Lecture 3: Mergeable Heaps

Binomial heaps

October 18, 2019

Mergeable heaps

Data structures designed to support well the following operations:

- `makeBinomialHeap()`: creates and returns a new empty heap.
- `insert`(H, x): inserts node referred by x, whose key field has already been filled in, into heap H.
- `minimum`(H): returns a pointer to the node with minimum key in heap H.
- `extractMin`(H): deletes the node from heap H whose key is minimum, returning a pointer to the node.
- `union`(H_1, H_2): creates and returns a new heap that contains all the nodes of heaps H_1 and H_2. Heaps H_1 and H_2 are "destroyed" by this operation.

and also

- `DecreaseKey`(H, x, k) assigns to node referred by x within heap H the new key value k. It is assumed that $key \leq x{-}{>}key$.
- `Delete`(H, x) deletes node referred by x from heap H.

Content of the lecture

In this lecture we will study binomial heaps. Later on, after we will explain the notion of *amortized time bounds* of algorithms, we will study Fibonacci heaps. If we also add binary heaps to the picture, we will be able to draw a comparison table of their running times:

Procedure	Binary heap (worst-case)	Binomial heap (worst case)	Fibonacci heap (amortized)
`makeBinomialHeap`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`insert`	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
`minimum`	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
`extractMin`	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
`union`	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
`DecreaseKey`	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
`Delete`	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

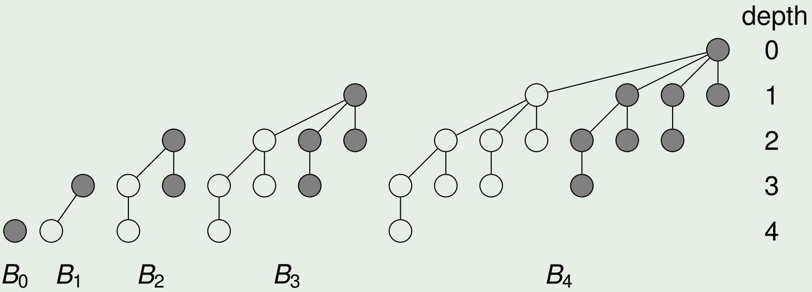
Note. The notion of *amortized time bounds* will be explained in another lecture.

Binomial trees

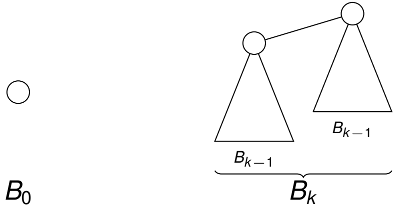
A binomial tree is an element of the set of ordered trees $\{B_k \mid k \in \mathbb{N}\}$ defined recursively as follows:

- B_0 consists of a single node.
- B_k consists of two binomial trees B_{k-1} that are linked such that one of them has the other one as left child of its root.

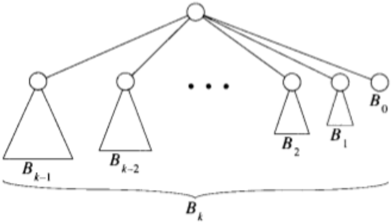
Example (B_0, B_1, B_2, B_3, and B_4)



- The recursive view (Cf. the definition)



- Another view



Properties of binomial trees

Lemma 1

The binomial tree B_k has the following properties:

1. It has 2^k nodes.
2. It has height 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 $k-1, k-2, \ldots, 0$, then child i is the root of a subtree B_i.

Binomial heaps

A binomial heap H is a set of binomial trees that satisfies the following **binomial heap properties**:

1. Each binomial tree in H is heap-ordered: the key of a node is greater than or equal to the key of its parent.
2. There is at most one binomial tree in H whose root has a given degree.

Binomial heaps

A binomial heap H is a set of binomial trees that satisfies the following **binomial heap properties**:

1. Each binomial tree in H is heap-ordered: the key of a node is greater than or equal to the key of its parent.
2. There is at most one binomial tree in H whose root has a given degree.

Consequences of the definition

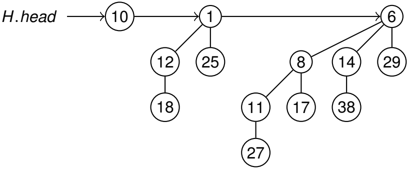
- The root of a heap-ordered tree contains the smallest key in the tree.
- The second property implied that an n-node binomial heap H has at most $\lfloor \log_2 n \rfloor + 1$ binomial trees. This is so because:
  - The binary representation of $n$has $\leq \lfloor \log_2 n \rfloor + 1$ bits, say $b_{\lfloor \log_2 n \rfloor}, b_{\lfloor \log_2 n \rfloor - 1}, \ldots, b_0$, so that $n = \sum_{i=0}^{\lfloor \log_2 n \rfloor} b_i \cdot 2^i$. From Lemma 1 we learn that B_i appears in H if and only if $b_i = 1$. Thus, H contains at most $\lfloor log_2 n \rfloor + 1$ binomial trees.

Representing binomial heaps

- Binomial trees are stored in the left-child, right-sibling representation.
- Each node x contains the following pointers:
  - $x.p$: pointer to its parent. It is nil if x is a root.
  - $x.child$: pointer to its left child. It is nil if x has no children.
  - $x.sibling$: pointer to its right sibling. It is nil if x is the rightmost child of its parent.

