Fibonacci Heaps

Manoj Kumar
DTU, Delhi

Fibonacci Heap

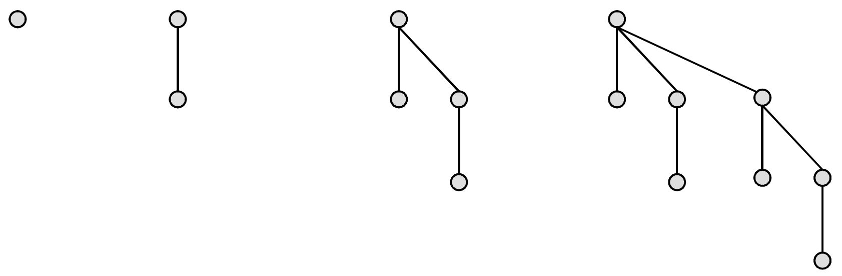
- A Fibonacci heap is a collection of trees satisfying the minimum-heap property, that is, the key of a child is always greater than or equal to the key of the parent.
- This implies that the minimum key is always at the root of one of the trees.
- Compared with binomial heaps, the structure of a Fibonacci heap is more flexible.

Fibonacci Heap

- 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.

Fibonacci Heap

- **Similar to binomial heaps, but less rigid structured.**
- Binomial heap: eagerly consolidate trees after each *insert*.



- Fibonacci heap: lazily defer consolidation until next *delete-min*.
- **Decrease-key and union run in O(1) time.**
- **"Lazy" unions.**

Fibonacci Heap

- Every node has degree at most D(n)=$O(\log n)$ and
- the size of a subtree rooted in a node of degree k is at least F_{k+2}, where F_k is the k^{th} Fibonacci number.
- This is achieved by the rule that we can cut at most one child of each non-root node.
- When a second child is cut, the node itself needs to be cut from its parent and becomes the root of a new tree.
- The number of trees is decreased in the operation *delete minimum*, where trees are linked together.

Fibonacci Heap

- As a result of a relaxed structure, some operations can take a long time while others are done very quickly.
- In the amortized running time analysis we pretend that very fast operations take a little bit longer than they actually do.
- This additional time is then later subtracted from the actual running time of slow operations.

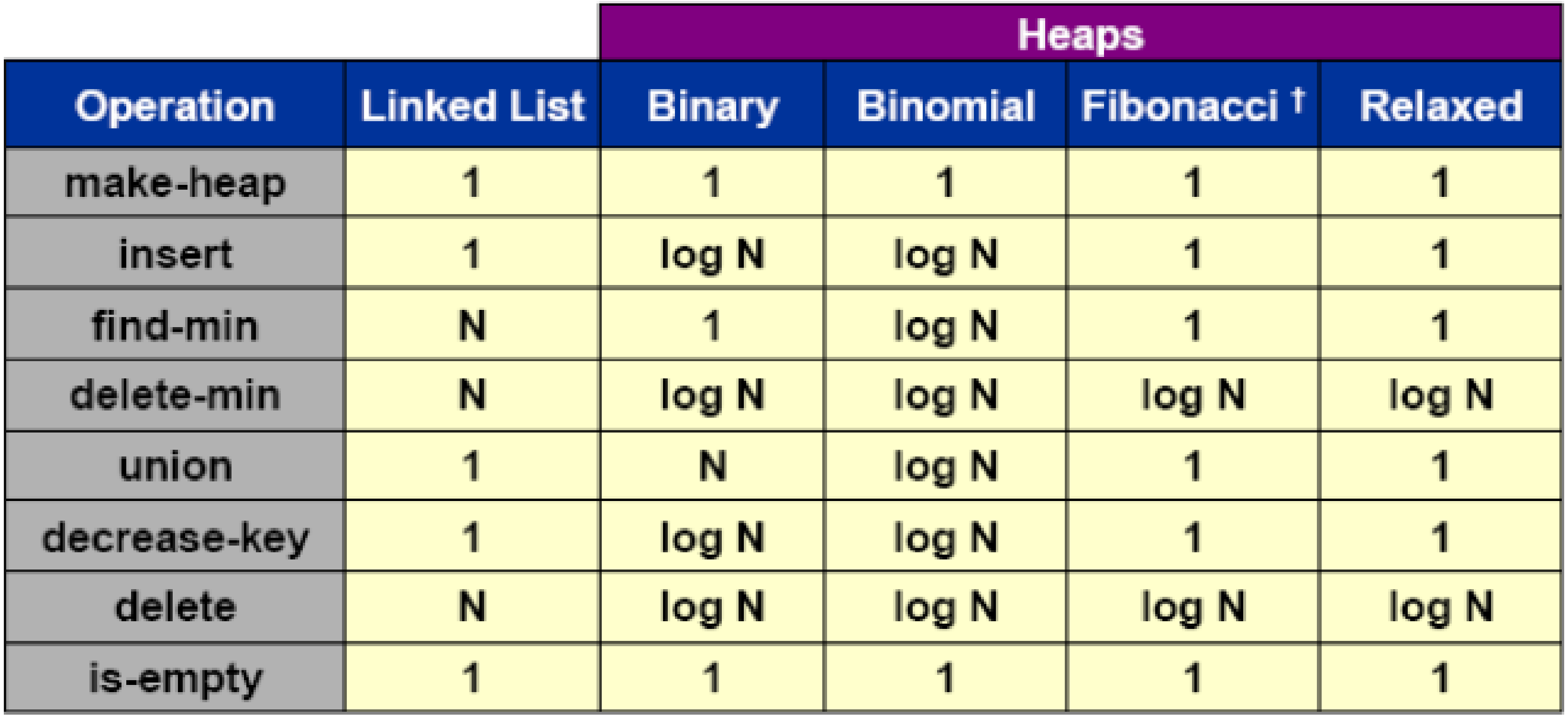
Fibonacci Heap

- The amount of time saved for later use is measured at any given moment by a potential function.
- The potential of a Fibonacci heap is given by

  $$\text{Potential} = t + 2m$$

  where t is the number of trees in the Fibonacci heap, and m is the number of 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.

Comparison



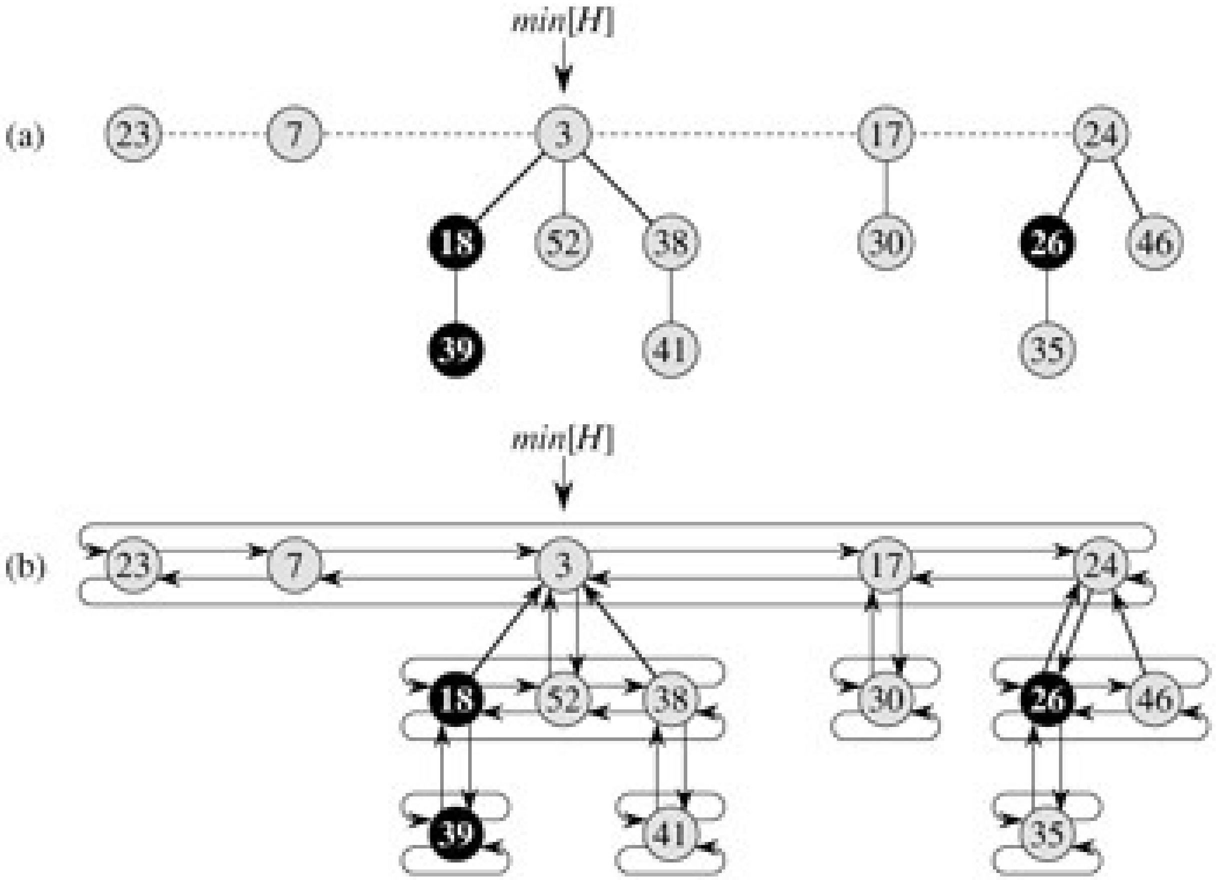
		Heaps			
Operation	Linked List	Binary	Binomial	Fibonacci †	Relaxed
make-heap	1	1	1	1	1
insert	1	log N	log N	1	1
find-min	N	1	log N	1	1
delete-min	N	log N	log N	log N	log N
union	1	N	log N	1	1
decrease-key	1	log N	log N	1	1
delete	N	log N	log N	log N	log N
is-empty	1	1	1	1	1

† **amortized**

Fibonacci heaps: Structure

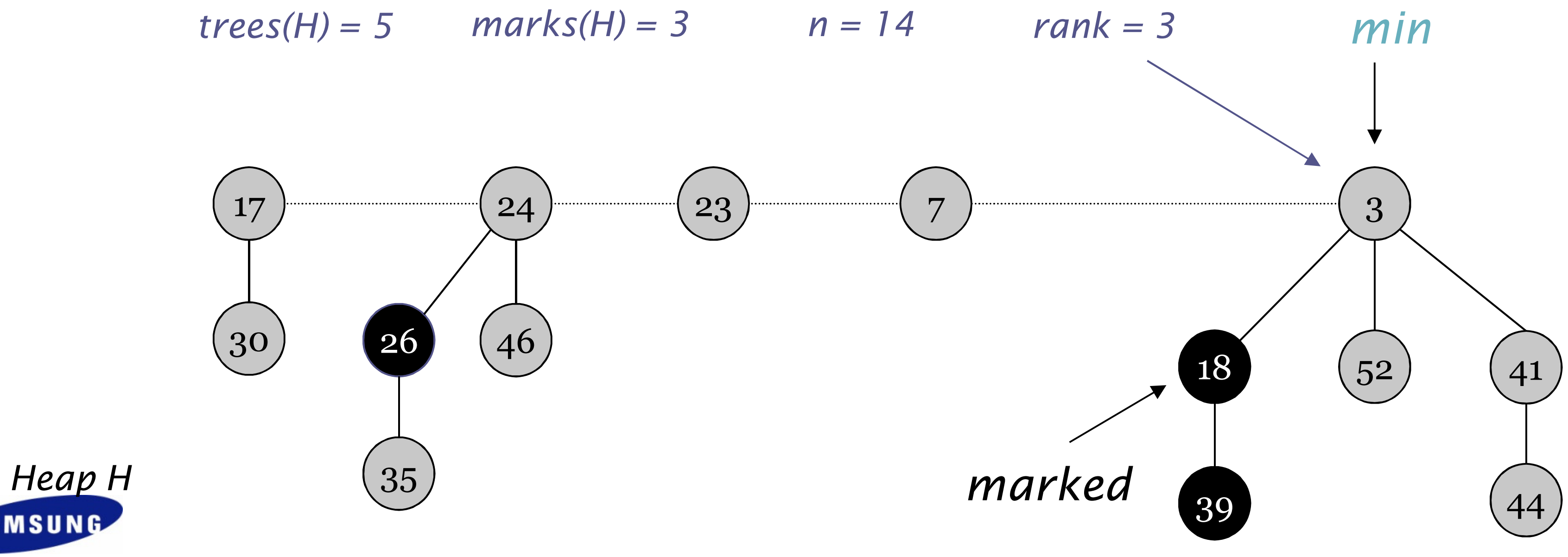
Fibonacci heap.

- Set of heap-ordered trees.
- Maintain pointer to minimum element.
- Set of marked nodes.



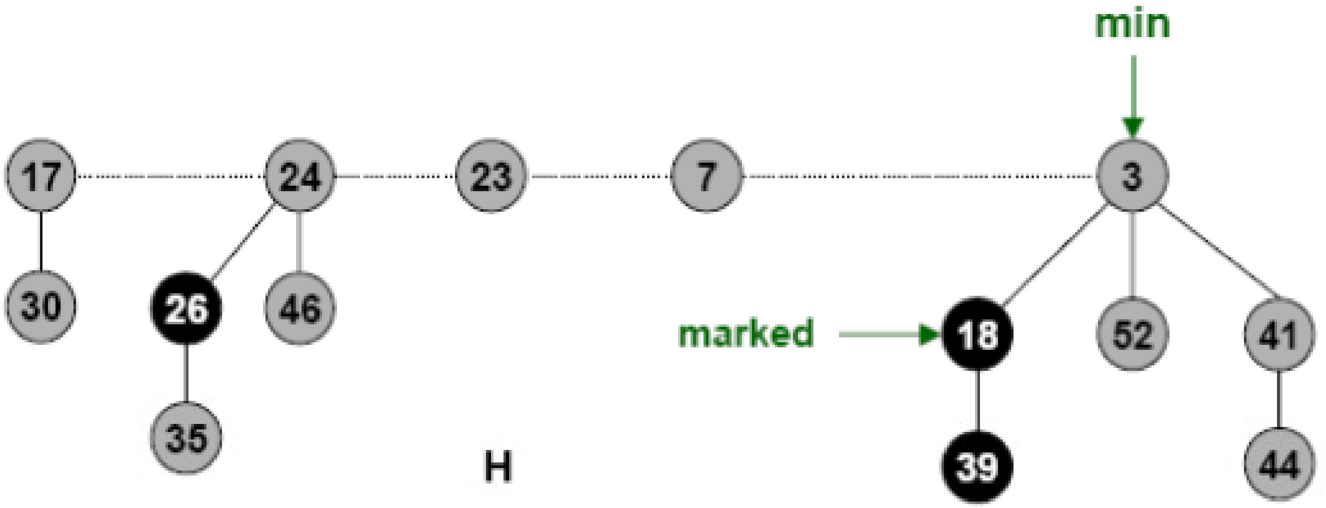
Notations

- *n(H)* = number of nodes in heap.
- *degree(x)* = number of children of node *x*.
- *t(H)* = number of trees in heap *H*.
- *m(H)* = number of marked nodes in heap *H*.



Implementation

- Represent trees using left-child, right sibling pointers and 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



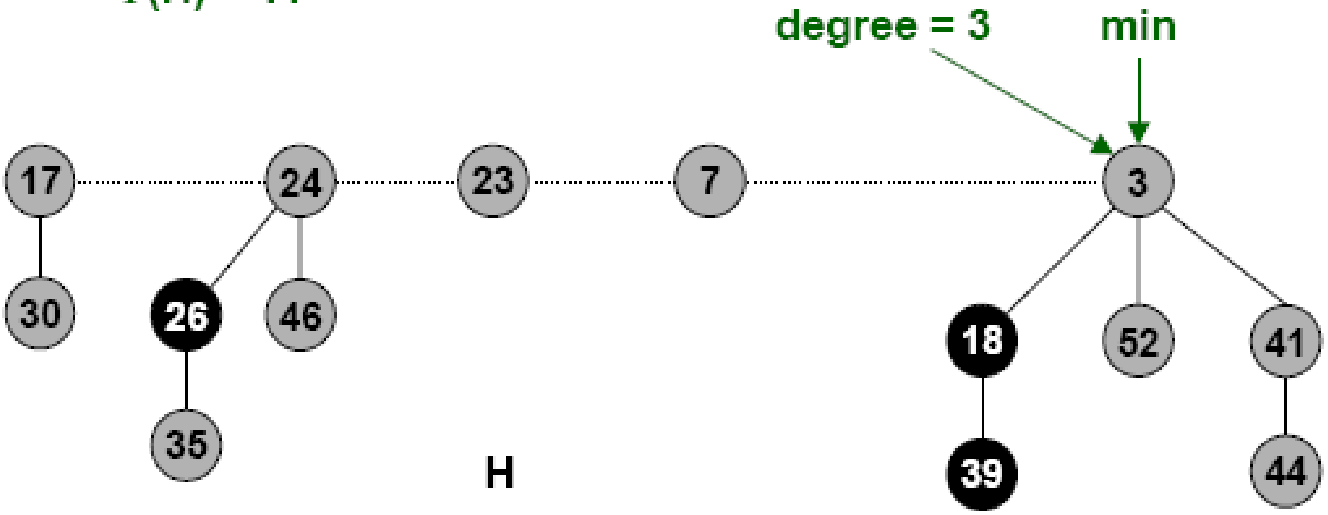
Potential Function

Key quantities.

