Introduction to Algorithms

Advanced Data Structures: II

Xiang-Yang Li and Haisheng Tan

School of Computer Science and Technology
University of Science and Technology of China (USTC)

Fall Semester 2025

Outline of Topics

Binomial Heaps

Fibonacci Heaps

Data Structures for Disjoint Sets

Mergeable Heap (min-heap by default)

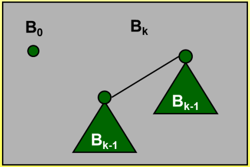
- A data structure supports the following operations:
  1. MAKE-HEAP(): Create and return a new heap containing no elements
  2. INSERT(H,x): Insert element x
  3. MINIMUM(H): Return min element
  4. EXTRACT-MIN(H): Return and delete minimum element
  5. UNION(H_1,H_2): Create and return a new heap that contains all the elements of heaps H_1 and H_2.
- Some other operations: Decrease key of element x to k; Delete an element.
- Applications: Dijkstra's shortest path algorithm, Prim's MST algorithm, Event-driven simulation, Huffman encoding, Heapsort...

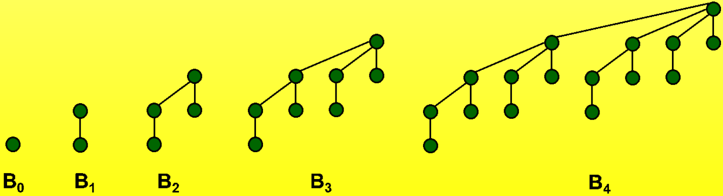
Mergeable Heap

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

Binomial Tree

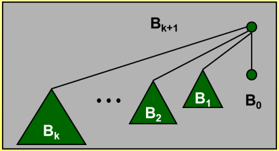
- Recursive definition: B_0 is a single node. B_k consists of 2 binomial trees B_{k-1} linked together, where the root of one subtree is the leftmost child of the other.



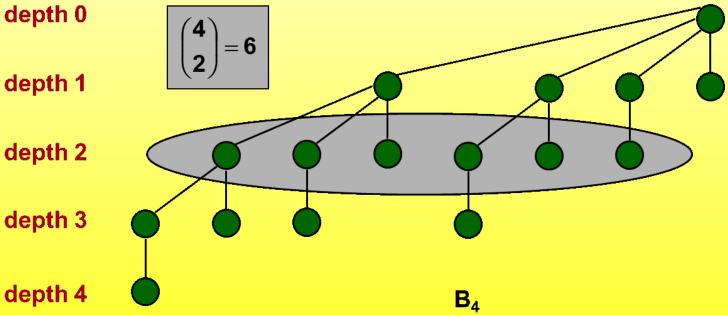


Useful Properties

- For order k binomial tree B_k
  1. Number of nodes $= 2^k$
  2. Height $= k$
  3. Degree of root $= k$
  4. Deleting root yields binomial trees $B_{k-1}, \ldots, B_0$
  5. B_k has $\binom{k}{i}$ nodes at depth i
- Proved by induction.

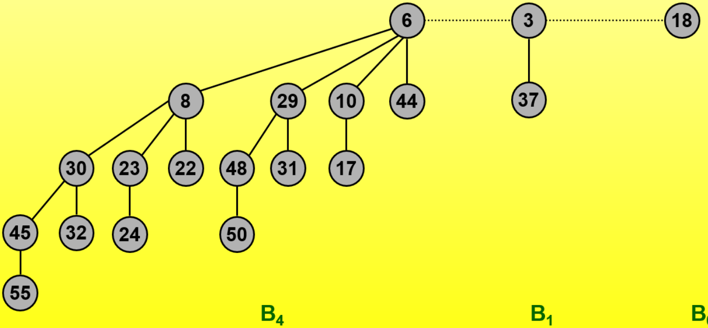


Useful Properties - Example



Binomial Heap: Overview

- Sequence of binomial trees that satisfy binomial heap property:
  1. Each tree is min-heap ordered
  2. 0 or 1 binomial tree of order k can be included.



Binomial Heap: Implementation

- Represent trees using left-child, right sibling pointers.
  Three links per node: *parent*, *left* (left-most child), *right* (right sibling).
- Roots of trees connected with singly linked list.
  Degrees of trees strictly increasing as we traverse the root list.

Binomial Heap: Implementation

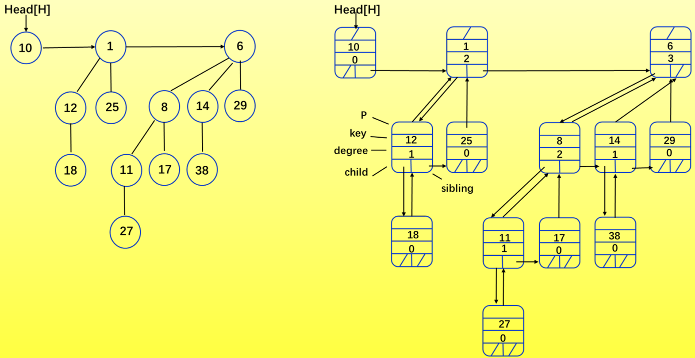
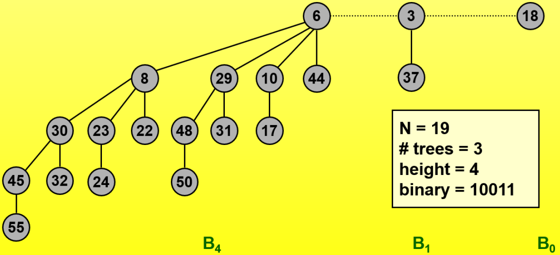


Figure: A binomial heap H and its more detailed representation. The heap consists of binmial tree B_0, B_2 and B_3 which have 1,4 and 8 nodes respectively.

