Binomial and Fibonacci Heaps

Deliverables

- Binomial Heap Basics
- Operations on Binomial heaps
- Fibonacci Heaps
- Operations on Fibonacci Heaps

Binomial Heaps

Binomial Heap is collection of Binomial trees.

Binomial tree is defined recursively.

Binomial tree of order 0 is a single node.

Binomial tree of order 1 has a root node and a binomial tree of order 0 as its children

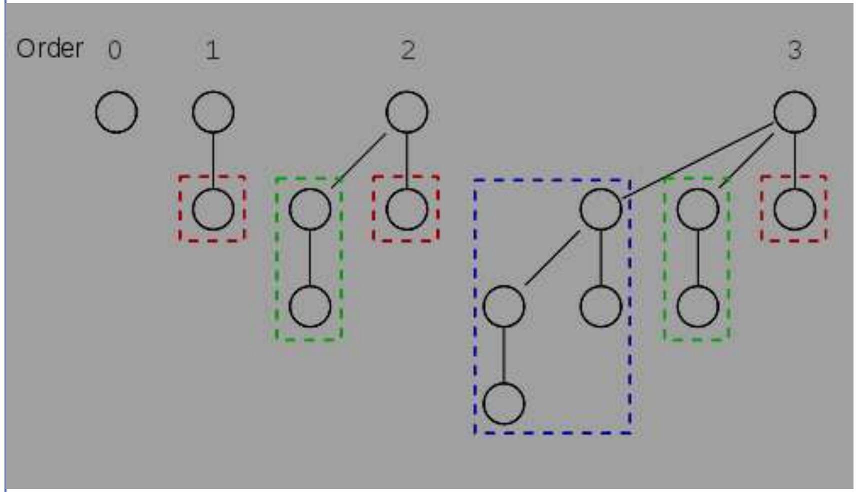
Binomial tree of order 2 has a root node and roots of binomial trees of 0 and 1 as its children

Binomial tree of order k has a root node whose children are roots of binomial trees of orders $k-1$, $k-2$, ..., 2, 1, 0

It uses a special tree structure to support quick merging of two heaps.

Binomial Tree: Definition

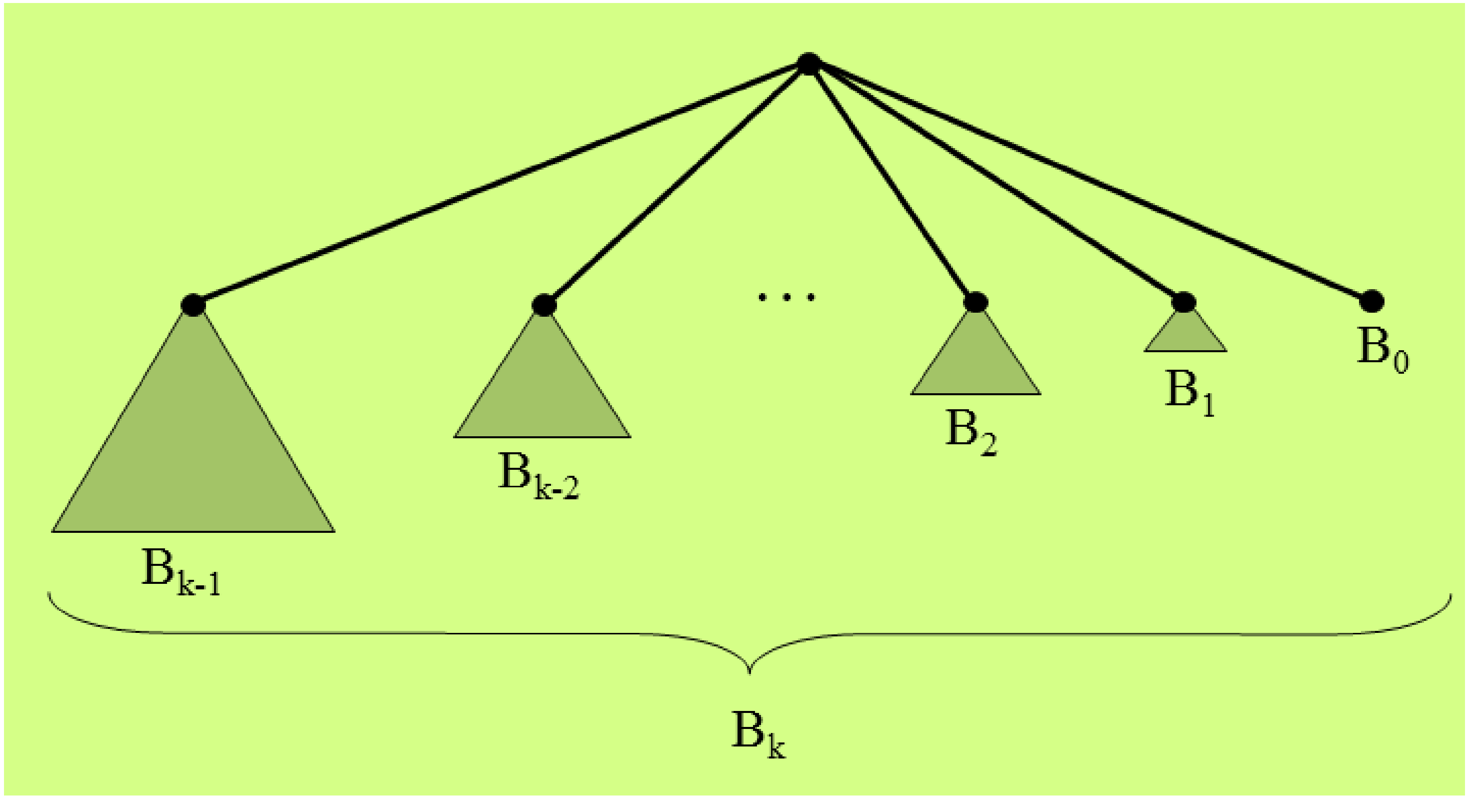
- A binomial tree of order *k* has 2^k nodes, height *k*.
- Due to its unique structure, a binomial tree of order *k* can be constructed from two trees of order *k*–1 by attaching one of them as the leftmost child of root of the other one.
- This feature is central to the merge operation of a binomial heap, which is major advantage over conventional heaps.



Operations on Binomial Heaps

- Make-Heap().
- Insert(H, x), where x is a node to be inserted in H.
- Minimum(H).
- Extract-Min(H).
- Union(H1, H2): merge H1 and H2, creating a new heap.
- Decrease-Key(H, x, k): decrease x.key (x is a node in H) to k

Binomial Heap of order k



Binomial Heap: key points

A binomial heap is implemented as a set of binomial trees satisfying properties:

Each binomial tree in a heap obeys the min-heap property: the key of a node is greater than or equal to key of its parent.

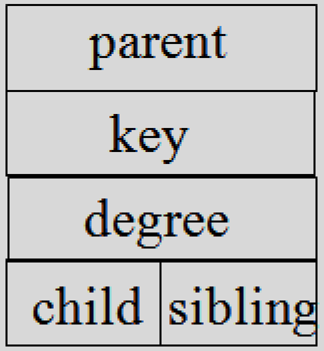
There can only be either one or zero binomial trees for each order, including zero order.

First property ensures that root of each binomial tree contains smallest key in the tree, that applies to entire heap

Roots of the binomial trees can be stored in a linked list, ordered by increasing order of the tree

Binomial heap Node contents

Each node is represented by a structure like this

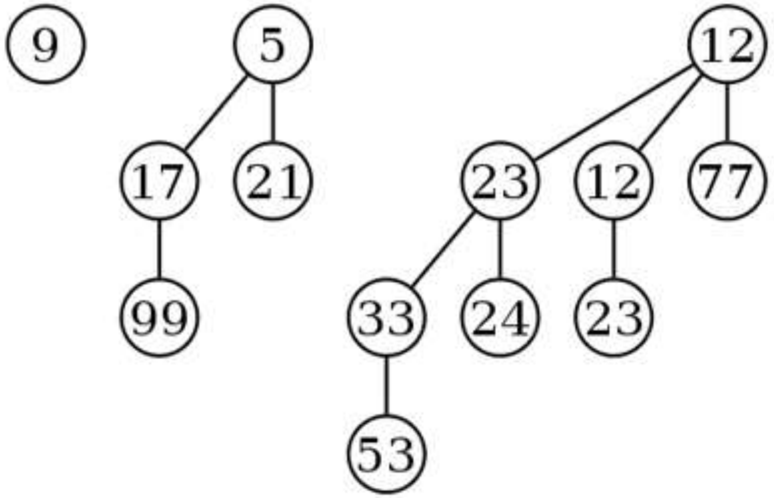


Binomial Heap Properties

A binomial heap with n nodes consists of at most log n + 1 binomial trees.

Number and orders of these trees are uniquely determined by number of nodes n: each binomial tree corresponds to one digit in binary representation of number n.

For example number 13 is 1101 in binary, $2^3 + 2^2 + 2^0$, thus a binomial heap with 13 nodes will consist of three binomial trees of orders 3, 2, and 0

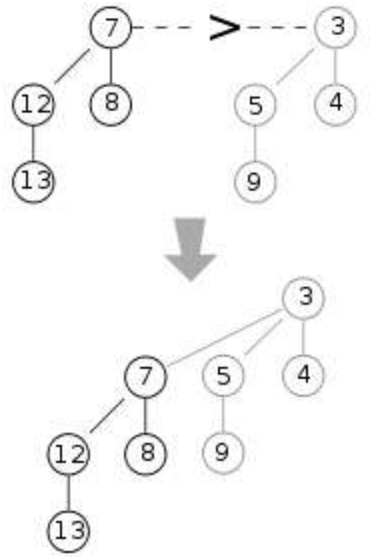


Merging two heaps

Simplest and most important operation is the merging of two binomial trees of the same order within two binomial heaps.

As their root node is the smallest element within the tree, by comparing the two keys, the smaller of them is the minimum key, and becomes new root node.

Then other tree become a subtree of the combined tree. This operation is basic to the complete merging of two binomial heaps



Merging Two heaps

If only one of the heaps contains a tree of order *j*, this tree is moved to the merged heap. If both heaps contain a tree of order *j*, the two trees are merged to one tree of order *j*+1 so that the minimum-heap property is satisfied. Later it is necessary to merge this tree with some other tree of order *j*+1 present in one of the heaps.

Merging Two heaps

Because each binomial tree in a binomial heap corresponds to a bit in the binary representation of its size, there is an analogy between the merging of two heaps and the binary addition of the *sizes* of the two heaps, from right-to-left.

Whenever a carry occurs during addition, this corresponds to a merging of two binomial trees during the merge.

Like binary addition:

```
  1 1 1     (carries)
0 0 1 1 1
1 0 0 1 1
---------
1 1 0 1 0
```

temporarily have three trees of this degree

Insert

Inserting a new element to a heap can be done by simply creating a new heap containing only this element and then merging it with the original heap. Due to the merge, insert takes O(log *n*) time, however it has an *amortized* time of O(1).

Find Min

To find the minimum element of the heap, find minimum among the roots of the binomial trees. This can be done in $O(\log n)$ time, as there are just $O(\log n)$ trees and hence roots to examine.

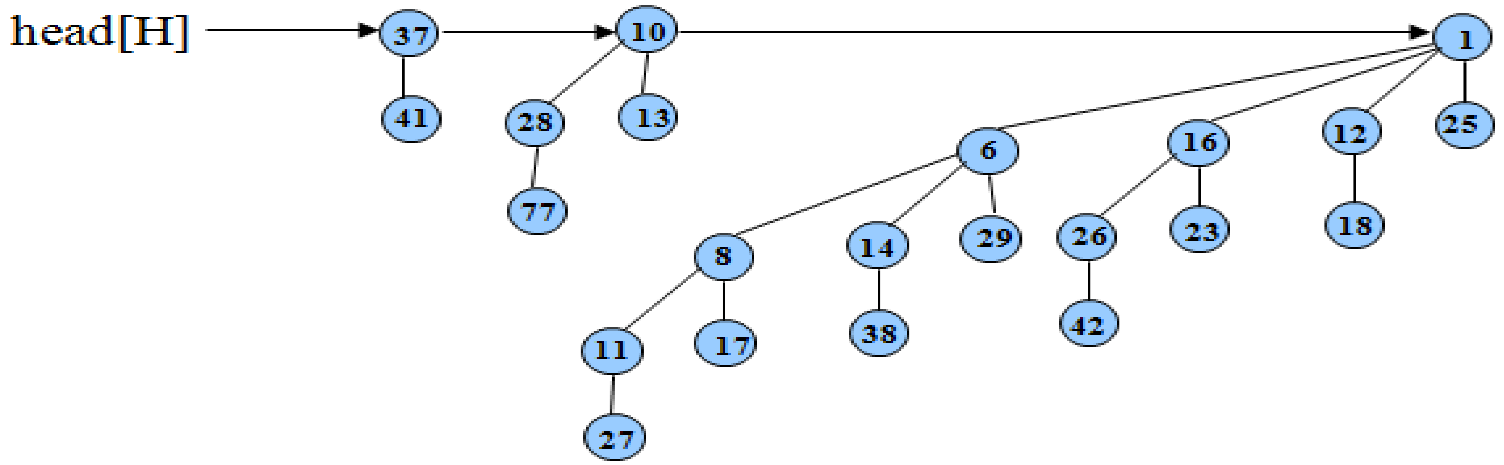
By using a pointer to the binomial tree that contains the minimum element, the time for this operation can be reduced to $O(1)$. The pointer must be updated when performing any operation other than Find minimum. This can be done in $O(\log n)$ without raising the running time of any operation.