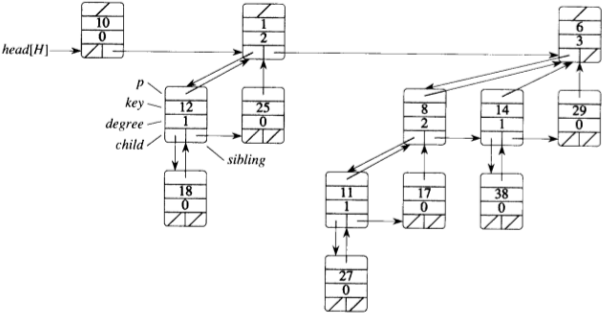
  and the fields
  - $x.key$: the key of the node.
  - possibly other fields for the satellite information required by the application.
  - $x.degree$: the number of children of x.
- The roots of the binomial trees of a binomial heap H for a linked list of nodes, connected through their *sibling* pointers. This list is called called the **root list** of the heap. The degrees of the roots strictly increase as we traverse the root list.
- A binomial heap H is accessed by the field $H.head$, which is a pointer to the first node in the root list of H.

Binomial heaps



A binomial heap H with $n = 13$ nodes, made of trees B_0, B_2, and B_3. The key of any node is $\geq$ the key of its parent.

The binomial heap is represented as a linked list of the roots of the trees it contains, in order of increasing degree.



A more detailed view of the illustrated heap.

In a binomial heap with n nodes, the degrees of the roots of its binomial trees are a subset of $\{0, 1, \ldots, \lfloor \log_2 n \rfloor\}$.

Operations on binomial heaps (1)

- `makeBinomialHeap()` creates an empty binomial heap h by allocating space for H and setting $H.head = \textsc{Nil}$. The running time is, obviously, $\Theta(1)$.
- `minimum`(H) returns a pointer to the node with the minimum key in an n-node heap. It is assumed that there are no keys with value ∞ in H.
  - Binomial heaps are heap-ordered $\Rightarrow$ the minimum key is in a node in the root list of the heap.
  - The root list had length $\leq \lfloor \log_2 n \rfloor + 1 \Rightarrow$ the running time of this operation is $O(\log_2 n)$.

```
minimum(H)
y := NIL
x := H.head
min := ∞
while x ≠ NIL
      if x->key < min
         min = x->key
         y := x
      x := x->sibling
return y
```

Operations on binomial heaps (2)

Uniting two binomial heaps

`union`(H_1, H_2) performs the union of binomial heaps H_1, H_2 by linking repeatedly binomial trees whose roots have the same degree.

- If y and z are roots of two B_{k-1}-trees, then `BinomialLink`(y, z) makes a B_k-tree with root z by setting the parent of y to be z, the leftmost child of z to be y, and increasing the degree of z by 1:

```
BinomialLink(y, z)
y->p := z
y->sibling := z->child
z->child := y
z->degree := z->degree + 1
```

`BinomialHeapMerge`(H_1, H_2) merges the root lists of binomial heaps H_1 and H_2 into a single linked list that is sorted by degree into monotonically increasing order.

Operations on binomial heaps (2)

```
union(H1, H2)
H := makeBinomialHeap()
H.head = BinomialHeapMerge(H1, H2)
free the objects H1 and H2, but not the lists they point to
if H.head = NIL
   return H
prev_x := NIL
x := H.head
next_x := x->sibling
while next_x ≠ NIL
      if x->degree ≠ next_x->degree or
         (next_x->sibling ≠ NIL and next_x->sibling->degree = x->degree)
            prev_x := x                                      // Cases 1 and 2
            x := next_x                                      // Cases 1 and 2
      else if x->key ≤ next_x->key
            x->sibling := next_x->sibling                    // Case 3
            BinomialLink(next_x, x)                          // Case 3
         else if prev_x = NIL                                // Case 4
                 H.head := next_x                            // Case 4
              else prev_x->sibling := next_x                 // Case 4
              BinomialLink(x, next_x)                        // Case 4
              x := next_x                                    // Case 4
      next_x := x->sibling
return H
```

It works in 2 phases:

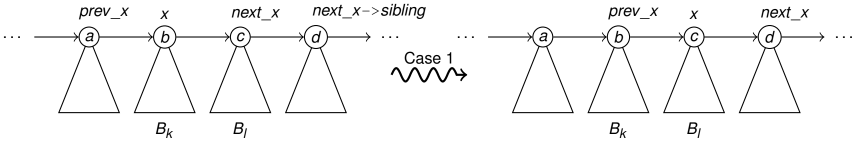
1. First, `BinomialHeapMerge`(H_1, H_2) merges the root lists of H_1 and H_2 into a single list H that is sorted by degree into monotonically increasing order.
2. There might be at most 2 roots of each degree $\Rightarrow$ the second phase links roots of equal degree until at most one root remains of each degree.

The meaning of the local pointers used in the algorithm is:

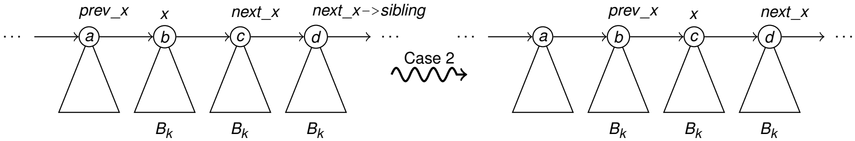
- x points to the root currently being examined.
- $prev\_x$ points to the root preceding x on the root list, thus $prev\_x{-}{>}sibling = x$.
- $next\_x$ points to the root following x on the root list, thus $x{-}{>}sibling = next\_x$.

