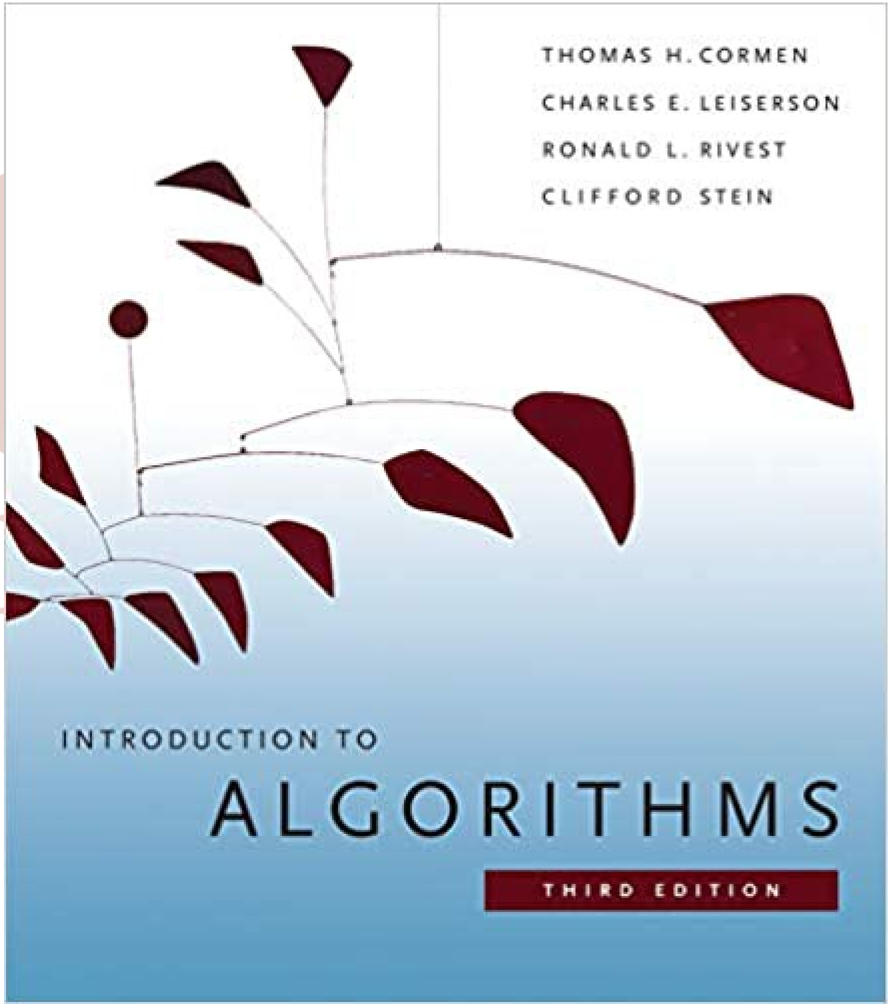
CS 07540 Advanced Design and Analysis of Algorithms

Week 11

- ADT Priority Queue
    - Data Structure Fibonacci Heap



Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms

MICHAEL L. FREDMAN

University of California, San Diego, La Jolla, California

AND

ROBERT ENDRE TARJAN

AT&T Bell Laboratories, Murray Hill, New Jersey

Abstract. In this paper we develop a new data structure for implementing heaps (priority queues). Our structure, *Fibonacci heaps* (abbreviated *F-heaps*), extends the binomial queues proposed by Vuillemin and studied further by Brown. F-heaps support arbitrary deletion from an n-item heap in $O(\log n)$ amortized time and all other standard heap operations in $O(1)$ amortized time. Using F-heaps we are able to obtain improved running times for several network optimization algorithms. In particular, we obtain the following worst-case bounds, where n is the number of vertices and m the number of edges in the problem graph:

(1) $O(n \log n + m)$ for the single-source shortest path problem with nonnegative edge lengths, improved from $O(m \log_{(m/n+2)} n)$;
(2) $O(n^2 \log n + nm)$ for the all-pairs shortest path problem, improved from $O(nm \log_{(m/n+2)} n)$;
(3) $O(n^2 \log n + nm)$ for the assignment problem (weighted bipartite matching), improved from $O(nm \log_{(m/n+2)} n)$;
(4) $O(m\beta(m, n))$ for the minimum spanning tree problem, improved from $O(m \log \log_{(m/n+2)} n)$, where $\beta(m, n) = \min \{i \mid \log^{(i)} n \leq m/n\}$. Note that $\beta(m, n) \leq \log^* n$ if $m \geq n$.

Of these results, the improved bound for minimum spanning trees is the most striking, although all the results give asymptotic improvements for graphs of appropriate densities.

Categories and Subject Descriptors: E.1 **[Data]**: Data Structures—*trees*; *graphs*; F.2.2 **[Analysis of Algorithms and Problem Complexity]**: Nonnumerical Algorithms and Problems—*computations on discrete structures*; *sorting and searching*; G.2.2 **[Discrete Mathematics]**: Graph Theory—*graph algorithms*; *network problems*; *trees*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Heap, matching, minimum spanning tree, priority queue, shortest path

Fibonacci Heaps

Fibonacci heaps are an implementation of the ADT priority queue. They are a mergeable heap data structure consisting of a collection of trees with the min heap (or max heap) property. Fibonacci heaps are named so due to the size of subtrees of a node of degree d being at least F_{d+2} (the $(d+2)^{\text{th}}$ Fibonacci number).

As mergeable heaps they support the following operations:

- **MAKE_HEAP**
- **INSERT(H, x)**
- **MINIMUM(H)**
- **DELETE_MIN(H)**
- **UNION(H1, H2)**

In addition, Fibonacci heaps support

- **DECREASE_KEY(H, x, k)**
- **DELETE(H,x)**

Fibonacci heaps have better asymptotic bounds than binary heaps for **INSERT**, **UNION**, and **DECREASE_KEY** (constant amortized time), and have the same asymptotic running times for the other operations. The price is a more complex structure.

Operation	Linked List	Binary Heap	Binomial Heap	Fibonacci Heap †	Relaxed Heap
make-heap	1	1	1	1	1
is-empty	1	1	1	1	1
insert	1	log *n*	log *n*	1	1
delete-min	*n*	log *n*	log *n*	log *n*	log *n*
decrease-key	*n*	log *n*	log *n*	1	1
delete	*n*	log *n*	log *n*	log *n*	log *n*
union	1	*n*	log *n*	1	1
find-min	*n*	1	log *n*	1	1

n = number of elements in priority queue

† amortized

This heap data structure was created by Fredman and Tarjan in 1984 with the goal of minimizing the number of operations needed to compute ***Minimum Spanning Trees*** (getting everywhere in a tree) and ***Shortest Paths*** (getting there fast) in the trees implementing heaps. These algorithms help improve the running times of **INSERT**, **DECREASE_KEY**, and **DELETE_MIN**. Asymptotically, they improved Dijkstra's SP algorithm from $O(E\log(V))$ to $O(E+V\log(V))$ where vertices are inserted and delete-min'ed and edges are used to decrease keys.

They achieved this goal with laziness: Do work only when we need to, and then simplify the data structure as much as possible so future work will be easy and fast. In contrast, binomial heaps consolidate trees after each **INSERT**.

Advantages of Fibonacci Heaps

Minimum spanning tree and shortest path algorithms rely on the **DECREASE_KEY** operation which runs in $\Theta(1)$ (amortized) in Fibonacci heaps. That is (asymptotically) a big improvement over Binomial Heaps which have $O(\log(n))$ (worst-case).

If we expect to have few **DEL_MIN** and **DELETE** operations (but many **DECREASE_KEY**) then almost all operations will be $\Theta(1)$ (amortized).

Dijkstra's algorithm and Prim/Kruskal's algorithm are graph algorithm examples where **DECREASE_KEY** is called for edges, which is a big improvement over Binary Heaps that have $\Theta(\log n)$ worst-case runtime.

Disadvantages of Fibonacci Heaps

Fibonacci heaps have a reputation for being slow in practical applications. Programming complexity is high compared to other structures we studied. The memory overhead per node is high compared to other priority queue implementations. There are also constant factors associated with amortized analysis. This leads to Fibonacci heaps to be primarily of theoretical interest. The **SEARCH** in Fibonacci Heaps is slow, so a pointer and a key are usually used as input to speed up the search.

However, an analysis titled "A Back-to-Basics Empirical Study of Priority Queues" in the *2014 Proceedings of the Meeting on Algorithm Engineering and Experiments* indicates that Fibonacci heaps are more efficient in practice than most of its later derivatives, such as quake heaps, violation heaps, strict Fibonacci heaps, rank pairing heaps, but less efficient than either pairing heaps or array-based heaps.