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

t(H) = 5, m(H) = 3

Φ(H) = 11



Maximum degree

- We assume that there is a known upper bound $D(n)$ on the maximum degree of any node in an n-node Fibonacci heap.
- $D(n) = O(\lg n)$

Operations on Fibonacci Heaps

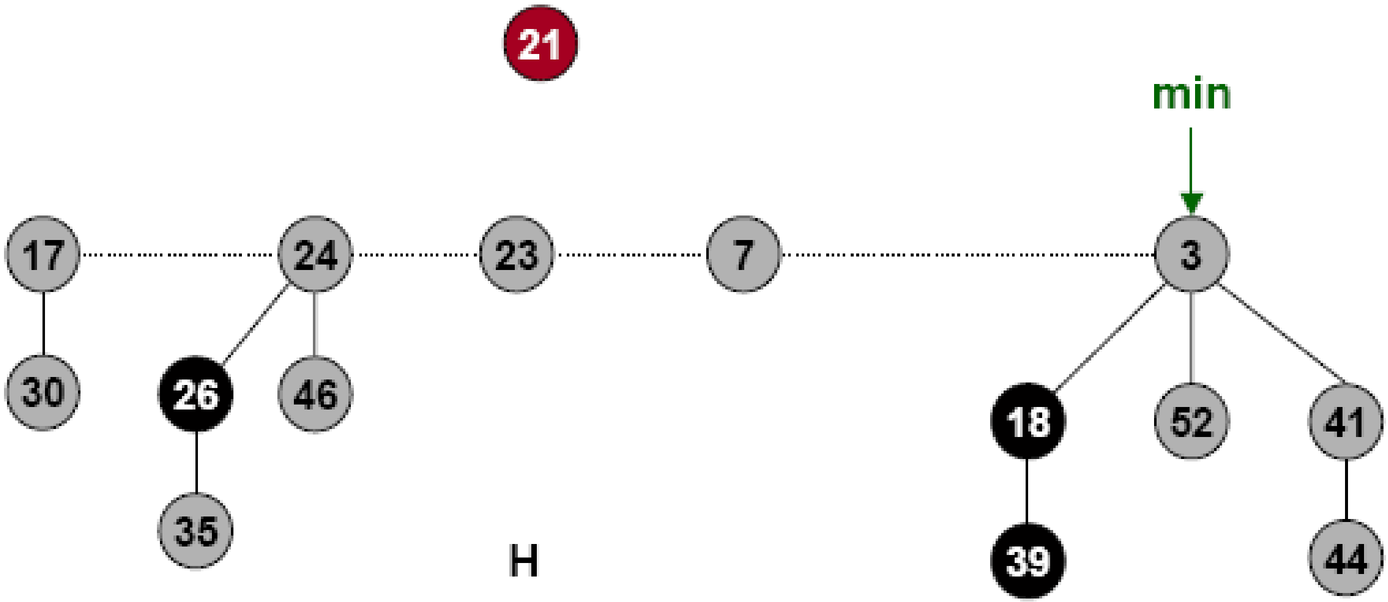
- Creating a new Fibonacci Heap
  - MAKE-FIB-HEAP()
  - Returns the Fabonacci heap object H, where
    - n[H]=0
    - min[H]=NIL
    - t[H] = 0
    - m[H] = 0
    - Φ[H] = 0
  - Amortized cost of MAKE-FIB-HEAP() is O(1).

Inserting a node: FIB-HEAP-INSERT(H , x)

Insert.

- Create a new singleton tree.
- Add to left of min pointer.
- Update min pointer.

Insert 21

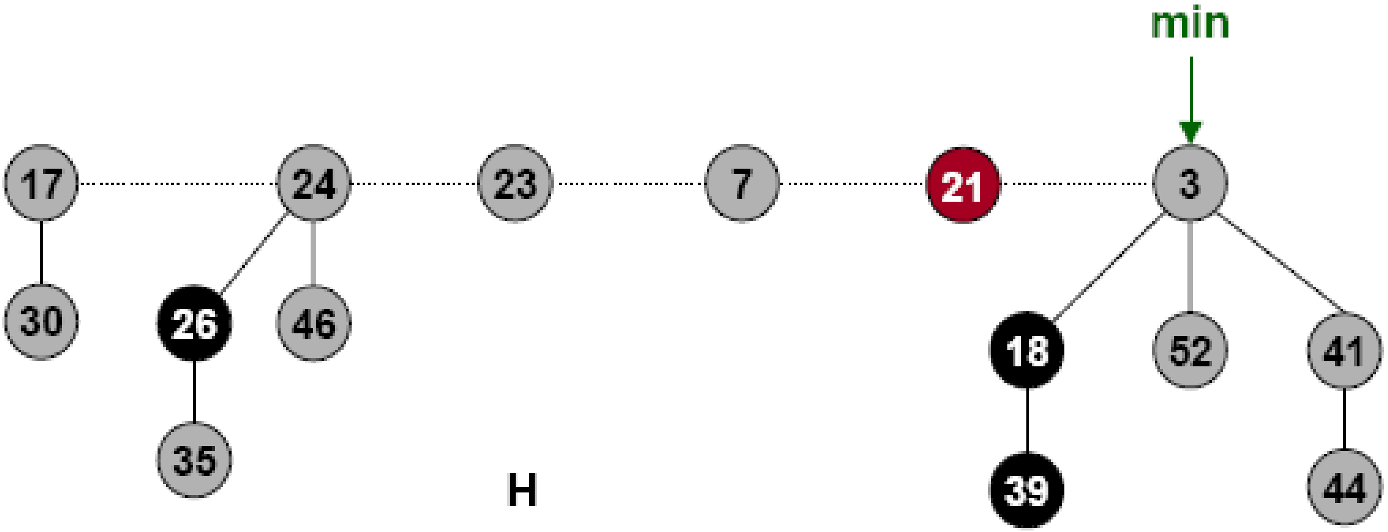


Insert…

Insert.

- Create a new singleton tree.
- Add to left of min pointer.
- Update min pointer.

Insert 21



Insert…

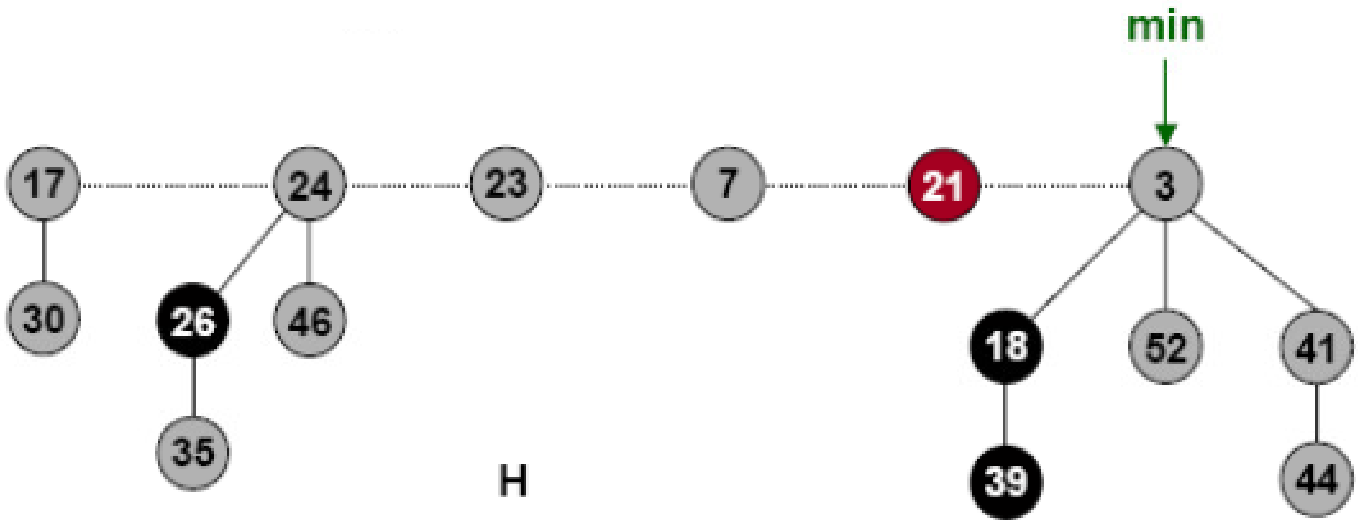
Running time:

Increase in potential function is

((t(H) + 1) + 2 m(H)) - (t(H) + 2 m(H)) = 1

Since actual cost is O(1),

The amortized cost is O(1) + 1 = O(1)



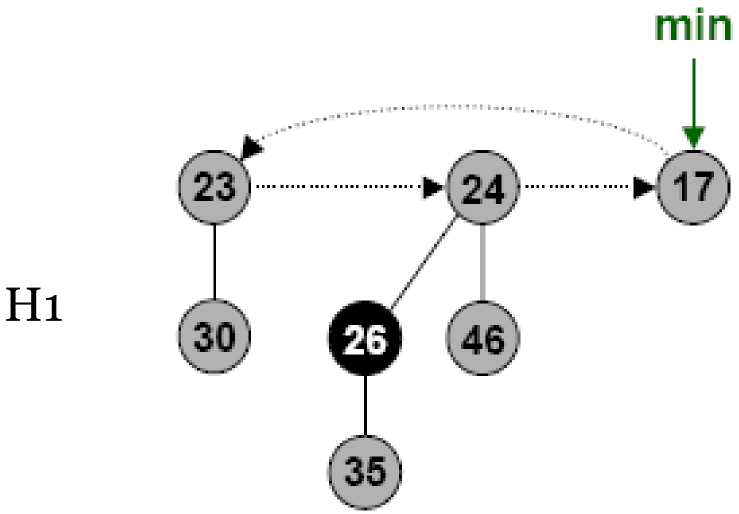
FIB-HEAP-INSERT(*H* , *x*)

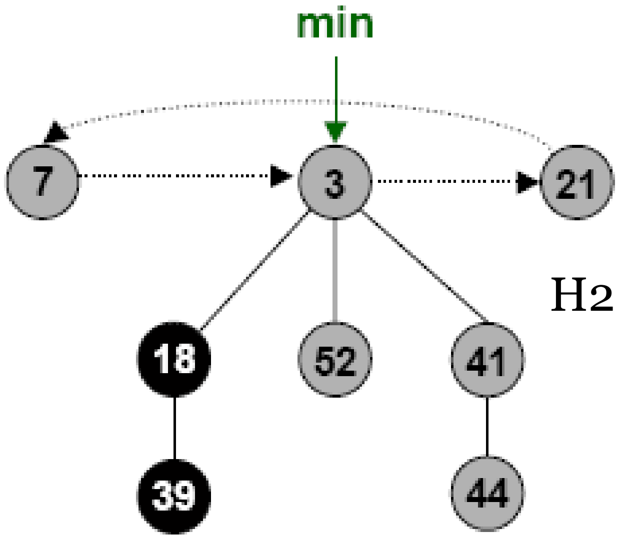
```
FIB-HEAP-INSERT(H,x)
degree[x] ← 0
P[x] ← NIL
child[x] ← NIL
left[x] ← x
right[x] ← x
mark[x] ← FALSE
Concatenate  the root list containing x with root list H
If  min[H] = NIL or key[x] < key[min[H]]
        then min[H] ← x
n[H] ← n[H] + 1
```

Union

Union.

- Concatenate two Fibonacci heaps.
- Root lists are circular, doubly linked lists.





Union…

Running time:

Increase in potential function is

$\Phi(H) - (\Phi(H_1) + \Phi(H_2))$

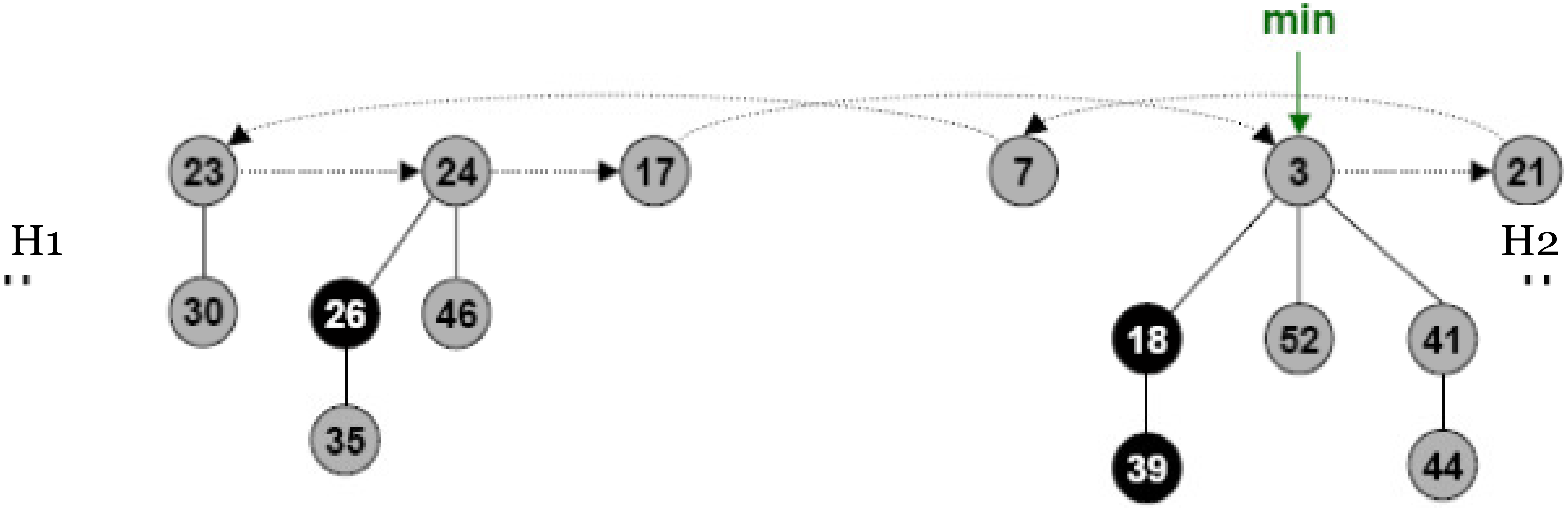
$= (t(H) + 2\,m(H)) - ((t(H_1) + 2\,m(H_1)) + (t(H_2) + 2\,m(H_2)))$

$= 0$

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

Since actual cost is O(1),

The amortized cost is O(1) + 0 = O(1)



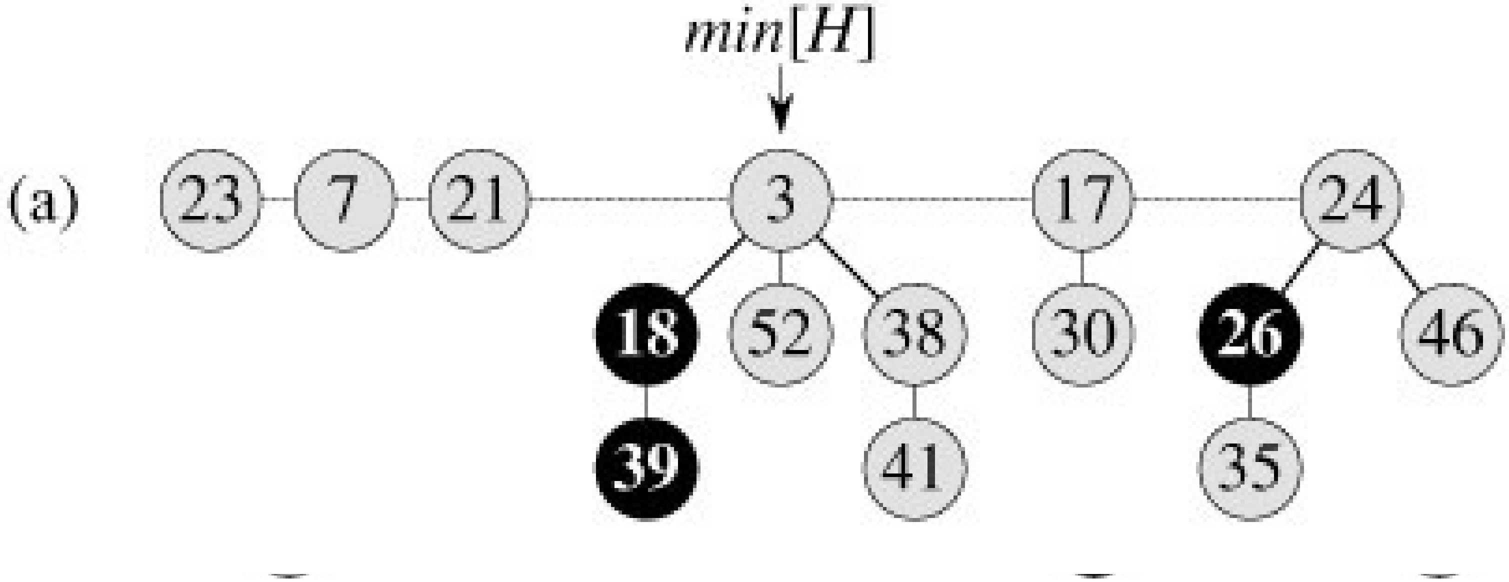
UNION

```
FIB-HEAP-UNION($H_1$,$H_2$)
    $H \leftarrow$ MAKE-FIB-HEAP( )
    $min[H] \leftarrow min[H_1]$
    Concatenate the root lists of $H_1$ with the root list of $H$
    If ($min[H_1] = NIL$) or ($min[H_2] \neq NIL$ and $min[H_2] < min[H_1]$)
        then $min[H] \leftarrow min[H_2]$
    $n[H] \leftarrow n[H_1] + n[H_2]$
    free the objects $H_1$ and $H_2$
    return $H$
```

Delete-Min.

Delete min.

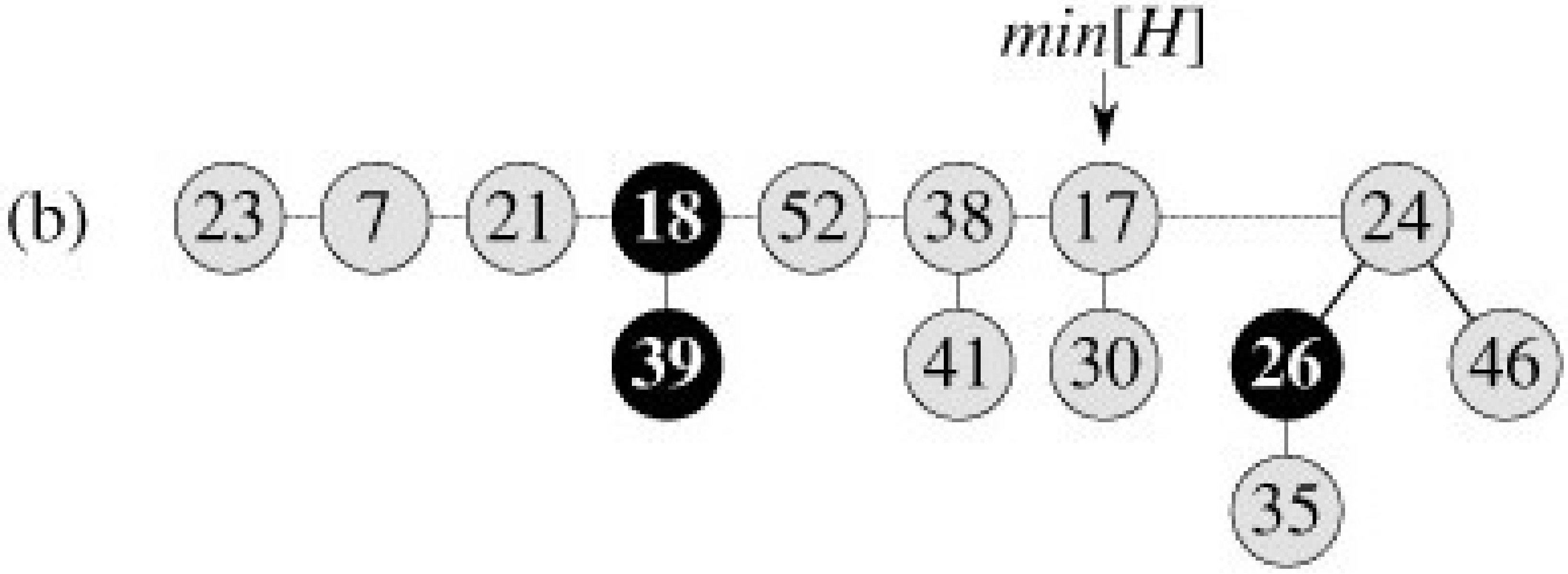
- Delete min and concatenate its children into root list.
- Consolidate trees so that no two roots have same degree.