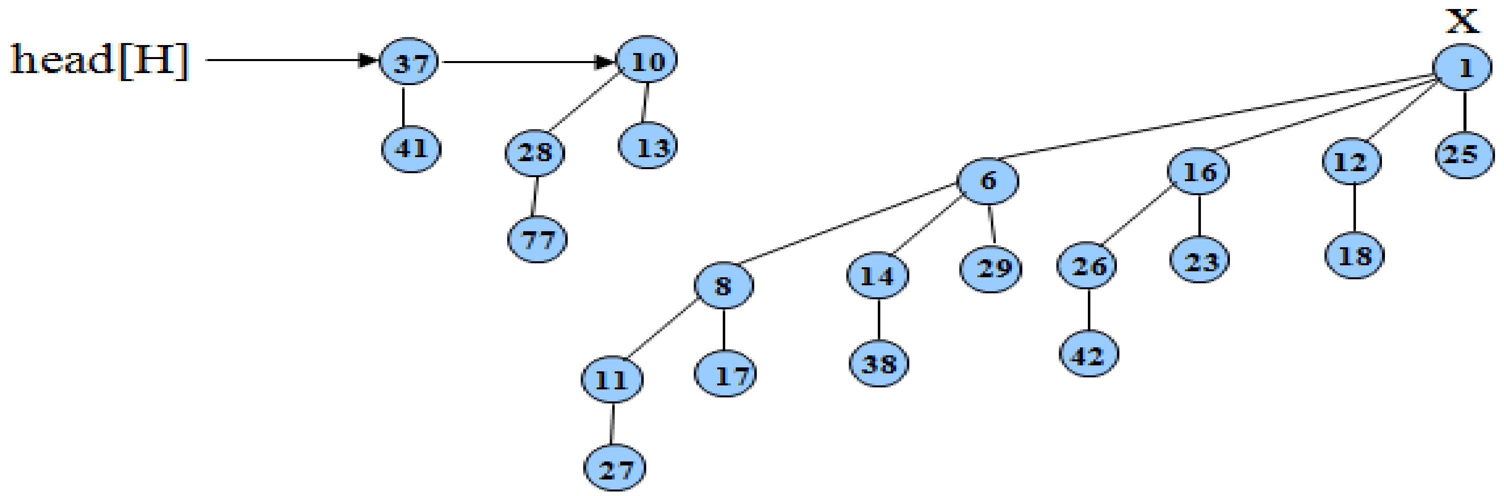
Delete Min

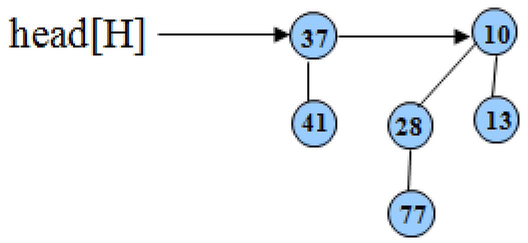
To delete the minimum element from the heap, first find this element, remove it from its binomial tree, and obtain a list of its subtrees.

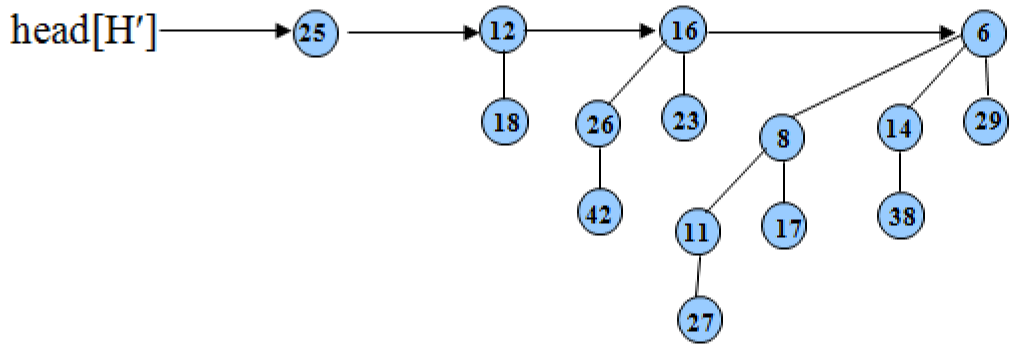
Then transform this list of subtrees into a separate binomial heap by reordering them from smallest to largest order.

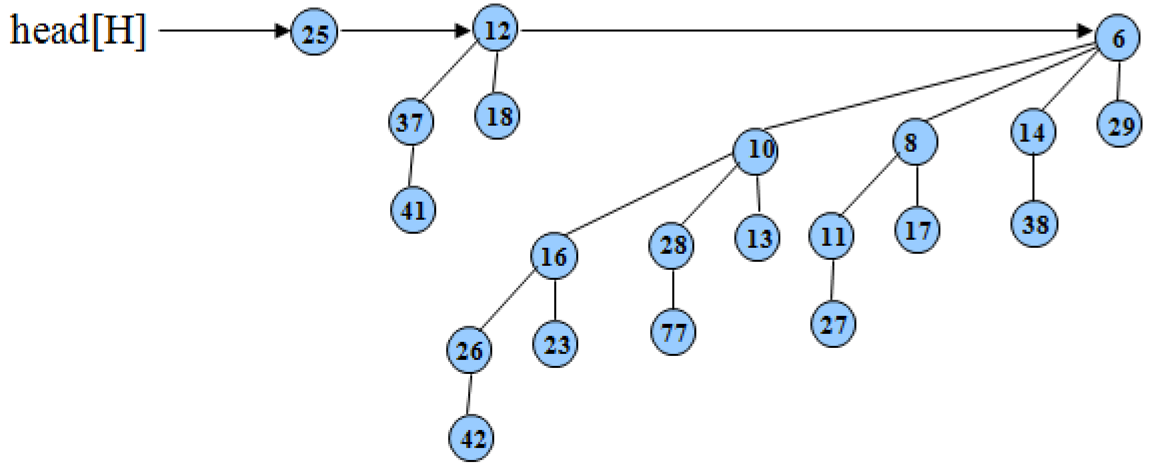
Then merge this heap with the original heap. Since each tree has at most log n children, creating this new heap is $O(\log n)$. Merging heaps is $O(\log n)$, so the entire delete minimum operation is $O(\log n)$