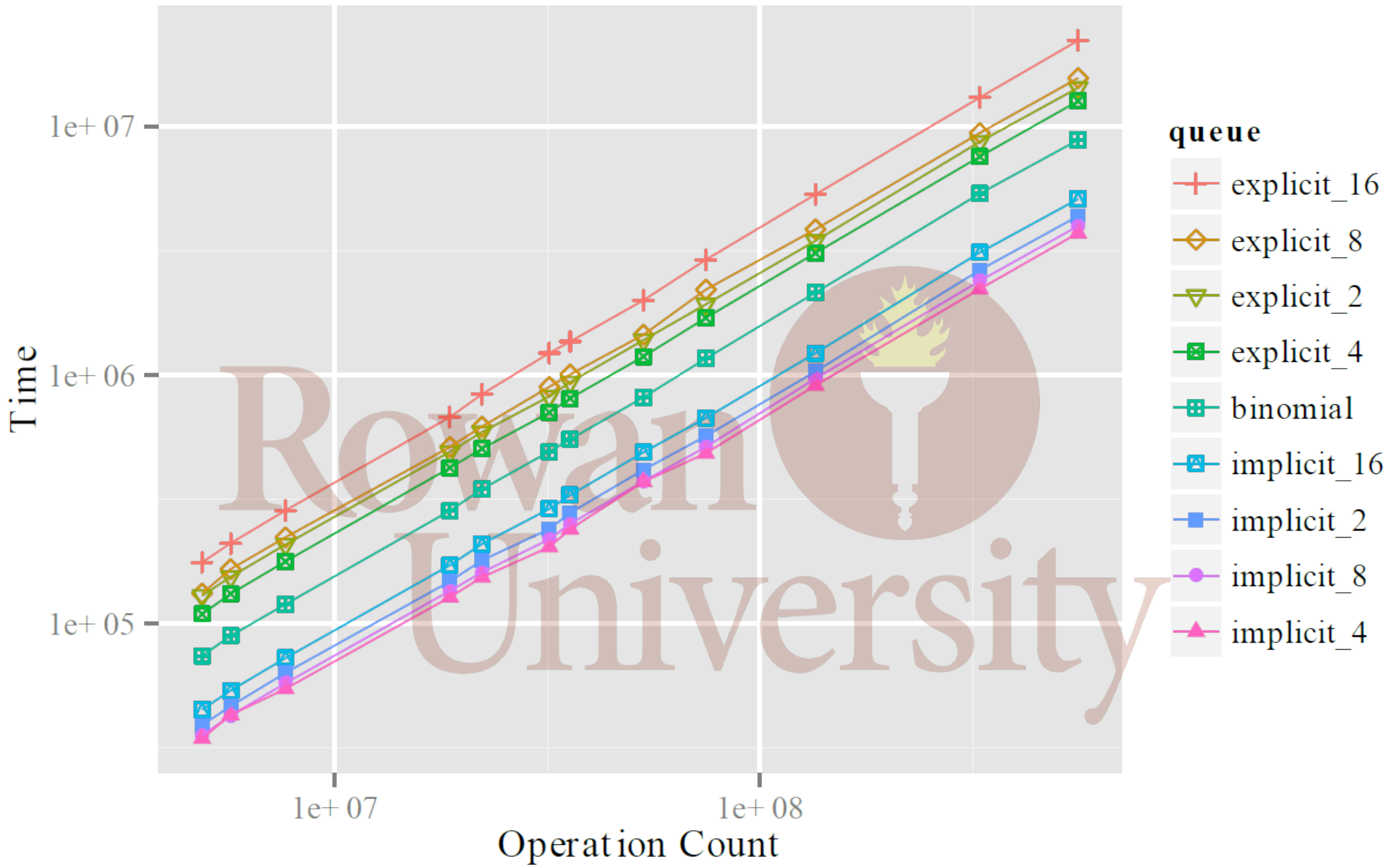


Figure 1: Dijkstra on the full USA road map. All operation counts are scaled by $\log n$.