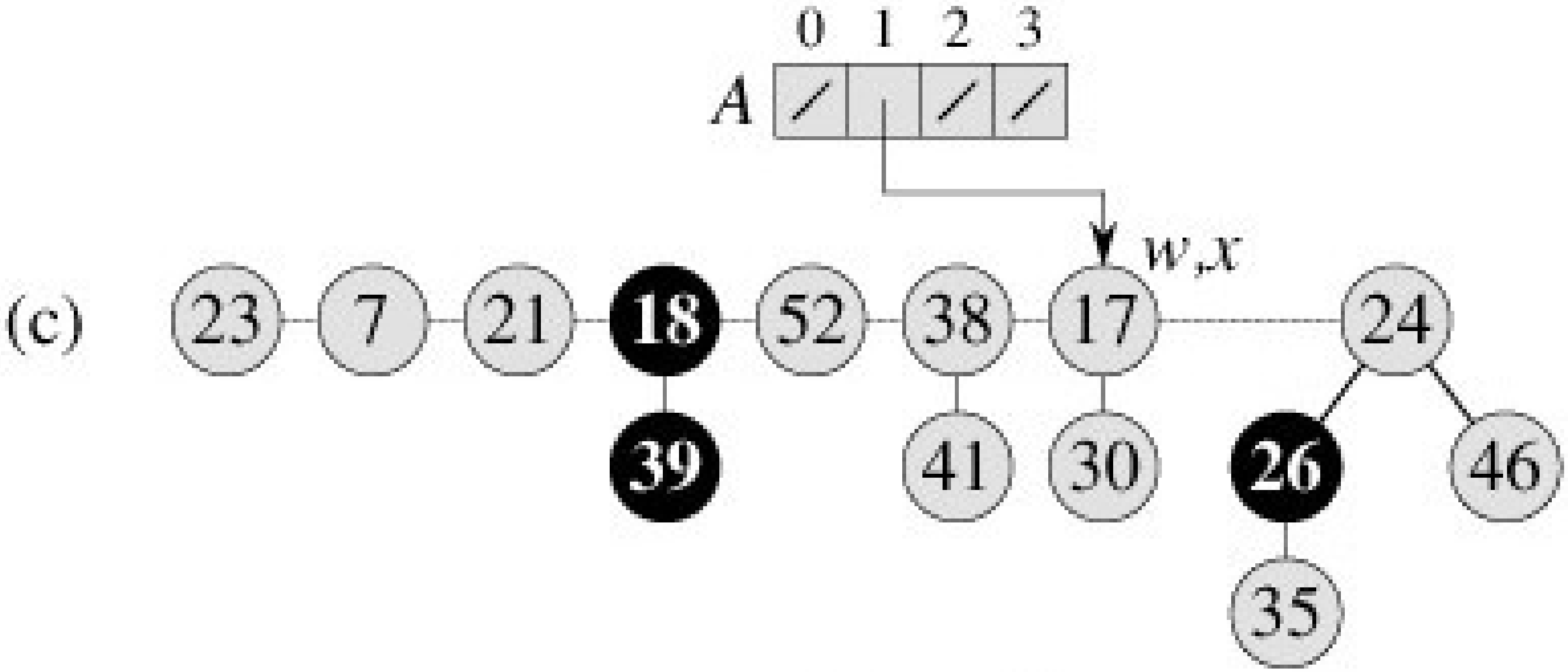
Delete-Min….

Delete min.

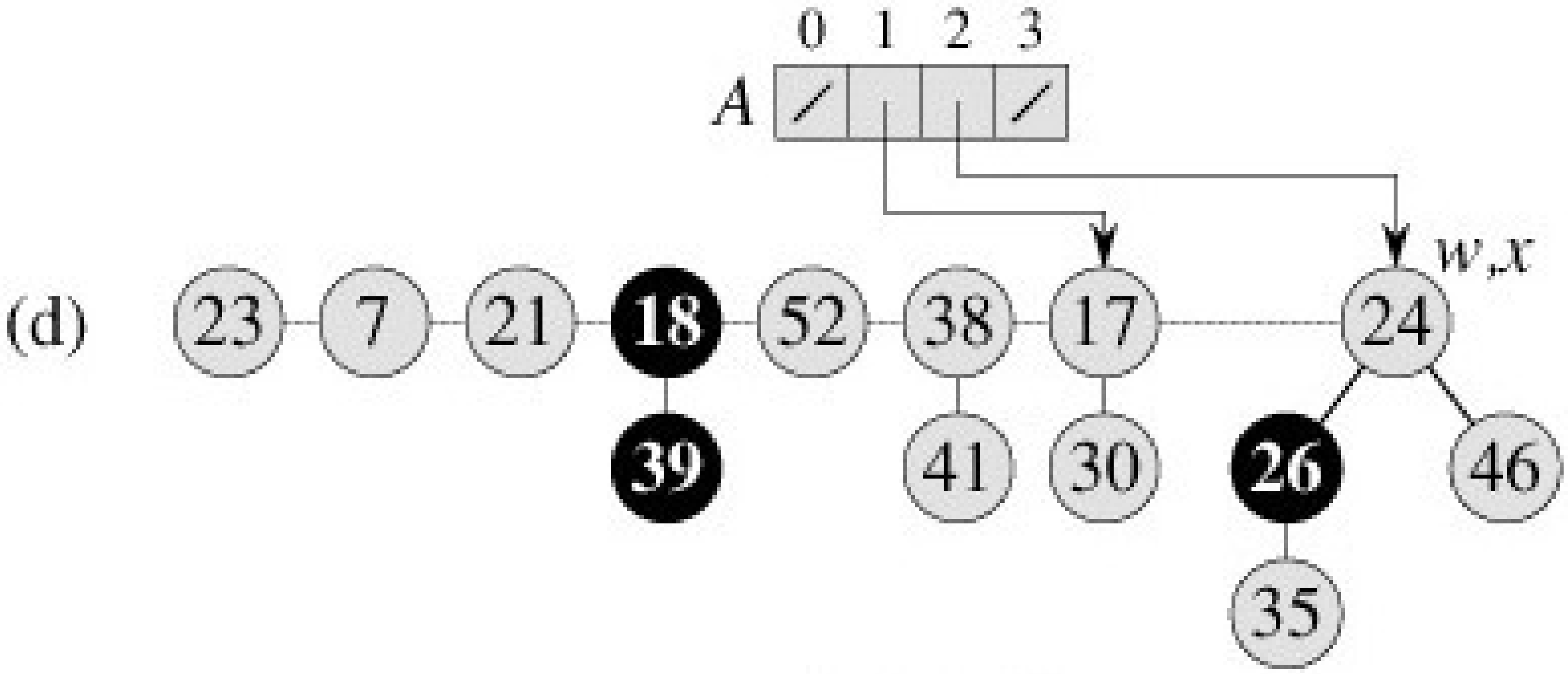
- Delete min and concatenate its children into root list.
- Consolidate trees so that no two roots have same degree.