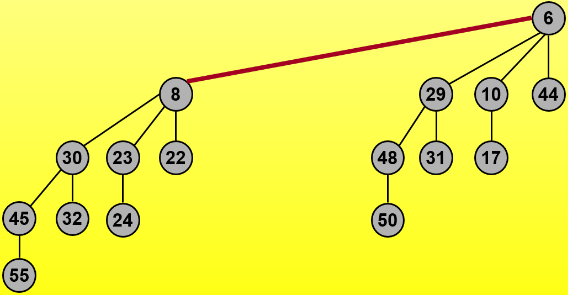
Binomial Heap: Properties

- Properties of N-node binomial heap
  1. Min key contained in root of B_0, B_1, ..., B_k
  2. Contains binomial tree B_i iff $b_i = 1$ where $b_n \cdot b_2 b_1 b_0$ is binary representation of $N = \sum_{i=0}^{\lfloor \log N \rfloor} b_i 2^i$.
  3. At most $\lfloor \log N \rfloor + 1$ binomial trees.
  4. Height $\leq \lfloor \log N \rfloor$

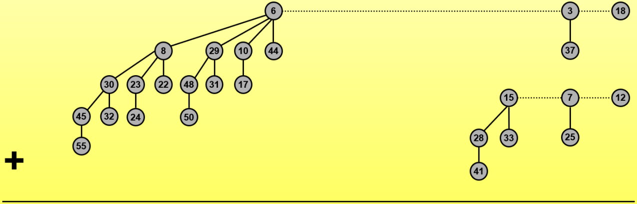


Binomial Heap: Union

- Create H that is union of heaps H' and H'' (in $O(1)$ time):
  1. "Mergeable heaps"
  2. Easy if H' and H'' are each an order k binomial tree.
     a. connect roots of H' and H''
     b. choose smaller key to be root of H



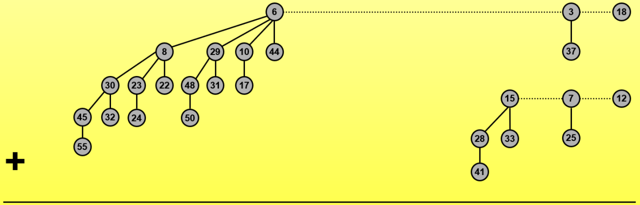
Binomial Heap: Union



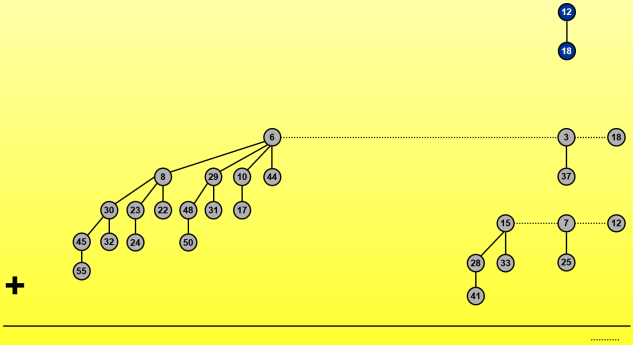
19 + 7 = 26

		1	1	1	
	1	0	0	1	1
+	0	0	1	1	1
	1	1	0	1	0

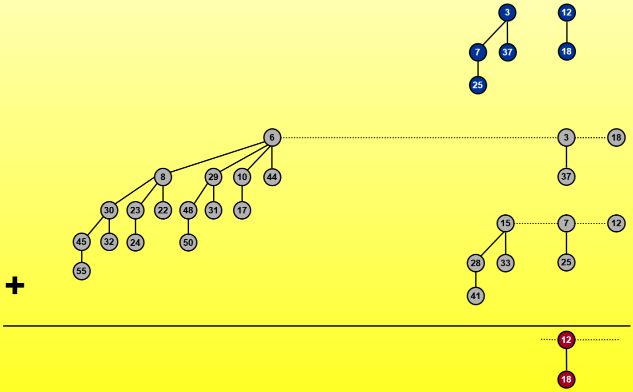
Binomial Heap: Union



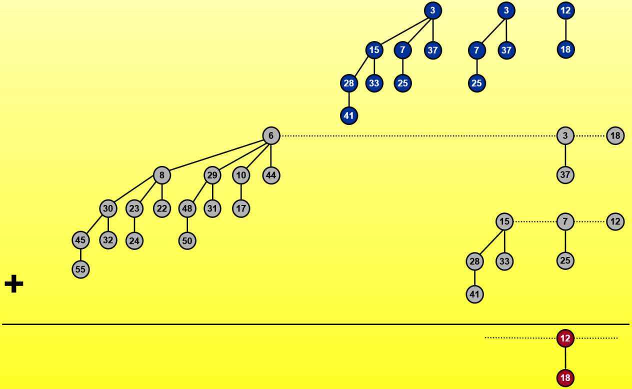
Binomial Heap: Union



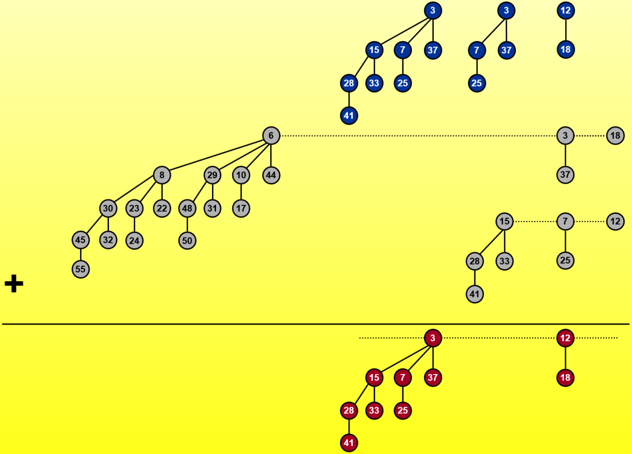
Binomial Heap: Union



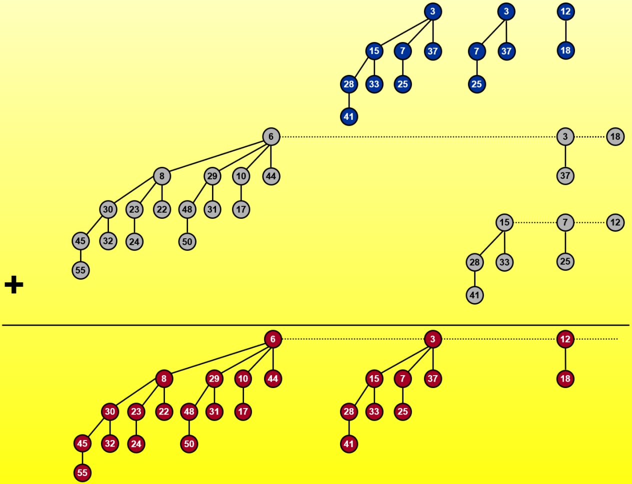
Binomial Heap: Union



Binomial Heap: Union



Binomial Heap: Union

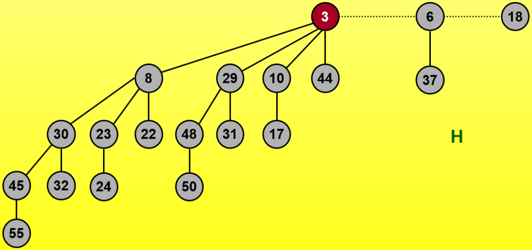


Analysis of Union

- Create heap H that is union of heaps H' and H"
  Analogous to binary addition.
- Running time: $O(\log N)$
  Proportional to number of trees in root lists
  $\lfloor \log N' \rfloor + 1 + \lfloor \log N'' \rfloor + 1 \leq 2(\lfloor \log N \rfloor + 1)$