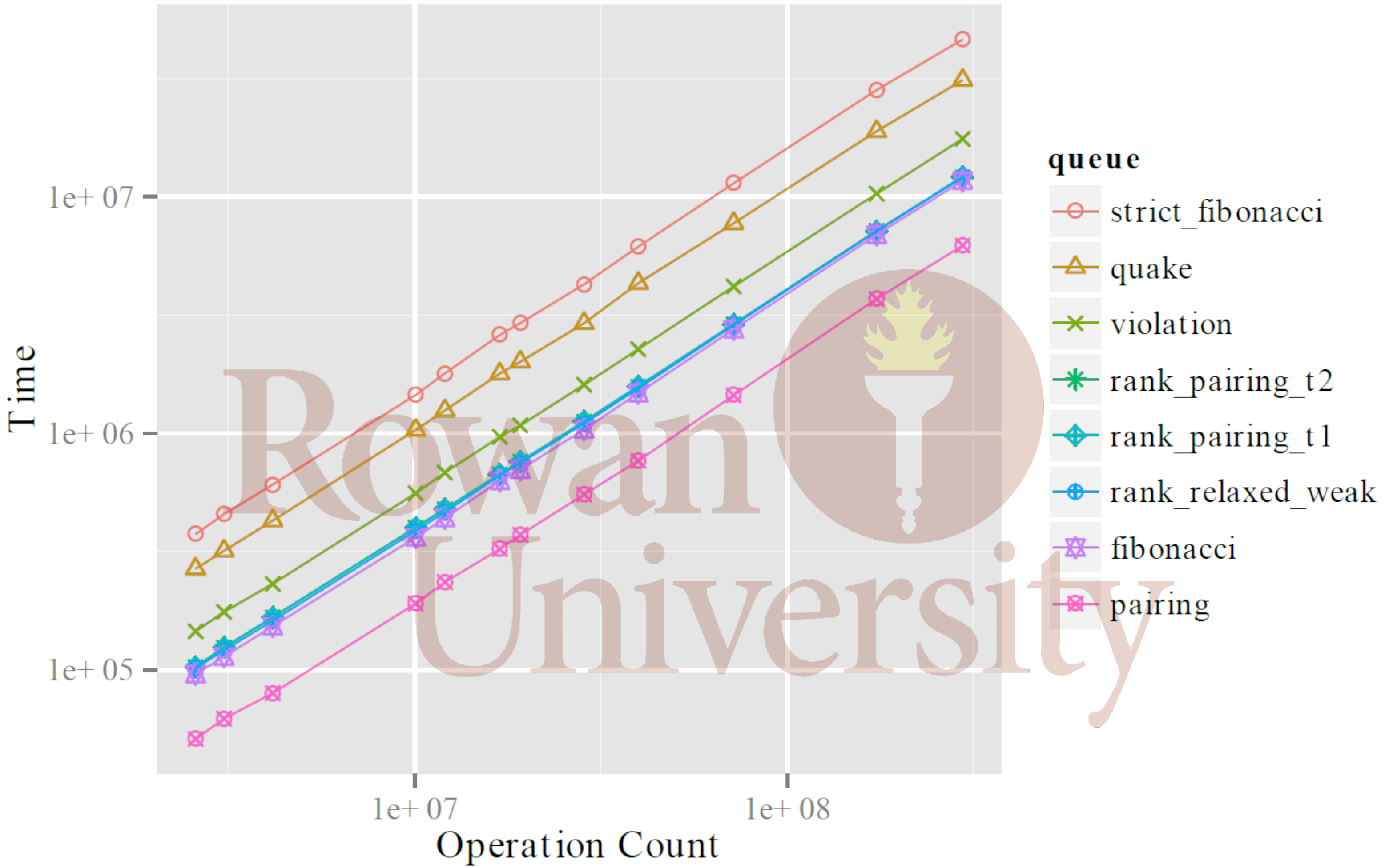


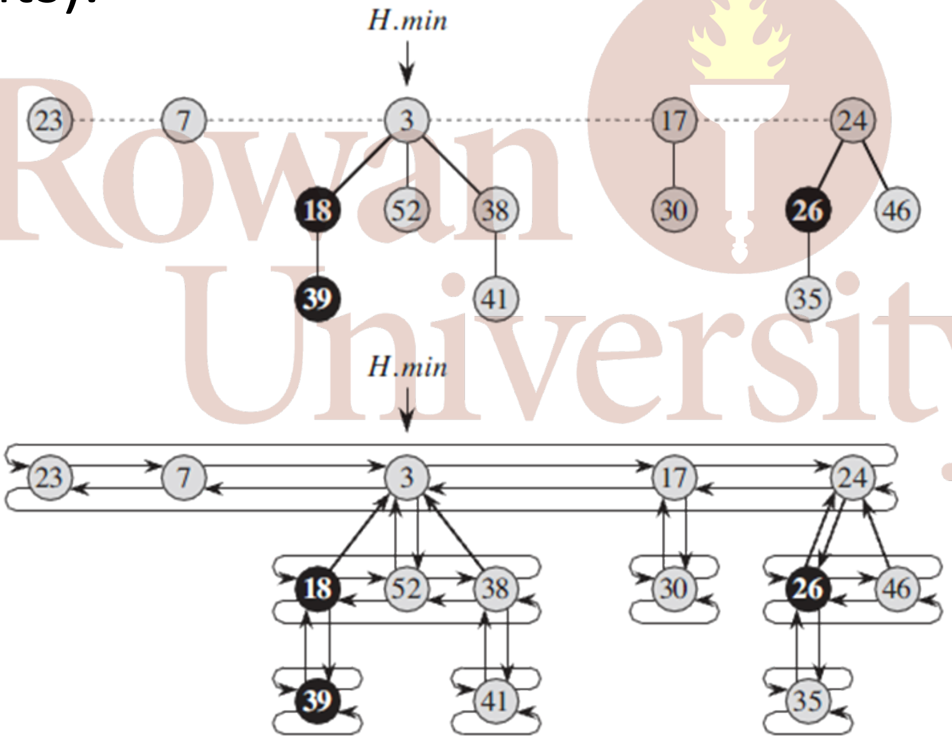
Figure 2: Dijkstra on the full USA road map. The DeleteMin count is scaled by $\log n$.

Structure of Fibonacci Heaps

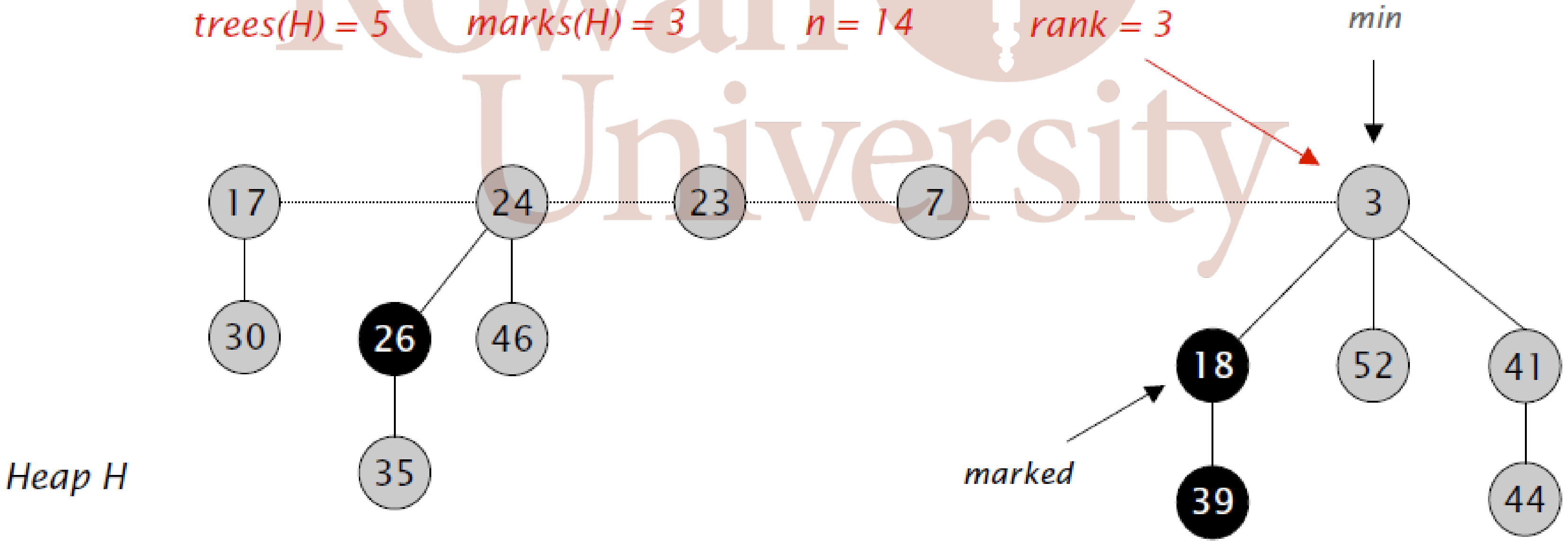
A Fibonacci heap H is a collection of rooted trees that are min-heap ordered. The trees have the following properties:

- The roots of these trees are stored in a doubly-linked "root" list;
- The root of each tree contains the minimum element in that tree;
- The heap has a pointer to the tree root with the global minimum key;
- For each node x, we store
  - its degree
  - its parent (pointer)
  - its children (pointer to **one** child or **`null`** if leaf)
  - its mark (Boolean, indicates whether node has lost a child since the last time x was made a child of another node)
- children are stored in a circular doubly-linked list

This example from our book represents a Fibonacci heap with five min-heap-ordered trees and 14 nodes. The top level is the root list. Black nodes are "marked" (used in **DECREASE_KEY** to keep track of where to make cascading cuts).



- n = number of nodes in heap.
- $rank(x)$ = number of children of node x.
- $rank(H)$ = max rank of any node in heap H.
- $trees(H)$ = number of trees in heap H.
- $marks(H)$ = number of marked nodes in heap H.



Implementation 📖

```
class Node {
    Node parent;
    Node left;
    Node right;
    Node child;
    int degree;
    boolean mark;
    int key;
}
public class FibonacciHeap {
    Node min;
    int n;
}
```

In this node structure, `left` and `right` point to the node itself if it is a leaf. The heap itself has a pointer to the minimum element, which is part of the circular root list. Trees may appear in any order in the list.

Why a Doubly-Linked Circular List?

The doubly-linked circular list makes it an $O(1)$ operation to insert an element. Elements are inserted at root list level and consolidated later.

```
private void insert(Node x) {
    if (this.min == null) {
        this.min = x;
        x.set_left(this.min);
        x.set_right(this.min);
    } else {
        x.set_right(this.min);
        x.set_left(this.min.get_left());
        this.min.get_left().set_right(x);
        this.min.set_left(x);
        if (x.get_key() < this.min.get_key())
            this.min = x;
    }
    this.n += 1;
}

public void insert(int key) {
    insert(new Node(key));
}
```

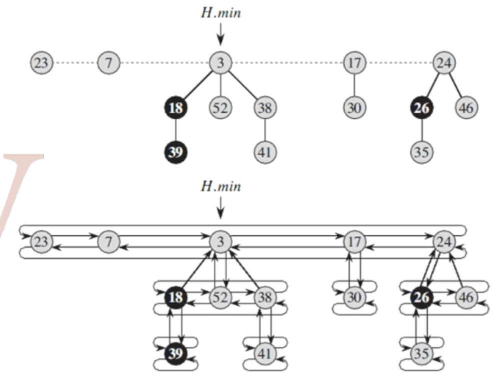
Potential Function for Amortized Analysis

Let $t(H)$ be the number of trees in a Fibonacci heap (the size of root list).

Let $m(H)$ be the number of marked nodes (which have to be visited to be consolidated).

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

The given example has potential $5 + 2 \cdot 3 = 11$.



MAKE_HEAP

We create a structure with null pointer for the minimum (tree node) and set the size of the heap to 0. The potential $\Phi(H)$ is 0 for an empty heap. The cost is $O(1)$.

```
FibonacciHeap() {
    min = null;
    n = 0;
}
```

Create a Node

We initialize a node with null parent, siblings, and children. Hence its degree will be 0 (number of connected nodes). This is also $O(1)$.

```
public Node() {
    this.degree = 0;
    this.mark = false;
    this.parent = null;
    this.left = this;
    this.right = this;
    this.child = null;
    this.key = Integer.MAX_VALUE;
}

Node(int x) {
    this();
    this.key = x;
}
```

Insert

Insert operations will create a new node x. If the heap is empty, we let H.min point to x. Otherwise, we insert the new node into the root list of the heap to the left (or right) of H.min. Then we update H.min to point to $\min(H.\text{min}, x.\text{key})$ and increase the degree of the heap. (Default operations are in class **Node**.)

```java
private void insert(Node x) {
    if (this.min == null) {
        this.min = x;
        x.set_left(this.min);
        x.set_right(this.min);
    } else {
        x.set_right(this.min);
        x.set_left(this.min.get_left());
        this.min.get_left().set_right(x);
        this.min.set_left(x);
        if (x.get_key() < this.min.get_key())
            this.min = x;
    }
    this.n += 1;
}
```

