we **must have had** $x.degree \geq i - 1$.
3. Node y_i is linked to x only if we had $x.degree = y_i.degree$, so we must have also had $y_i.degree \geq i - 1$ when node y_i was linked to x.
4. Since then, node y_i has lost at most one child.
    - Note: It would have been cut from x if it had lost two children.
5. We conclude that $y_i.degree \geq i - 2$.

Now...

Lemma 19.1

Let x be any node in a Fibonacci heap, and suppose that $x.degree = k$. Let $y_1, y_2, ..., y_k$ denote the children of x in the order in which they were linked to x, from the earliest to the latest. Then $y_1.degree \geq 0$ and $y_i.degree \geq i - 2$ for $i = 2, 3, ..., k$.

Proof

1. Obviously, $y_1.degree \geq 0$.
2. For $i \geq 2$, y_i was linked to x, all of $y_1, y_2, ..., y_{i-1}$were children of x, so we **must have had** $x.degree \geq i - 1$.
3. Node y_i is linked to x only if we had $x.degree = y_i.degree$, so we must have also had $y_i.degree \geq i - 1$ when node y_i was linked to x.
4. Since then, node y_i has lost at most one child.
    - Note: It would have been cut from x if it had lost two children.
5. We conclude that $y_i.degree \geq i - 2$.

Now...

Lemma 19.1

Let x be any node in a Fibonacci heap, and suppose that $x.degree = k$. Let $y_1, y_2, ..., y_k$ denote the children of x in the order in which they were linked to x, from the earliest to the latest. Then $y_1.degree \geq 0$ and $y_i.degree \geq i - 2$ for $i = 2, 3, ..., k$.

Proof

1. Obviously, $y_1.degree \geq 0$.
2. For $i \geq 2$, y_i was linked to x, all of $y_1, y_2, ..., y_{i-1}$were children of x, so we **must have had** $x.degree \geq i - 1$.
3. Node y_i is linked to x only if we had $x.degree = y_i.degree$, so we must have also had $y_i.degree \geq i - 1$ when node y_i was linked to x.
4. Since then, node y_i has lost at most one child.
    - Note: It would have been cut from x if it had lost two children.
5. We conclude that $y_i.degree \geq i - 2$.

Now...

Lemma 19.1

Let x be any node in a Fibonacci heap, and suppose that $x.degree = k$. Let $y_1, y_2, ..., y_k$ denote the children of x in the order in which they were linked to x, from the earliest to the latest. Then $y_1.degree \geq 0$ and $y_i.degree \geq i - 2$ for $i = 2, 3, ..., k$.

Proof

1. Obviously, $y_1.degree \geq 0$.
2. For $i \geq 2$, y_i was linked to x, all of $y_1, y_2, ..., y_{i-1}$were children of x, so we **must have had** $x.degree \geq i - 1$.
3. Node y_i is linked to x only if we had $x.degree = y_i.degree$, so we must have also had $y_i.degree \geq i - 1$ when node y_i was linked to x.
4. Since then, node y_i has lost at most one child.
    - Note: It would have been cut from x if it had lost two children.
5. We conclude that $y_i.degree \geq i - 2$.

Now...

Lemma 19.1

Let x be any node in a Fibonacci heap, and suppose that $x.degree = k$. Let $y_1, y_2, ..., y_k$ denote the children of x in the order in which they were linked to x, from the earliest to the latest. Then $y_1.degree \geq 0$ and $y_i.degree \geq i-2$ for $i = 2, 3, ..., k$.

Proof

1. Obviously, $y_1.degree \geq 0$.
2. For $i \geq 2$, y_i was linked to x, all of $y_1, y_2, ..., y_{i-1}$were children of x, so we **must have had** $x.degree \geq i-1$.
3. Node y_i is linked to x only if we had $x.degree = y_i.degree$, so we must have also had $y_i.degree \geq i-1$ when node y_i was linked to x.
4. Since then, node y_i has lost at most one child.
    - Note: It would have been cut from x if it had lost two children.
5. We conclude that $y_i.degree \geq i-2$.

Outline

1. Introduction
   - Basic Definitions
   - Ordered Trees
2. Binomial Trees
   - Example
3. Fibonacci Heap
   - Operations
   - Fibonacci Heap
   - Why Fibonacci Heaps?
   - Node Structure
   - Fibonacci Heaps Operations
     - Mergeable-Heaps operations - Make Heap
     - Mergeable-Heaps operations - Insertion
     - Mergeable-Heaps operations - Minimum
     - Mergeable-Heaps operations - Union
   - Complexity Analysis
   - Consolidate Algorithm
   - Potential cost
   - Operation: Decreasing a Key
   - Why Fibonacci?
4. Exercises
   - Some Exercises that you can try

Why Fibonacci?

The kth Fibonacci number is defined by the recurrence

$$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

Lemma 19.2

For al integers $k \geq 0$

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

Why Fibonacci?

The kth Fibonacci number is defined by the recurrence

$$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

Lemma 19.2

For al integers $k \geq 0$

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

Proof by induction $k = 0$....

The use of the Golden Ratio

Lemma 19.3

Let x be any node in a Fibonacci heap, and let $k = x.degree$. Then, $F_{k+2} \geq \Phi^k$, where $\Phi = \frac{1+\sqrt{5}}{2}$ (The golden ratio).

Golden Ratio

Building the Golden Ratio

1. Construct a unit square.
2. Draw a line from the midpoint of one side to an opposite corner.
3. Use that line as a radius for a circle to define a rectangle.