head[H]
37
41
10
28
77
13
1
6
8
11
27
17
14
38
29
16
26
42
23
12
18
25


head[H]
37
41
10
28
77
13
x
1
6
8
11
27
17
14
38
29
16
26
42
23
12
18
25

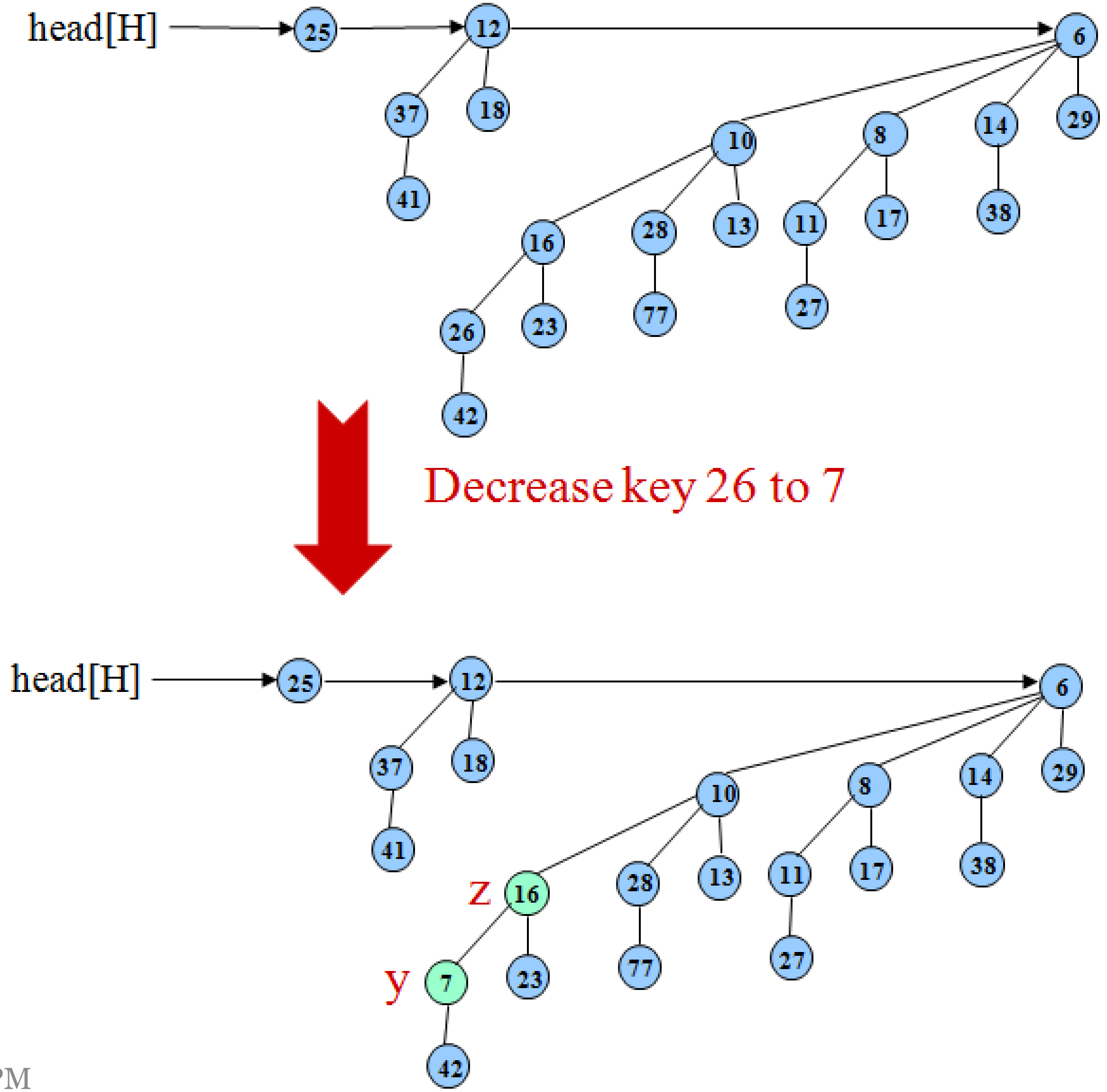

head[H]
37
10
41
28
13
77

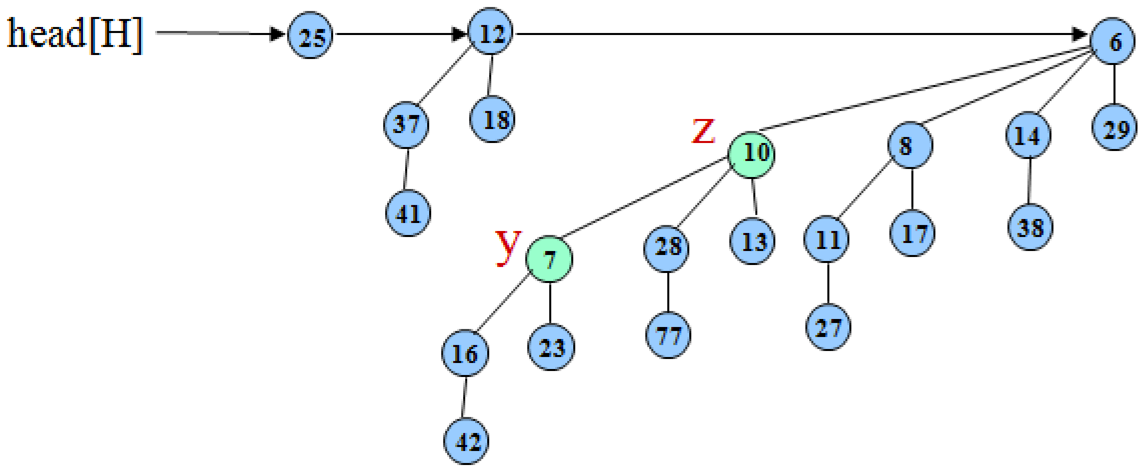

head[H′]
25
12
16
6
18
26
23
8
14
29
42
11
17
38
27

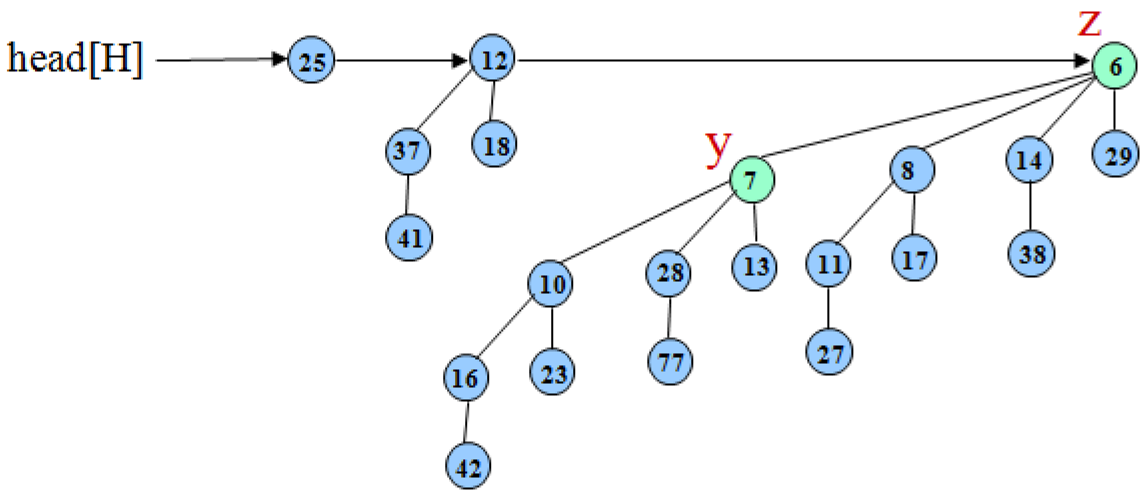

head[H]
25
12
6
37
18
10
8
14
29
41
16
28
13
11
17
38
26
23
77
27
42

Decrease Key

- Decreasing the key of an element may make it smaller than the key of its parent, violating the minimum-heap property.
- In this case, exchange the element with its parent, and possibly also with its grandparent, and so on, until the minimum-heap property is no longer violated.
- Each binomial tree has height at most log n, so this takes $O(\log n)$ time.


head[H]
25
12
6
37
18
10
8
14
29
41
16
28
13
11
17
38
26
23
77
27
42
Decrease key 26 to 7
head[H]
25
12
6
37
18
10
8
14
29
41
z
16
28
13
11
17
38
y
7
23
77
27
42


head[H]
25
12
6
37
18
z
10
8
14
29
41
y
7
28
13
11
17
38
16
23
77
27
42


head[H]
25
12
z
6
37
18
y
7
8
14
29
41
10
28
13
11
17
38
16
23
77
27
42

Delete

To delete an element from the heap, decrease its key to negative infinity (that is, some value lower than any element in the heap) and then delete the minimum in the heap.

Fibonacci Heaps

It is a collection of trees satisfying the minimum-heap property. This implies that the minimum key is always at the root of one of the trees.

Compared with binomial heaps, the structure of a Fibonacci heap is more flexible. The trees do not have a prescribed shape and in the extreme case the heap can have every element in a separate tree.

This flexibility allows some operations to be executed in a "lazy" manner, postponing the work for later operations.

For example merging heaps is done simply by concatenating the two lists of trees, and operation decrease key sometimes cuts a node from its parent and forms a new tree.

Node Representation

Each node x in a Fibonacci heap contains:

A pointer p(x) to its parent node

A pointer child(x) to any one of its children

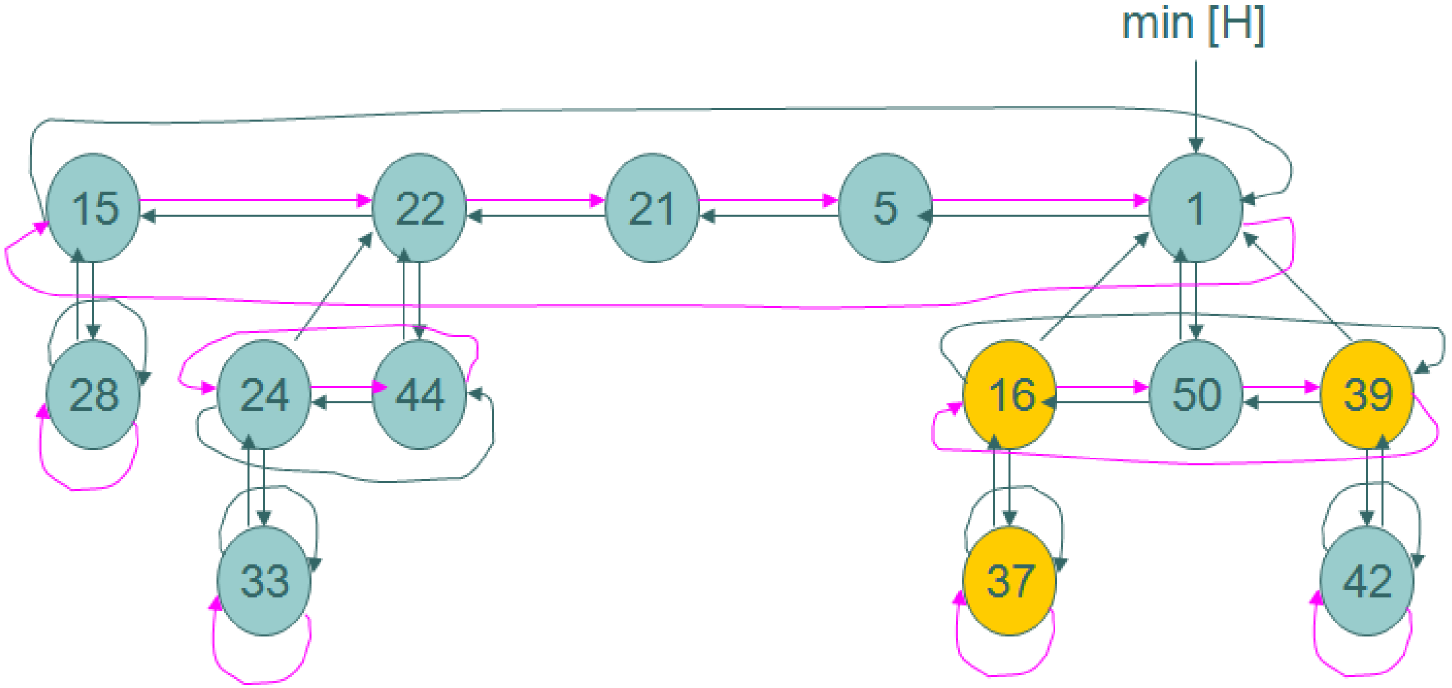
A left pointer left(x) to its left sibling

A right pointer right(x) to its right sibling

Variable deg(x), gives number of children in the child list of x

Variable mark(x), Boolean value indicating if node x is marked or not

Fibonacci Heap



Fibonacci Heaps

Represent trees using left child, right sibling pointers and circular doubly linked list that helps in quickly splicing off sub trees

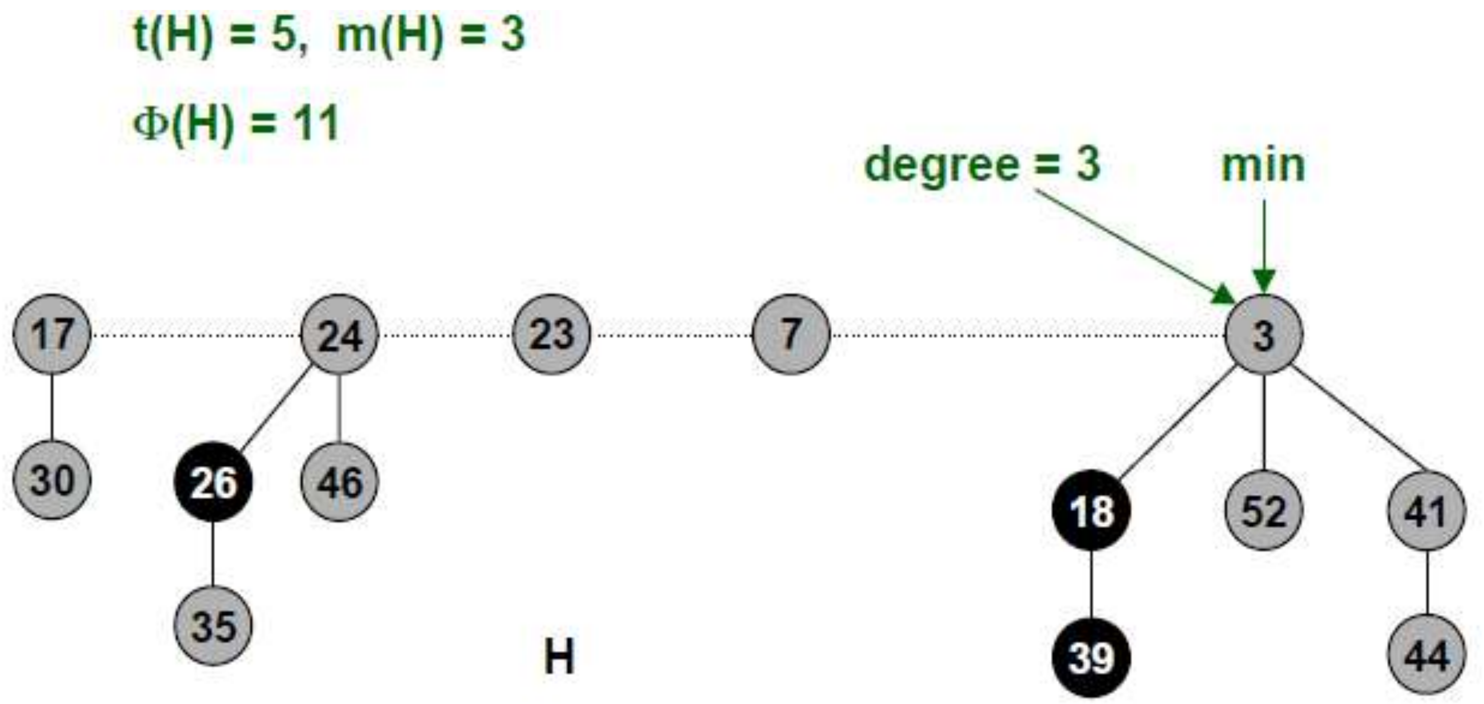
Roots of trees connected with circular doubly linked list help in fast union

Pointer to root of tree with min element helps in fast find min

Key quantities

- Degree[x]= degree of node x
- Mark[x]= mark of node x (black or gray)
- t(H)=#of trees
- m(H)=# of marked nodes
- Φ(H)=t(H)+2m(H)=potential function