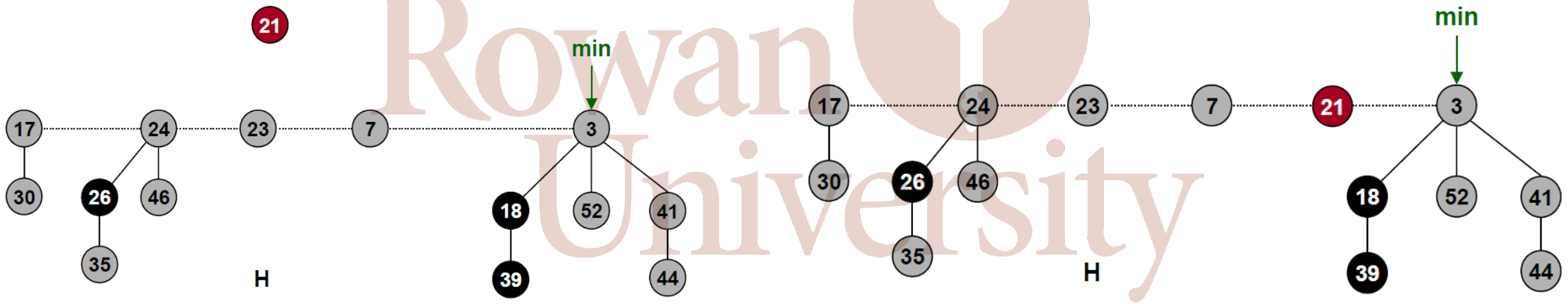
FIB-HEAP-INSERT(H, x)

```
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
```

The immediate (actual) cost is $O(1)$ while the amortized cost is actual plus change in potential.

$$\Delta\Phi(H) = [(t(H) + 1) + 2m(H)] - [t(H) + 2m(H)] = 1$$

Still $O(1)$.



Fibonacci heaps behave like a doubly linked list until they are forced to consolidate (organize into a tree structure).

```
FibonacciHeap H = FibonacciHeap.create_heap();
H.insert(17);
H.display();
H.insert(26);
H.display();
H.insert(30);
H.display();
H.insert(39);
H.display();
H.insert(10);
H.display();
H.consolidate();
H.display();
```

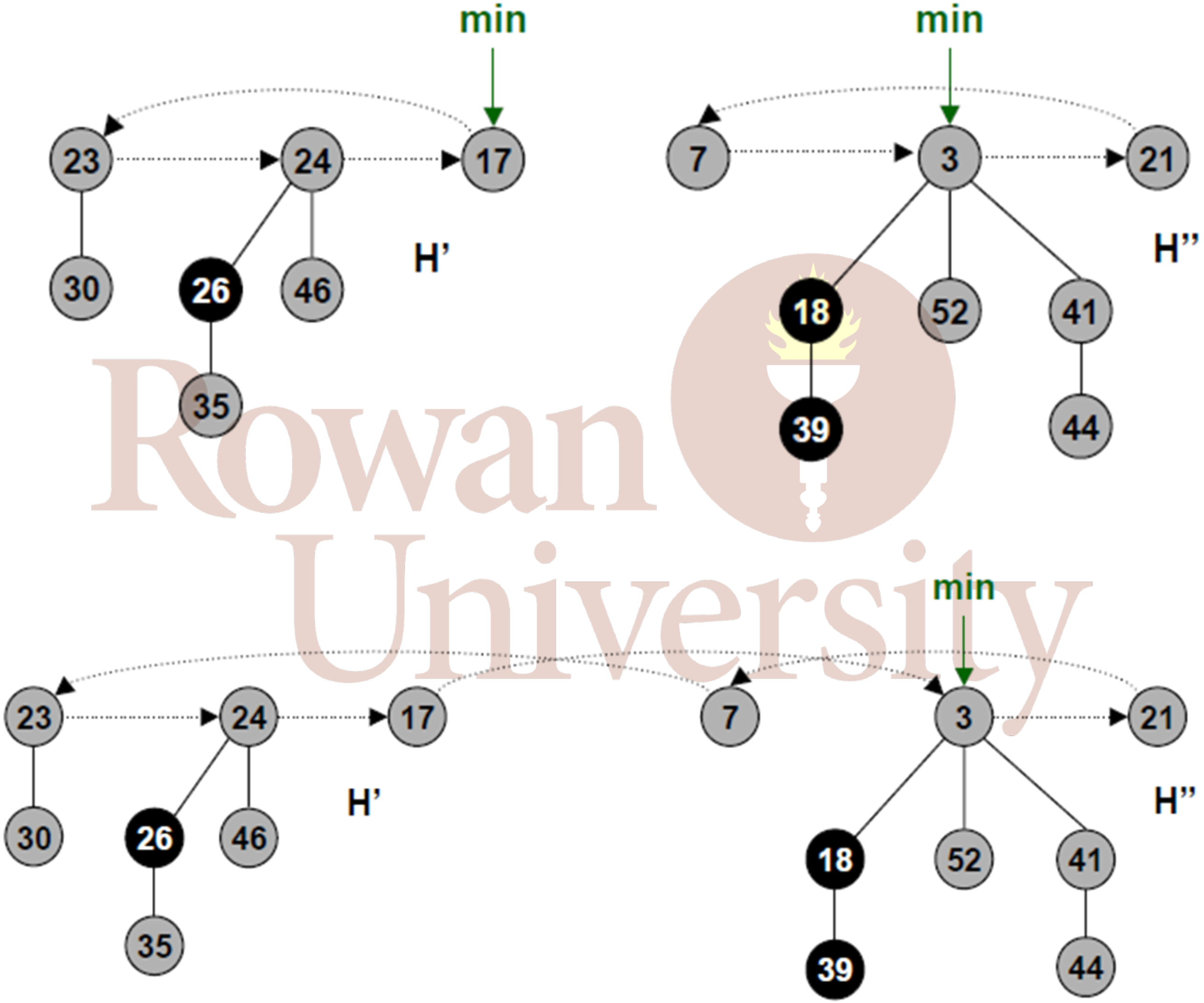
```
(17()->)
(17()->26()->)
(17()->26()->30()->)
(17()->26()->30()->39()->)
(10()->17()->26()->30()->39()->)
(10(26(30()->)->17()->)->39()->)
```

Union

In order to combine two Fibonacci heaps H_1 and H_2 into a single heap, we concatenate their root lists and update the $H.\mathrm{min}$ and $H.\mathrm{n}$ of the resulting heap H. The heaps H_1 and H_2 are consumed in the process to avoid copying (they are technically still there, but access leads to undefined behavior). The actual cost is $O(1)$. The amortized cost is also $O(1)$ since

$$\Delta\Phi(H) = \Phi(H) - \big(\Phi(H_1) + \Phi(H_2)\big) = 0$$

The number of trees in the root list of H is equal to the sum of trees in the root lists of H_1 and H_2, and the number of marked nodes in H is equal to the sum of marked nodes in H_1 and H_2.


min
23
24
17
30
26
46
35
H'
min
7
3
21
18
52
41
39
44
H''
min
23
24
17
7
3
21
30
26
46
35
H'
18
52
41
39
44
H''

```java
public static void merge_heap(FibonacciHeap H1, FibonacciHeap H2, FibonacciHeap H3)
{
    H3.min = H1.min;

    if (H1.min != null && H2.min != null) {
        Node t1 = H1.min.get_left();
        Node t2 = H2.min.get_left();
        H1.min.set_left(t2);
        t1.set_right(H2.min);
        H2.min.set_left(t1);
        t2.set_right(H1.min);
    }
    if (H1.min == null || (H2.min != null && H2.min.get_key() < H1.min.get_key()))
        H3.min = H2.min;
    H3.n = H1.n + H2.n;
}
```

Example

We create two Fibonacci heaps, consolidate them, merge them, and extract the three smallest values.

```
(17()->)
(17()->26()->)
(17()->26()->30()->)
(17()->26()->30()->39()->)
(10()->17()->26()->30()->39()->)
(10(26(30()->)->17()->)->39()->)
(1()->)
(1()->2()->)
(1()->2()->30()->)
(1()->2()->30()->19()->)
(1()->2()->30()->19()->17()->)
(1(19(30()->)->2()->)->17()->)
(1(19(30()->)->2()->)->17()->10(26(30()->)->17()->)->39()->)
1
2
10
```