There are four cases, illustrated below, where $l > k$.
Labels a, b, c, d serve only to identify the roots involved; they do not indicate the degrees or keys of these roots.

Case 1: $x\text{->}degree \neq next\_x\text{->}degree$.

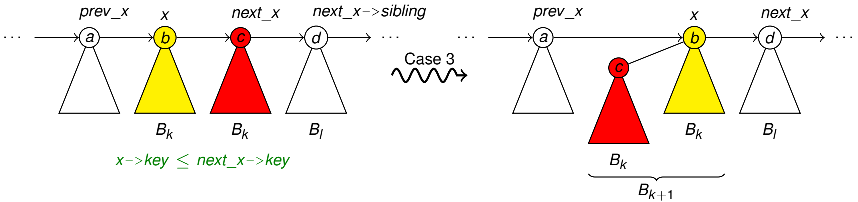


Case 2: $x\text{->}degree = next\_x\text{->}degree = next\_x\text{->}sibling\text{->}degree$.

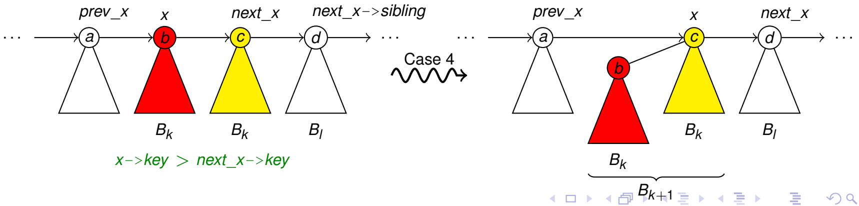


union
A study by cases (continued)

Case 3: $x\text{->}degree = next\_x\text{->}degree \neq next\_x\text{->}sibling\text{->}degree$ and $x\text{->}key \leq next\_x\text{->}key$.

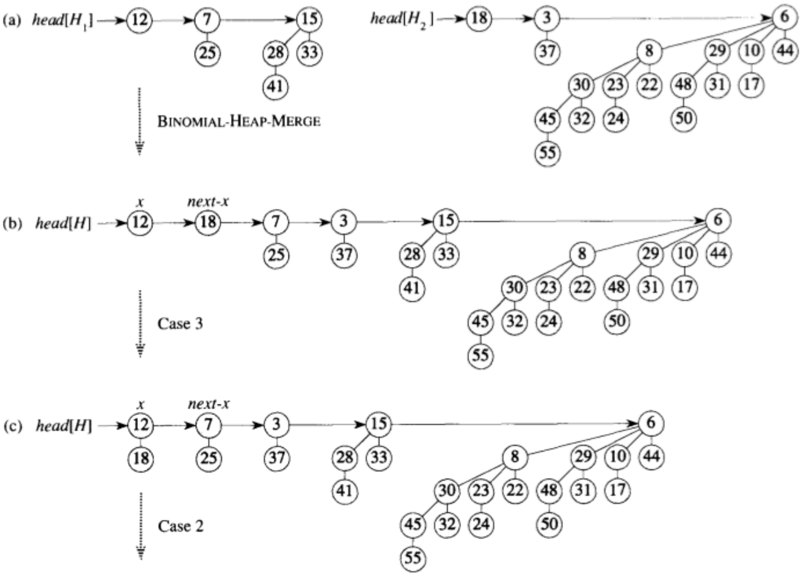


Case 4: $x\text{->}degree = next\_x\text{->}degree \neq next\_x\text{->}sibling\text{->}degree$ and $x\text{->}key > next\_x\text{->}key$.



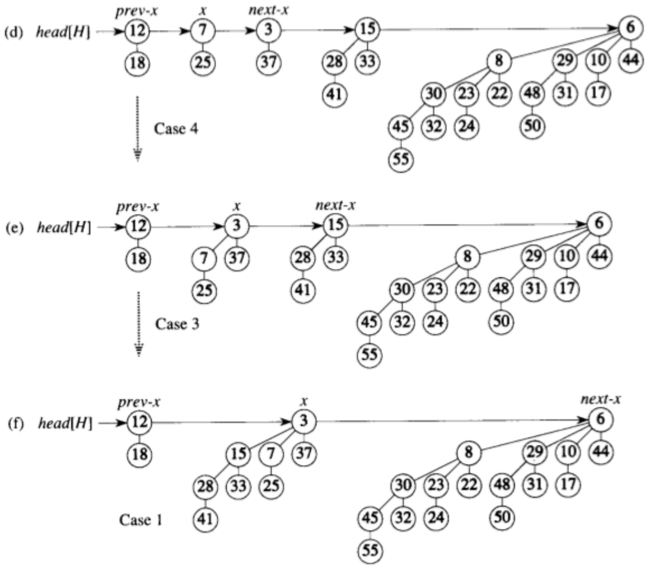
Execution of `union`

Running example



Execution of `union`

Running example



Run time complexity of `union`

Assumptions. H_1 contains n_1 nodes, H_2 contains n_2 nodes, and let $n = n_1 + n_2$.
Then