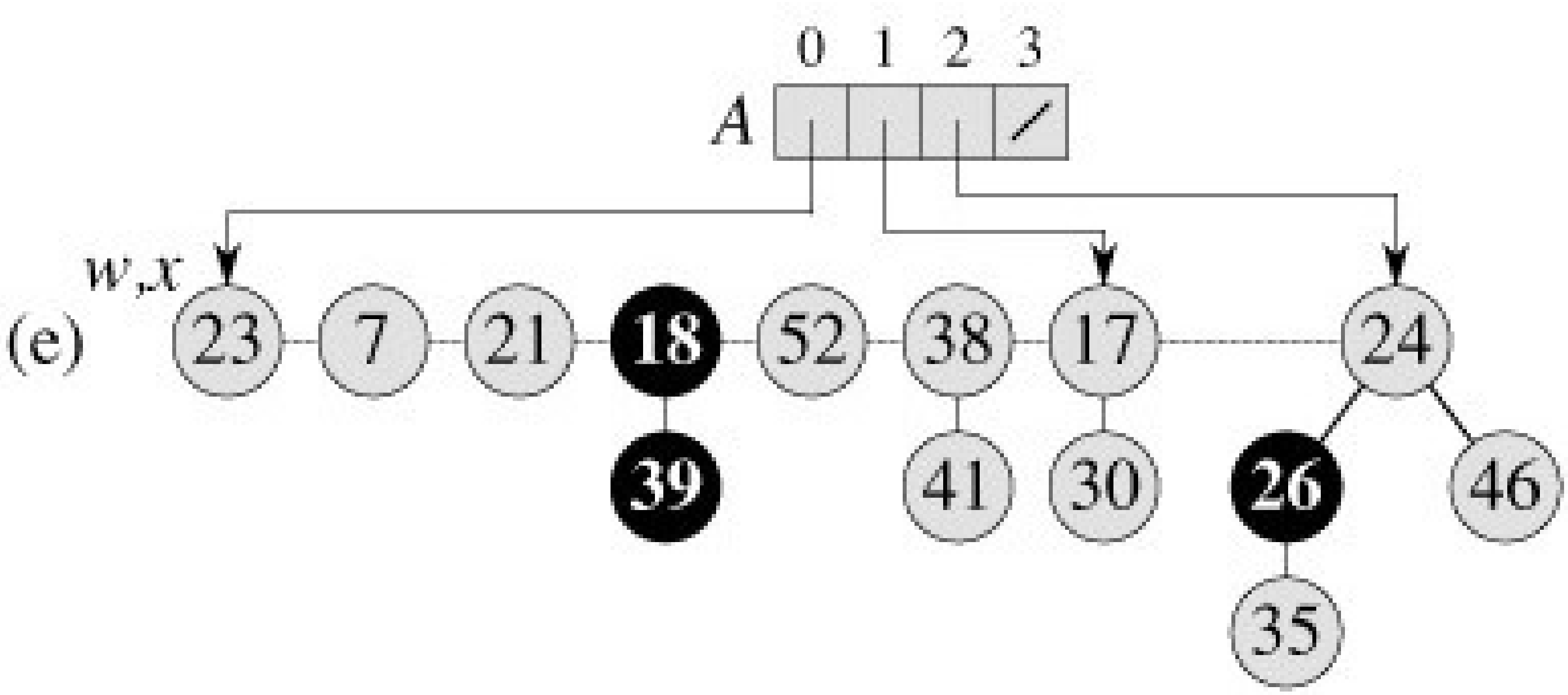
Delete-Min : Consolidate



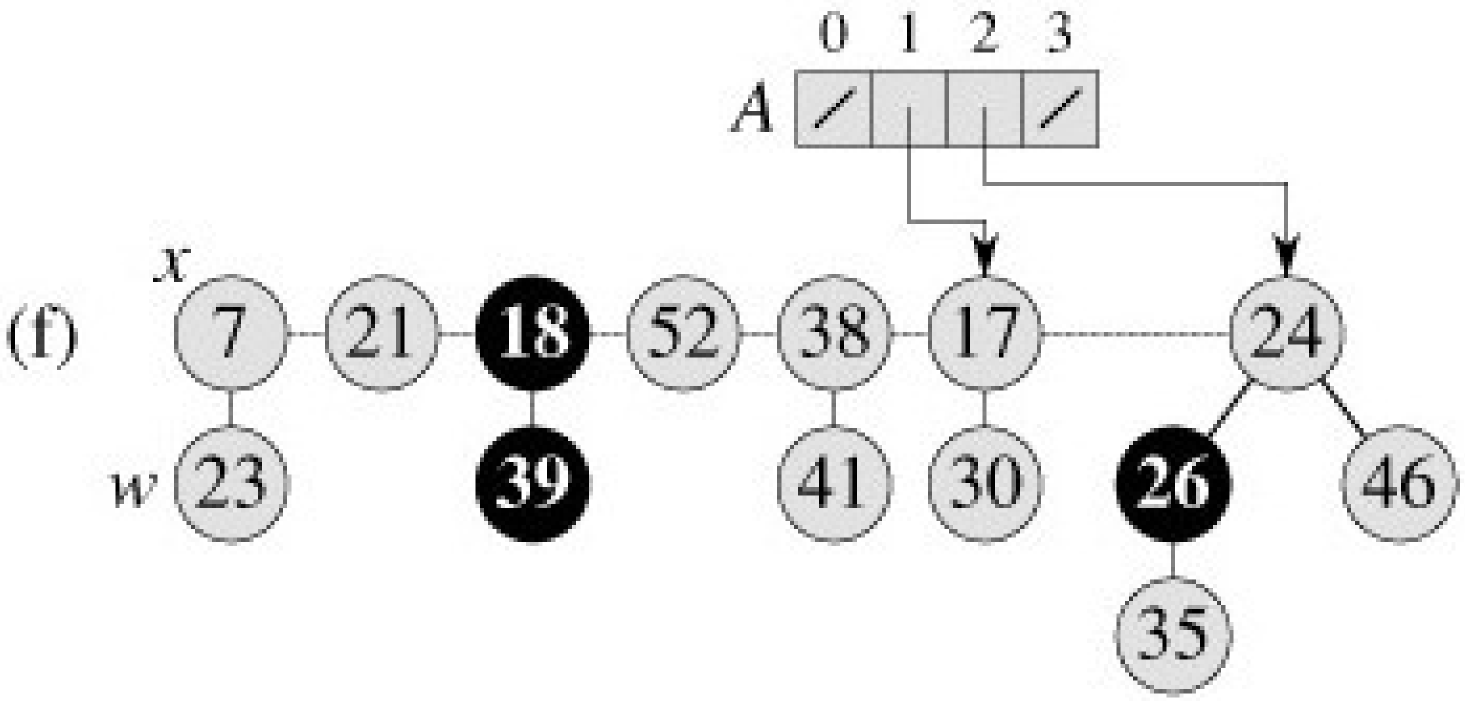
Delete-Min…



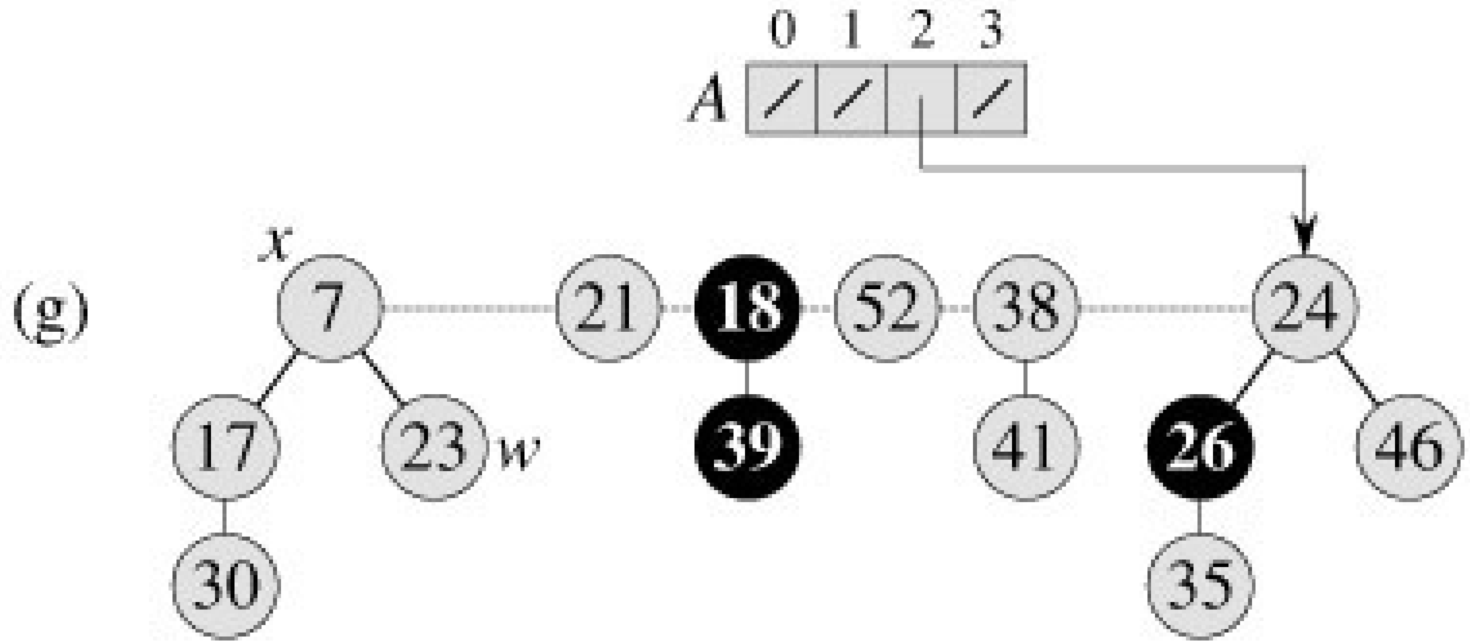
Delete-Min…



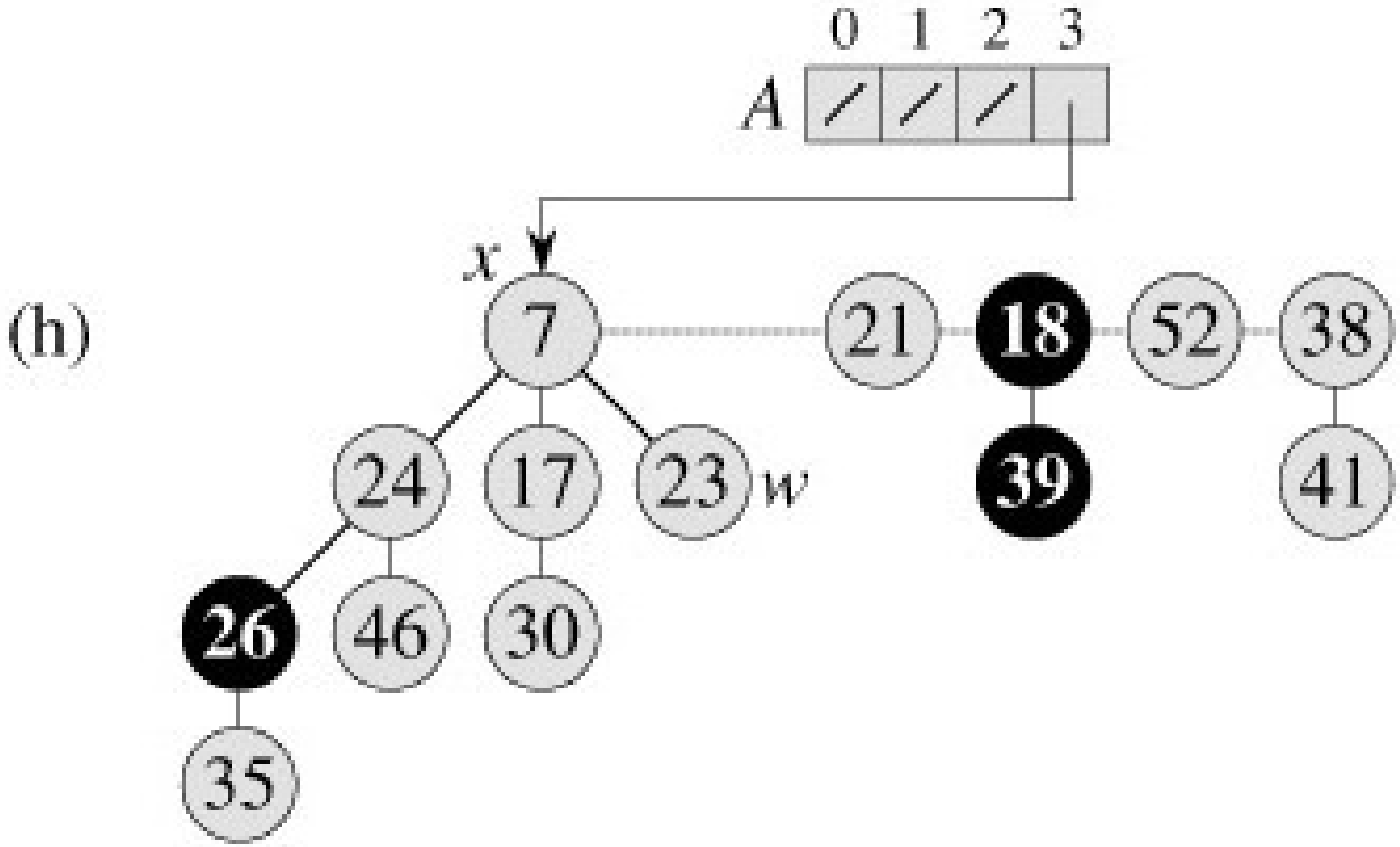
Delete-Min…



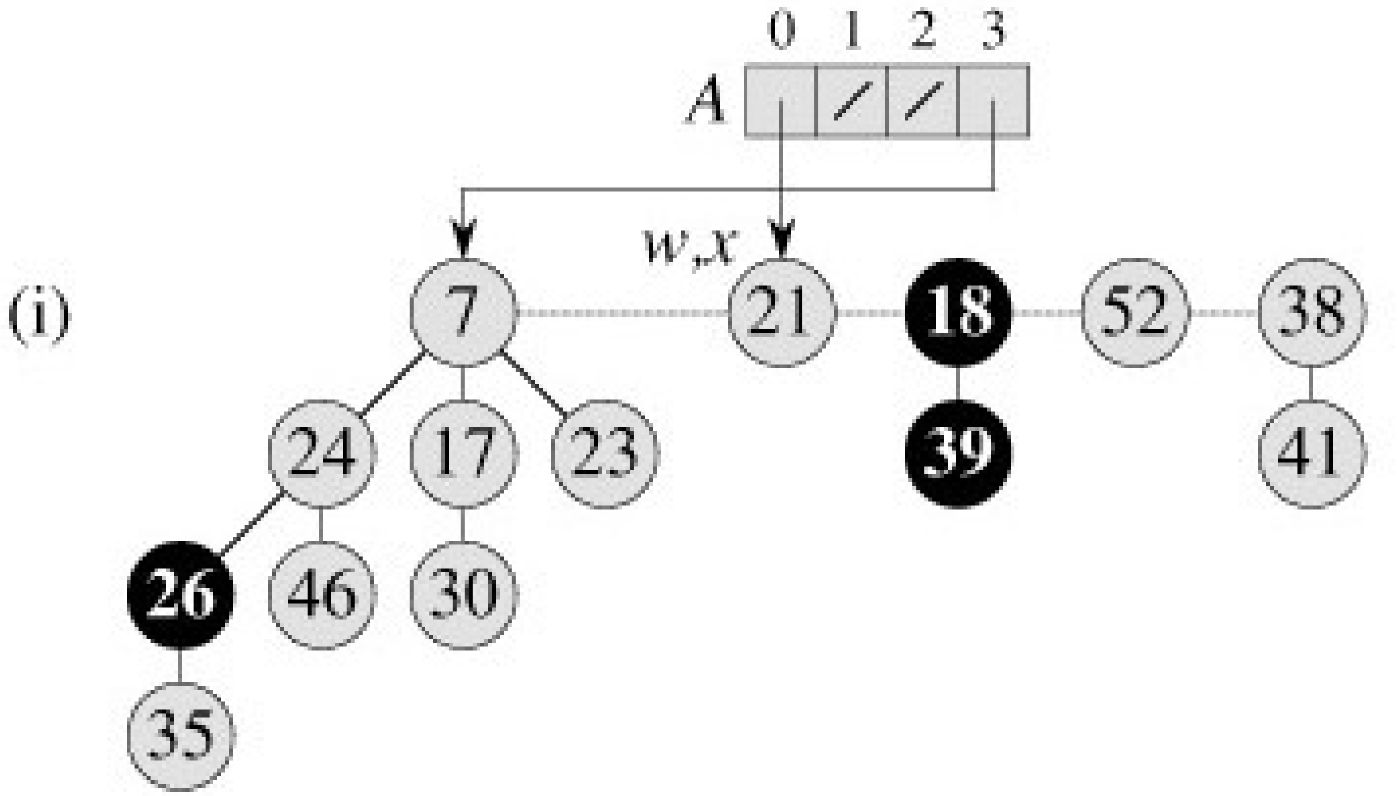
Delete-Min…



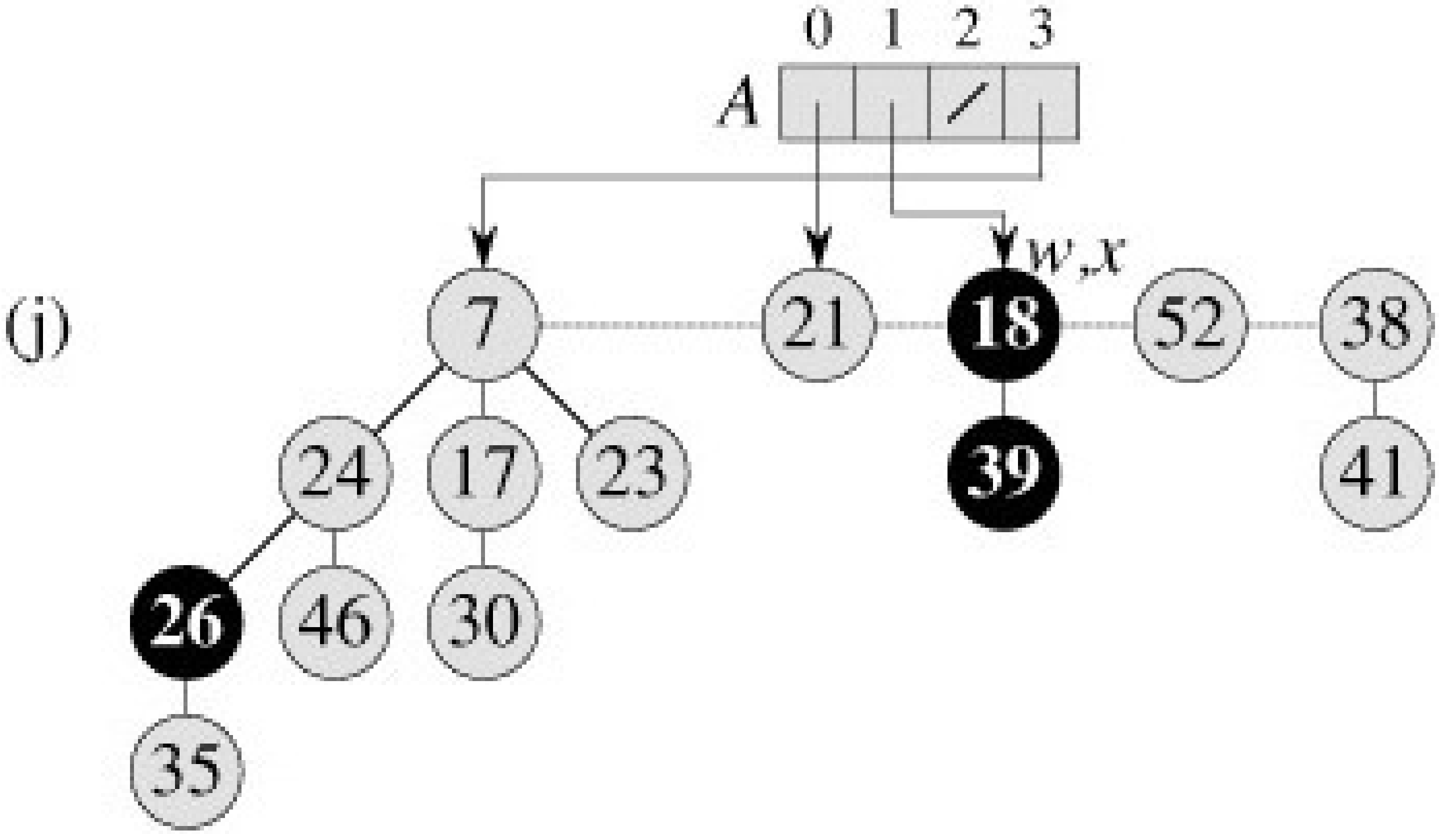
Delete-Min…



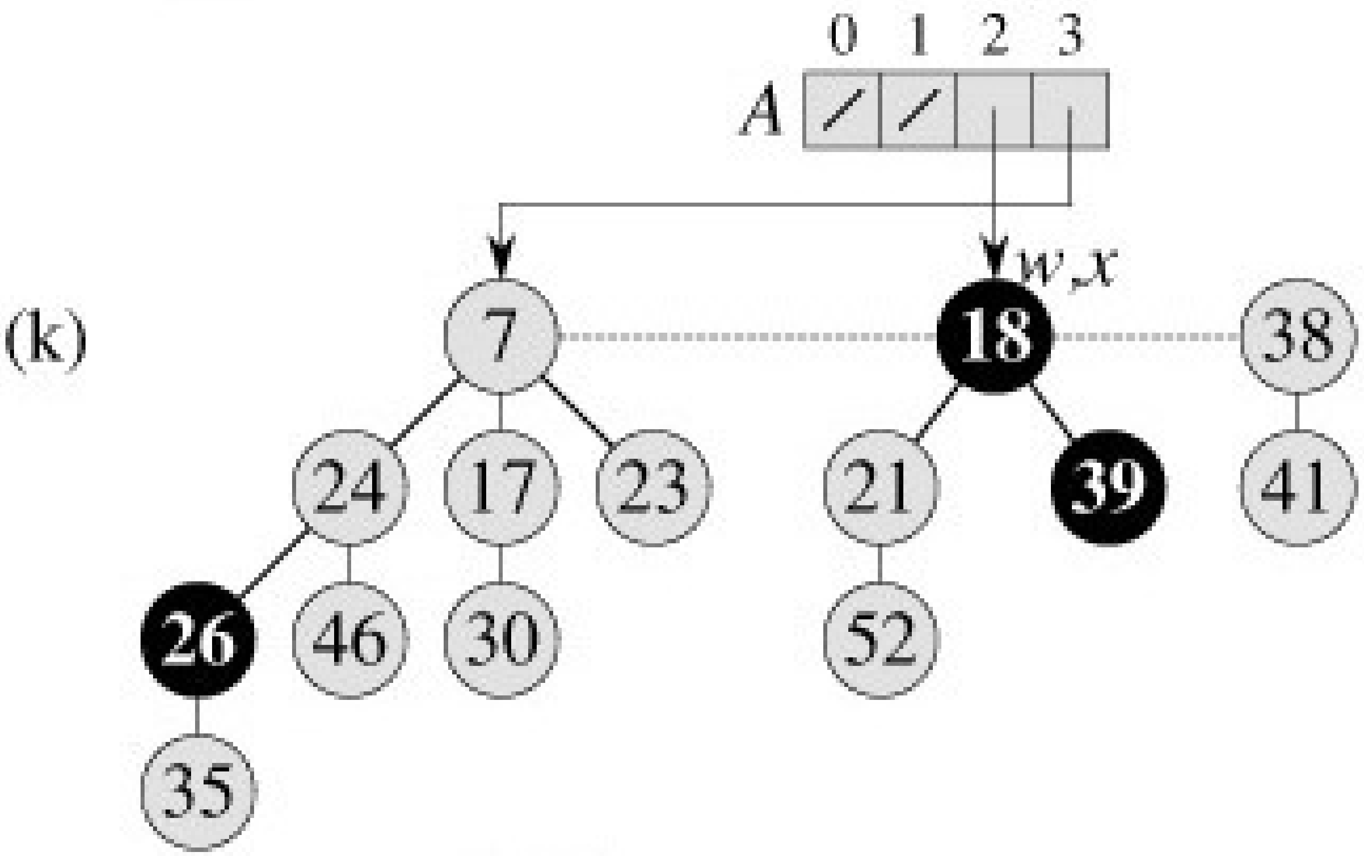
Delete-Min…



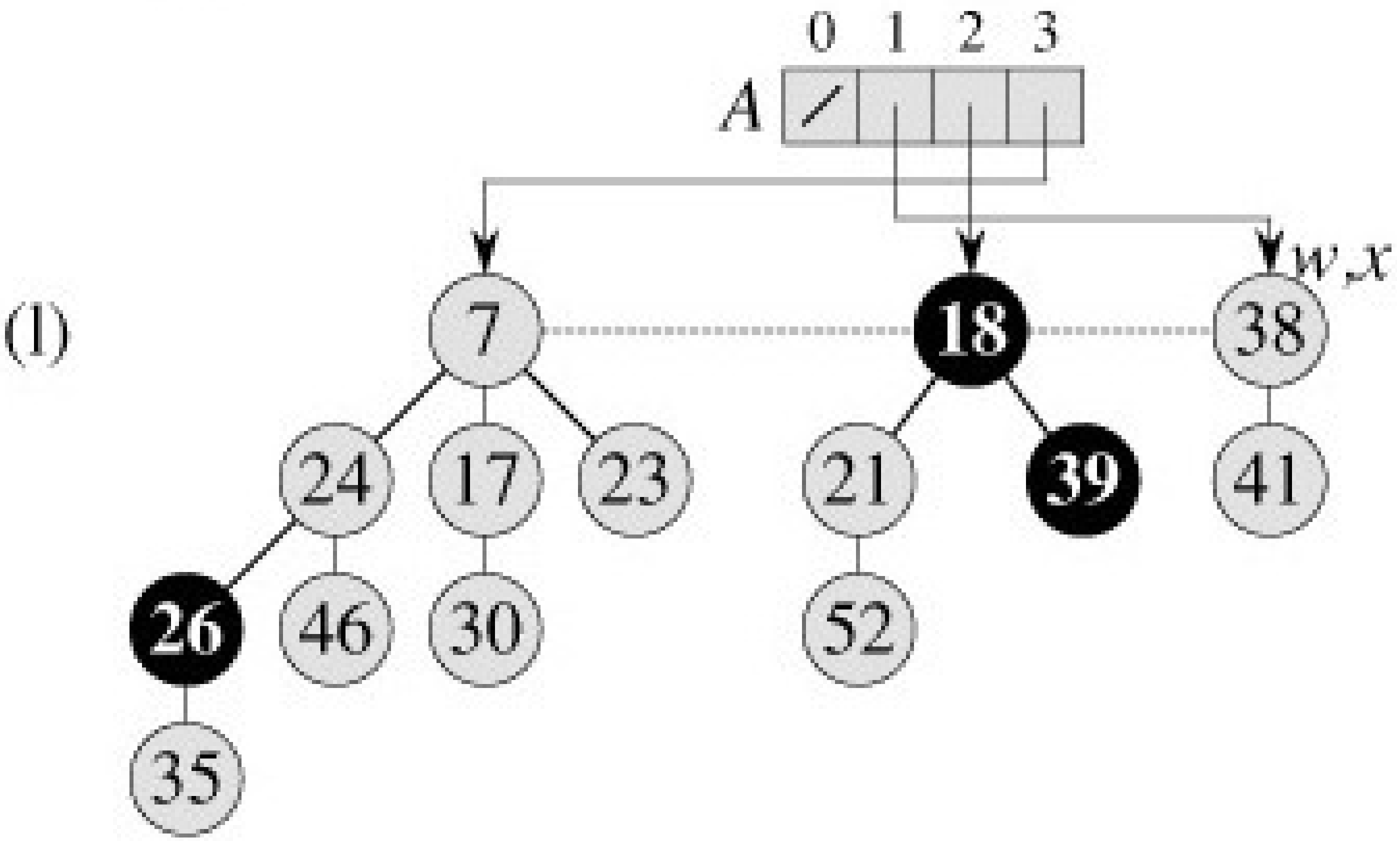
Delete-Min…



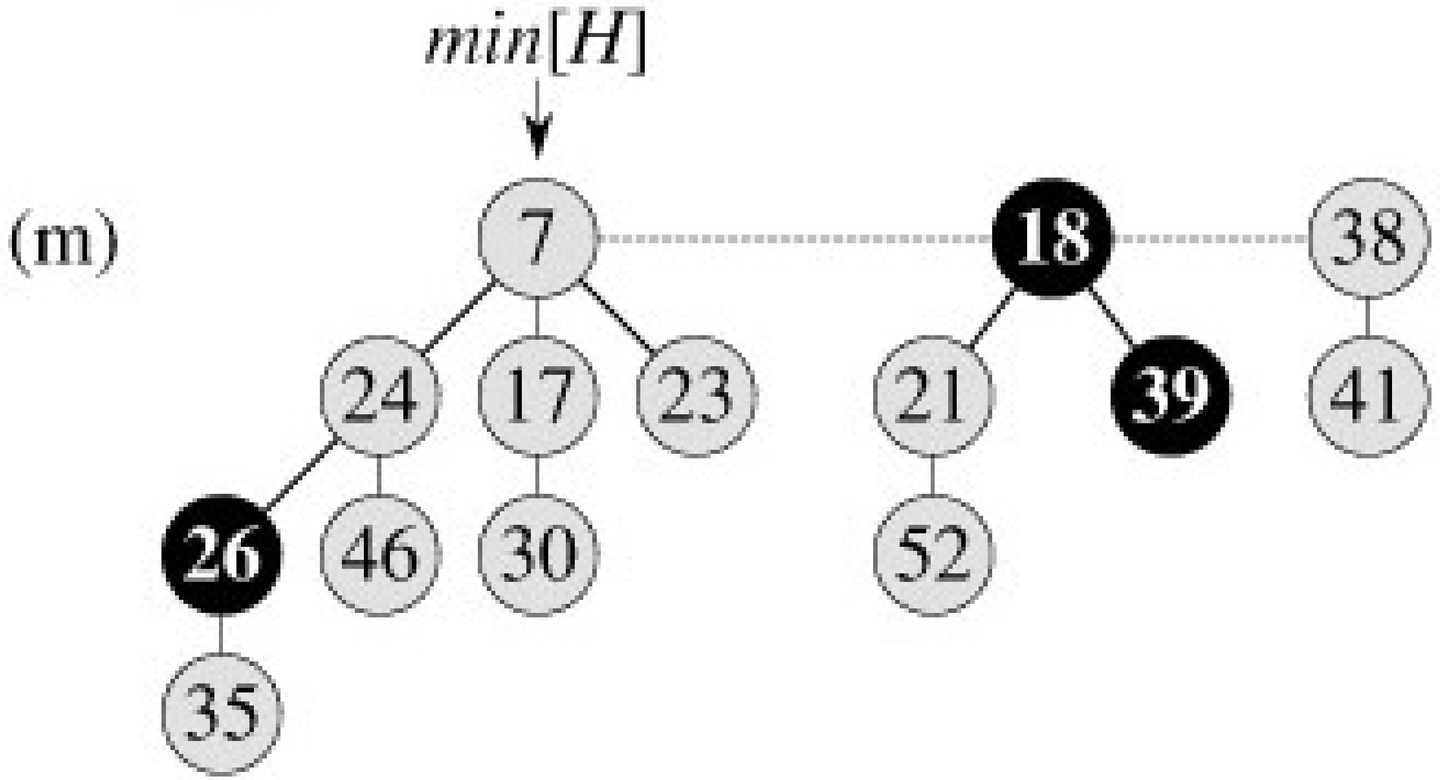
Delete-Min…



Delete-Min…



Delete-Min…



Delete-Min Analysis

Actual Cost

- If degree of min[H] is D(n), then D(n) children are added into root list of H and min[H] is removed from root list of H.
- Consolidate operation is called on a root list of length D(n) + t(H) -1
- In consolidation, every time through the while loop (line 6-12), one of the roots is linked to another. Total work in for loop is at most O(D(n) + t(H)).
- Thus actual cost is O(D(n) + t(H)).

Delete-Min Analysis

Change in Potential

- Potential before extraction: $t(H) + 2\ m(H)$.
- At most $D(n) + 1$ roots remain and no nodes become marked during the operation, potential after the operation is $((D(n) + 1) + 2\ m(H))$.
- Change in potential

  $((D(n) + 1) + 2\ m(H)) - (t(H) + 2\ m(H))$

$= (D(n) + 1 - t(H))$

Amortized cost $= O(D(n) + t(H)) + D(n) + 1 - t(H))$

$= O(D(n)) + O(t(H) + D(n) + 1 - t(H)$

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

FIB-HEAP-EXTRACT-MIN(H)

```
FIB-HEAP-EXTRACT-MIN(H)
$z \leftarrow min[H]$
if $z \neq NIL$
   then for each child x of z
            do add x to the root list of H
               $p[x] \leftarrow NIL$
        remove z from the root list of H
        if $z = right[z]$
           then $min[H] \leftarrow NIL$
           else $min[H] \leftarrow right[z]$
                CONSOLIDATE(H)
        $n[H] \leftarrow n[H] - 1$