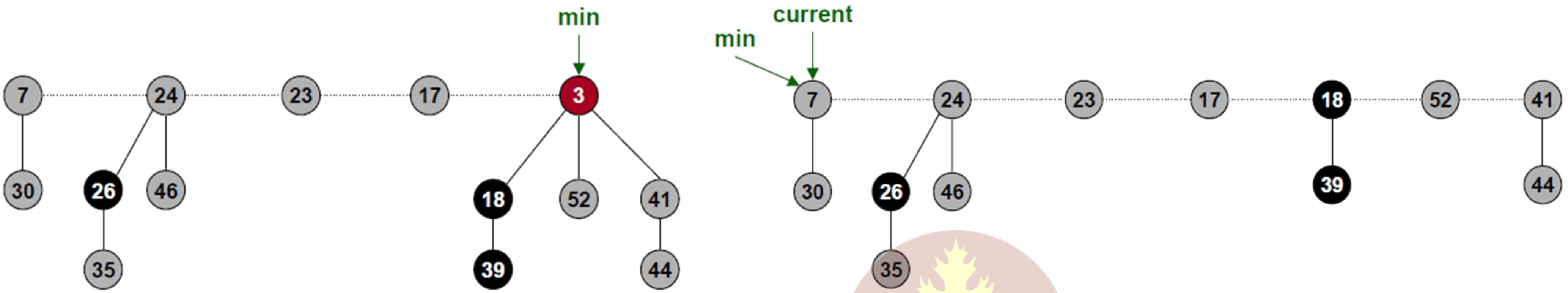
DELETE_MIN

DELETE_MIN retrieves and removes the smallest element in heap H. That is followed up by a tree consolidation. The idea is to

- delete the root element
- concatenate each of the root's children into the root list of H
- consolidate in the root list all the trees of the same degree
  - consolidating the root list consists of repeatedly consolidating until every root in the root list has a distinct degree value
- update H.min and H.n of the new heap

The resulting Fibonacci heap will have no trees in the root list with the same degree, similar to Binomial Heaps.

```
FIB-HEAP-EXTRACT-MIN(H)
z = H.min
if z ≠ 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
```



After removal of the node with key 3, its children are elevated to roots in the root list. The new min pointer is 7 (which needs to be established as true min – in any case we consolidate the trees to reestablish the min heap property).

```java
public int extract_min() {
    Node z = this.min;
    if (z != null) {
        Node c = z.get_child();
        Node k = c, p;
        if (c != null) {
            do {
                p = c.get_right();
                insert(c);
                c.set_parent(null);
                c = p;
            } while (c != null && c != k);
        }
        z.get_left().set_right(z.get_right());
        z.get_right().set_left(z.get_left());
        z.set_child(null);
        if (z == z.get_right())
            this.min = null;
        else {
            this.min = z.get_right();
            this.consolidate();
        }
        this.n -= 1;
        return z.get_key();
    }
    return Integer.MAX_VALUE;
}
```

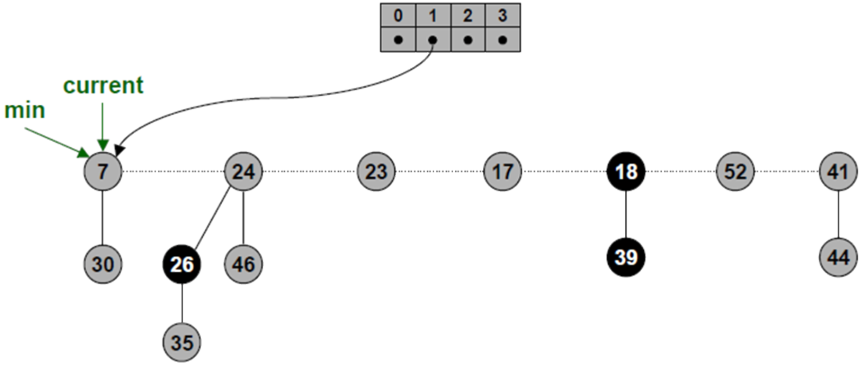
Consolidate

Consolidation combines all trees of the same degree in the root list.

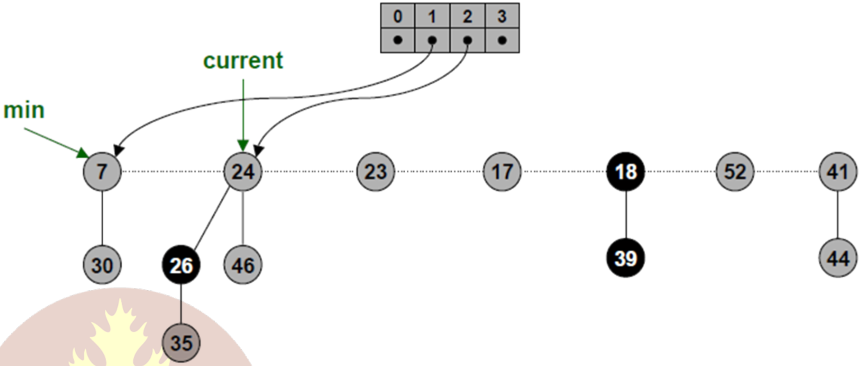
- Find two roots x and y in the root list with the same degree. Without loss of generality, let $x.\mathrm{key} \leq y.\mathrm{key}$
- Link y to x by removing y from the root list and making it a child of x. Increment $x.\mathrm{degree}$, and clear the mark on y.
- Repeat until there are no two trees with the same degree.

There will be a supporting data structure: An array of pointers to the trees in the root list, where an array item in position j points to the tree of degree j. We need to know the upper bound on the maximum degree of the tree in Fibonacci heap H (denoted $D(H.\mathrm{n})$).

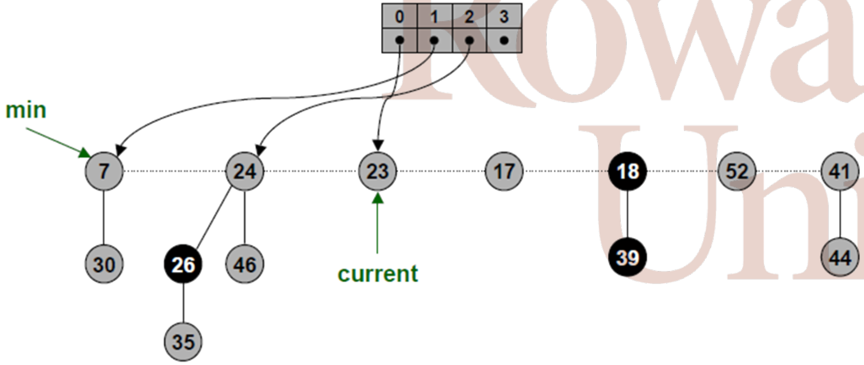
1.


0 1 2 3
min
current
7 24 23 17 18 52 41
30 26 46 39 44
35

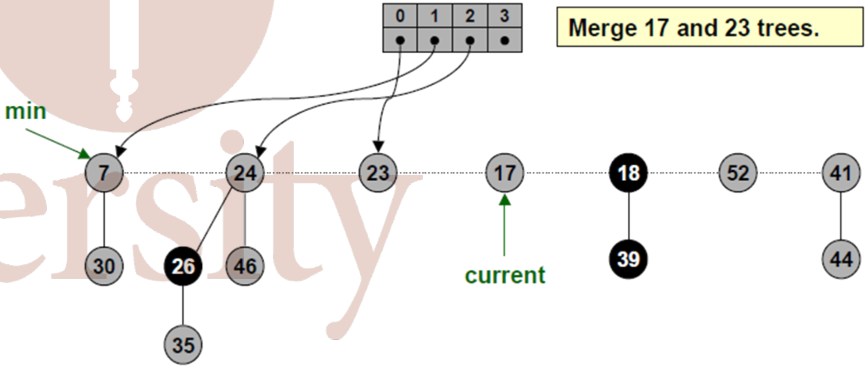
2.


0 1 2 3
current
min
7 24 23 17 18 52 41
30 26 46 39 44
35

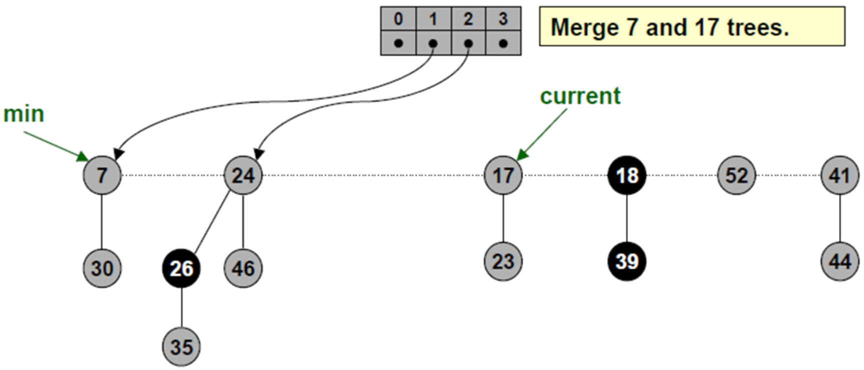
3.