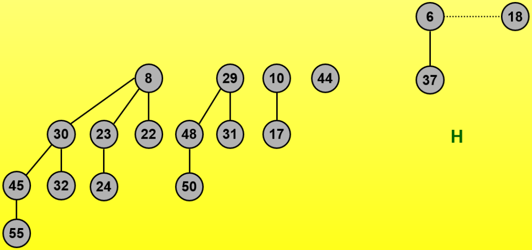
Binomial Heap: Delete Min

- Delete node with minimum key in binomial heap H:
  1. Find root x with min key in root list of H, and delete
  2. $H' \leftarrow$ broken binomial trees
  3. $H \leftarrow \text{UNION}(H', H)$
- Running time: $O(\log N)$



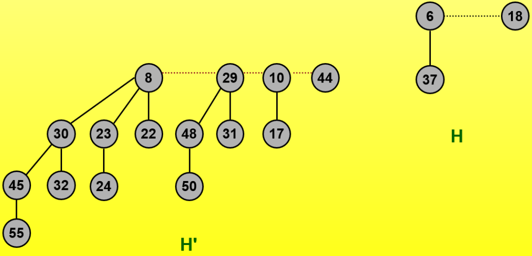
Binomial Heap: Delete Min

- Delete node with minimum key in binomial heap H:
  1. Find root x with min key in root list of H, and delete
  2. $H' \leftarrow$ broken binomial trees
  3. $H \leftarrow \text{UNION}(H', H)$
- Running time: $O(\log N)$



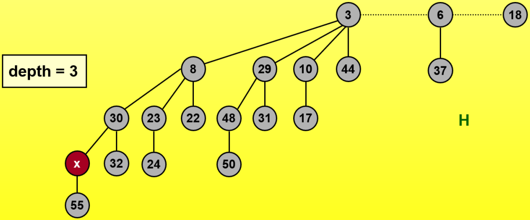
Binomial Heap: Delete Min

- Delete node with minimum key in binomial heap H:
  1. Find root x with min key in root list of H, and delete
  2. $H' \leftarrow$ broken binomial trees
  3. $H \leftarrow \textsc{Union}(H', H)$
- Running time: $O(\log N)$



Binomial Heap: Decrease Key

- Decrease key of node x in binomial heap H:
  1. Suppose x is in binomial tree B_k
  2. Bubble node x up the tree if x is too small
- Running time: $O(\log N)$
  Proportional to depth of node $x \leq \lfloor \log_2 N \rfloor$

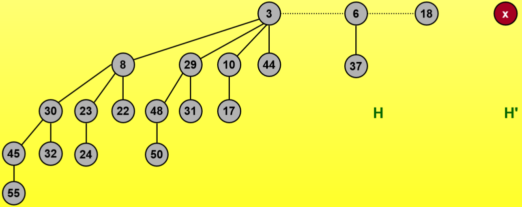


Binomial Heap: Delete

- Delete node x in binomial heap H:
  1. Decrease key of x to $-\infty$
  2. DELETEMIN
- Running time: $O(\log N)$

Binomial Heap: Insert

- Insert a new node x into binomial heap H
  1. $H' \leftarrow \text{MAKEHEAP}(x)$
  2. $H \leftarrow \text{UNION}(H', H)$
- Running time: $O(\log N)$



Recall

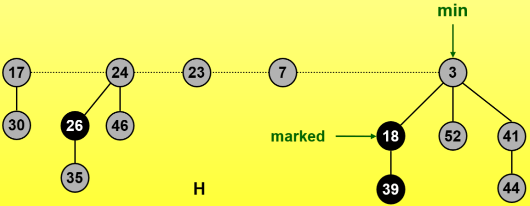
		Heaps			
Operation	Linked List	Binary	Binomial	Fibonacci	Relaxed
make-heap	1	1	1	1	1
insert	1	log N	log N	1	1
find-min	N	1	log N	1	1
delete-min	N	log N	log N	log N	log N
union	1	N	log N	1	1
decrease-key	1	log N	log N	1	1
delete	N	log N	log N	log N	log N
is-empty	1	1	1	1	1

Fibonacci Heaps: Overview

- Fibonacci heap history: Fredman and Tarjan (1986)
  1. Ingenious data structure and analysis
  2. Original motivation: $O(m + n \log n)$ shortest path algorithm, also led to faster algorithms for MST, weighted bipartite matching
  3. Still ahead of its time
- Fibonacci heap intuition:
  1. Similar to binomial heaps, but less structured
  2. Decrease-key and union run in $O(1)$ time (amortized)
  3. "Lazy" unions
- Fibonacci heaps are named after the Fibonacci numbers, which are used in their running time analysis.