return z
```

CONSOLIDATE(H)

```
CONSOLIDATE(H)
for i ← 0 to D(n[H])
    do A[i] ← NIL
for each node w in the root list of H
    do x ← w
       d ← degree[x]
       while A[d] ≠ NIL
           do  y ← A[d]
               if key[x] > key[y]
                  then exchange  x ←→ y
               FIB-HEAP-LINK(H,y,x)
               A[d] ← NIL
               d ← d +1
       A[d] ← x
min[H] ← NIL
for i ← 0 to D(n[H])
    do if A[i] ≠ NIL
          then add A[i] to the root list of H
               if min[H] = NIL or key[A[i]] < key[min[H]]
                  then min[H] ← A[i]
```

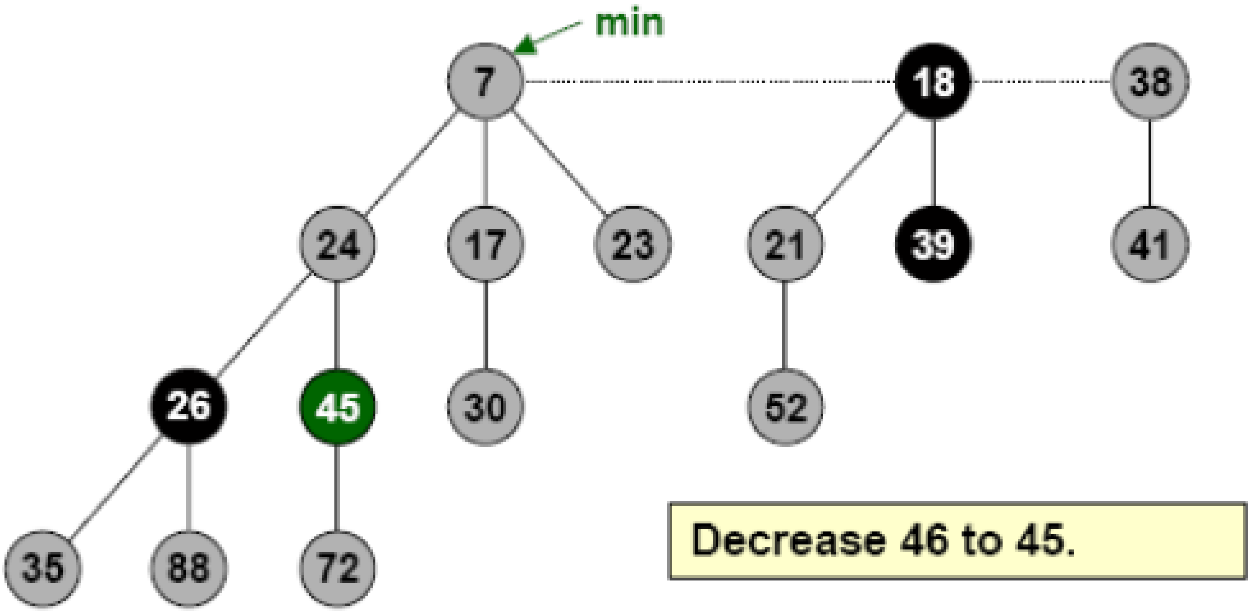
FIB-HEAP-LINK(H, y, x)

1. *Remove y from the root list of H*
2. *Make y a child of x, incrementing degree[x]*
3. *Mark[y]n ← FALSE*

Decrease key

Decrease key of element x to k.

- Case 0: min-heap property not violated.
  - decrease key of x to k
  - change heap min pointer if necessary

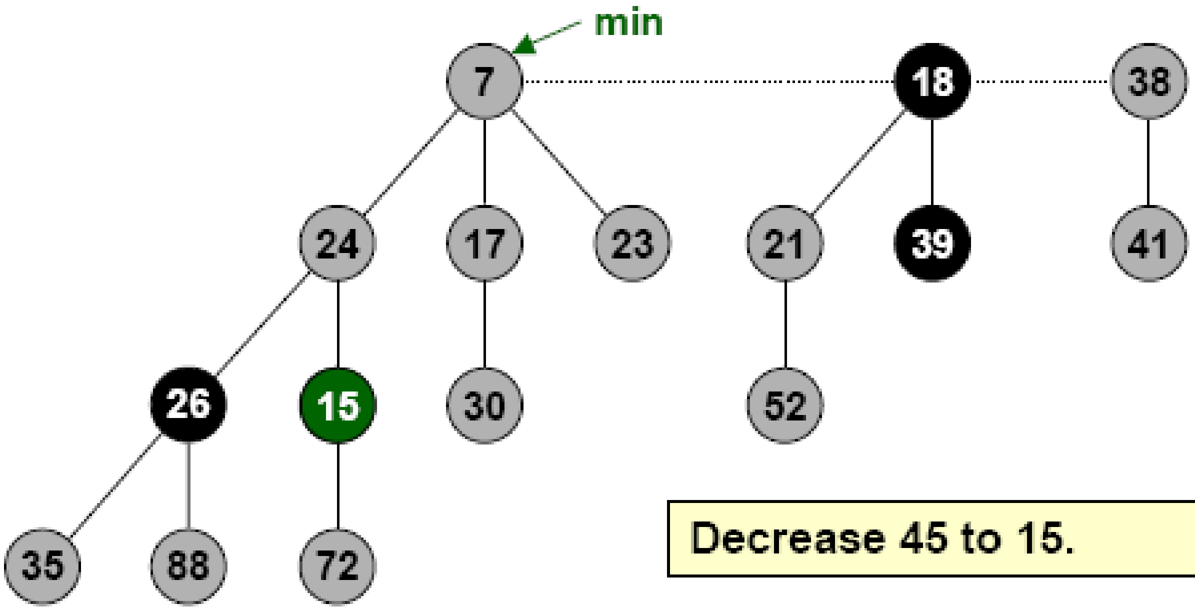


Decrease 46 to 45.

Decrease key

Decrease key of element x to k.

- Case 1: parent of x is unmarked.
  - decrease key of x to k
  - cut off link between x and its parent
  - mark parent
  - add tree rooted at x to root list, updating heap min pointer

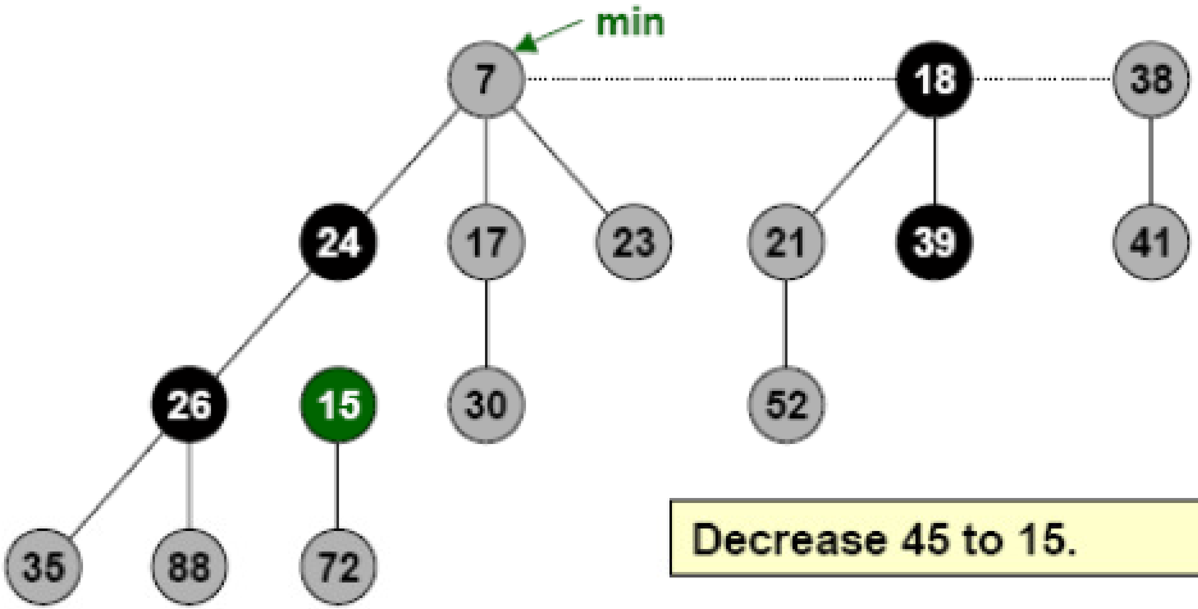


Decrease 45 to 15.

Decrease key

Decrease key of element x to k.

- Case 1: parent of x is unmarked.
  - decrease key of x to k
  - cut off link between x and its parent
  - mark parent
  - add tree rooted at x to root list, updating heap min pointer

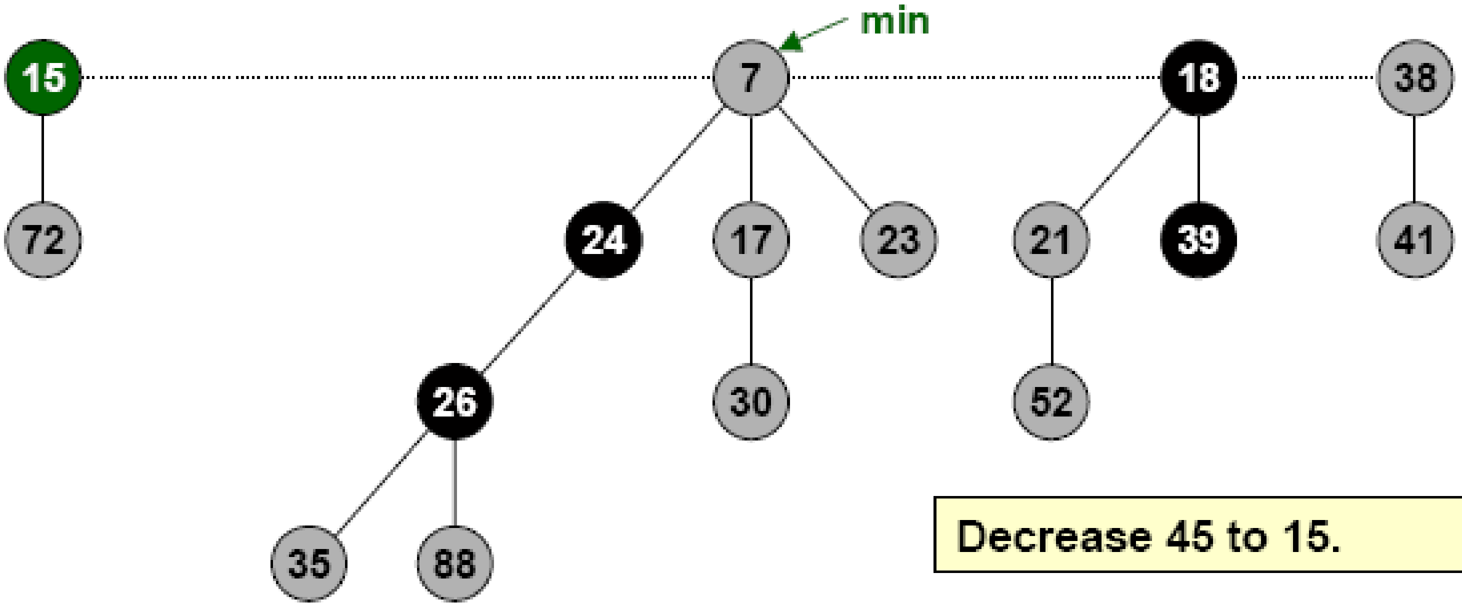


Decrease 45 to 15.

Decrease key

Decrease key of element x to k.

- Case 1: parent of x is unmarked.
  - decrease key of x to k
  - cut off link between x and its parent
  - mark parent
  - add tree rooted at x to root list, updating heap min pointer

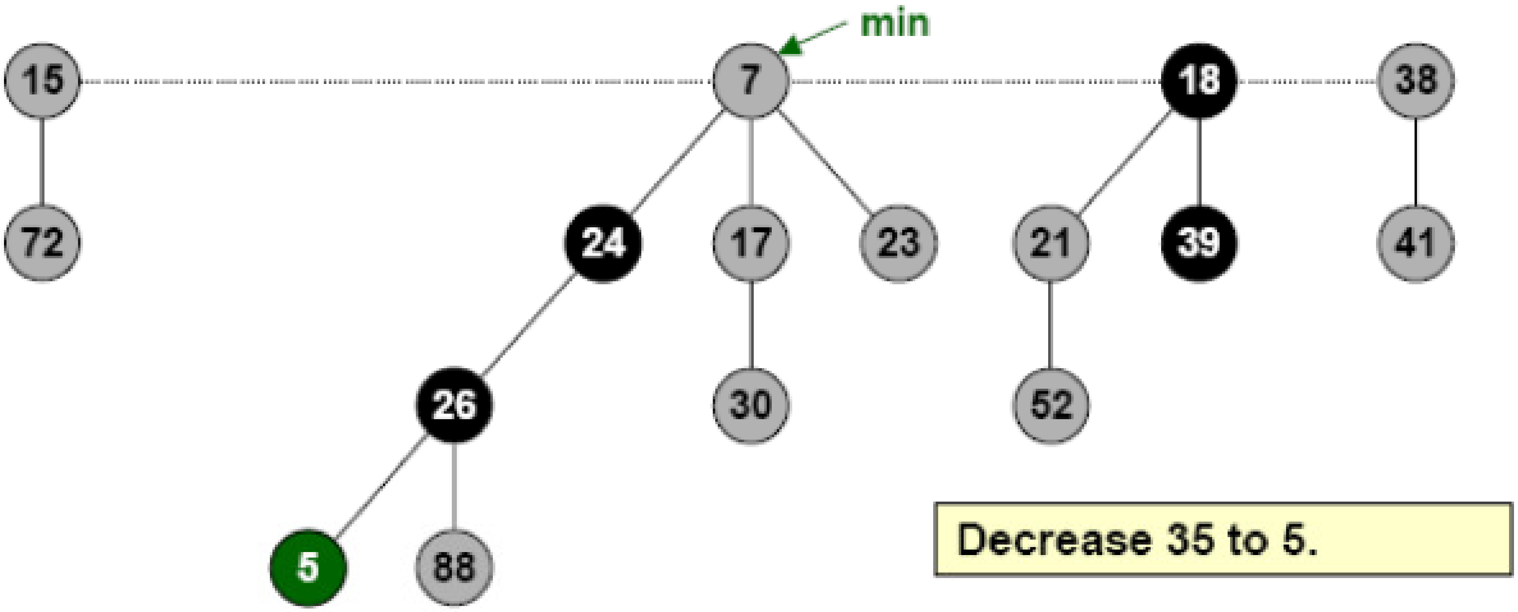


Decrease 45 to 15.