0 1 2 3
min
7 24 23 17 18 52 41
30 26 46 39 44
current
35

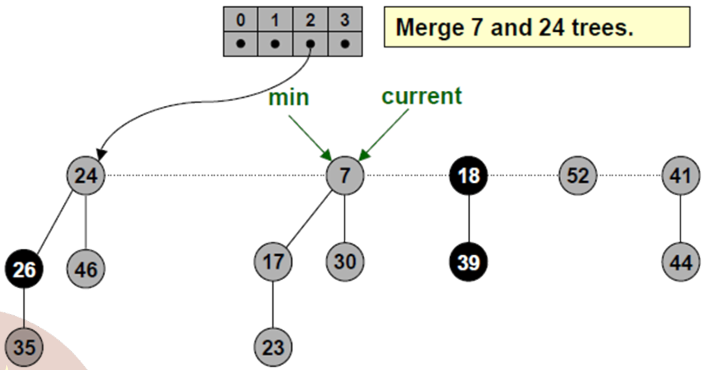
4.


0 1 2 3
Merge 17 and 23 trees.
min
7 24 23 17 18 52 41
30 26 46 39 44
current
35

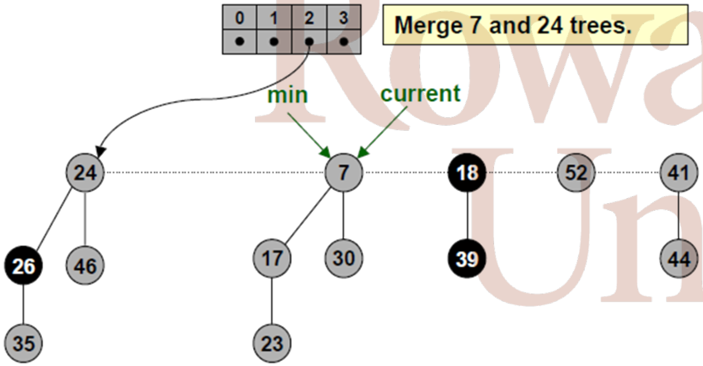
5.


0 1 2 3
Merge 7 and 17 trees.
min
current
7 24 17 18 52 41
30 26 46 23 39 44
35

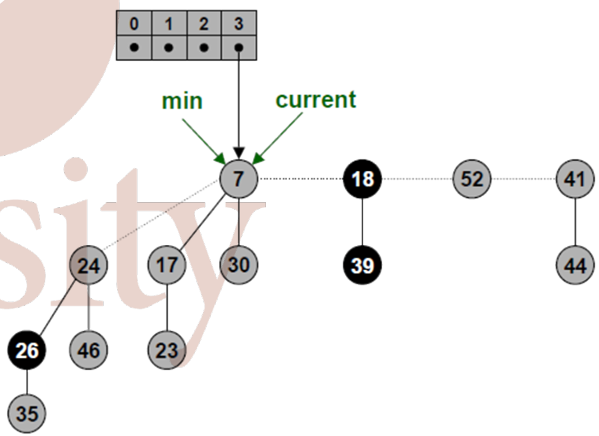
6.


0 1 2 3
Merge 7 and 24 trees.
min
current
24 7 18 52 41
26 46 17 30 39 44
35 23

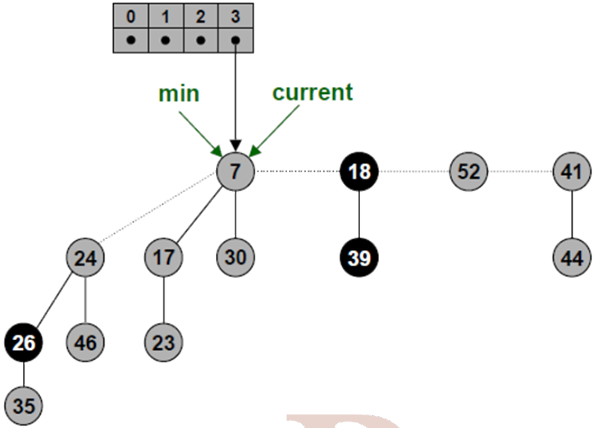
7.


0 1 2 3
Merge 7 and 24 trees.
min
current
24 7 18 52 41
26 46 17 30 39 44
35 23

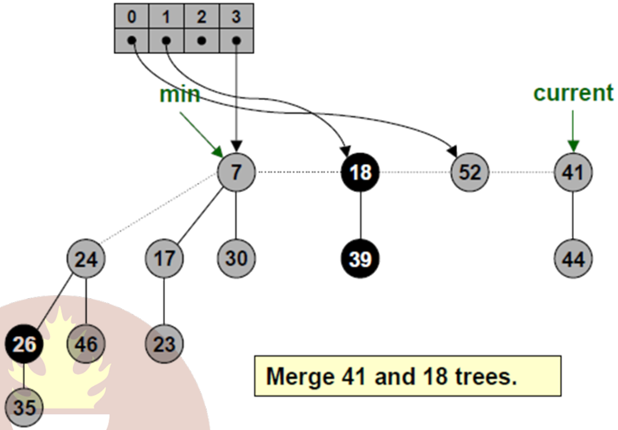
8.


0 1 2 3
min
current
7 18 52 41
24 17 30 39 44
26 46 23
35

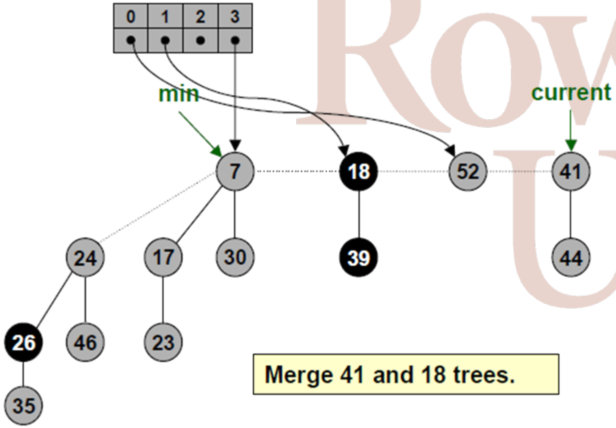
9.


0 1 2 3
min
current
7 18 52 41
24 17 30 39 44
26 46 23
35

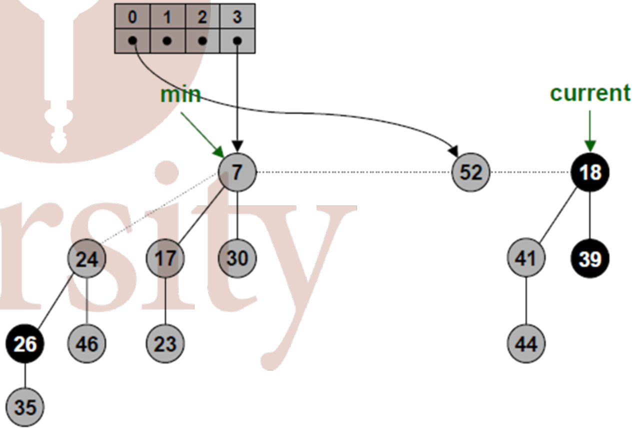
10.


0 1 2 3
min
current
7 18 52 41
24 17 30 39 44
26 46 23
35
Merge 41 and 18 trees.

11.


0 1 2 3
min
current
7 18 52 41
24 17 30 39 44
26 46 23
35
Merge 41 and 18 trees.

12.


0 1 2 3
min
current
7 52 18
24 17 30 41 39
26 46 23 44
35

```
CONSOLIDATE(H)
let A[0..D(H.n)] be a new array
for i = 0 to D(H.n)
    A[i] = NIL
for each node w in the root list of H
    x = w
    d = x.degree
    while A[d] ≠ NIL
        y = A[d]          // another node with the same degree as x
        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] ≠ 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(H, y, x)
remove y from the root list of H
make y a child of x, incrementing x.degree
y.mark = FALSE
```

```java
public void consolidate() {
    double phi = (1 + Math.sqrt(5)) / 2;
    int Dofn = (int) (Math.log(this.n) / Math.log(phi));
    Node[] A = new Node[Dofn + 1];
    for (int i = 0; i <= Dofn; ++i)
        A[i] = null;
    Node w = min;
    if (w != null) {
        Node check = min;
        do {
            Node x = w;
            int d = x.get_degree();
            while (A[d] != null) {
                Node y = A[d];
                if (x.get_key() > y.get_key()) {
                    Node temp = x;
                    x = y;
                    y = temp;
                    w = x;
                }
                fib_heap_link(y, x);
                check = x;
```

```
                A[d] = null;
                d += 1;
            }
            A[d] = x;
            w = w.get_right();
        } while (w != null && w != check);
        this.min = null;
        for (int i = 0; i <= Dofn; ++i) {
            if (A[i] != null) {
                insert(A[i]);
            }
        }
    }
}
```

```java
private void fib_heap_link(Node y, Node x) {
    y.get_left().set_right(y.get_right());
    y.get_right().set_left(y.get_left());

    Node p = x.get_child();
    if (p == null) {
        y.set_right(y);
        y.set_left(y);
    } else {
        y.set_right(p);
        y.set_left(p.get_left());
        p.get_left().set_right(y);
        p.set_left(y);
    }
    y.set_parent(x);
    x.set_child(y);
    x.set_degree(x.get_degree() + 1);
    y.set_mark(false);
}
```

DELETE_MIN Cost Analysis

Let $D(n)$ be the maximum degree of a node in the heap and $t(H)$ the number of trees in the root list. The actual cost of adding children of min into the root list is $O\big(D(n)\big)$ since the min node can have at most $D(n)$ children. When we consolidate trees, we get a running time of $O\big(D(n) + t(H)\big)$ since we need to "touch" all the trees in the root list (all $t(H)$ of them) as well as $D(n)$ nodes, which is the maximum number of added nodes (less one, which is the min node just removed).