- H_1 contains $\leq \lfloor \log_2 n_1 \rfloor + 1$ roots, and H_2 contains $\leq \lfloor \log_2 n_2 \rfloor + 1$ roots
- $\Rightarrow$ H contains at most $\lfloor log_2 n_1 \rfloor + \lfloor \log_2 n_2 \rfloor + 2 \leq 2\lfloor \log_2 n \rfloor + 2 = O(\log_2 n)$ roots immediately after the call of BINOMIALHEAPMERGE.
- `BinomialHeapMerge`(H_1, H_2) takes $O(\log_2 n)$ time. Each iteration of the **while** loop takes $O(1)$ time, and there are at most $\lfloor log_2 n_1 \rfloor + \lfloor \log_2 n_2 \rfloor + 2$ iterations because each iteration either advances the pointers one position down the root list or removes a root from the root list.
- $\Rightarrow$ total run time is $O(\log_2 n)$.

Inserting a node

Assumptions: Node x has already been allocated, and key $x\text{->}key$ has already been filled in.

`insert`(H, x)
$H' :=$ `makeBinomialHeap`$()$
$x\text{->}p := \textsc{Nil}$
$x\text{->}child := \textsc{Nil}$
$x\text{->}sibling := \textsc{Nil}$
$x\text{->}degee := 0$
$H'.head = x$
$H :=$ `union`(H, H')

Extracting the node with minimum key

`extractMin`(H)

Find the root x with the minimum key in the root list of H
    and remove x from the root list of H

$H' :=$ `makeBinomialHeap`$()$

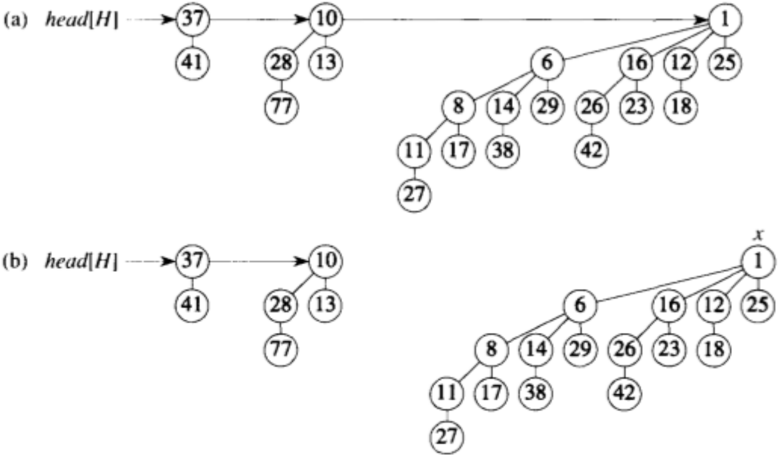
Reverse the order of the linked list of x's children,
    and set $H'.head$ to point to the head of the resulting list

$H :=$ `union`(H, H')

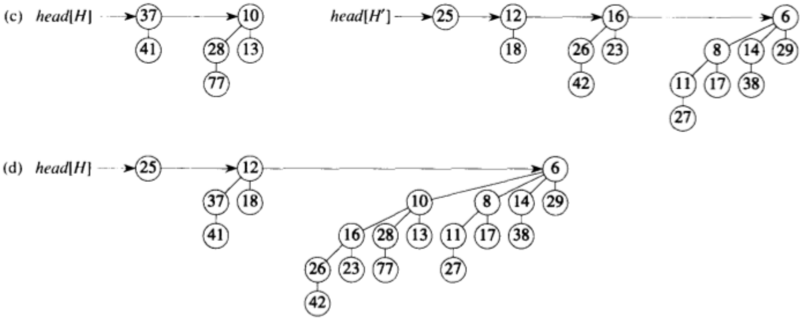
return x

Extracting the node with minimum key

Example



Extracting the node with minimum key
Example continued



Decreasing a key

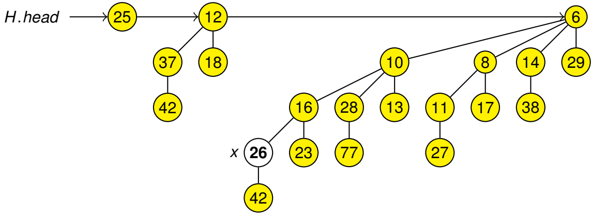
DecreaseKey(H, x, k) decreases the key of a node x in a binomial heap to a new value k. It signals an error if $k > x\text{->}key$.

DecreaseKey(H, x, k)

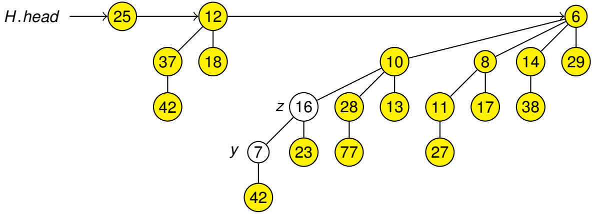
if $k > x\text{->}key$
    error "new key is greater than current key"
$x\text{->}key := k$
$y := x$
$z := y\text{->}p$
while $z \neq$ NIL and $y\text{->}key < z\text{->}key$
    exchange $y\text{->}key \leftrightarrow z\text{->}key$
    if y and z have satellite fields, exchange them, too
    $y := z$
    $z := y\text{->}p$

DecreaseKey(H, x, k)
Example

Let's decrease the key of node x in H to $k = 7$.