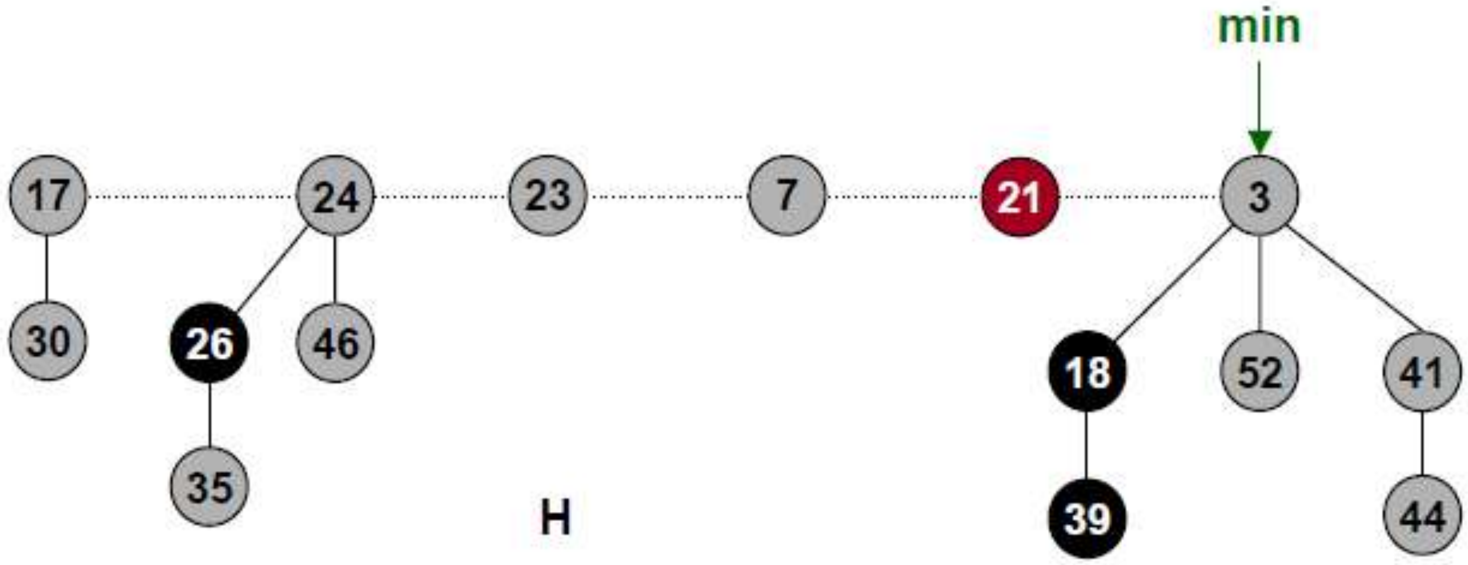
Fibonacci Heap Example



Insert

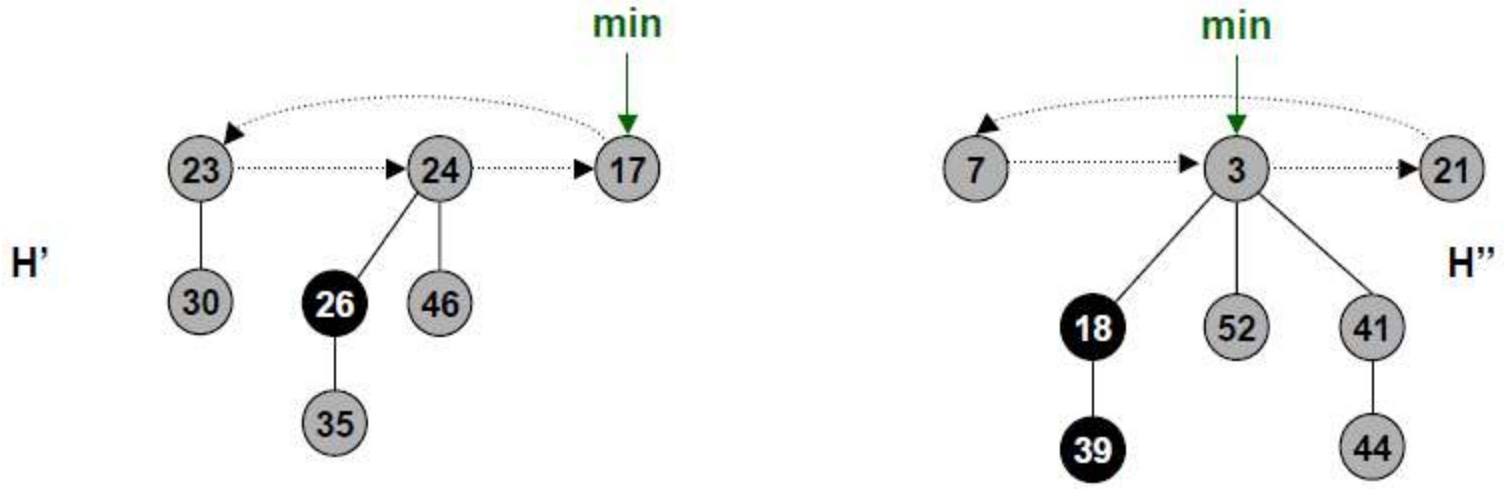
- Create a new singleton tree
- Add to the left of min pointer
- Update min pointer
- Actual Cost=O(1)
- Change in Potential=+1
- Amortized Cost=O(1)

Insert 21



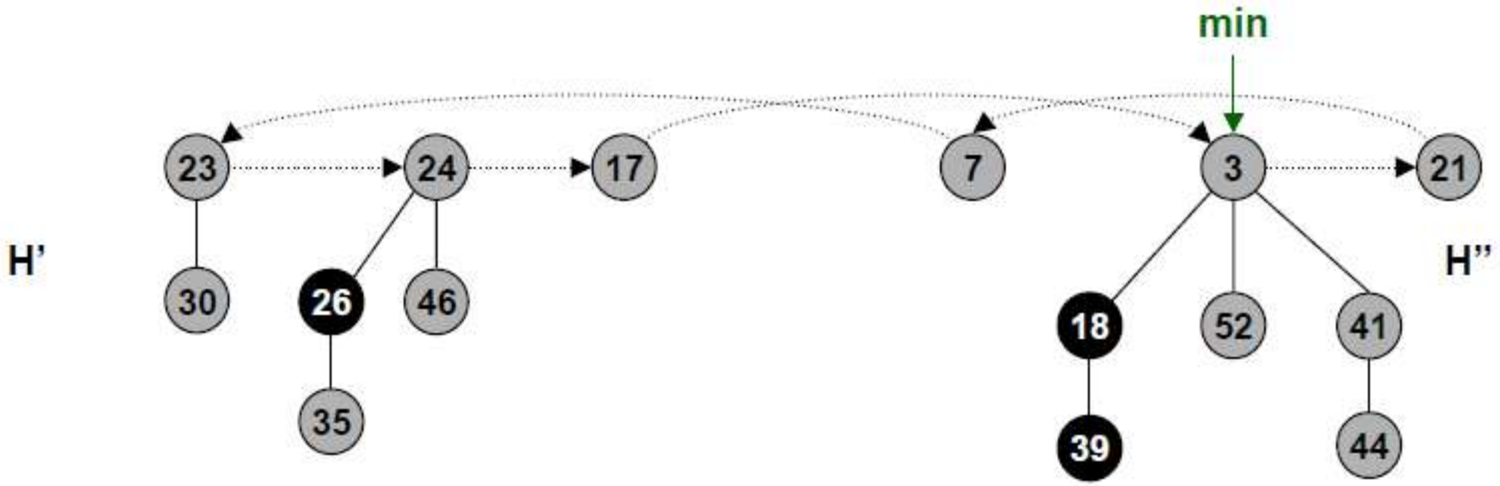
Union

- Concatenate two Fibonacci Heaps



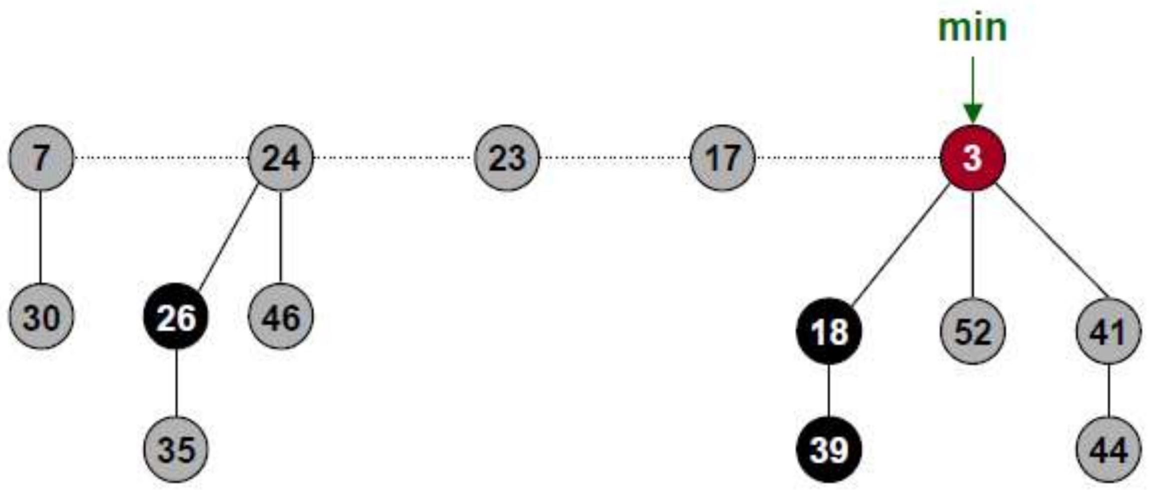
Union of two heaps

- Actual Cost = O(1)
- Change in Potential=0
- Amortized Cost=O(1)