Decrease key

Decrease key of element x to k.

- Case 2: parent of x is marked.
  - decrease key of x to k
  - cut off link between x and its parent p[x], and add x to root list
  - cut off link between p[x] and p[p[x]], add p[x] to root list
    - If p[p[x]] unmarked, then mark it.
    - If p[p[x]] marked, cut off p[p[x]], unmark, and repeat.

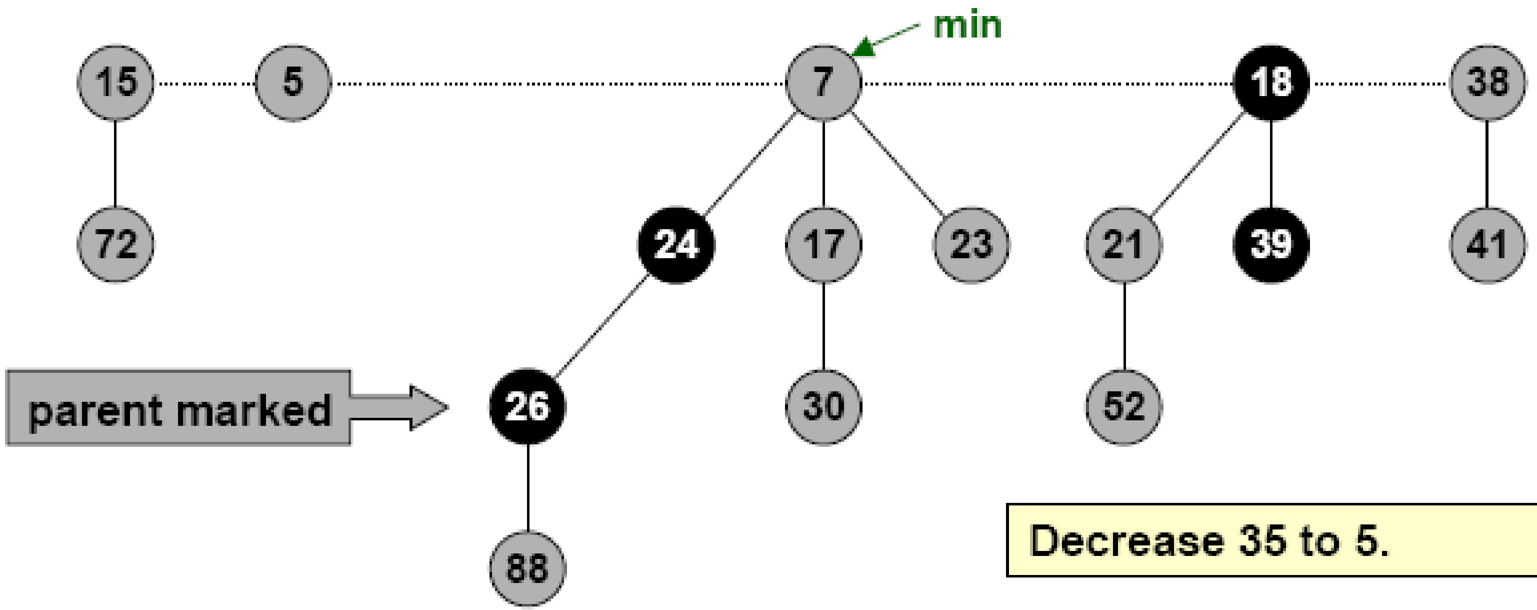


Decrease 35 to 5.

Decrease key

Decrease key of element x to k.

- Case 2: parent of x is marked.
  - decrease key of x to k
  - cut off link between x and its parent p[x], and add x to root list
  - cut off link between p[x] and p[p[x]], add p[x] to root list
    - If p[p[x]] unmarked, then mark it.
    - If p[p[x]] marked, cut off p[p[x]], unmark, and repeat.



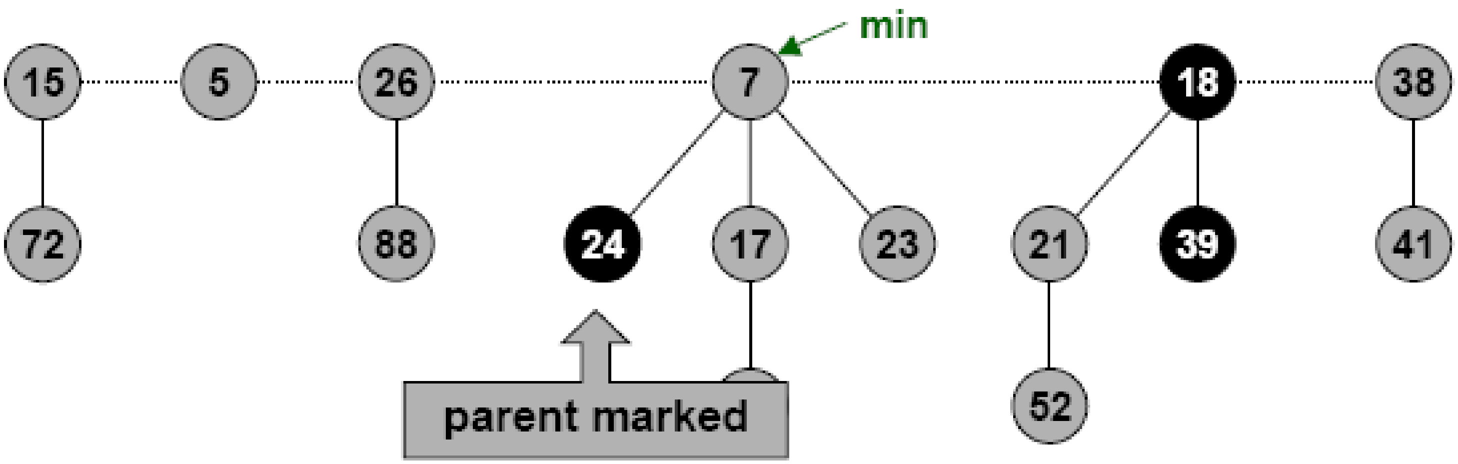
parent marked

Decrease 35 to 5.

Decrease key

Decrease key of element x to k.

- Case 2: parent of x is marked.
  - decrease key of x to k
  - cut off link between x and its parent p[x], and add x to root list
  - cut off link between p[x] and p[p[x]], add p[x] to root list
    - If p[p[x]] unmarked, then mark it.
    - If p[p[x]] marked, cut off p[p[x]], unmark, and repeat.



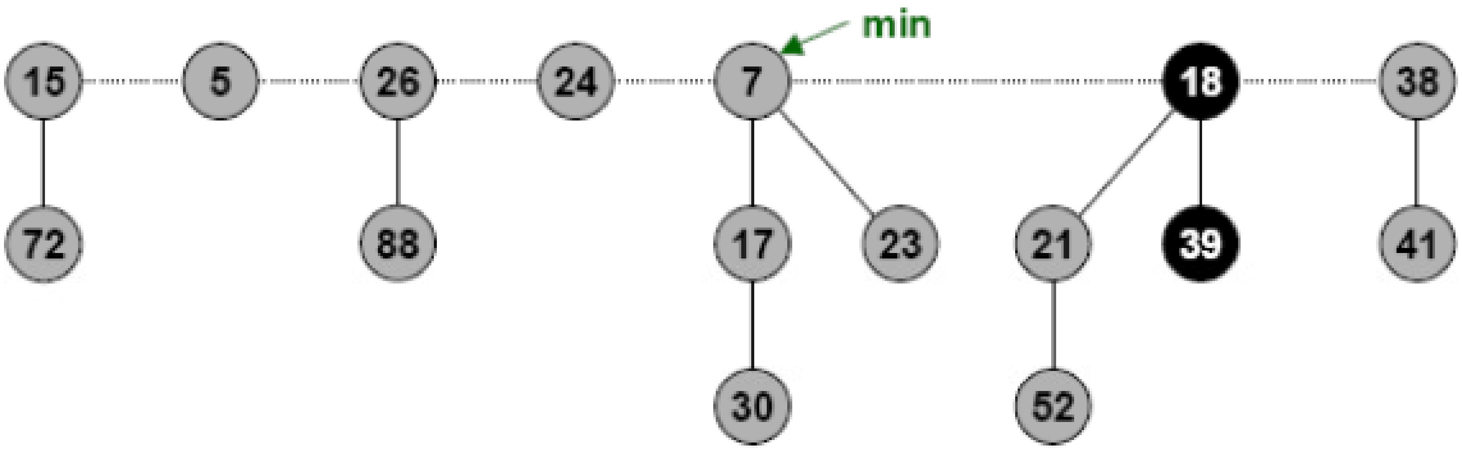
parent marked

Decrease 35 to 5.

Decrease key

Decrease key of element x to k.

- Case 2: parent of x is marked.
  - decrease key of x to k
  - cut off link between x and its parent p[x], and add x to root list
  - cut off link between p[x] and p[p[x]], add p[x] to root list
    - If p[p[x]] unmarked, then mark it.
    - If p[p[x]] marked, cut off p[p[x]], unmark, and repeat.



Decrease 35 to 5.

FIB-HEAP-DECREASE-KEY(H, x, k)

```
FIB-HEAP-DECREASE-KEY(H, x, k)
if $k > key[x]$
    then error "new key is greater than current key"
$key[x] \leftarrow k$
$y \leftarrow p[x]$
if $y \neq NIL$ and $key[x] < key[y]$
    then CUT(H,x,y)
         CASCADEING-CUT(H,y)
if $key[x] < key[min[H]]$
    then $min[H] \leftarrow x$
```

CUT( ) & CASCADING-CUT( )

```
CUT(H, x, k)
1. remove x from the child list of y, decreasing degree[y]
2. add x to the root list of H
3. p[x] ← NIL
4. mark[x] ← FALSE
```

```
CASCADING-CUT(H, y)
1. z ← p[y]
2. if z ≠ NIL
3.     then if mark[y] = FALSE
4.             then mark[y] ← TRUE
5.             else CUT(H,y,z)
6.                  CASCADING-CUT(H,z)
```

Decrease key Analysis

Actual cost

- FIB-HEAP-DECREASE-KEY procedure takes O(1) time plus time to perform CASCADING-CUT.
- Let CASCADING-CUT is called c times, each time it takes $O(1)$ time excluding recursive calls.
- Thus actual cost of FIB-HEAP-DECREASE-KEY is $O(c)$.

Decrease key Analysis

- Change in potential.
- Each CASCADING-CUT except the last one , cuts a marked node and clears mark bit.
- Afterward there are $t(H) + c$ trees. ($c-1$ trees created from by cascading cuts and at most $m(H) - c + 2$ marked. ($c-1$ were unmarked by cascading cuts and the last call of CASCADING-CUT may have marked a node)
- Change in potential= $((t(H) + c) + 2\,(m(H) - c + 2)) - (t(H) + 2\,m(H)) = 4 - c$.
- Amortized Cost = Actual Cost + change in potential

  $= O(c) + 4 - c = O(1)$

Delete x

Delete node x.

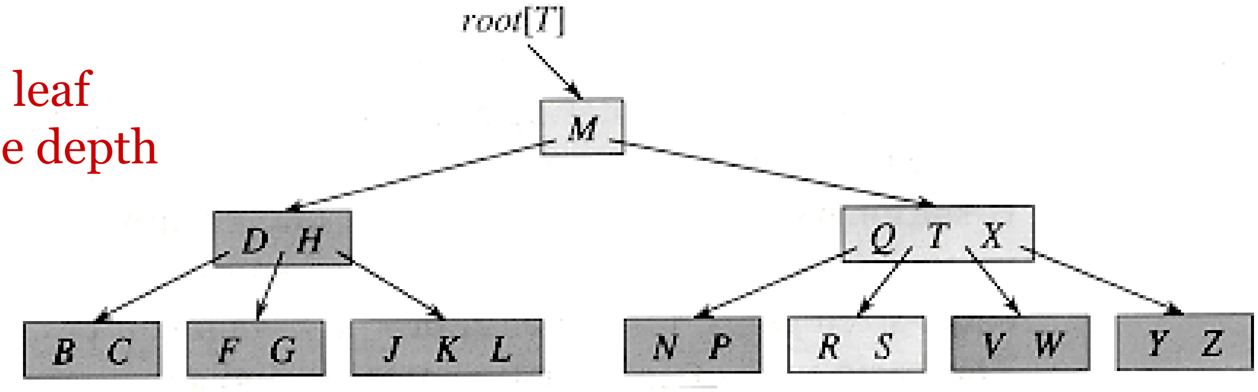
- Decrease key of x to -∞.
- Delete min element in heap.

Amortized cost. O(D(n))

- O(1) for decrease-key.
- O(D(n)) for delete-min.
- D(n) = max degree of any node in Fibonacci heap.

B-Trees

Note: Each leaf
has the same depth



A B-tree whose keys are the consonants of English. All leaves are at the same depth in the tree. The lightly shaded nodes are examined in a search for the letter R

Definition of B-trees

- A ***B-tree*** *T* is a rooted tree (with root *root*[*T*]) having the following properties.

1. Every node *x* has the following fields:

   a. $n[x]$, the number of keys currently stored in node *x*,

   b. the $n[x]$ keys themselves, stored in nondecreasing order:

   $key_1[x] \leq key_2[x] \ldots. \leq key_n[x][x]$, and

   c. *leaf* [*x*], a boolean value that is TRUE if *x* is a leaf and FALSE if *x* is an internal node.

2. If *x* is an internal node, it also contains $n[x] + 1$ pointers $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$ to its children. Leaf nodes have no children, so their c_i fields are undefined.

3. The keys $key_i[x]$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $c_i[x]$, then

   $k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \ldots.. \leq \mathrm{key}_{n[x]}[x]\ k_n[x]+1$.

Definition of B-trees

4. Every leaf has the same depth, which is the tree's height h.

5. There are lower and upper bounds on the number of keys a node can contain. These bounds can be expressed in terms of a fixed integer t 2 called the ***minimum degree*** of the B-tree:

a. Every node other than the root must have at least t - 1 keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have at least one key.

b. Every node can contain at most $2t$ - 1 keys. Therefore, an internal node can have at most $2t$ children. We say that a node is ***full*** if it contains exactly $2t$ - 1 keys.

The simplest B-tree occurs when $t = 2$. Every internal node then has either 2, 3, or 4 children, and we have a ***2-3-4 tree.*** In practice, however, much larger values of t are typically used.

Definition of B-trees

- $\exists\ t \geq 2$ called the **minimum degree**.

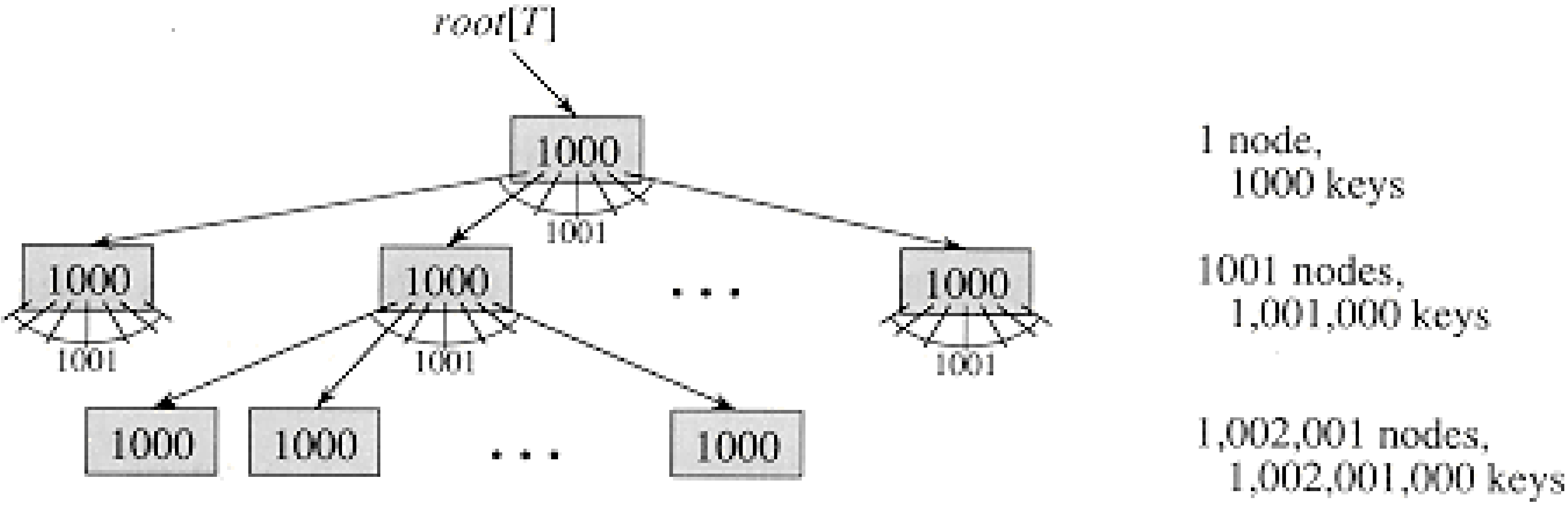
  $x \neq \text{root} \Rightarrow t-1 \leq n[x] \leq 2t-1$

  $x = \text{root} \Rightarrow n[x] \leq 2t-1$

Application: Disk Accesses

- Each node is stored as a page.
- Page size determines t.
  - t is usually large
  - Implies branching factor is large, so height is small.
- Disk accesses dominate performance in this application.