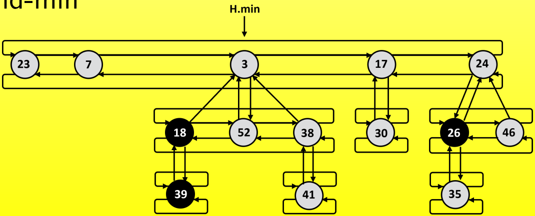
Fibonacci Heaps: Structure

- Fibonacci heap:
  Set of min-heap ordered trees



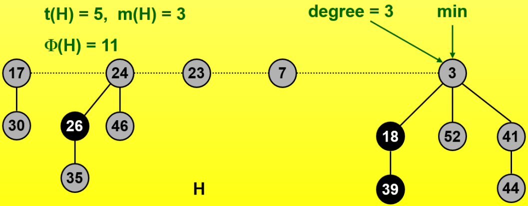
Fibonacci Heaps: Implementation

- Each node contains a pointer to its parent and a pointer to any one of its children. The children are linked together in a circular, doubly linked list:
  Can quickly splice off subtrees
- Roots of trees connected with circular doubly linked list:
  Fast union
- Pointer to root of tree with min element:
  Fast find-min



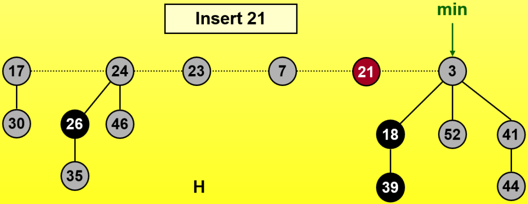
Fibonacci Heaps: Potential Function

- $Degree[x]$ = degree of node x
- $D(n)$ = max degree of any node in Fibonacci heap with n nodes
- $Mark[x]$ = mark of node x (black or gray)
- $t(H) = \#$ trees
- $m(H) = \#$ marked nodes
- $\Phi(H) = t(H) + 2m(H)$ = potential function



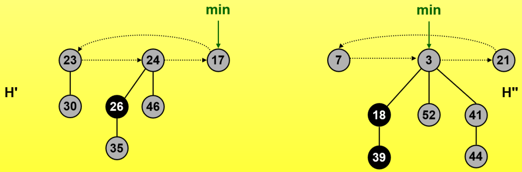
Fibonacci Heaps: Insert

- Insert:
  1. Create a new singleton tree
  2. Add to left of min pointer
  3. Update min pointer
- Running time: $O(1)$ amortized



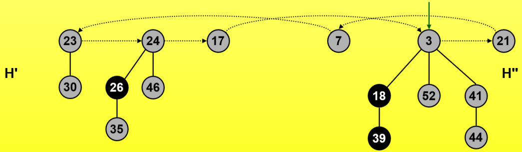
Fibonacci Heaps: Union

- Union:
  1. Concatenate two Fibonacci heaps
  2. Root lists are circular, doubly linked lists



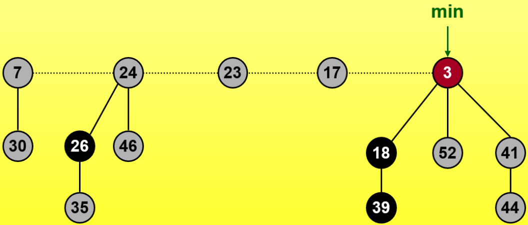
Fibonacci Heaps: Union

- Union:
  1. Concatenate two Fibonacci heaps
  2. Root lists are circular, doubly linked lists
- Concatenate the two root lists, and update the min pointer.
- Running time: $O(1)$ amortized



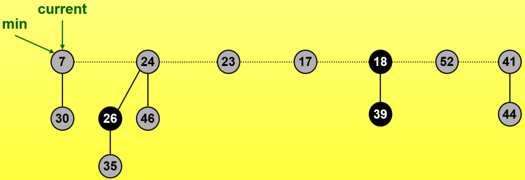
Fibonacci Heaps: Delete Min

- Delete min and concatenate its children into root list
- Consolidate trees so that no two roots have same degree



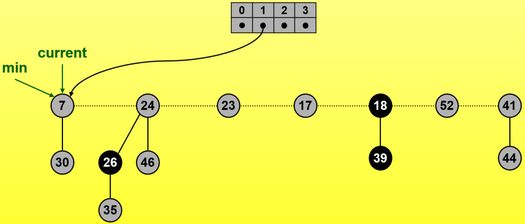
Fibonacci Heaps: Delete Min

- Delete min and concatenate its children into root list
- Consolidate trees so that no two roots have same degree



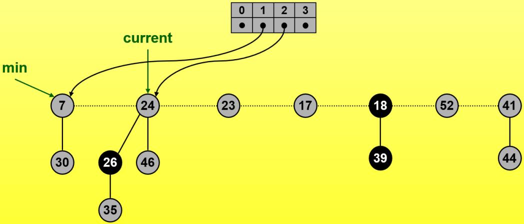
Fibonacci Heaps: Delete Min

- Delete min and concatenate its children into root list
- Consolidate trees so that no two roots have same degree



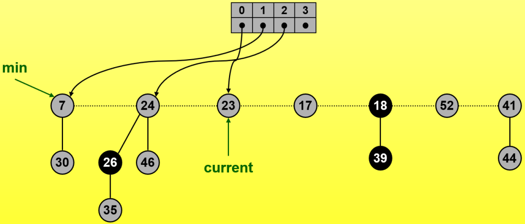
Fibonacci Heaps: Delete Min

- Delete min and concatenate its children into root list
- Consolidate trees so that no two roots have same degree



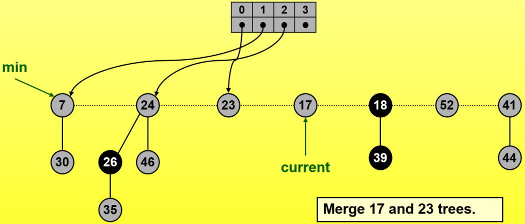
Fibonacci Heaps: Delete Min

- Delete min and concatenate its children into root list
- Consolidate trees so that no two roots have same degree



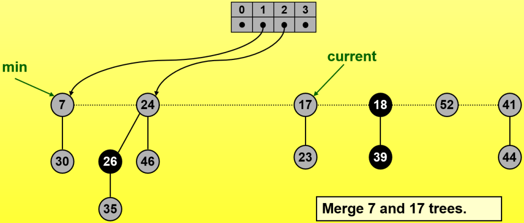
Fibonacci Heaps: Delete Min

- Delete min and concatenate its children into root list
- Consolidate trees so that no two roots have same degree



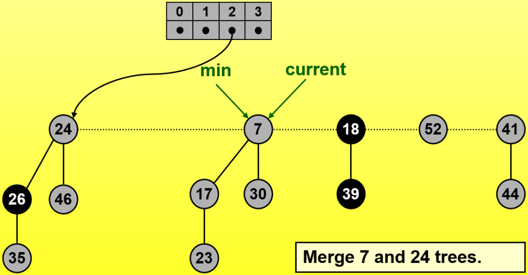
Fibonacci Heaps: Delete Min

- Delete min and concatenate its children into root list
- Consolidate trees so that no two roots have same degree