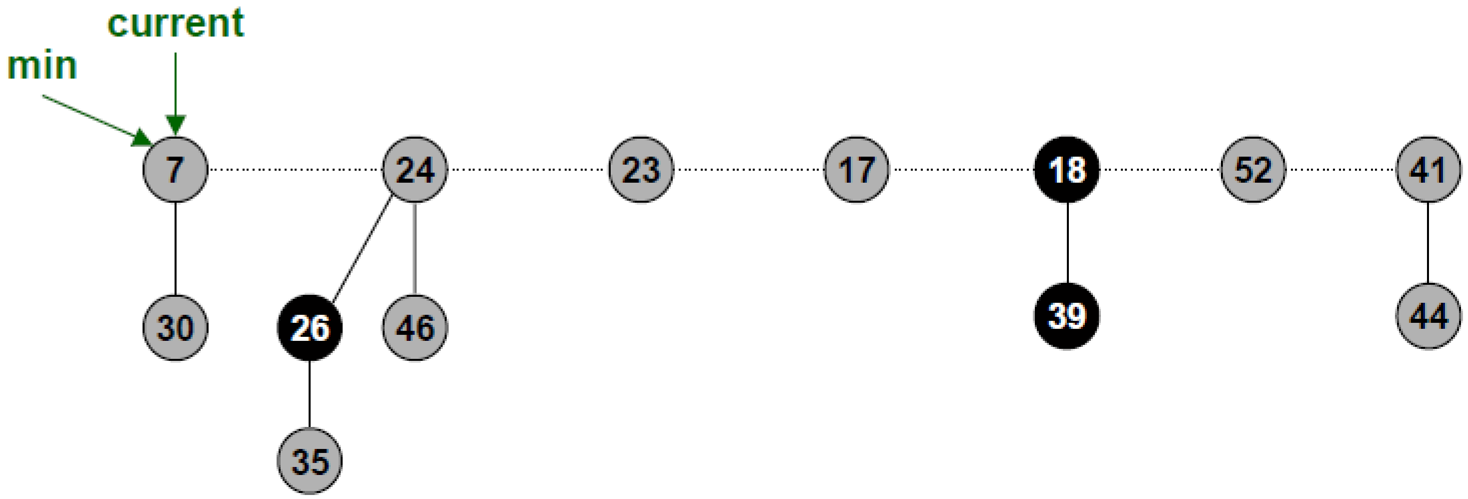
Delete Min

- Delete Min and concatenate its children into root list



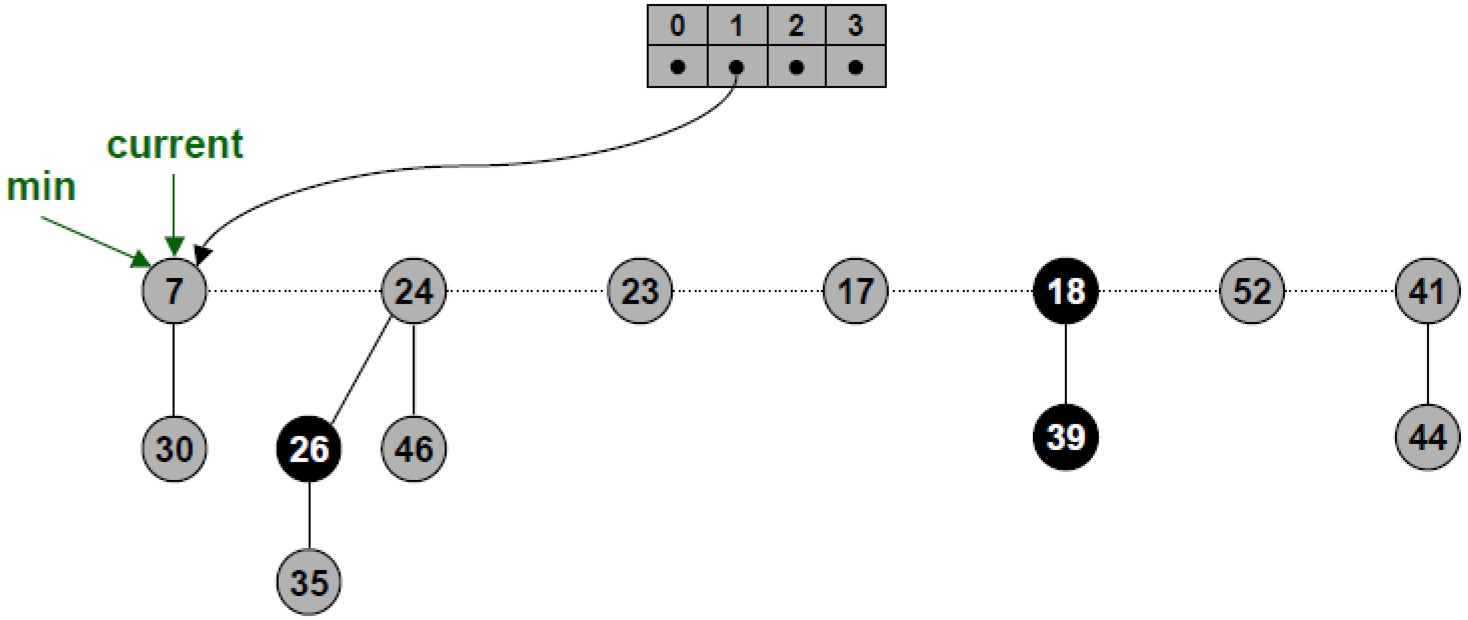
Delete Min

- Min Pointer is pointing to the new min

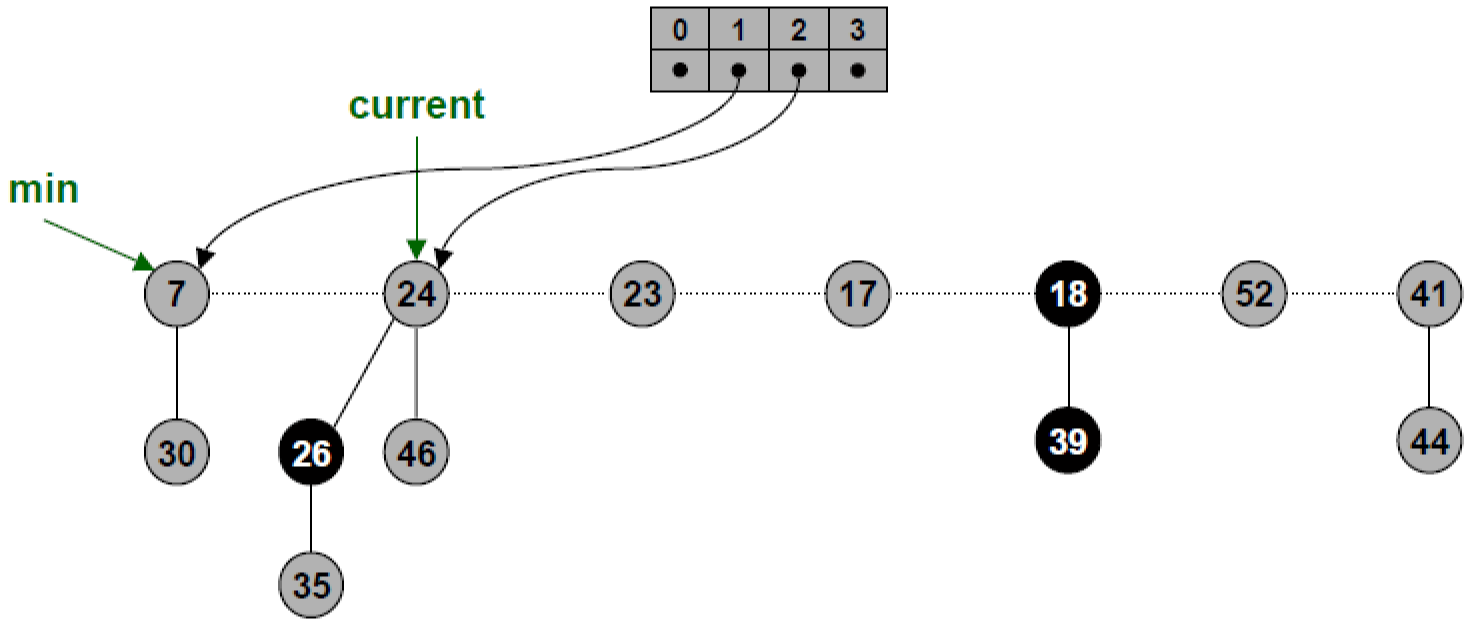


Delete Min

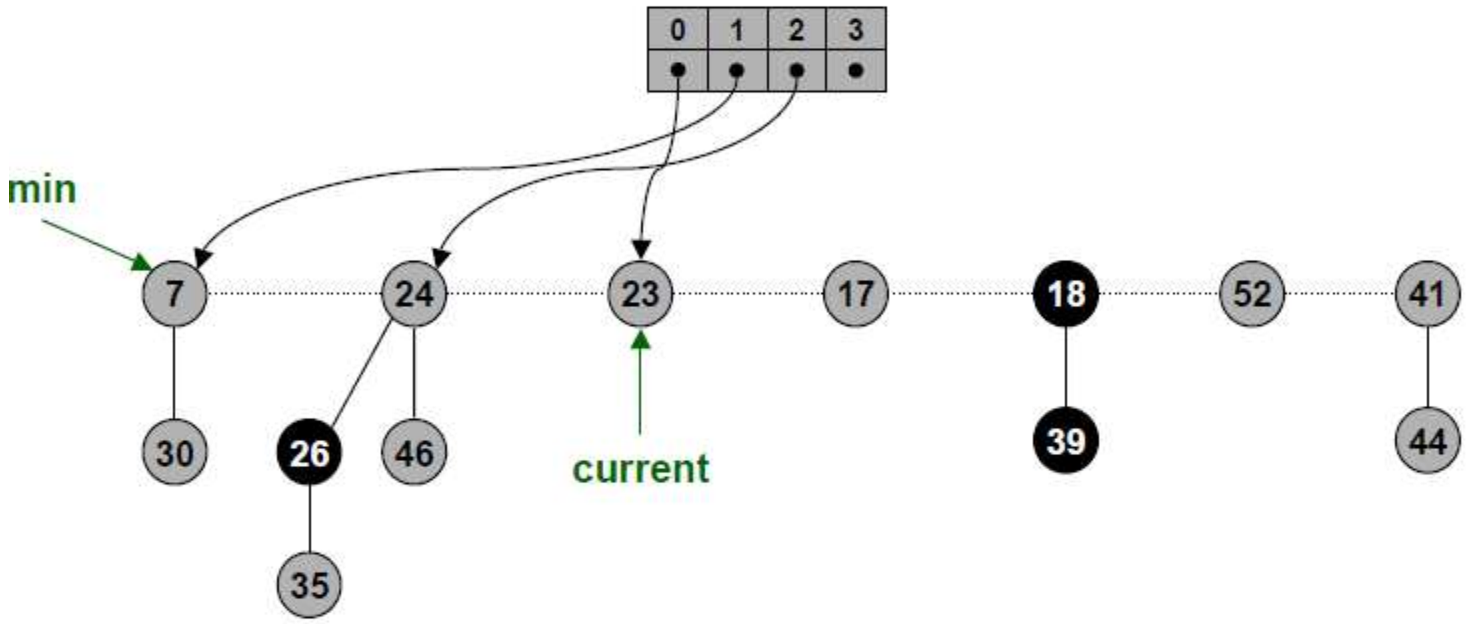
- Consolidate trees so that no two trees have same degree