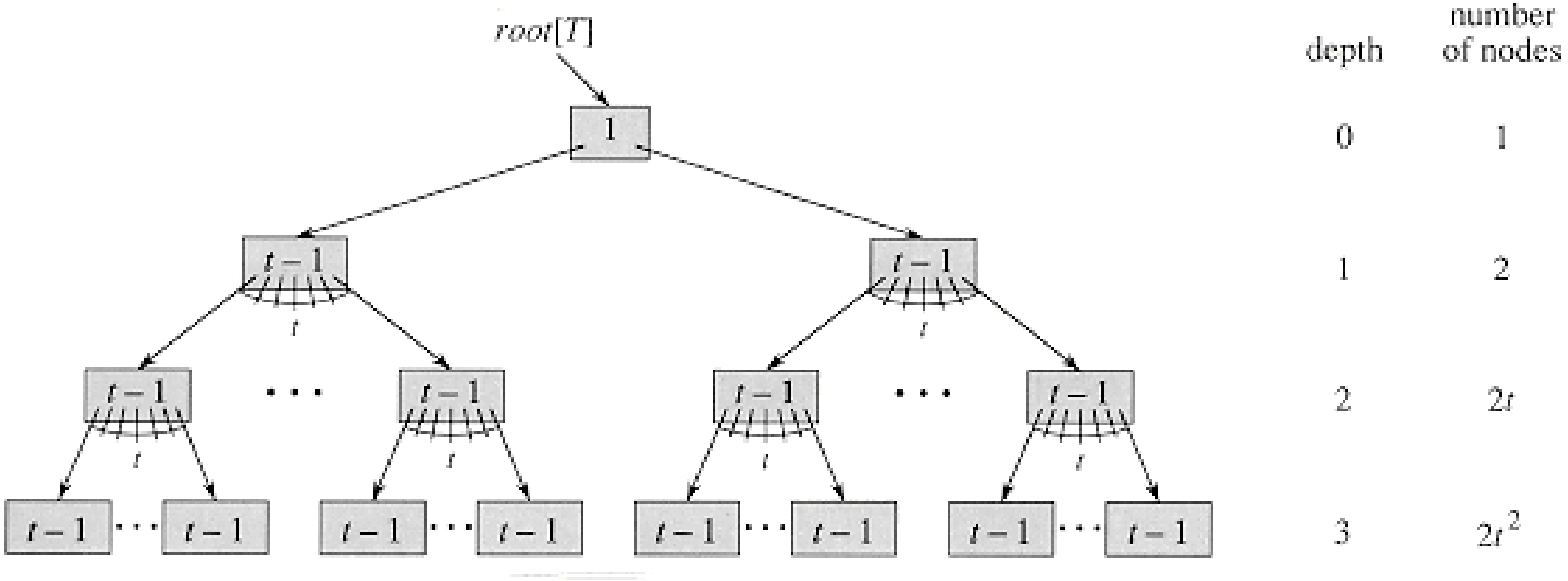
B-Tree: Example



A B-tree of height 2 containing over one billion keys. Each internal node and leaf contains 1000 keys. There are 1001 nodes at depth 1 and over one million leaves at depth 2. Shown inside each node x is n[x], the number of keys in x.

If *n* 1, then for any *n*-key B-tree *T* of height *h* and minimum degree *t* 2,

Height of B-tree



If n ≥ 1, then for any *n*-key B-tree *T* of height *h* and minimum degree t ≥ 2

$$h \le \log_t \frac{n+1}{2} .$$

Proof

- If a B-tree has height *h*, the number of its nodes is minimized when the root contains one key and all other nodes contain *t* - 1 keys.
- In this case, there are 2 nodes at depth 1, 2*t* nodes at depth 2, $2t^2$ nodes at depth 3, and so on, until at depth *h* there are $2t^{h-1}$ nodes. Thus, the number *n* of keys satisfies the inequality

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h - 1}{t - 1}\right) \\ &= 2t^h - 1 , \end{aligned}$$

B-Tree Operations

- **Search:**
  - $\Theta(\log_t n)$ disk accesses.
  - $O(t \log_t n)$ CPU time.
- **Create:**
  - $O(1)$ disk accesses.
  - $O(1)$ CPU time.
- **Insert and Delete:**
  - $O(\log_t n)$ disk accesses.
  - $O(t \log_t n)$ CPU time.

- In the code that follows, we use:
  - **Disk-Read:** To move node from disk to memory.
  - **Disk-Write:** To move node from memory to disk.
- We assume root is in memory.

Search

```
B-TREE-SEARCH(x, k)
    i ← 1;
    while $i \le n[x]$ and $k > key_i[x]$
        do i ← i + 1
    if $i \le n[x]$ and $k = key_i[x]$
        then return(x, i)
    if leaf[x]
        then return NIL
        else DiskRead($c_i[x]$);
            return B-TREE-SEARCH($c_i[x]$, k)
```

Search(root[T], k) returns (y,i) s.t. $key_i[y] = k$ or NIL if no such key.

Worst-case:

$\Theta(\log_t n)$ disk reads.

$\Theta(t \log_t n)$ CPU time.

Create

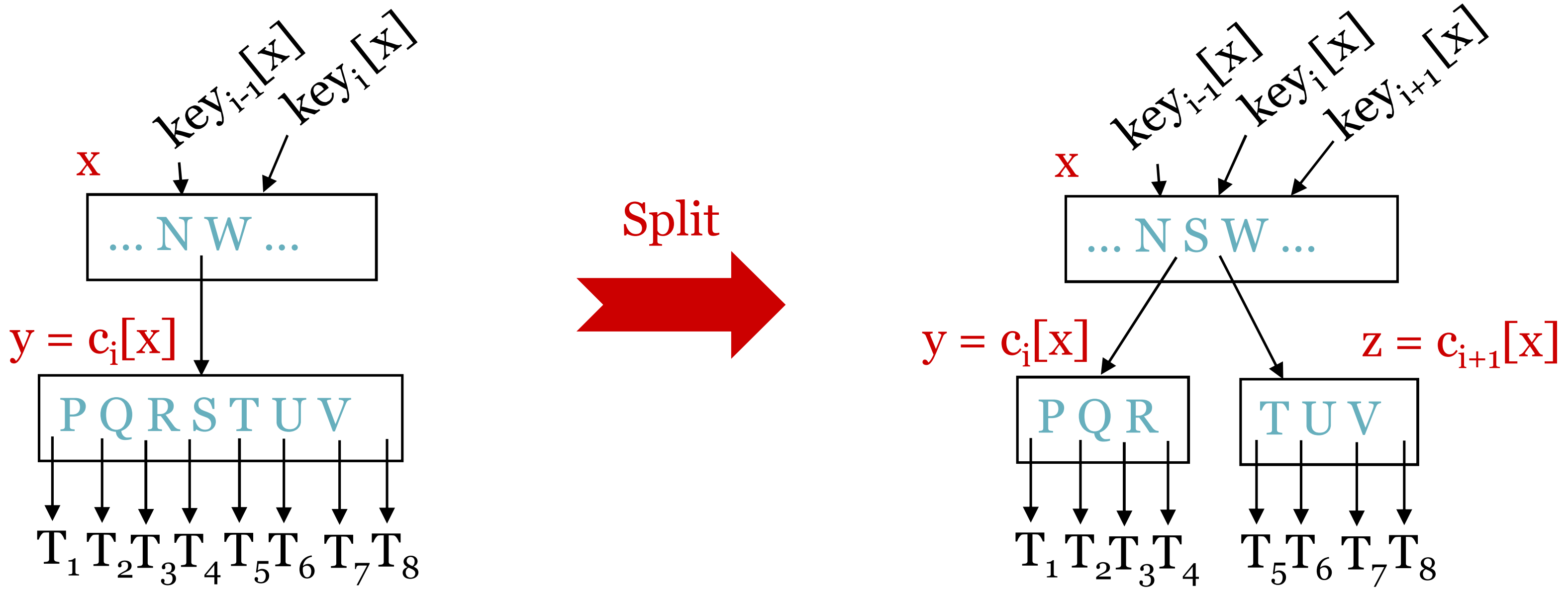
```
B-TREE-CREATE(T)
1.  x ← ALLOCATE-NODE()
2.  leaf[x] ←TRUE
3.  n[x] ← 0
4.  DISK-WRITE(x)
5.  root[T] ← x
```

- To create a nonempty tree, first create an empty tree, then insert nodes.
- O(1) time

Splitting

Applied to a "full" child of a "nonfull" parent. "full" ≡ 2t−1 keys.

Example: (t=4)



Split Child

```
B-TREE-SPLIT-CHILD(x, i, y)
z ← ALLOCATE-NODE()
leaf[z] ← leaf[y]
n[z] ← t−1
for j ← 1 to t−1
    do $key_j[z] ← key_{j+t}[y]$
if not leaf[y]
    then for j ← 1 to t
            do $c_j[z] ← c_{j+t}[y]$
n[y] ← t−1
for j ← n[x] + 1 downto i+1
    do $c_{j+1}[x] ← c_j[x]$
$c_{i+1}[x] ← z$
for j ← n[x] downto i
    do $key_{j+1}[x] ← key_j[x]$
$key_i[x] ← key_t[y]$
n[x] ← n[x] + 1
Disk-Write(y)
Disk-Write(z)
Disk-Write(x)
```

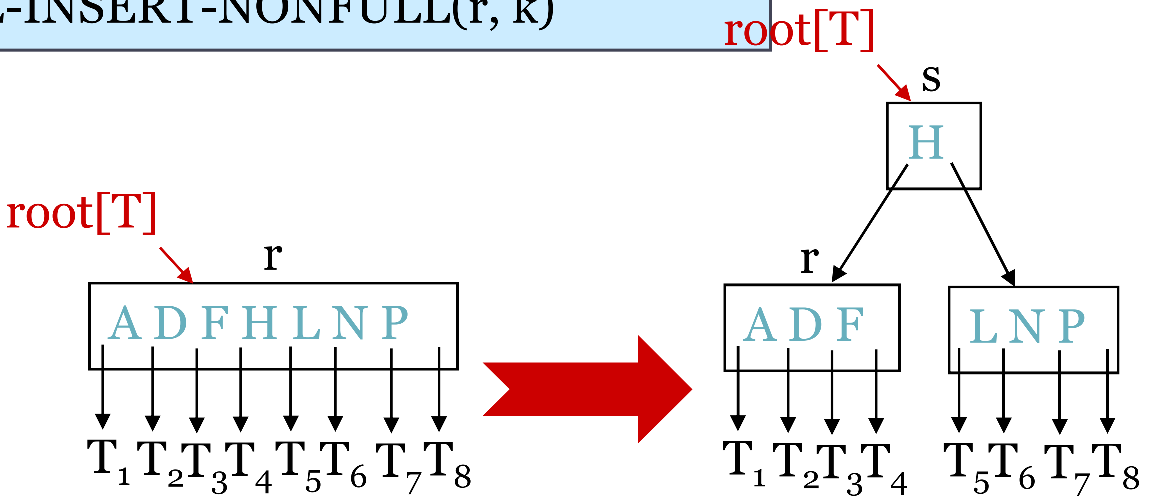
$\Theta(t)$ CPU time.
$O(1)$ disk writes.

Insert

```
B-TREE-INSERT(T, k)
    r ← root[T]
    if n[r] = 2t–1
        then s ← Allocate-Node()
             root[T] ← s
             leaf[s] ← false
             n[s] ← 0
             $c_1$[s] ← r
             B-TREE-SPLIT-CHILD(s, 1, r);
             B-TREE-INSERT-NONFULL(s, k)
        else B-TREE-INSERT-NONFULL(r, k)
```

First, modify tree
(if necessary) to create
room for new key.
Then, call Insert-Nonfull().

Example:



Insert-Nonfull

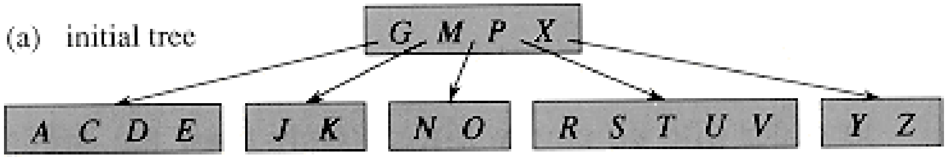
```
B-TREE-INSERT-NONFULL(x, k)
i ← n[x]
if leaf[x]
   then while i ≥ 1 and k < $key_i[x]$
               do $key_{i+1}[x]$ ← $key_i[x]$
                  i ← i–1
        $key_{i+1}[x]$ ← k
        n[x] ← n[x] + 1
        DISK-WRITE(x)
   else while i ≥ 1 and k < $key_i[x]$
               do i ← i–1
        i ← i + 1
        DISK-WRITE($c_i[x]$)
        if n[$c_i[x]$] = 2t–1
           then B-TREE-SPLIT-CHILD(x, i, $c_i[x]$)
                if k > $key_i[x]$
                   then i ← i + 1
        B-TREE-INSERT-NONFULL($c_i[x]$, k)
```

Worst Case:

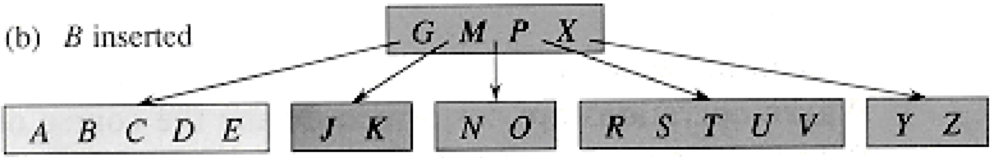
$\Theta(t \log_t n)$ CPU time.

$\Theta(\log_t n)$ disk writes.

(a) initial tree

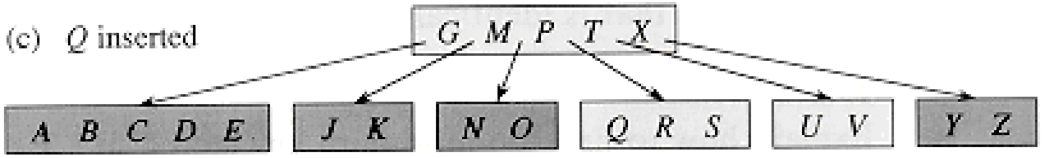


(b) *B* inserted

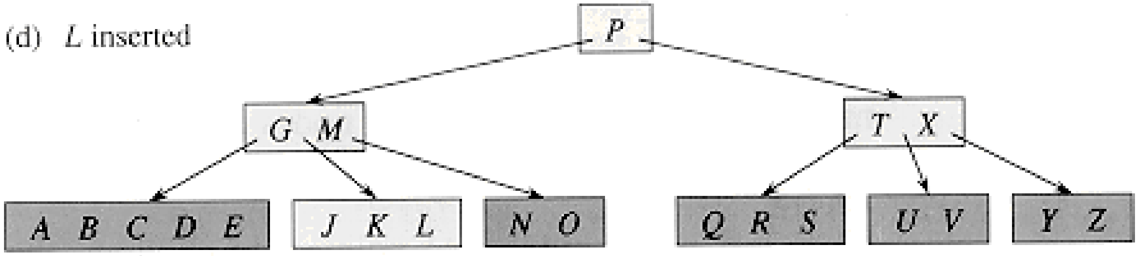


t = 3

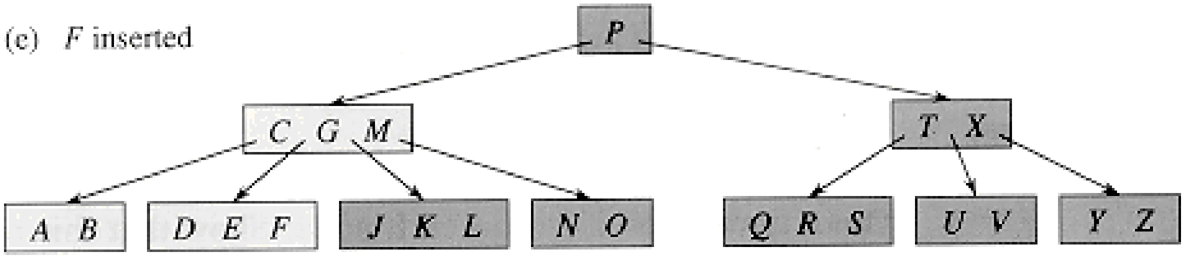
(c) *Q* inserted



(d) *L* inserted



(e) *F* inserted

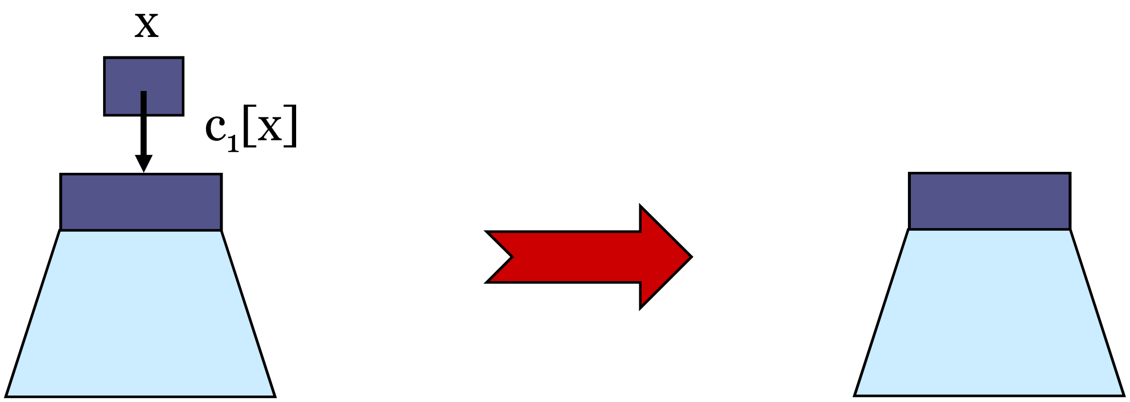


Deletion

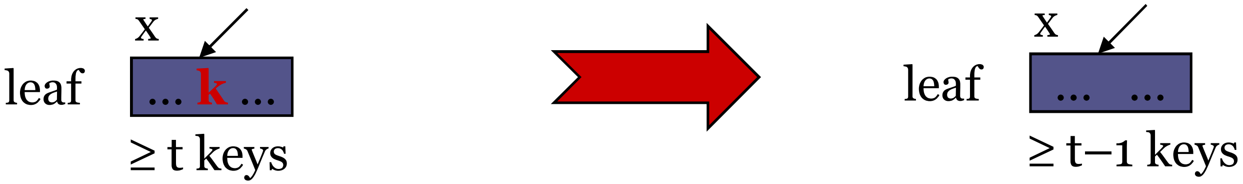
- **Main Idea:** Recursively descend tree.
- **Ensure any non-root node x that is considered has at least t keys.**
- May have to move key down from parent.