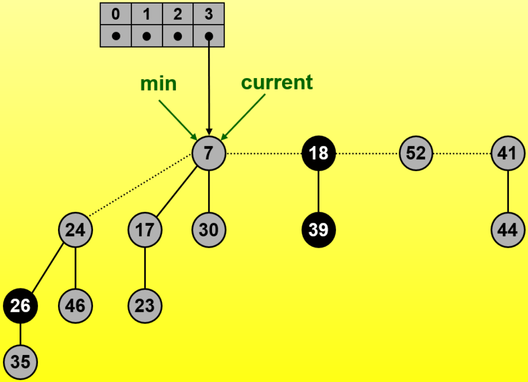
Fibonacci Heaps: Delete Min

- Delete min and concatenate its children into root list
- Consolidate trees so that no two roots have same degree



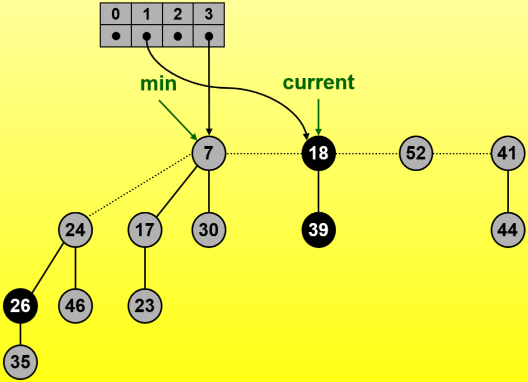
Fibonacci Heaps: Delete Min

- Delete min and concatenate its children into root list
- Consolidate trees so that no two roots have same degree



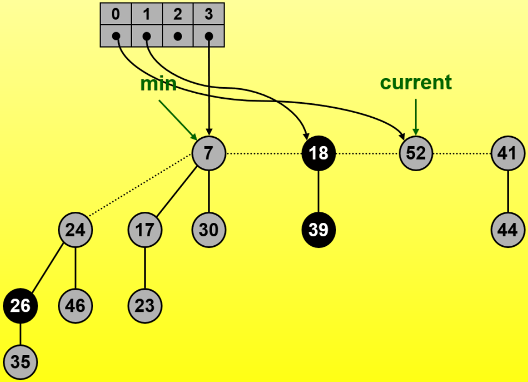
Fibonacci Heaps: Delete Min

- Delete min and concatenate its children into root list
- Consolidate trees so that no two roots have same degree



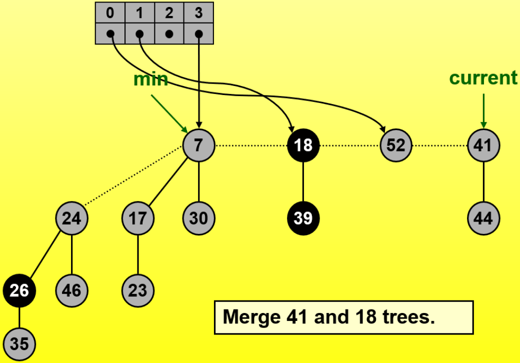
Fibonacci Heaps: Delete Min

- Delete min and concatenate its children into root list
- Consolidate trees so that no two roots have same degree



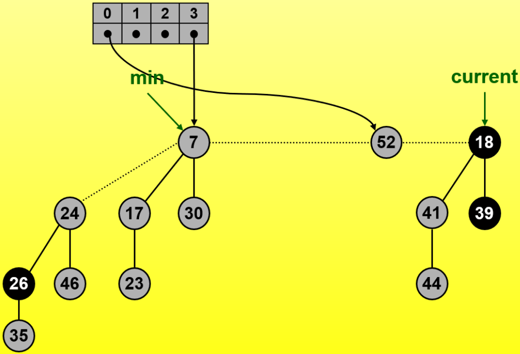
Fibonacci Heaps: Delete Min

- Delete min and concatenate its children into root list
- Consolidate trees so that no two roots have same degree



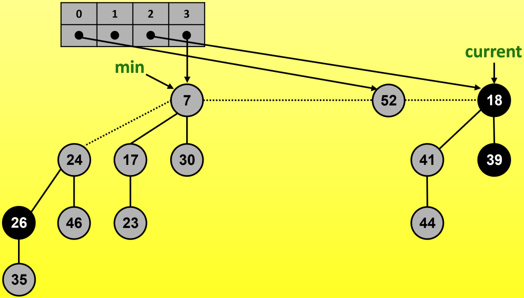
Fibonacci Heaps: Delete Min

- Delete min and concatenate its children into root list
- Consolidate trees so that no two roots have same degree



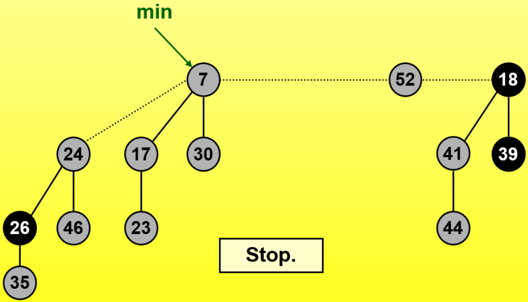
Fibonacci Heaps: Delete Min

- Delete min and concatenate its children into root list
- Consolidate trees so that no two roots have same degree



Fibonacci Heaps: Delete Min

- Delete min and concatenate its children into root list
- Consolidate trees so that no two roots have same degree



Fibonacci Heaps: Delete Min Analysis

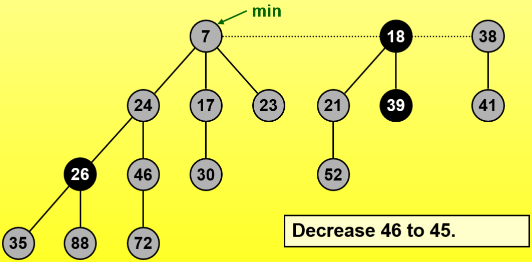
- Actual cost: $O(D(n) + t(H))$
  1. $O(D(n))$ work adding min's children into root list and updating min
  2. $O(D(n) + t(H))$ work consolidating trees
- Amortized cost: $O(D(n))$
  1. $t(H') \leq D(n) + 1$ since no two trees have same degree
  2. $\Delta\Phi(H) \leq D(n) + 1 - t(H)$

Fibonacci Heaps: Delete Min Analysis

- Is amortized cost of $O(D(n))$ good?
  1. Yes, if only Insert, Delete-min, and Union operations supported
     a. In this case, Fibonacci heap contains only binomial trees since we only merge trees of equal root degree
     b. This implies $D(n) \leq \lfloor \log_2 N \rfloor$
  2. Yes, if we support Decrease-key in clever way
     a. We'll show that $D(n) \leq \lfloor \log_\phi N \rfloor$ where ϕ is golden ratio
     b. Limiting ratio between successive Fibonacci numbers!