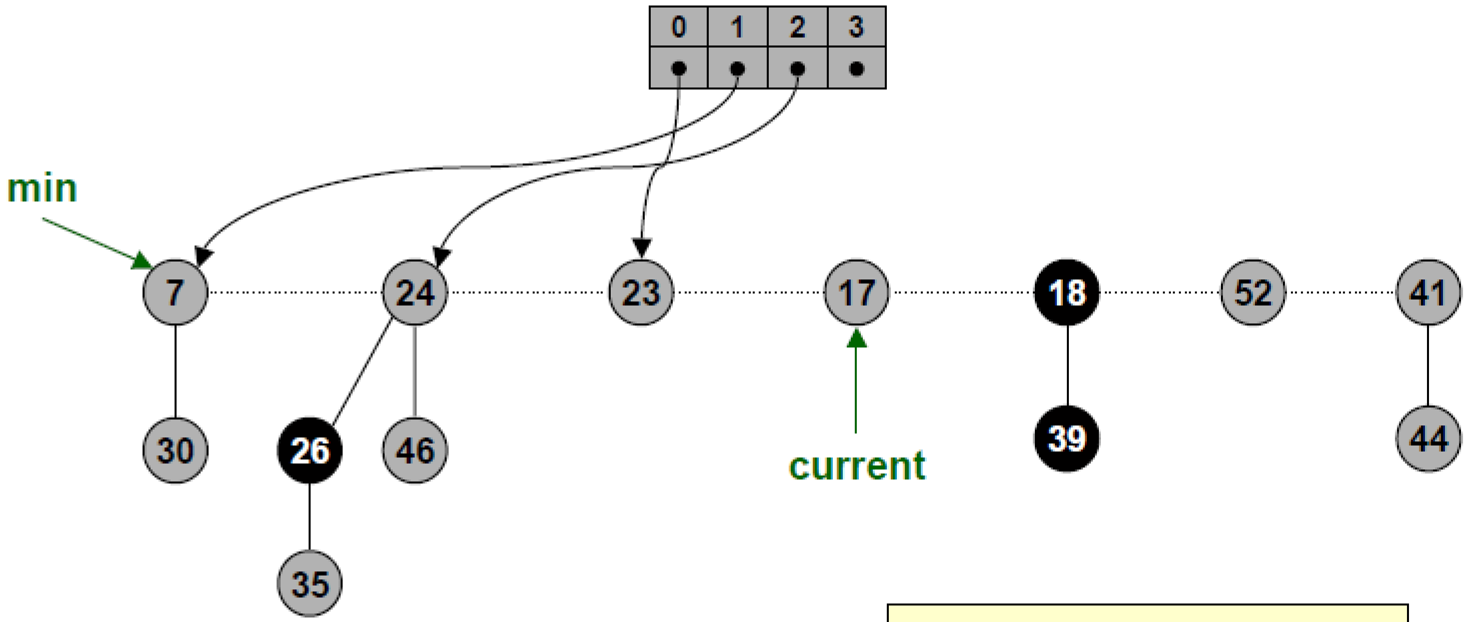
Delete Min



Delete Min

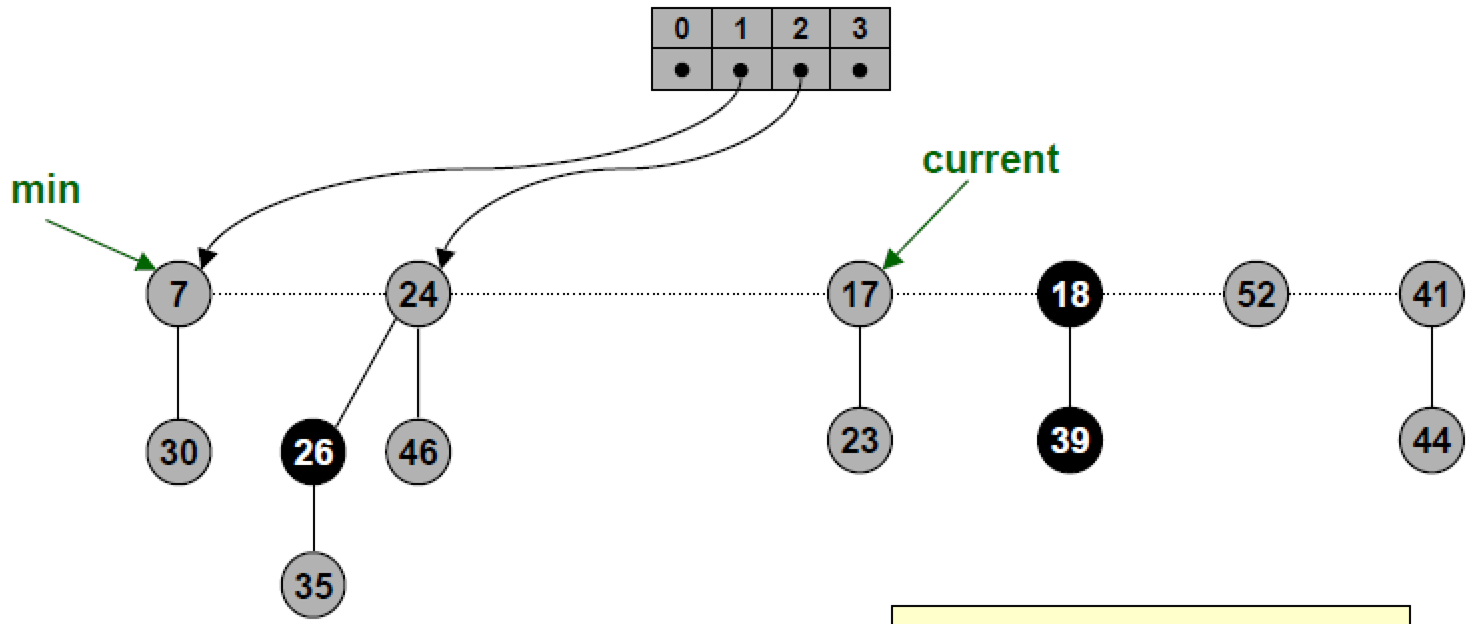


Delete Min



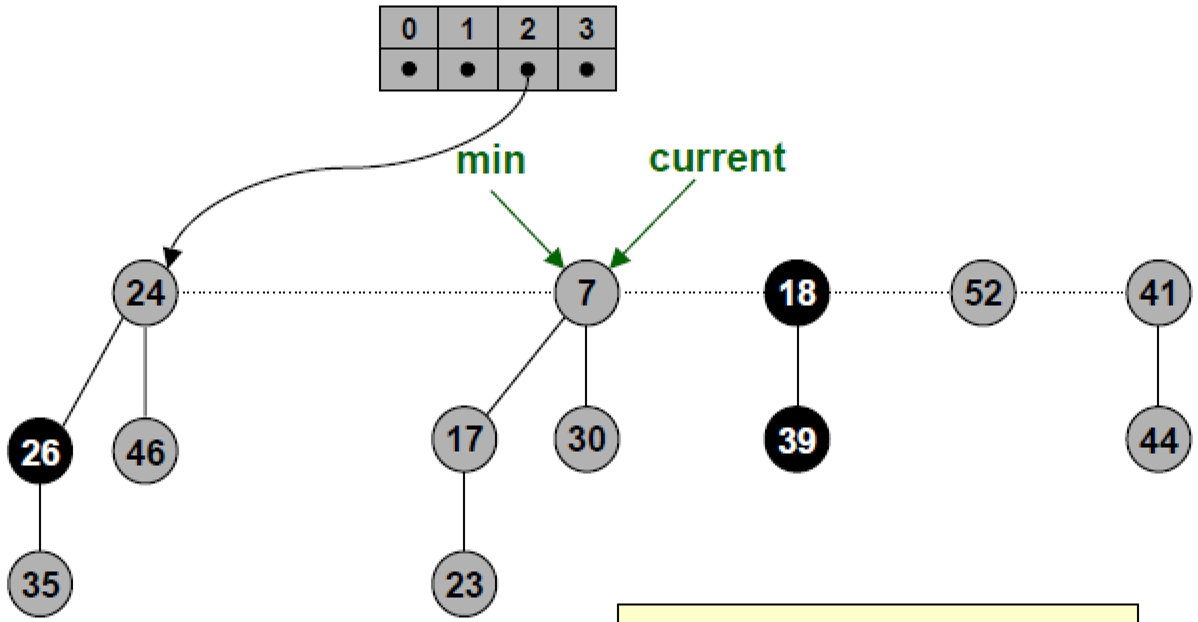
Merge 17 and 23 trees.

Delete Min



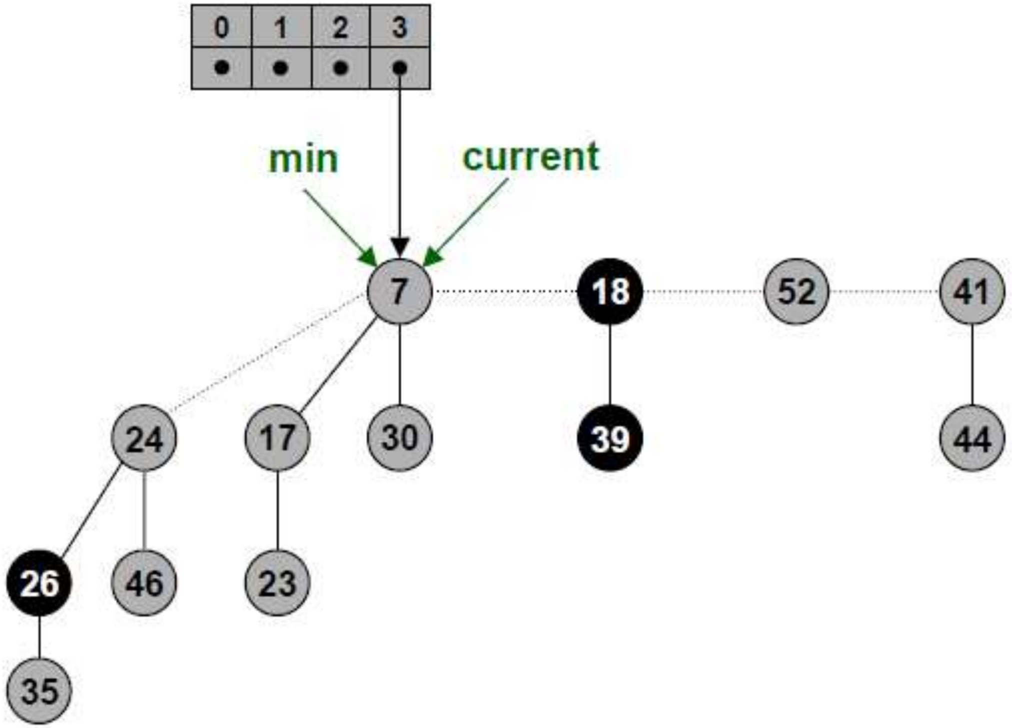
Merge 7 and 17 trees.

Delete Min

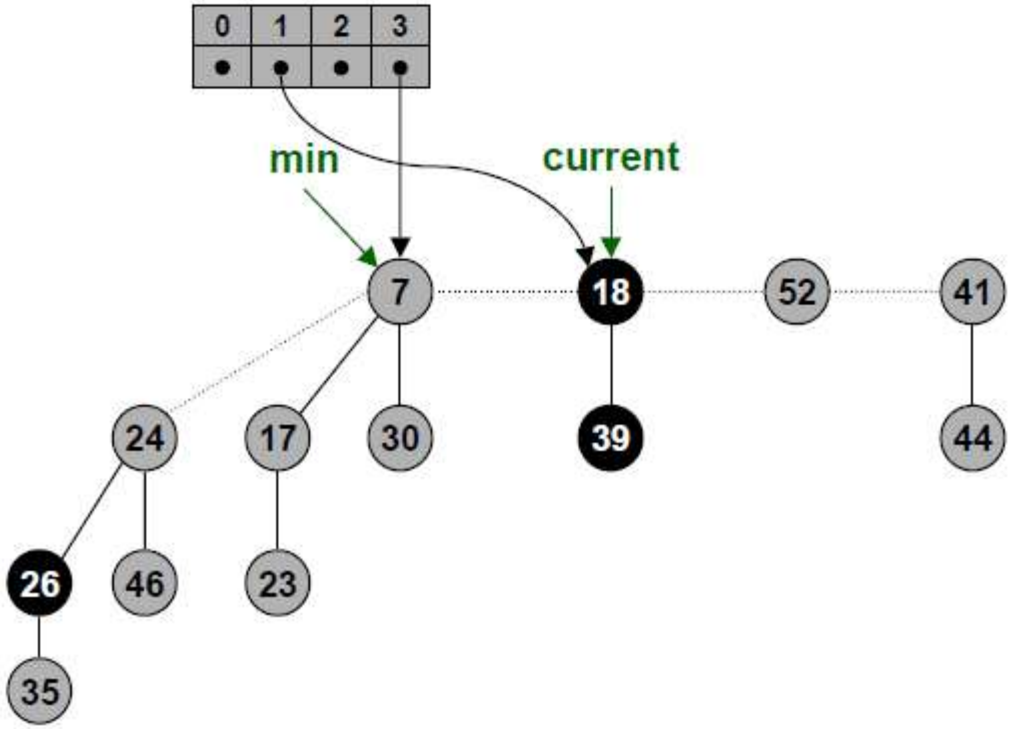


Merge 7 and 24 trees.