Golden Ratio

Building the Golden Ratio

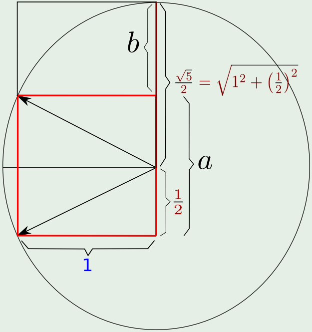
1. Construct a unit square.
2. Draw a line from the midpoint of one side to an opposite corner.
3. Use that line as a radius for a circle to define a rectangle.

Golden Ratio

Building the Golden Ratio

1. Construct a unit square.
2. Draw a line from the midpoint of one side to an opposite corner.
3. Use that line as a radius for a circle to define a rectangle.

$$\frac{a+b}{a} = \frac{a}{b} = \varphi \quad a > b$$



The use of the Golden Ratio

Lemma 19.4

Let x be any node in a Fibonacci Heap, $k = x.degree$. Then $size(x) \geq F_{k+2} \geq \Phi^k$.

- Restrictions for s_k:
  - $s_k \leq size(x)$ and $s_k \leq s_{k+1}$ (monotonically increasing)

The use of the Golden Ratio

Lemma 19.4

Let x be any node in a Fibonacci Heap, $k = x.degree$. Then $size(x) \geq F_{k+2} \geq \Phi^k$.

Proof

- Let s_k denote the minimum value for $size(x)$ over all nodes x such that $x.degree = k$.
- Restrictions for s_k:
  - $s_k \leq size(x)$ and $s_k \leq s_{k+1}$ (monotonically increasing).

The use of the Golden Ratio

Lemma 19.4

Let x be any node in a Fibonacci Heap, $k = x.degree$. Then $size(x) \geq F_{k+2} \geq \Phi^k$.

Proof

- Let s_k denote the minimum value for $size(x)$ over all nodes x such that $x.degree = k$.
- Restrictions for s_k:
  - $s_k \leq size(x)$ and $s_k \leq s_{k+1}$ (monotonically increasing).

As in the previous lemma

- $y_1, y_2, ..., y_k$ denote the children of x in the order they were linked to x.

The use of the Golden Ratio

Lemma 19.4

Let x be any node in a Fibonacci Heap, $k = x.degree$. Then $size(x) \geq F_{k+2} \geq \Phi^k$.

Proof

- Let s_k denote the minimum value for $size(x)$ over all nodes x such that $x.degree = k$.
- Restrictions for s_k:
  - $s_k \leq size(x)$ and $s_k \leq s_{k+1}$ (monotonically increasing).

As in the previous lemma

- $y_1, y_2, \ldots, y_k$ denote the children of x in the order they were linked to x.

The use of the Golden Ratio

Lemma 19.4

Let x be any node in a Fibonacci Heap, $k = x.degree$. Then $size(x) \geq F_{k+2} \geq \Phi^k$.

Proof

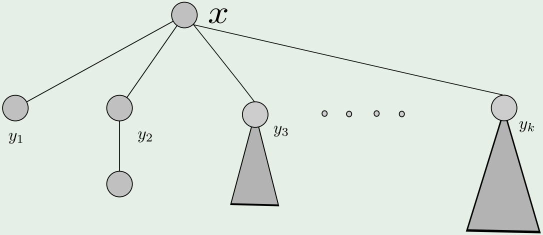
- Let s_k denote the minimum value for $size(x)$ over all nodes x such that $x.degree = k$.
- Restrictions for s_k:
    - $s_k \leq size(x)$ and $s_k \leq s_{k+1}$ (monotonically increasing).

As in the previous lemma

- $y_1, y_2, ..., y_k$ denote the children of x in the order they were linked to x.

Example

For example

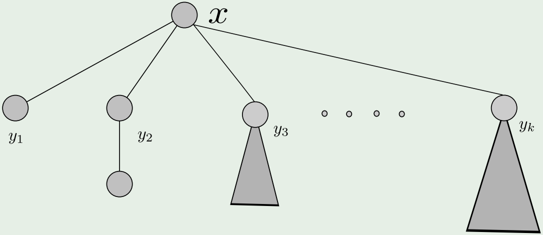


Now

Look at the board

Example

For example



Now

Look at the board

The use of the Golden Ratio

Proof continuation

- Now, we need to proof that $s_k \geq F_{k+2}$
- The cases for $k = 0 \rightarrow s_0 = 1$, and $k = 1 \rightarrow s_1 = 2$ are trivial because

$$F_2 = F_1 + F_0 = 1 + 0 = 1 \tag{4}$$

$$F_3 = F_2 + F_1 = 1 + 1 = 2 \tag{5}$$

The use of the Golden Ratio

Proof continuation

- Now, we need to proof that $s_k \geq F_{k+2}$
- The cases for $k = 0 \to s_0 = 1$, and $k = 1 \to s_1 = 2$ are trivial because

$$F_2 = F_1 + F_0 = 1 + 0 = 1 \tag{4}$$

$$F_3 = F_2 + F_1 = 1 + 1 = 2 \tag{5}$$

The use of the Golden Ratio

Proof continuation

- Now, we need to proof that $s_k \geq F_{k+2}$
- The cases for $k = 0 \to s_0 = 1$, and $k = 1 \to s_1 = 2$ are trivial because

$$F_2 = F_1 + F_0 = 1 + 0 = 1 \tag{4}$$

$$F_3 = F_2 + F_1 = 1 + 1 = 2 \tag{5}$$

Finally

Corollary 19.5

The maximum degree $D(n)$ of any node in an n-node Fibonacci heap is $O(\log n)$.

Proof

Look at the board!

Finally

Corollary 19.5

The maximum degree $D(n)$ of any node in an n-node Fibonacci heap is $O(\log n)$.

Proof

Look at the board!

Outline

Exercises

From Cormen's book solve the following

- 20.2-2
- 20.2-3
- 20.2-4
- 20.3-1
- 20.3-2
- 20.4-1
- 20.4-2