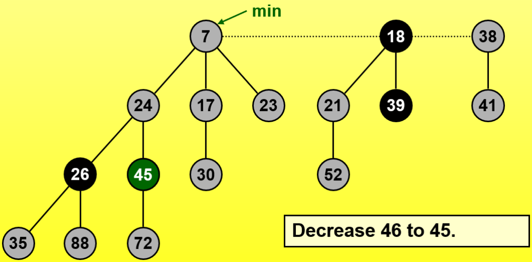
Fibonacci Heaps: Decrease Key

- **Case 0**: min-heap property not violated
  1. Decrease key of x to k
  2. Change heap min pointer if necessary



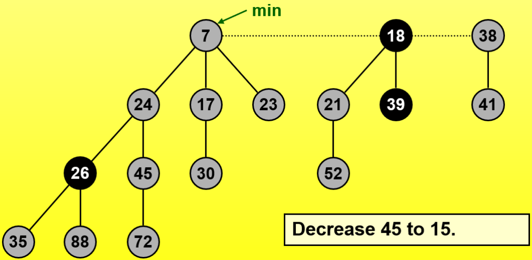
Fibonacci Heaps: Decrease Key

- **Case 0**: min-heap property not violated
  1. Decrease key of x to k
  2. Change heap min pointer if necessary



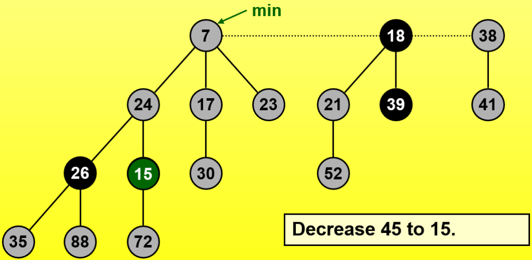
Fibonacci Heaps: Decrease Key

- **Case 1**: min-heap property violated; and parent of x is unmarked
  1. Decrease key of x to k
  2. Cut off link between x and its parent
  3. Mark parent
  4. Add tree rooted at x to root list, updating heap min pointer



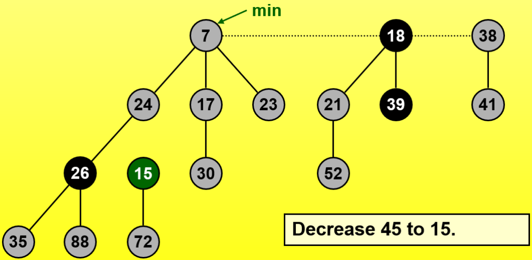
Fibonacci Heaps: Decrease Key

- **Case 1**:min-heap property violated; and parent of x is unmarked
  1. Decrease key of x to k
  2. Cut off link between x and its parent
  3. Mark parent
  4. Add tree rooted at x to root list, updating heap min pointer



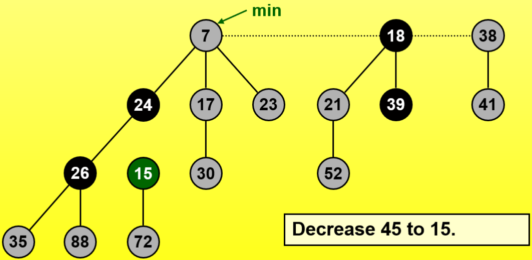
Fibonacci Heaps: Decrease Key

- **Case 1**:min-heap property violated; and parent of x is unmarked
  1. Decrease key of x to k
  2. Cut off link between x and its parent
  3. Mark parent
  4. Add tree rooted at x to root list, updating heap min pointer



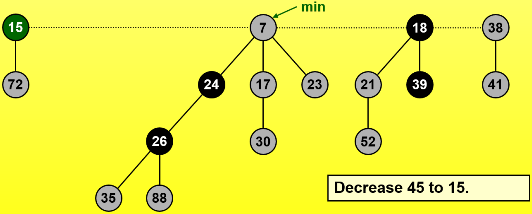
Fibonacci Heaps: Decrease Key

- **Case 1**:min-heap property violated; and parent of x is unmarked
  1. Decrease key of x to k
  2. Cut off link between x and its parent
  3. Mark parent
  4. Add tree rooted at x to root list, updating heap min pointer



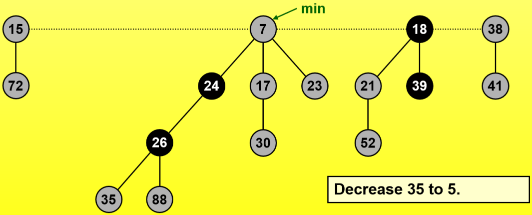
Fibonacci Heaps: Decrease Key

- **Case 1**:min-heap property violated; and parent of x is unmarked
  1. Decrease key of x to k
  2. Cut off link between x and its parent
  3. Mark parent
  4. Add tree rooted at x to root list, updating heap min pointer



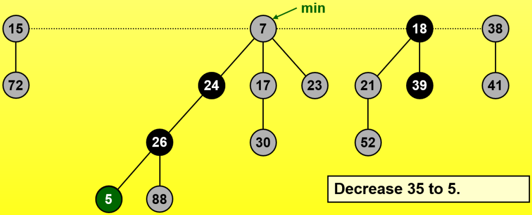
Fibonacci Heaps: Decrease Key

- **Case 2**:min-heap property violated; and parent of x is marked
  1. Decrease key of x to k
  2. Cut off link between x and its parent $p[x]$, and add x to root list
  3. Cut off link between $p[x]$ and $p[p[x]]$, add $p[x]$ to root list
     a. If $p[p[x]]$ unmarked, then mark it
     b. If $p[p[x]]$ marked, cut off $p[p[x]]$, unmark, and repeat