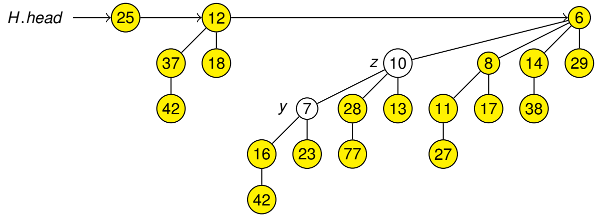
(a)



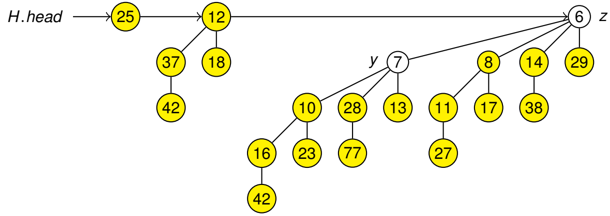
DecreaseKey(H, x, k)

Example (continued)

(b)



(c)



The decreased key "bubbles up" in the heap:

- After ensuring that $k \leq x{-}{>}key$ and then assigning key k to x, the procedure goes up the tree, with y initially pointing to x.
- In each iteration of the **while** loop of lines 6-10, $y{-}{>}key$ is checked against the key of y's parent z:
  - If y is the root or $y{-}{>}key \geq z{-}{>}key$, the tree is heap-ordered.
  - Otherwise, node y violates the heap ordering, therefore its key is exchanged with the key of its parent z, along with any other satellite information.

  The procedure then sets y to z, going up one level in the tree, and continues with the next iteration.
- The time complexity of DecreaseKey(H, x, k) is $O(\log_2 n)$ because the maximum depth of x is $\lfloor \log_2 n \rfloor$, so the **while** loop iterates at most $\lfloor \log_2 n \rfloor$ times.

Deleting a node

Deleting a node x from a binomial heap H is trivial:

- First, decrease the key of x to a value smaller than any key in H, e.g., $-\infty$.
- Next, extract from H the node with minimal key, which is x with key $-\infty$.

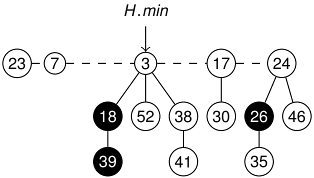
```
Delete(H, x)
1 DecreaseKey(H, x, −∞)
2 extractMin(H)
```

- If H has n nodes, then Delete(H, x) takes $O(\log_2 n)$ time.

A Fibonacci heap is a collection of heap-ordered trees. The trees are rooted, but unordered:

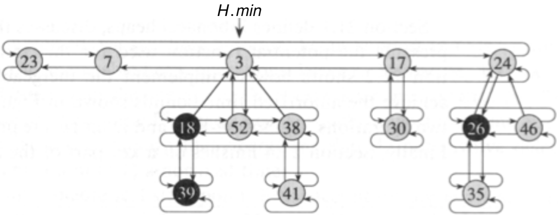
- Each node x contains
  - the fields
    - $x.key$ and possibly more fields for satellite data associated with the key.
    - $x.degree$: number of children in the child list of node x.
    - $x.mark$: a boolean value, which indicates whether node x has lost a child since the last time x was made the child of another node. Newly created nodes are unmarked.
  - the pointers
    - $x.p$: pointer to the parent node; NIL if x is a root node.
    - $x.child$: pointer to any one of its children.
    - The children of a node x are linked together in a circular, doubly-linked list, called the **child list** of x.
    - Each node n in a child list has pointers $n.left$ and $n.right$ which point to its left and right siblings, respectively. If y is a reference to the only child, then $y{-}{>}left = y{-}{>}right = y$.
    - The order in which children appear in the child list is arbitrary.

- The roots of all trees in a Fibonacci heap H are linked together into a circular, doubly linked list, using the *left* and *right* fields of the root nodes. This circular list is called the **root list** of the Fibonacci heap.
- The order of the trees in the root list is arbitrary.
- A Fibonacci heap H has 2 fields:
  - $H.min$: a pointer to the root of the tree in H with minimum key; this node is called the **minimum node** of the Fibonacci heap.
  - $H.n$: the number of nodes currently in H.



A Fibonacci heap made of 5 heap-ordered trees. The dashed line indicates the root list. The 3 marked nodes are blackened.

A more complete representation of the previous heap, showing the pointers *p* (up arrows), *child* (down arrows), and *left* and *right* (sideways arrows), is illustrated below.



For a Fibonacci heap H, we define:

- $t(H)$: the number of trees in the root list of H
- $m(H)$: the number of marked nodes in H.
- $\Phi(H) := t(H) + 2 \cdot m(H)$ is called the potential of H.
- The potential of a set of Fibonacci heaps is the sum of the potentials of its constituent Fibonacci heaps.

Example

The Fibonacci heap illustrated before has $t(H) = 5$ and $m(H) = 3$. Thus, the potential of H is

$$\Phi(H) = 5 + 2 \cdot 3 = 5 + 6 = 11.$$

- Φ will be used for amortized analysis with the potential method (See Lecture 7.)
- 1 unit of potential can pay for a constant amount of work, where the constant is sufficiently large to cover the cost of any specific constant-time piece of work that we might encounter.
- The Fibonacci heap is initially empty $\Rightarrow$ initial potential is 0.
- By definition, $\Phi(H) \geq 0$ always holds $\Rightarrow$ for a sequence of heap operations, the total amortized cost is an upper bound of the total actual cost.
- $D(n)$ is an upper bound on the maximum degree of any node in an n-node Fibonacci heap.