Amortized Cost Analysis

Before consolidation we have

$$\Phi(H_{i-1}) = t(n) + 2m(H)$$

After consolidation we have

$$\Phi(H_i) \leq D(n) + 1 + 2m(H)$$

where $D(n) + 1$ is an upper bound on consolidating $t(n)$ trees.

- No new marked nodes
- No two trees with the same degree
- At most $D(n) + 1$ root nodes remain after consolidation

$$\Delta\Phi(H) \leq D(n) + 1 - t(n)$$

The actual cost was $O\big(D(n)+t(H)\big)$, so the amortized cost is bounded by

$$[D(n)+t(H)]+[D(n)+1-t(H)]=2D(n)+1=O\big(D(n)\big)$$

The degree is bounded by $\log(n)$ (Fibonacci heaps get their name from Fibonacci numbers, and there is a formula for $F_n=\frac{\left(1+\sqrt{5}\right)^n-\left(1-\sqrt{5}\right)^n}{\sqrt{5}}$).

COROLLARY 1. *A node of rank k in an F-heap has at least $F_{k+2}\geq\phi^k$ descendants, including itself, where F_k is the kth Fibonacci number ($F_0=0$, $F_1=1$, $F_k=F_{k-2}+F_{k-1}$ for $k\geq 2$), and $\phi=(1+\sqrt{5})/2$ is the golden ratio. (See Figure 7.)*

PROOF. Let S_k be the minimum possible number of descendants of a node of rank k. Obviously, $S_0=1$, and $S_1=2$. Lemma 1 implies that $S_k\geq\sum_{i=0}^{k-2}S_i+2$ for $k\geq 2$. The Fibonacci numbers satisfy $F_{k+2}=\sum_{i=2}^{k}F_i+2$ for $k\geq 2$, from which $S_k\geq F_{k+2}$ for $k\geq 0$ follows by induction on k. The inequality $F_{k+2}\geq\phi^k$ is well known [14]. □

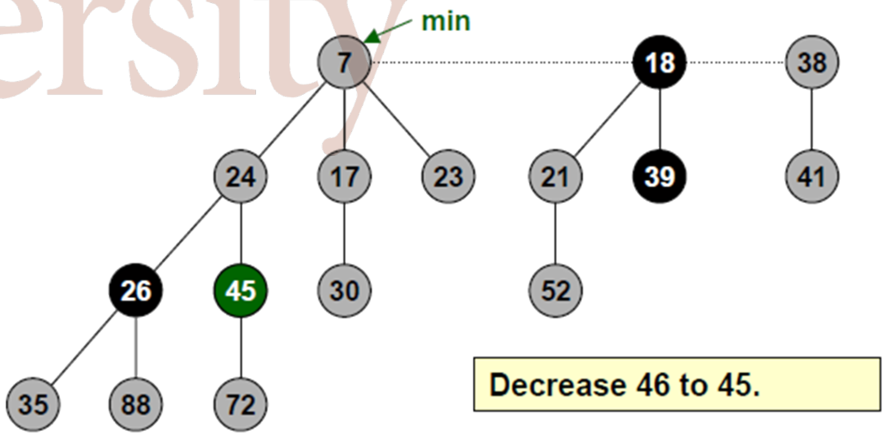
Remark. This corollary is the source of the name "Fibonacci heap."

DECREASE_KEY

The amortize running time for **DECREASE_KEY** is $O(1)$. A main design point of Fibonacci heaps was to achieve this improvement over other priority queue implementations.

Case 0

The new key does not violate the heap property. We just set $x.\text{key} = k$ and are done.



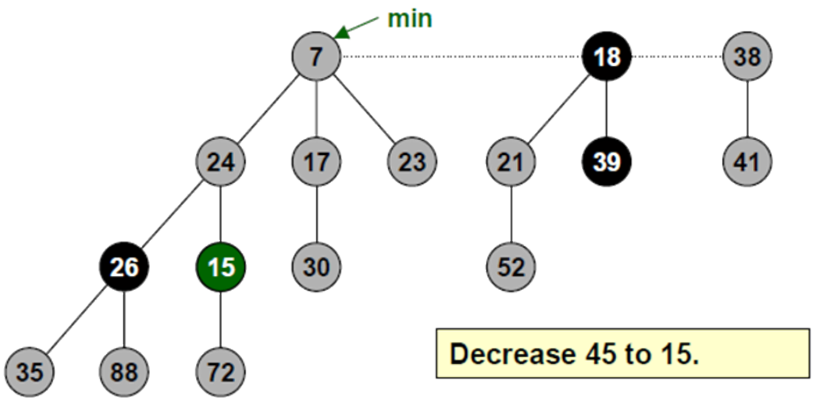
Case 1

The new key violates the heap property. We set $x.\text{key} = k$ and update the heap to maintain the min heap property.

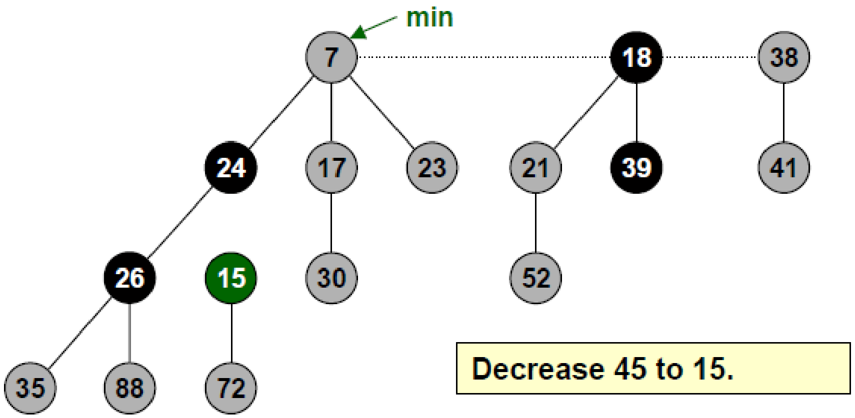
If parent node of x is not marked then

- Decrease key of x to k ($x.\text{key} = k$)
- Cut off the node x and place it into root list
- Unmark the node placed into the root list ($x.\text{marked} = \text{false}$)
- Decrease the degree of the parent by 1 ($x.\text{parent.degree} = x.\text{parent.degree} - 1$)
- Mark the parent ($x.\text{parent.marked} = \text{true}$) indicating that the node has lost one of its children
- Update $H.\text{min} = \min(H.\text{min.key}, x.\text{key})$

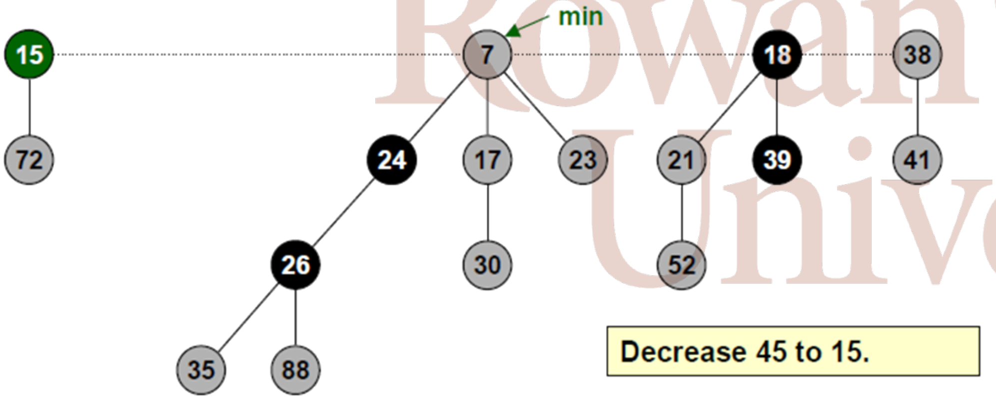
1


min
7
18
38
24
17
23
21
39
41
26
15
30
52
35
88
72
Decrease 45 to 15.