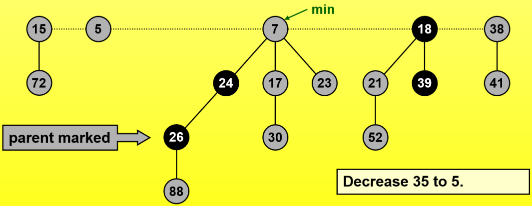
Fibonacci Heaps: Decrease Key

- **Case 2**:min-heap property violated; and parent of x is marked
  1. Decrease key of x to k
  2. Cut off link between x and its parent $p[x]$, and add x to root list
  3. Cut off link between $p[x]$ and $p[p[x]]$, add $p[x]$ to root list
     a. If $p[p[x]]$ unmarked, then mark it
     b. If $p[p[x]]$ marked, cut off $p[p[x]]$, unmark, and repeat



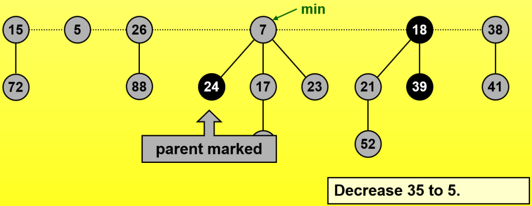
Fibonacci Heaps: Decrease Key

- **Case 2**:min-heap property violated; and parent of x is marked
  1. Decrease key of x to k
  2. Cut off link between x and its parent $p[x]$, and add x to root list
  3. Cut off link between $p[x]$ and $p[p[x]]$, add $p[x]$ to root list
     a. If $p[p[x]]$ unmarked, then mark it
     b. If $p[p[x]]$ marked, cut off $p[p[x]]$, unmark, and repeat



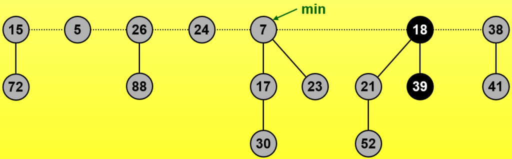
Fibonacci Heaps: Decrease Key

- **Case 2**:min-heap property violated; and parent of x is marked
  1. Decrease key of x to k
  2. Cut off link between x and its parent $p[x]$, and add x to root list
  3. Cut off link between $p[x]$ and $p[p[x]]$, add $p[x]$ to root list
     a. If $p[p[x]]$ unmarked, then mark it
     b. If $p[p[x]]$ marked, cut off $p[p[x]]$, unmark, and repeat



Fibonacci Heaps: Decrease Key

- **Case 2**: parent of x is marked
  1. Decrease key of x to k
  2. Cut off link between x and its parent $p[x]$, and add x to root list
  3. Cut off link between $p[x]$ and $p[p[x]]$, add $p[x]$ to root list
     a. If $p[p[x]]$ unmarked, then mark it
     b. If $p[p[x]]$ marked, cut off $p[p[x]]$, unmark, and repeat



Decrease 35 to 5.

Fibonacci Heaps: Decrease Key Analysis

- Actual cost: $O(c)$
  1. $O(1)$ time for decrease key
  2. $O(1)$ time for each of c cascading cuts, plus reinserting in root list
- Amortized cost: $O(1)$
  1. $t(H') = t(H) + c$
  2. $m(H') \leq m(H) - c + 2$
  3. $\Delta\Phi(H) \leq c + 2(-c + 2) = 4 - c$

Fibonacci Heaps: Delete

- Delete node x:
  1. Decrease key of x to $-\infty$
  2. Delete min element in heap
- Amortized cost: $O(D(n))$

Fibonacci Heaps: Bounding Max Degree

- **Key lemma**: In a Fibonacci heap with N nodes, the maximum degree of any node, denoted as $D(N)$, is at most $\log_\phi N$, where $\phi = \frac{(1+\sqrt{5})}{2}$.
- **Corollary**: Delete and Delete-min take $O(\log\ N)$ amortized time

Fibonacci Facts

- **Definition**: The Fibonacci sequence is

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

- **Fact 1**: $F_{k+2} \geq \phi^k$

Fibonacci Facts

- **Definition**: The Fibonacci sequence is

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

- **Fact 1**: $F_{k+2} \geq \phi^k$
  Proved by induction, and $\phi^2 = \phi + 1$.
- **Fact 2**: For $k \geq 0$, $F_{k+2} = 1 + \sum_{i=0}^{k} F_i = 2 + \sum_{i=2}^{k} F_i$

Proof of Key Lemma

- **Lemma**: Let x be a node with degree k, and let y_1, ..., y_k denote the children of x in the order in which they were linked to x. Then:

$$degree(y_i) \geq \begin{cases} 0 & \text{if } i = 1 \\ i-2 & \text{if } i \geq 2 \end{cases}$$

- **Proof**:
  1. When y_i is linked to x, $y_1, \ldots, y_{i-1}$ already linked to x,
     $\Rightarrow degree(x) = i - 1$
     $\Rightarrow degree(y_i) = i - 1$ since we only link nodes of equal degree (in CONSOLIDATE)
  2. Since then, y_i has lost at most one child (or else, CASCADING-CUT will be triggered)
  3. Thus, $degree(y_i) = i - 1$ or $i - 2$

Proof of Key Lemma

- **Proof of Key Lemma:**:
  1. For any node x, we show that $size(x) \geq \phi^{degree(x)}$
     a. $size(x) = \#$ node in subtree rooted at x
     b. Taking base ϕ logs, $degree(x) \leq \log_\phi(size(x)) \leq \log_\phi N$
  2. Let s_k be min size of tree rooted at any degree k node
     a. Trivial to see that $s_0 = 1, \; s_1 = 2$
     b. s_k monotonically increases with k
  3. Let z be a degree k node and size(z)$=s_k$, and let $y_1, \ldots, y_k$ be children in order that they were linked to z

Proof of Key Lemma

- **Proof of Key Lemma:** :

4. Since $y_i.degree \geq i - 2$ for $i \geq 2$, we have

$$
\begin{aligned}
size(x) \geq s_k &\geq 2 + \sum_{i=2}^{k} s_{y_i.degree} \\
&\geq 2 + \sum_{i=2}^{k} s_{i-2} \qquad (since\ y_i.degree \geq i - 2) \\
&\geq 2 + \sum_{i=2}^{k} F_i \qquad (prove\ s_k \geq F_{k+2} by induction) \\
&= F_{k+2} \geq \phi^k.
\end{aligned}
$$

Data Structures for Disjoint Sets: Overview

- Some applications involve grouping n distinct elements into a collection of disjoint sets
- Two important operations are then finding which set a given element belongs to and uniting two sets
- This chapter explores methods for maintaining a data structure that supports these operations
- Application: connected components in an undirected graph, data clustering...