Mergeable heap operations

Assumption. Only MAKEBINOMIALHEAP, INSERT, MINUMUM, EXTRACTMIN and BINOMIALHEAPUNION are supported. $\Rightarrow$ Fibonacci heap = collection of unordered binomial trees, that is, elements of the set $\{U_n \mid n \in \mathbb{N}\}$ defined recursively as follows:

- U_0 has a single node.
- U_k is obtained from two trees U_{k-1}, for which the root of one is made into *any* child of the root of the other.

Properties of unordered binomial trees

For the unordered binomial tree U_k,

1. there are 2^k nodes,
2. the height 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. The children of the root are roots of subtrees $U_0, U_1, \ldots, U_{k-1}$ in some order.

Properties

If an n-node Fibonacci heap is made of unordered binomial trees, then $D(n) = \log_2 n$.

PROOF. Obvious. $\square$

Main idea. To make mergeable-heap operations performant, delay their work as long as possible.

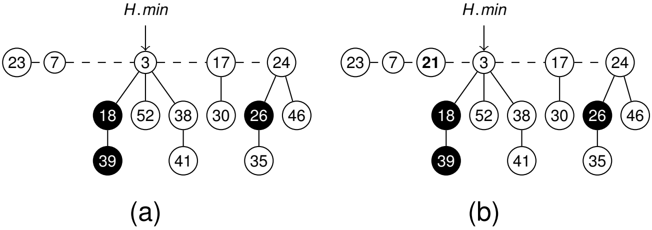
MakeFibHeap allocates and returns the Fibonacci heap object H with $H.n = 0$ and $H.min = \text{Nil}$.

- $\Phi(H) = 0$
- For MakeBinHeap: amortized cost = actual cost = $O(1)$.

Inserting a node
FibHeapInsert

```
FibHeapInsert(H, x)
x->degree := 0
x->p := Nil
x->child := Nil
x->left := x
x->right := x
x->mark := false
concatenate the root list containing x with the root list H
if H.min = Nil or x->key < (H.min)->key
    H.min = x
H.n := H.n + 1
```

Note. Unlike BinomialHeapInsert, FibHeapInsert does not attempt to merge trees within the Fibonacci heap.



(a) A Fibonacci heap H. (b) Fibonacci heap H' produced after inserting the node with key 21 in H. The node becomes its own heap-ordered tree and is then added to the root list, becoming the left sibling of the root.

$$\left.\begin{array}{r} t(H') = t(H) + 1 \\ m(H') = m(H) \end{array}\right\} \Rightarrow \begin{array}{l} \text{the increase in potential is} \\ ((t(H) + 1 + 2\,m(H)) - (t(H) + 2\,m(H)) = 1. \end{array}$$

Actual cost of FIBHEAPINSERT is $O(1) \Rightarrow$
Amortized cost of FIBHEAPINSERT is $O(1) + 1 = O(1)$.

H.min points to the minimum node of $H \Rightarrow$ finding it takes $O(1)$ actual time.

- $\Phi(H)$ does not change $\Rightarrow$ the amortized cost of FIBHEAPMINIMUM is $O(1)$.

Merging (or uniting) Fibonacci heaps

FibHeapBinomialHeapUnion

FibHeapBinomialHeapUnion(H_1, H_2)

$H :=$ MakeFibHeap()
$H.min := H_1.min$
concatenate the root list of H_2 with the root list of H
if $(H_1.min =$ Nil$)$ or $(H_2.min \neq$ Nil and $H_2.min < H_1.min)$
    $H.min := H_2.min$
$H.n := H_1.n + H_2.n$
free the objects H_1 and H_2
return H

Note. No consolidation of trees occurs in the Fibonacci heap.

- The change in potential is

$$\begin{aligned}\Phi(H) - (\Phi(H_1) + \Phi(H_2)) &= (t(H) + 2\,m(H)) - ((t(H_1) + 2\,m(H_1)) \\ &\quad + ((t(H_2) + 2\,m(H_2)) \\ &= 0,\end{aligned}$$

because $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$

$\Rightarrow$ amortized cost of FIBHEAPBINOMIALHEAPUNION = actual cost = $O(1)$.

Extracting the minimum node
FIBHEAPEXTRACTMIN

Preliminary remark. This is the operation where all work delayed by other operations is done.

- delayed work = consolidation (or merging) of the trees in the root list.

```
FIBHEAPEXTRACTMIN(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
```

Extracting the minimum node
Description of the pseudocode

- FIBHEAPEXTRACTMIN makes a root out of each of the minimum node-s children and removes the minimum node from the root list.
- Next, it consolidates the root list by linking roots of equal degree until at most one root remains of each degree.
- Consolidate(H) consolidates the root list of H by executing repeatedly the following steps until every root in the root list has a distinct degree value:
  1. Find two roots x and y from the root list with the same degree, and with $x{-}{>}key \leq y{-}{>}key$.
  2. Link y to x: remove y from the root list, and make y a child of x. This operation is performed by FIBHEAPLINK.