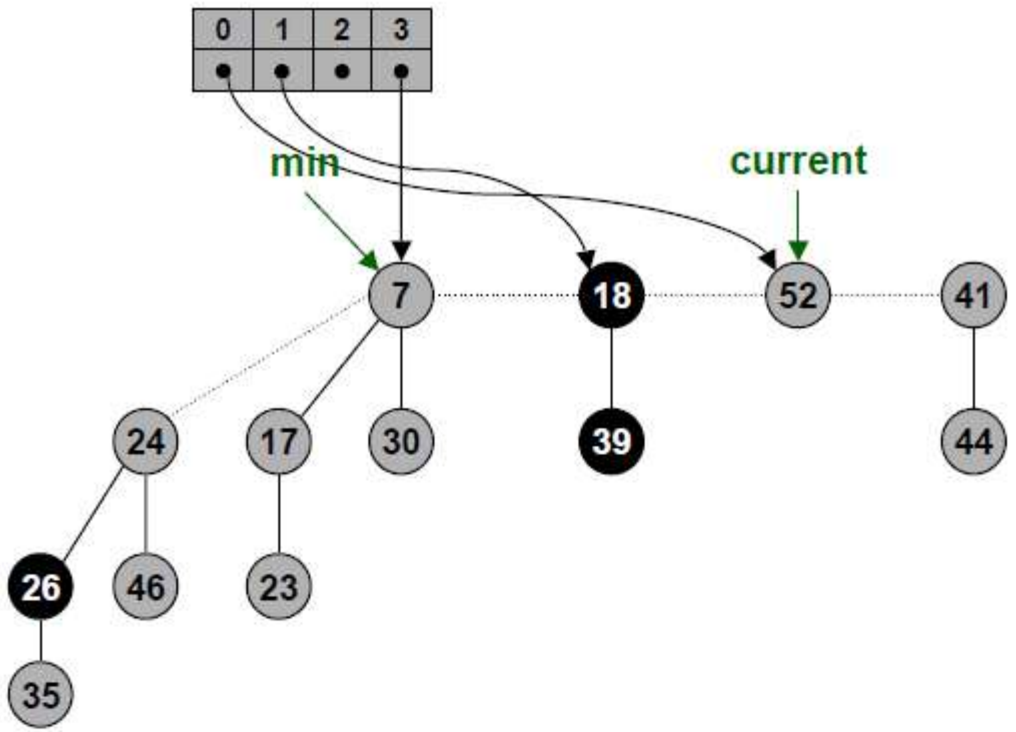
Delete Min



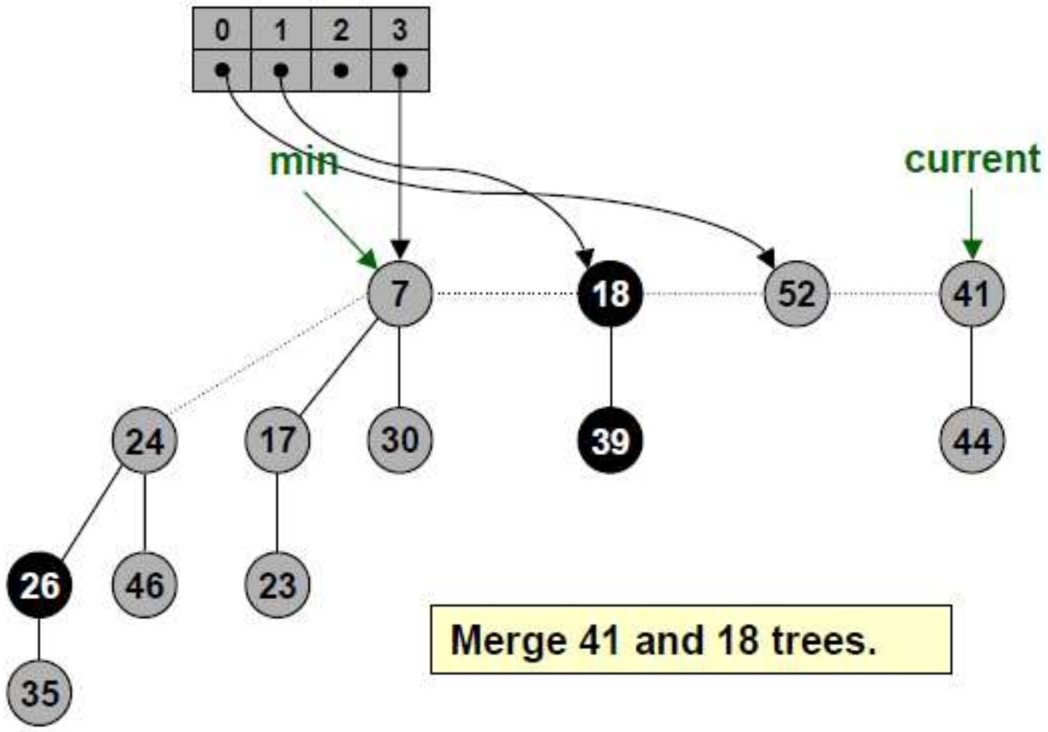
Delete Min



Delete Min

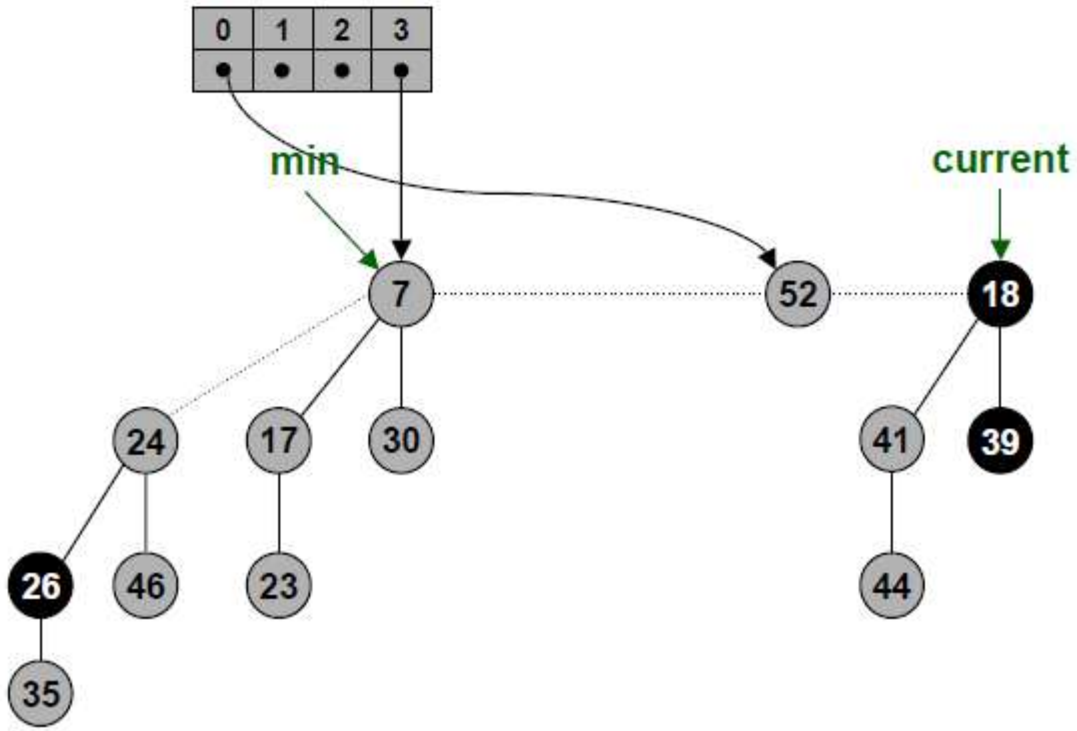


Delete Min

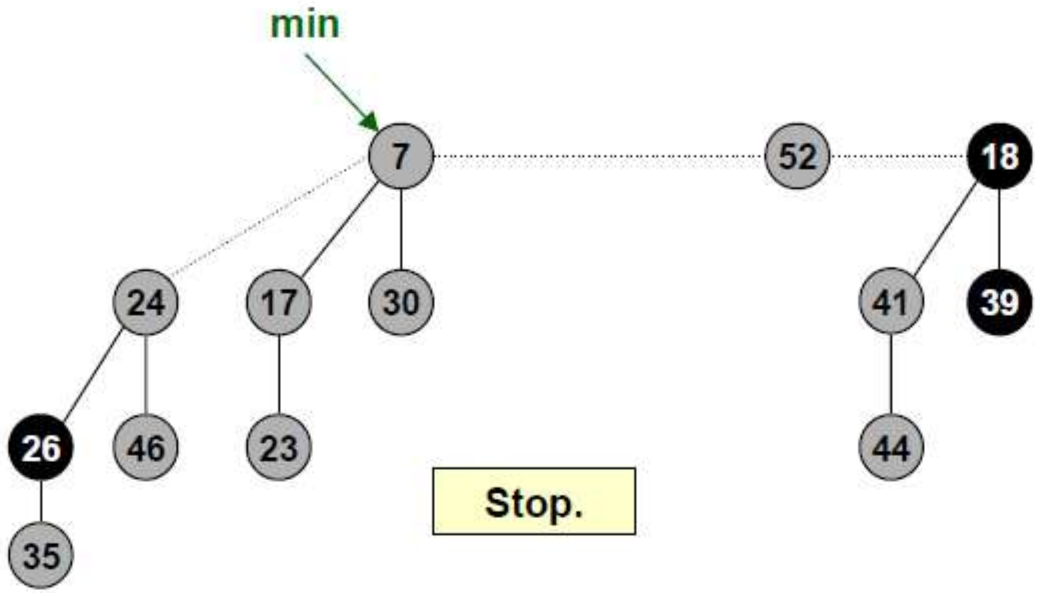


Merge 41 and 18 trees.

Delete Min



Delete Min



Delete Min Analysis

- Actual Cost = O(D(n)+t(H))
- O(D(n)) work adding min's children into root list and updating min.

- At most D(n) children of min node

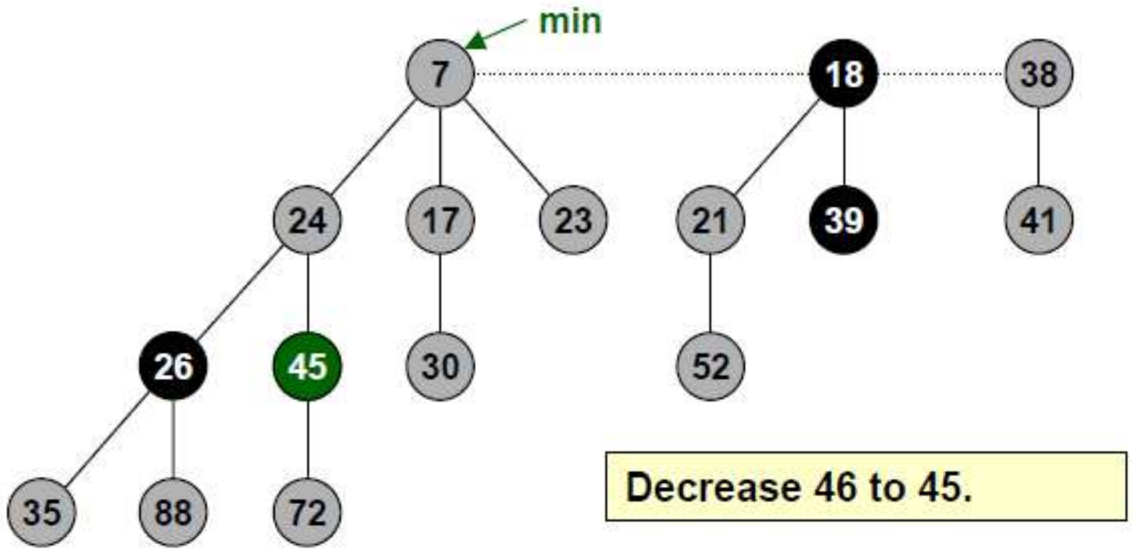
- O(D(n)+t(H)) work consolidating trees
- Work is proportional to size of root list since number of roots decreases by one after each merging
- $\leq$ D(n) + t(H) -1 root nodes at beginning of consolidation
- Amortized Cost O(D(n))

- t(H') $\leq$ D(n) +1 since no two trees have same degree

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

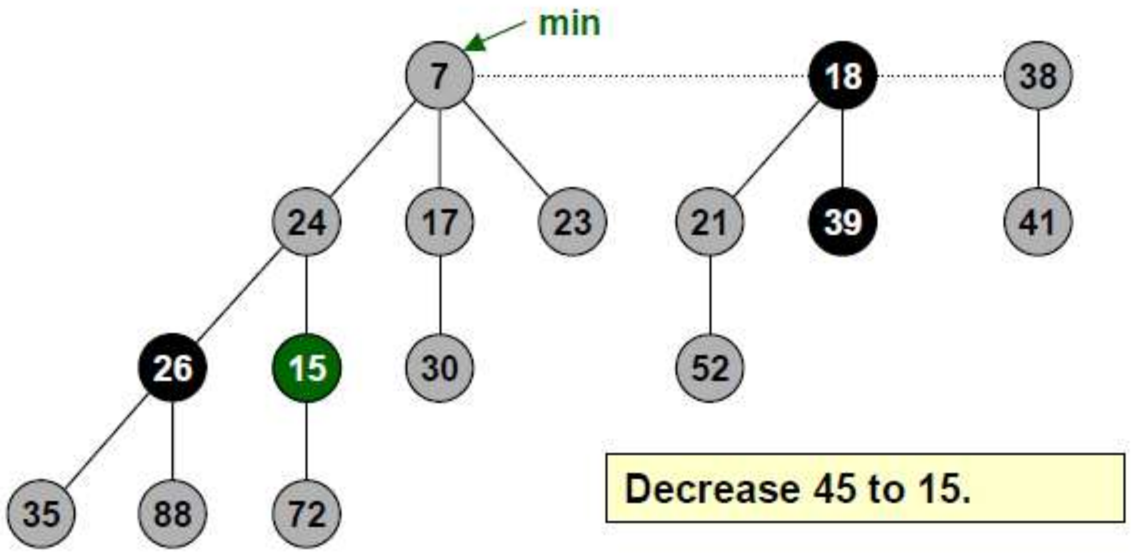
Decrease Key

- Case 1: Min Heap Property is not violated
- Decrease key x to K
- Change heap min pointer if necessary