Disjoint-Set Operations

- Letting x denote an object, we wish to support the following operations:
  1. MAKESET(x) creates a new set whose only member is x. We require that x not already be in some other set
  2. UNION(x, y) unites the dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets, then we remove sets S_x and S_y from S
  3. FINDSET(x) returns a pointer to the representative of the (unique) set containing x

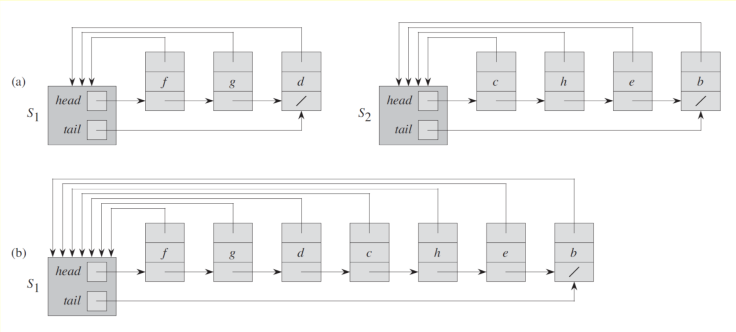
Running Time Analysis

- The running times of disjoint-set data structures shall be analyzed in terms of two parameters:
  1. n: the number of MAKESET operations
  2. m: the total number of MAKESET, UNION, and FINDSET operations
- The number of UNION operations is at most $n - 1$
- We have $m \geq n$

Linked-List Representation

- A simple way to implement a disjoint-set data structure is to represent each set by a linked list
- The first object in each linked list serves as its set's representative
- Each object in the linked list contains a set member, a pointer to the object containing the next set member, and a pointer back to the representative
- Each list maintains pointers *head*, to the representative, and *tail*, to the last object in the list

Linked-List - Example



- The result of UNION(g, e), which appends the linked list containing e to the linked list containing g. The representative of the resulting set is f. The set object for e's list, S_2, is destroyed

Running Time Analysis

- Both MakeSet and FindSet only require $O(1)$ time
- **The worst case**: suppose there are objects x_1, x_2, ..., x_n, we first execute n MakeSet operations, then $n-1$ Union operations: Union(x_2, x_1),...,Union(x_n, x_{n-1})
  1. The n MakeSet operations takes $\Theta(n)$ time
  2. Because the i th Union operation updates i objects, the total number of objects updated by all $n-1$ UNION operations is

$$\sum_{i=1}^{n-1} i = \Theta(n^2)$$

  3. The amortized time of an operation is $\Theta(n)$

Smaller into Larger

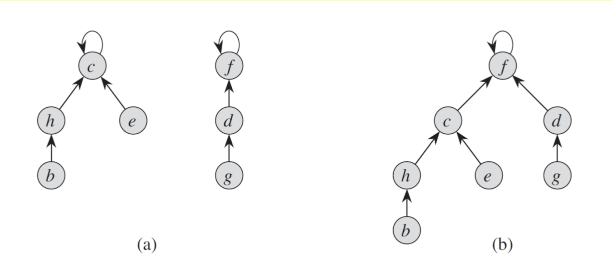
- **A weighted-union heuristic**: suppose that each list also includes the length of the list and that we always append the shorter list onto the longer, breaking ties arbitrarily
- **Theorem**: Using the linked-list representation of disjoint sets and the weighted-union heuristic, a sequence of m MAKESET, UNION, and FINDSET operations, n of which are MAKESET operations, takes $O(m + n \log n)$ time
- Proof?

Smaller into Larger

- **A weighted-union heuristic**: suppose that each list also includes the length of the list and that we always append the shorter list onto the longer, breaking ties arbitrarily
- **Theorem**: Using the linked-list representation of disjoint sets and the weighted-union heuristic, a sequence of m MAKESET, UNION, and FINDSET operations, n of which are MAKESET operations, takes $O(m + n \log n)$ time
- Proof?
  For any $k \leq n$, after an object x's pointer has been updated $\lceil \log k \rceil$ times, the resulting set must have at least k members. So, each element will at most be updated $\lceil \log n \rceil$ times in UNION operations.

Disjoint-Set Forests

- In a faster implementation of disjoint sets, we represent sets by rooted trees, with each node containing one member and each tree representing one set
- The straightforward algorithms that use this representation are no faster than ones that use the linked-list representation



Representing Sets as Trees

- MAKESET: create a tree with just one node
- FINDSET:follow parent pointers until we find the root of the tree. The nodes visited on this simple path toward the root constitute the find path
- UNION: cause the root of one tree to point to the root of the other

Heuristics to Improve the Running Time

- **Union by rank**: for each node, we maintain a rank, which is an upper bound on the height of the node. In union by rank, we make the root with smaller rank point to the root with larger rank during a UNION operation
- **Path compression**: we use it during FINDSET operations to make each node on the find path point directly to the root. Path compression does not change any ranks

Disjoint-Set Forests - Pseudocode I

MakeSet(x)

1: $p[x] \leftarrow x$
2: $rank[x] \leftarrow 0$

Union(x, y)

1: Link(FindSet(x), FindSet(y))

Disjoint-Set Forests - Pseudocode II

Link(x, y)

if $rank[x] > rank[y]$ **then**
    $p[y] \leftarrow x$
else
    $p[x] \leftarrow y$
    if $rank[x] = rank[y]$ **then**
        $rank[y] \leftarrow rank[y] + 1$
    end if
end if

FindSet(x)

if $x \neq p[x]$ **then**
    $p[x] \leftarrow$ FindSet($p[x]$)
end if
return $p[x]$

Running Time Analysis

- **Theorem**: In general, amortized cost is $O(\alpha(n))$, where $\alpha(n)$ grows really, really, really slow
  proof: Really, really, really long
- In any conceivable application of a disjoint-set data structure, $\alpha(n) \leq 4$