  Consolidate(H) uses an auxiliary array $A[0..D(H.n)]$; if $A[i] = y$ then y is currently a root with $y{-}{>}degree = i$.

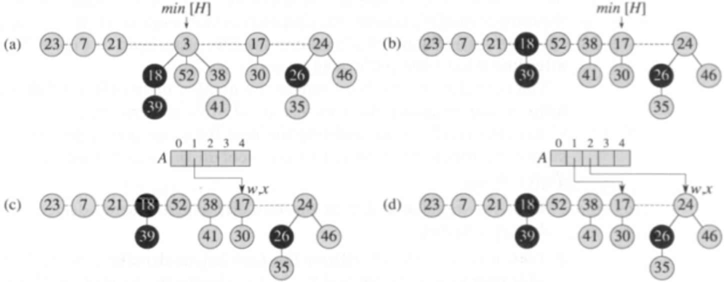
Extracting the minimum node

The CONSOLIDATE procedure

```
CONSOLIDATE(H)
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]
        if x->key > y->key
            exchange x ↔ y
        FIBHEAPLINK(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
        add A[i] to the root list of H
        if H.min = NIL or A[i]->key < H.min->key
            H.min := A[i]
```

FIBHEAPLINK(H, y, x)

1. remove y form the root list of H
2. make y a child of x, incrementing x−>*degree*
3. y−>*mark* $:=$ *false*

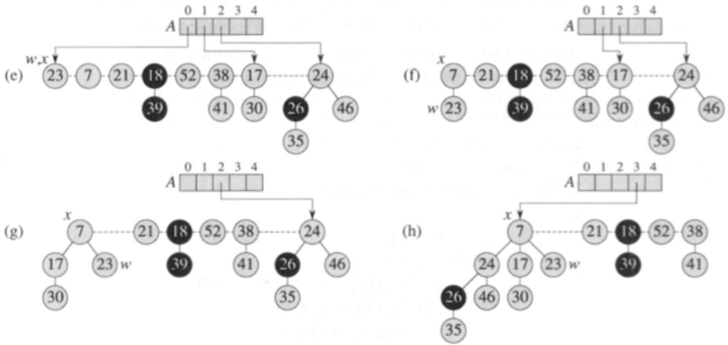


(b) The situation after the minimum node z is removed from the root list and its children are added to the root list.

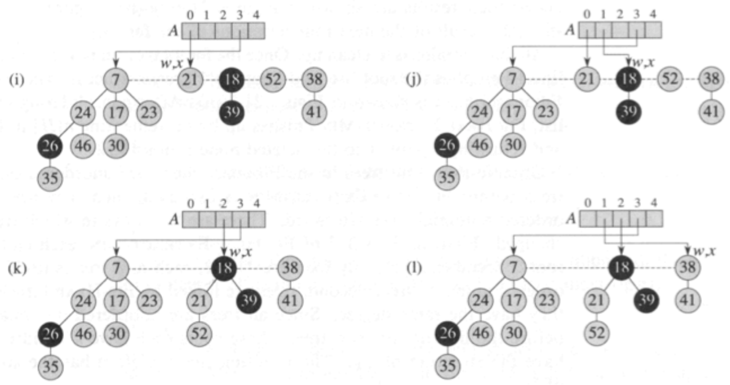
(c)-(d) The array A and the trees after each of the first 2 iterations of the **for** loop of lines 3-13 of CONSOLIDATE.

Extracting the minimum node

Illustrated example continued

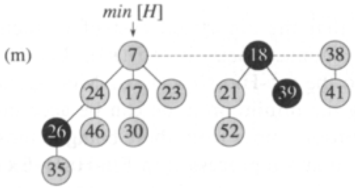


(e) The array *A* and the trees after each of the 3rd iteration of the **for** loop of lines 3-13 of CONSOLIDATE. (f)-(h) The next iteration of the **for** loop, with the values of *w* and *x* shown at the end of each iteration of the **while** loop of lines 6-12. Part (f) shows the situation after the first time through the **while** loop. The node with key 23 has been linked to the node with key 7, which is now pointed to by *x*. In part (g), the node with key 17 has been linked to the node with key 7, which is still pointed to by *x*.



(i)-(l) The situation after each of the next four iterations of the **while** loop.

Extracting the minimum node
Illustrated example continued



(m) Fibonacci heap after reconstruction of the root list from the array A and determination of the new $H.min$ pointer.

Let H be the Fibonacci heap before the call of FIBHEAPEXTRACTMIN(H).

- FIBHEAPEXTRACTMIN(H) contributes $O(D(n))$ from the extraction of at most $D(n)$ children of the minimum node that are processed in FIBHEAPEXTRACTMIN and from the work done in lines 1-2 and 14-19 of CONSOLIDATE.
- It remains to analyze the contribution of the **for** loop of lines 3-13.
  - When CONSOLIDATE is called, the root list has size $\leq D(n) + t(H) - 1$.
  - Every **while** loop of lines 6-12 links one root to another $\Rightarrow$ the amount of work performed in the **while** loop is $\leq D(n) + t(H)$.

  $\Rightarrow$ total actual work is $O(D(n) + t(H))$.

- The potential before extracting the minimum node is $t(H) + 2\,m(H)$.
- At most $D(n) + 1$ roots remain and no nodes become marked during the operation $\Rightarrow$ the potential after extracting the minimum node is $\leq (D(n) + 1) + 2\,m(H)$.