2


min
7
18
38
24
17
23
21
39
41
26
15
30
52
35
88
72
Decrease 45 to 15.

3


min
15
7
18
38
72
24
17
23
21
39
41
26
30
52
35
88
Decrease 45 to 15.

If the parent node of x is marked then

- Decrease key of x to k ($x.\text{key} = k$)
- Cut off the node x and place it into root list
- Unmark the node placed into the root list ($x.\text{marked} = \text{false}$)
- Decrease the degree of the parent by 1 ($x.\text{parent.degree} = x.\text{parent.degree} - 1$)
- Repeat steps 1 – 4 for x.parent until
    - we reached the root or
    - reached unmarked parent ($x.\text{parent.marked} == \text{false}$), in which case mark the parent ($x.\text{parent.marked} = \text{true}$)

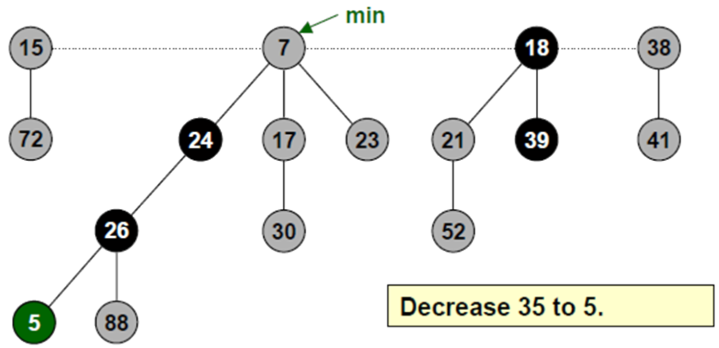
Step 5 is often referred to as Cascading Cuts. No node can lose more than one of its children before being placed into the root list.

```
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 ≠ NIL and x.key < y.key
    CUT(H, x, y)
    CASCADING-CUT(H, y)
if x.key < H.min.key
    H.min = x
```

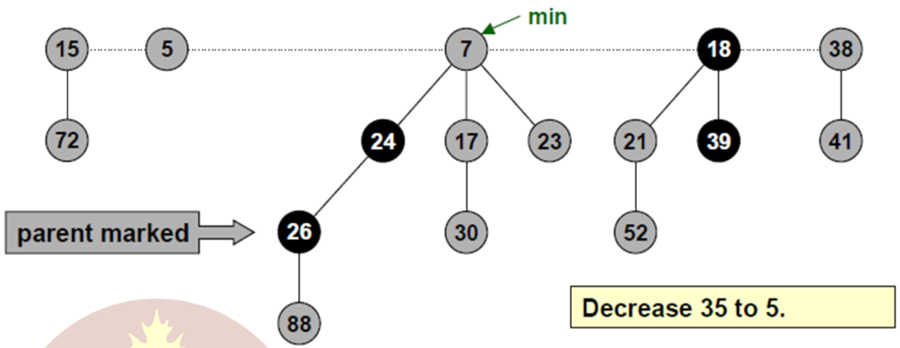
```
CUT(H, x, y)
remove x from the child list of y, decrementing y.degree
add x to the root list of H
x.p = NIL
x.mark = FALSE
```

```
CASCADING-CUT(H, y)
z = y.p
if z ≠ NIL
    if y.mark == FALSE
        y.mark = TRUE
    else CUT(H, y, z)
        CASCADING-CUT(H, z)
```

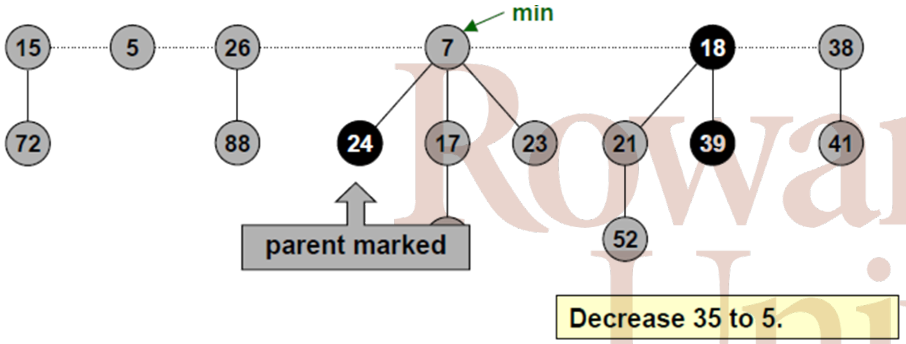
1


min
15 7 18 38
72 24 17 23 21 39 41
26 30 52
5 88
Decrease 35 to 5.

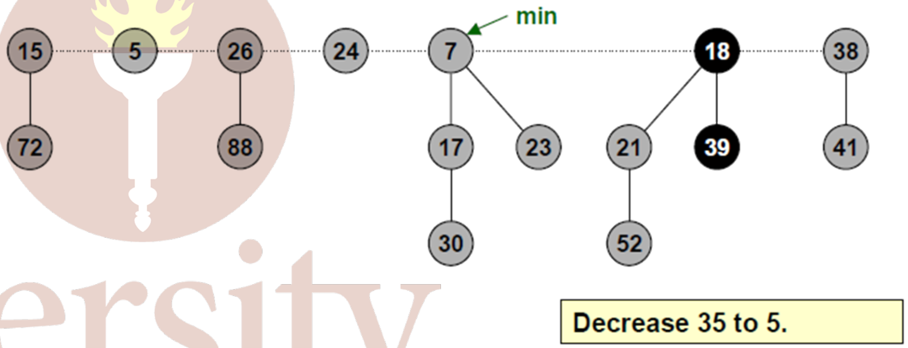
2


min
15 5 7 18 38
72 24 17 23 21 39 41
parent marked
26 30 52
88
Decrease 35 to 5.

3


min
15 5 26 7 18 38
72 88 24 17 23 21 39 41
parent marked
52
Decrease 35 to 5.

4


min
15 5 26 24 7 18 38
72 88 17 23 21 39 41
30 52
Decrease 35 to 5.

```
// Search operation
private void find(int key, Node c) {
    if (found != null || c == null)
        return;
    else {
        Node temp = c;
        do {
            if (key == temp.get_key())
                found = temp;
            else {
                Node k = temp.get_child();
                find(key, k);
                temp = temp.get_right();
            }
        } while (temp != c && found == null);
    }
}
```

```java
public Node find(int k) {
    found = null;
    find(k, this.min);
    return found;
}

public void decrease_key(int key, int nval) {
    Node x = find(key);
    decrease_key(x, nval);
}

private void decrease_key(Node x, int k) {
    if (k > x.get_key())
        return;
    x.set_key(k);
    Node y = x.get_parent();
    if (y != null && x.get_key() < y.get_key()) {
        cut(x, y);
        cascading_cut(y);
    }
    if (x.get_key() < min.get_key())
        min = x;
}
```

```java
private void cut(Node x, Node y) {
    x.get_right().set_left(x.get_left());
    x.get_left().set_right(x.get_right());

    y.set_degree(y.get_degree() - 1);

    x.set_right(null);
    x.set_left(null);
    insert(x);
    x.set_parent(null);
    x.set_mark(false);
}

private void cascading_cut(Node y) {
    Node z = y.get_parent();
    if (z != null) {
        if (y.get_mark() == false)
            y.set_mark(true);
        else {
            cut(y, z);
            cascading_cut(z);
        }
    }
}
```

The actual cost of a cut is $O(1)$ and there are a total of c cuts (one to cut x and $c-1$ cascading cuts). The amortized cost after **DECREASE_KEY** is

- $t(H) + c$ for total number of trees in the heap after cuts
- $m(H) - c + 2$ for the total number of marked nodes
- $c - 1$ nodes that were unmarked by cascading cuts, plus last node marked

$$\Delta\Phi = [t(H) + c + 2(m(H) - c + 1)] - [t(H) + 2m(H)]$$

$$\Delta\Phi = 4 - c$$

Hence the amortized cost is $O(c) + 4 - c = O(1)$.

Delete

We decrease the key to $-\infty$ and use **Delete_Min**.

```
public void delete(Node x) {
    decrease_key(x, Integer.MIN_VALUE);
    int p = extract_min();
}
```

Summary of Fibonacci Heaps

- Fibonacci heaps implement ADT Priority Queue.
- Fibonacci trees are a collection of min-ordered heaps.
- Similar to binomial heaps, but less rigid structure.
- Asymptotically optimal, practically slower.
- **DECREASE_KEY** with amortized $\Theta(1)$.
- Siblings (roots and children) are doubly-linked.
- Designed to optimized **DECREASE_KEY** operation for MST and SP problems.
    - Lazy operations.