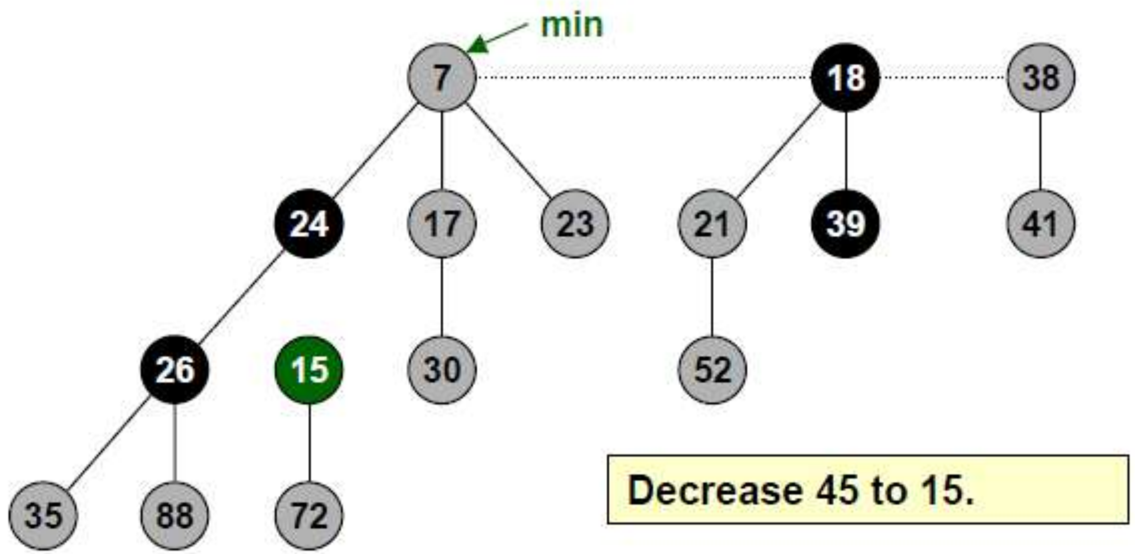
Decrease Key Case 2

- Parent of x is unmarked
- Decrease key of x to k



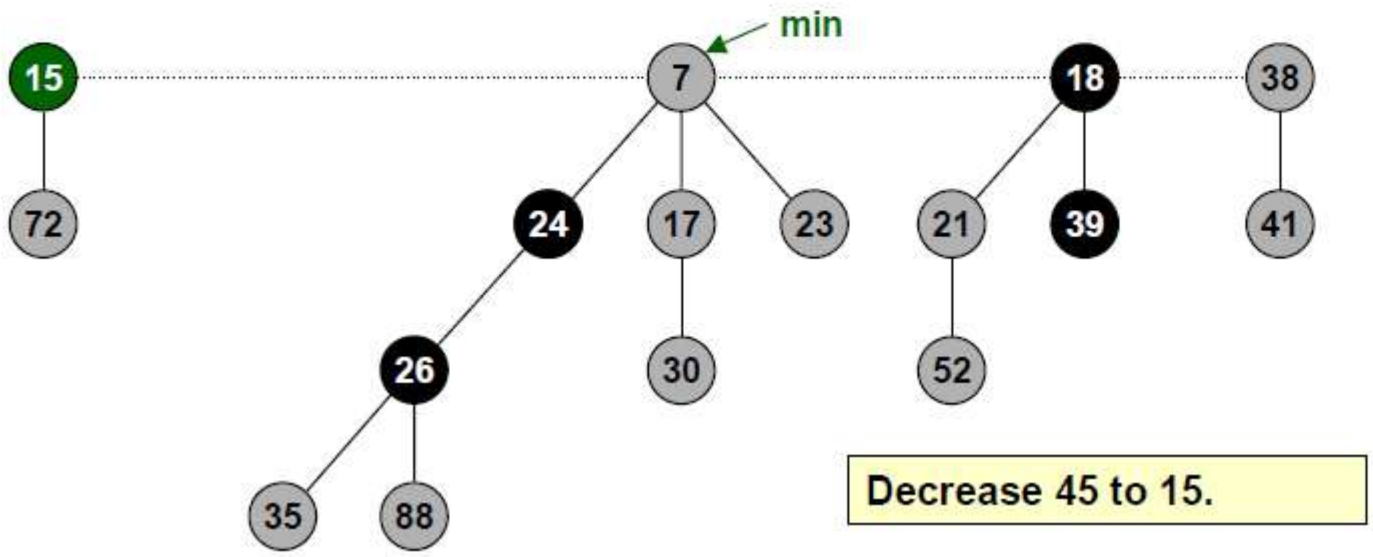
Decrease key case 2

- Cut off link between x and its parent
- Mark Parent



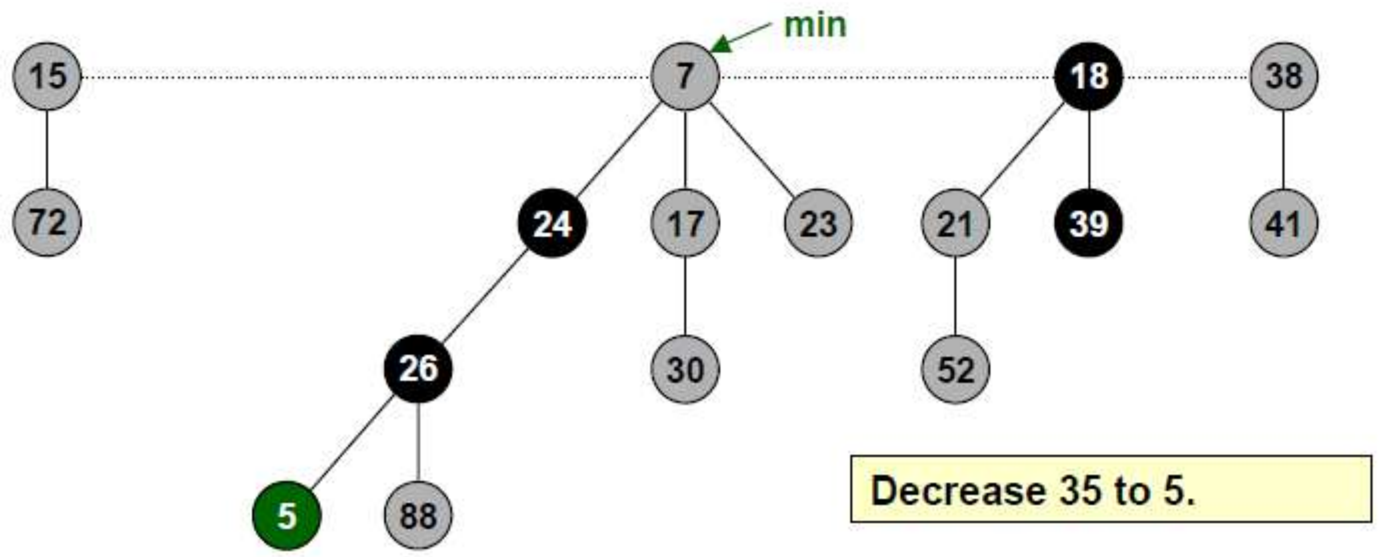
Decrease key case 2

- Add tree rooted at x to the root list updating min pointer



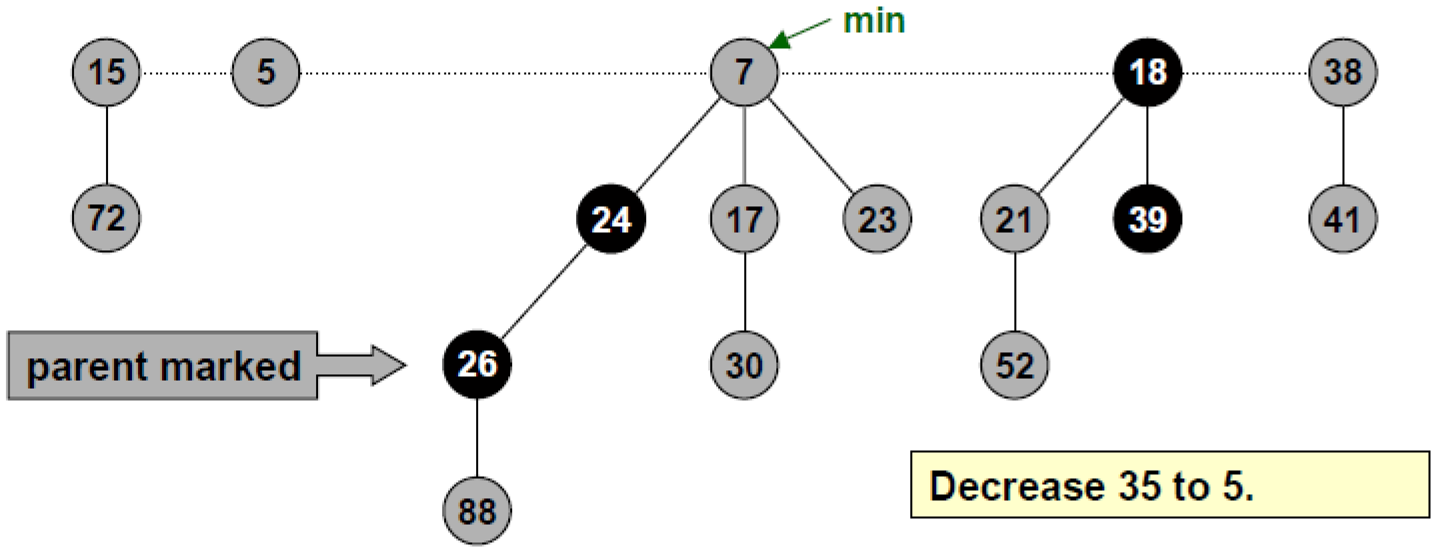
Decrease Key Case 3

- Parent of x is marked
- Decrease key x to k



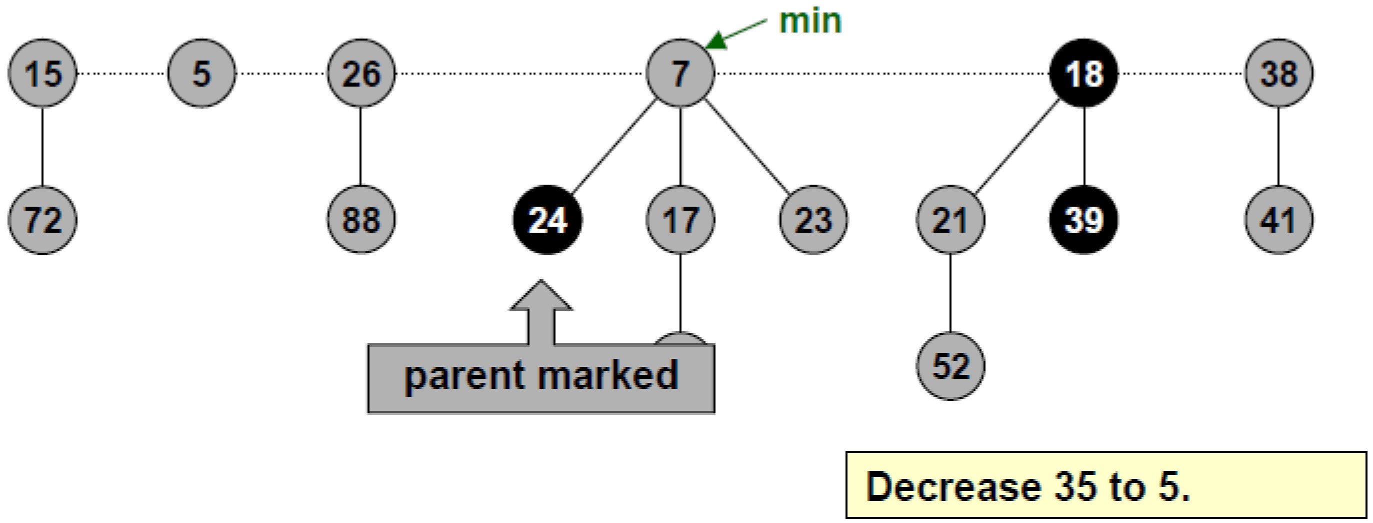
Decrease key case 3

- Cut off link between x and its parent p[x] and add x to root list



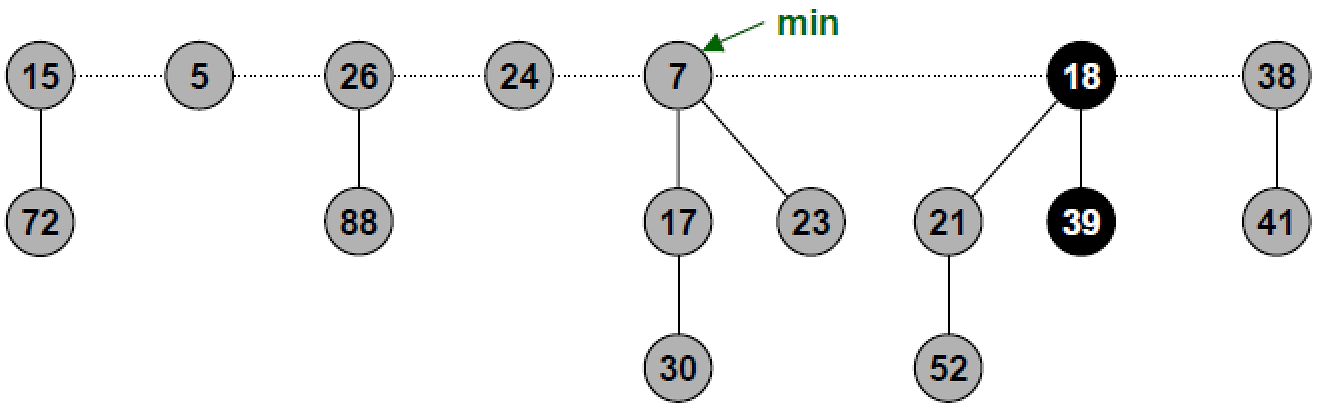
Decrease key case 3

- Cut off link between p[x] and p[p[x]] and add p[x] to root list



Decrease Key case 3

- If p[p[x]] is unmarked then mark it
- If p[p[x]] marked cut off p[p[x]] , unmark and repeat



Decrease 35 to 5.

Marked Nodes

A node is marked if at least one of its children was cut since this node was made a child of another node (all roots are unmarked).

Decrease key Analysis

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

- each cascading cut unmarks a node

- last cascading cut could potentially mark a node

- $\triangle\Phi \leq c + 2\ (-c + 2) = 4 - c$

Questions, Comments and Suggestions

Question 1

Using Fibonacci heaps for ________ improves the asymptotic running time of important algorithms.

A) Priority Queues
B) Stacks
C) Link Lists
D) Binary Search Trees

Question 2

Binomial heap is a heap similar to a _______ but also supports quickly merging two heaps

A) Fibonacci Heap

B) Binary Heap

C) Max Heap

D) Min Heap

Question 3

Show an valid Binomial Heap with the following nodes 3, 5, 7, 10, 12, 15.