$\Rightarrow$ the amortized cost is at most

$$\begin{aligned} O(D(n) + t(H)) + ((D(n) + 1) + 2\,m(H)) - (t(H) + 2\,m(H)) \\ &= O(D(n)) + O(t(H)) - t(H) \\ &= O(D(n)), \end{aligned}$$

because the units of potential can be scaled to dominate the constant hidden in $O(t(H))$.

Decreasing a key and deleting a node

- The operations presented so far for Fibonacci heaps did preserve the property that all trees in the Fibonacci heap are unordered binomial trees U_n.
- The operations that will be presented do not preserve this property.

Decreasing a key
FIBHEAPDECREASEKEY

FIBHEAPDECREASEKEY(H, x, k) decreases the key of a node x in a binomial heap to a new value k. It signals an error if $k > x{-}{>}key$.

FIBHEAPDECREASEKEY(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)
    CASCADINGCUT(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
$x{-}{>}mark := false$

Decreasing a key (2)

```
CascadingCut(H, y)
z := y->p
if z ≠ NIL
   if y->mark = false
      y->mark := true
   else Cut(H, y, z)
         CascadingCut(H, z)
```

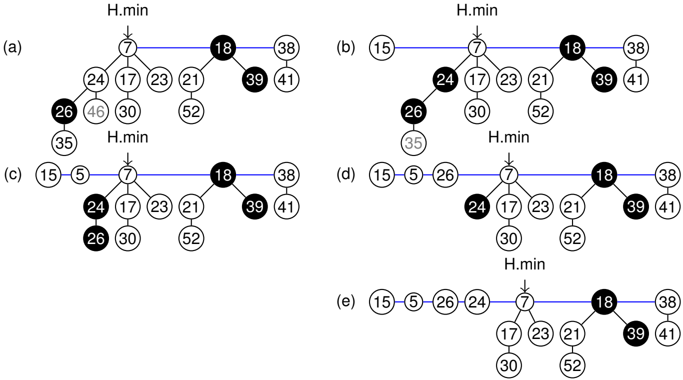
Lines 1-3 of FIBHEAPDECREASEKEY ensure that new key should be < current key.
If x is a root (that is, x->p = NIL) or else x->$key \geq x$->p->key, then no structural changes need occur because the heap order is preserved by the key replacement.
If the heap order is violated $\Rightarrow$ x is cut out from its siblings in line 6, and moved into the root list of the heap.
The purpose of the *mark* fields is to ensure short time bounds for the heap operations. To understand how it is used, let's consider that x is a pointer to a node that went through the following situations:

1. At some point, x was a root.
2. Later on, x was linked to another node.
3. Later on, two children of x were removed by cuts.

When the 2nd child is cut, x is cut from its parent and becomes a new root. We have x->$mark = true$ if steps 1 and 2 happened and one child of x has been cut. Thus, CUT sets x->$mark = false$ in line 4 because it performs step 1.

The CASCADINGCUT calls in line 7 of FIBHEAPDECREASEKEY take care of the situation when x might be the second child cut from its parent y since the time that y was linked to another node. It recurses up the tree until either a root or an unmarked node is found. After all cascading cuts were done, lines 8-9 of FIBHEAPDECREASEKEY update the value of $H.min$ accordingly.

Decreasing a key: Example



(a) The initial Fibonacci heap. (b) key 46 was decreased to 15. (3) The node becomes a root, and its parent (with key 24) gets marked. (c)–(e) the node with key 35 has its key decreased to 5. Its parent (key 26) is marked, and a cascading cut occurs. The node with key 26 is cut from its parent and becomes an unmarked root in (d). Another cascading occurs since node with key 24 is marked too. This node gets cut from its parent and made an unmarked root in (e). At this stage, cascading stops.

- Actual cost = $O(1)$ + time for cascading cuts $= O(c)$ where $c =$ max. number of recursive calls of CASCADINGCUT from a call of FIBHEAPDECREASEKEY
- Change in potential during a FIBHEAPDECREASEKEY operation is at most (see References)

$$((t(H) + c) + 2(m(H) - c + 2)) - (t(H) + 2m(H)) = 4 - c$$

$\Rightarrow$ the amortized cost of FIBHEAPDECREASEKEY is at most $O(c) + 4 - c = O(1)$

FIBHEAPDELETE(H, x)

```
FIBHEAPDECREASEKEY(H, x, −∞)
FIBHEAPEXTRACTMIN(H)
```

- The amortized execution time of FIBHEAPDELETE(H, x) is the sum of the amortized time $O(1)$ to perform FIBHEAPDECREASEKEY$(H, x, -\infty)$, with the amortized time $O(D(n))$ to perform FIBHEAPEXTRACTMIN(H).

Bounding the maximum degree

- The last thing to do is to compute an upper bound $D(n)$ for the maximal degree of the unordered trees in the Fibonacci heap.
- We noticed that if all trees in the Fibonacci heap are unordered binomial trees, then $D(n) = \lfloor \log_2 n \rfloor$. But the (cascading) cuts may cause the occurrence of trees that are not unordered binomial.
- Therefore, a slightly weaker result still holds: $D(n) \leq \lfloor \log_\phi(n) \rfloor$ where $\phi = (1 + \sqrt{5})/2$.
- For a proof of this result, see Chapter 21 of the book *Introduction to Algorithms* by Cormen *et al*.