Deletion:cases

Case 0: Empty root -- make root’s only child the new root.

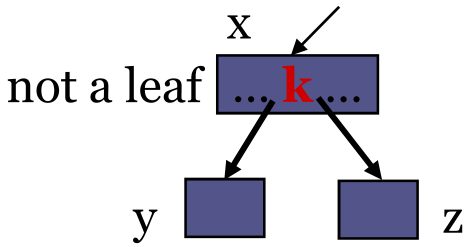


Case 1: k in x, x is a leaf -- delete k from x.

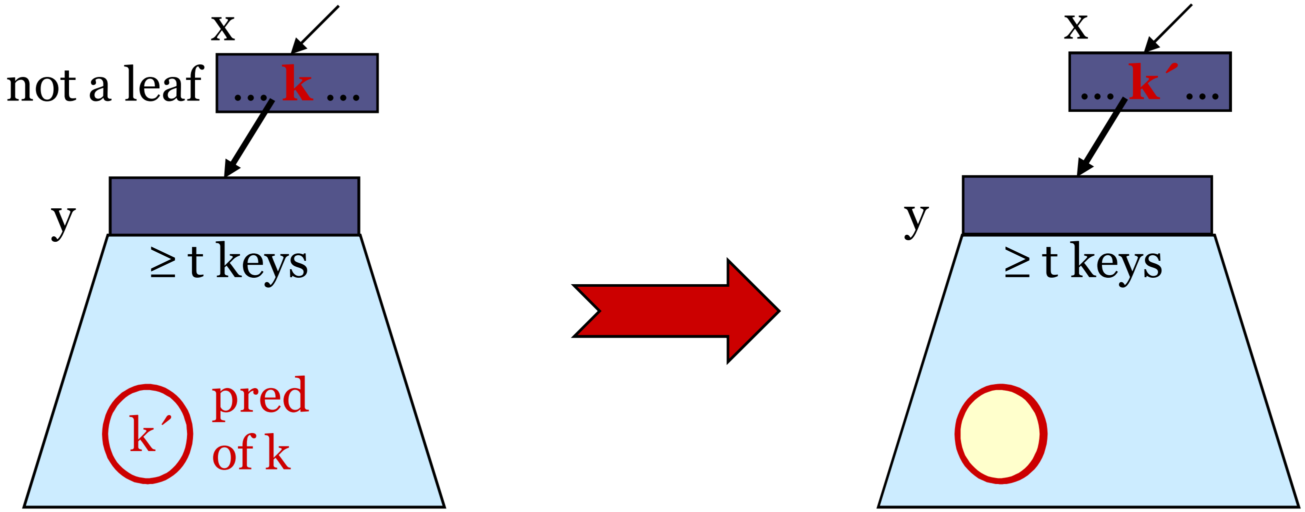


Deletion:cases

Case 2: k in x, x internal.

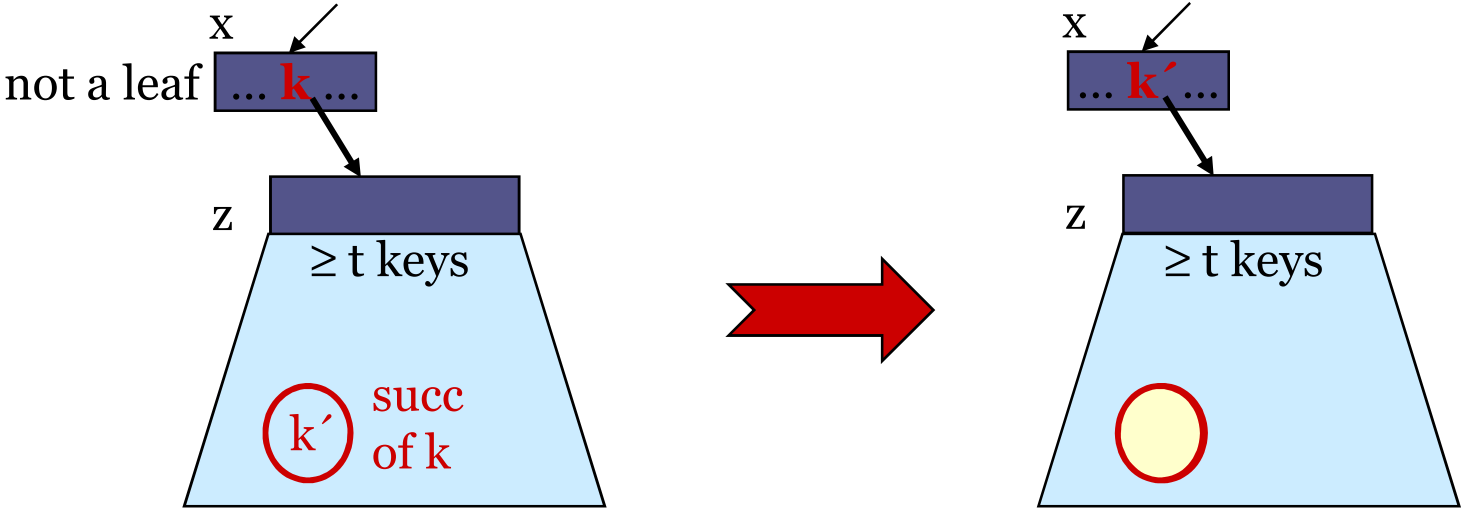


Subcase A: y has at least t keys -- find predecessor k´ of k in subtree rooted at y, recursively delete k´, replace k by k´ in x.

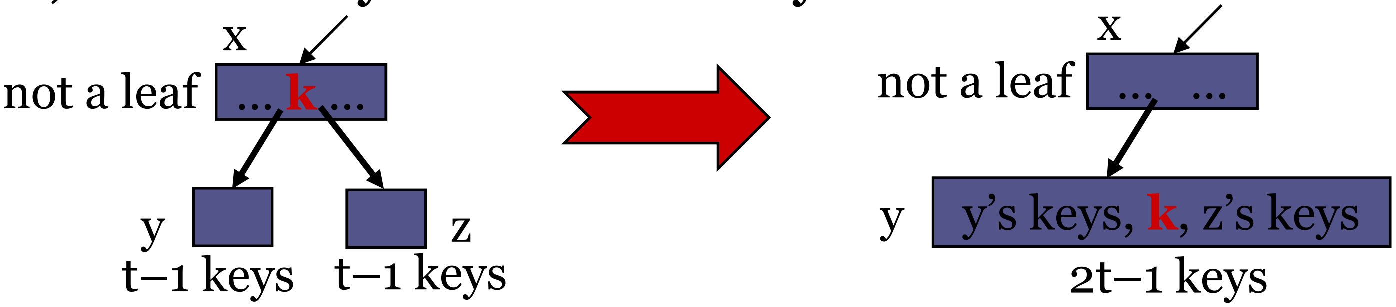


Deletion:cases

Subcase B: z has at least t keys -- find successor k´ of k in subtre rooted at z, recursively delete k´, replace k by k´ in x.



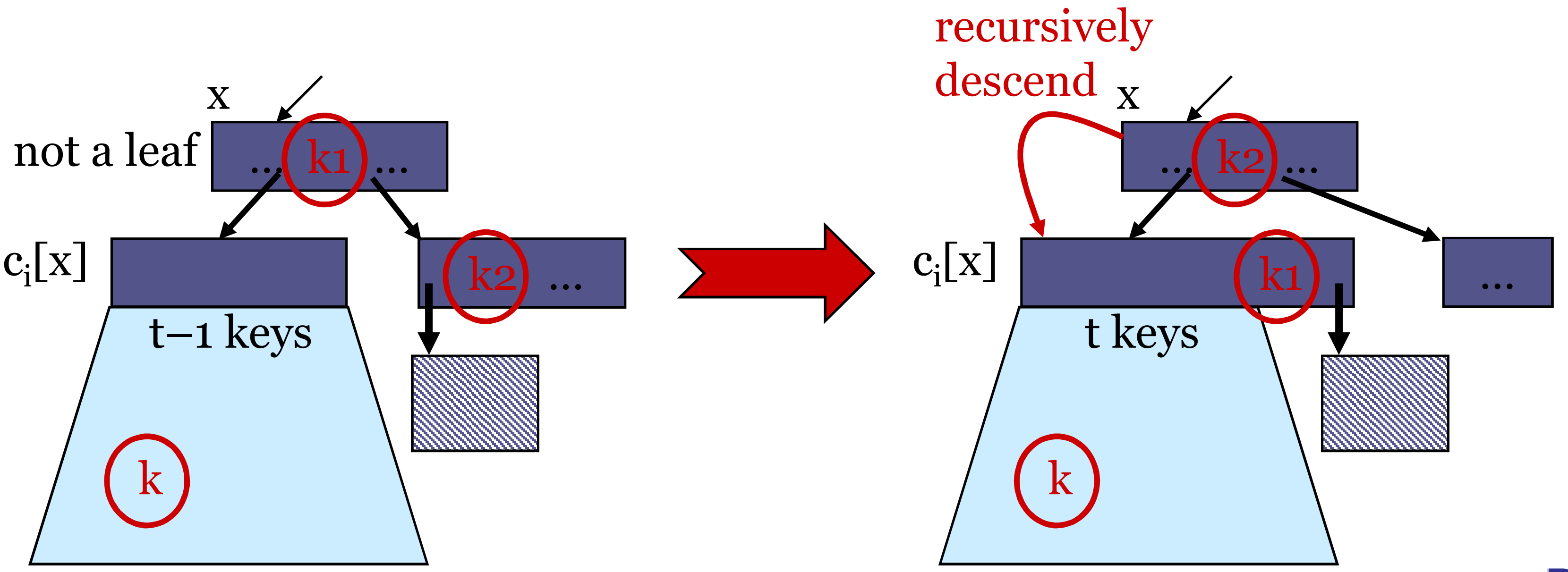
Subcase C: y and z both have t–1 keys -- merge k and z into y, fre z, recursively delete k from y.



Deletion:cases

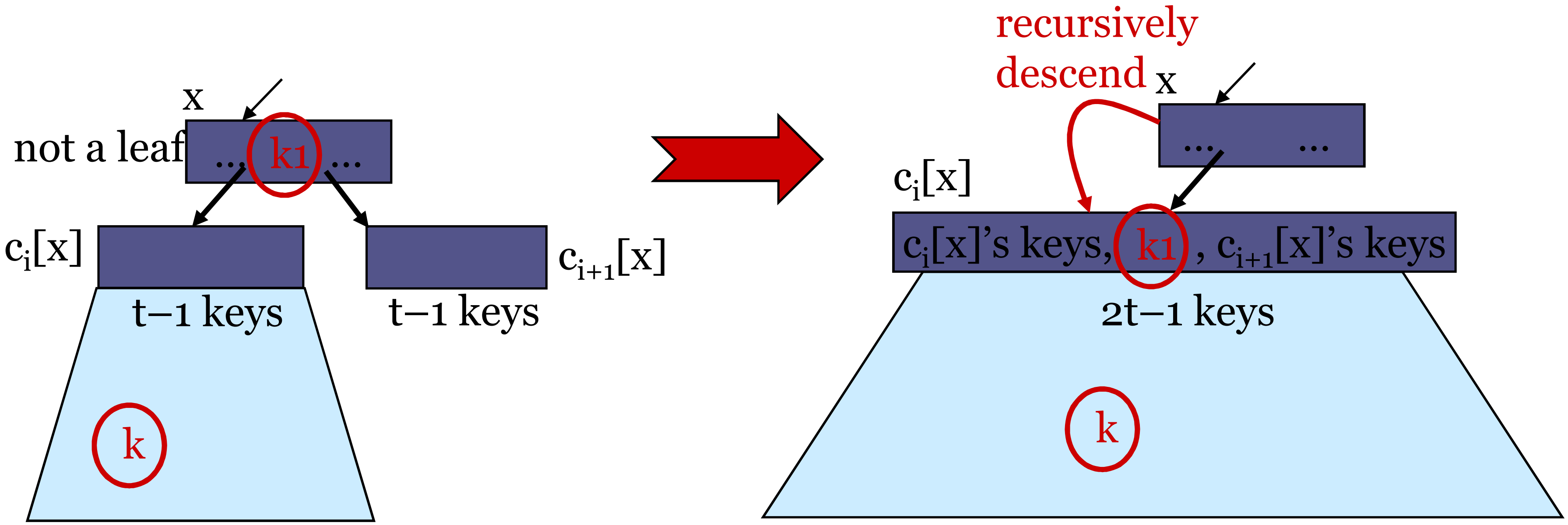
Case 3: k not in internal node. Let $c_i[x]$ be the root of the subtree that must contain k, if k is in the tree. If $c_i[x]$ has at least t keys, then recursively descend; otherwise, execute 3.A and 3.B as necessary.

Subcase A: $c_i[x]$ has t–1 keys, some sibling has at least t keys.



Deletion:cases

Subcase B: $c_i[x]$ and sibling both have t–1 keys.

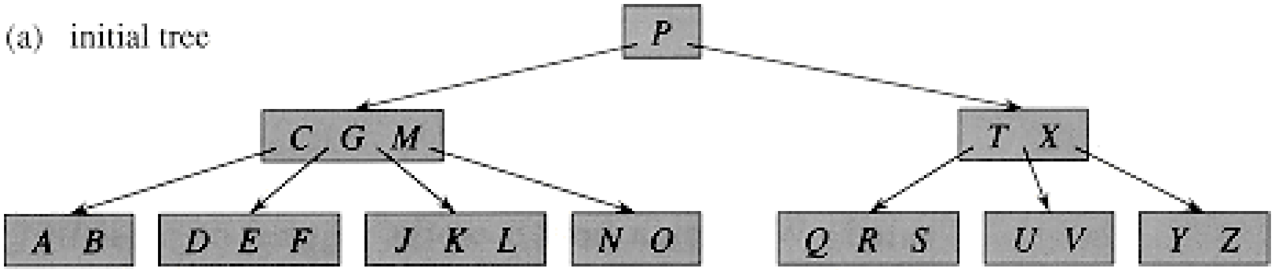


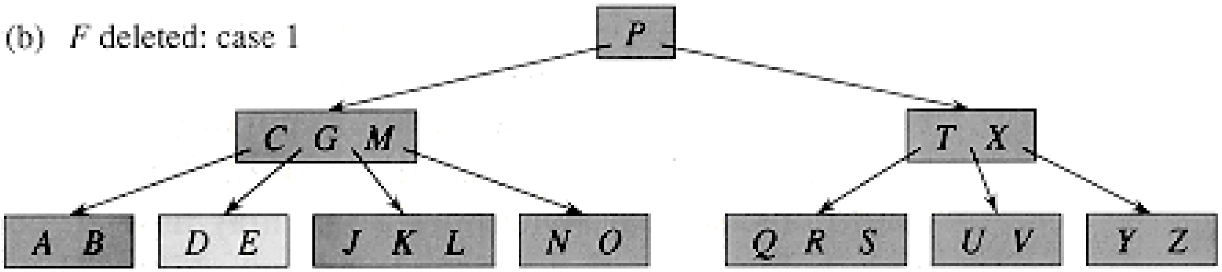
Deletion: Summary

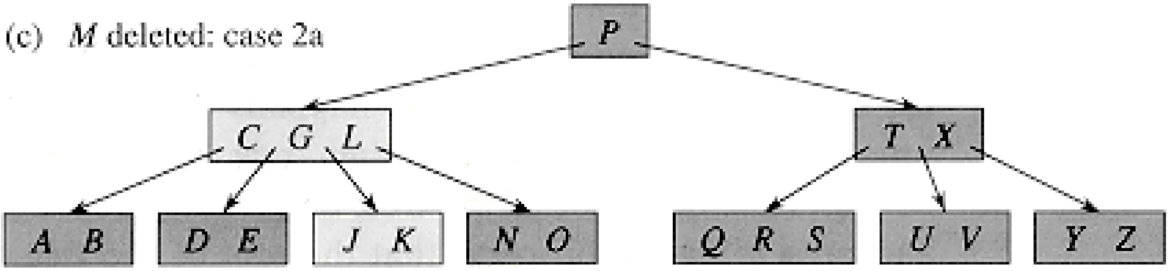
1. If the key k is in node x and x is a leaf, delete the key k from x.
2. If the key k is in node x and x is an internal node, do the following.
   a. If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k', and replace k by k' in x. (Finding k' and deleting it can be performed in a single downward pass.)
   b. Symmetrically, if the child z that follows k in node x has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k', and replace k by k' in x. (Finding k' and deleting it can be performed in a single downward pass.)
   c. Otherwise, if both y and z have only t- 1 keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains $2t$ - 1 keys. Then, free z and recursively delete k from y.
3. If the key k is not present in internal node x, determine the root $c_i[x]$ of the appropriate subtree that must contain k, if k is in the tree at all. If $c_i[x]$ has only t - 1 keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then, finish by recursing on the appropriate child of x.
   a. If $c_i[x]$ has only t - 1 keys but has a sibling with t keys, give $c_i[x]$ an extra key by moving a key from x down into $c_i[x]$, moving a key from$c_i[x]$'s immediate left or right sibling up into x, and moving the appropriate child from the sibling into $c_i[x]$.
   b. If $c_i[x]$ and all of $c_i[x]$'s siblings have t - 1 keys, merge c_i with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

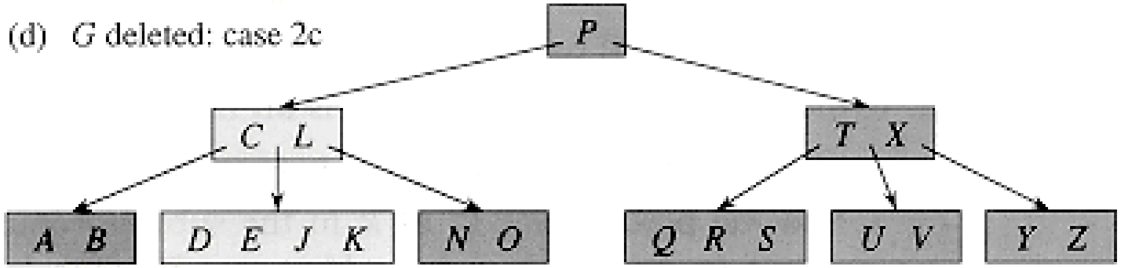
Deletion

t=3
Minimum keys = 2
Maximum keys = 5

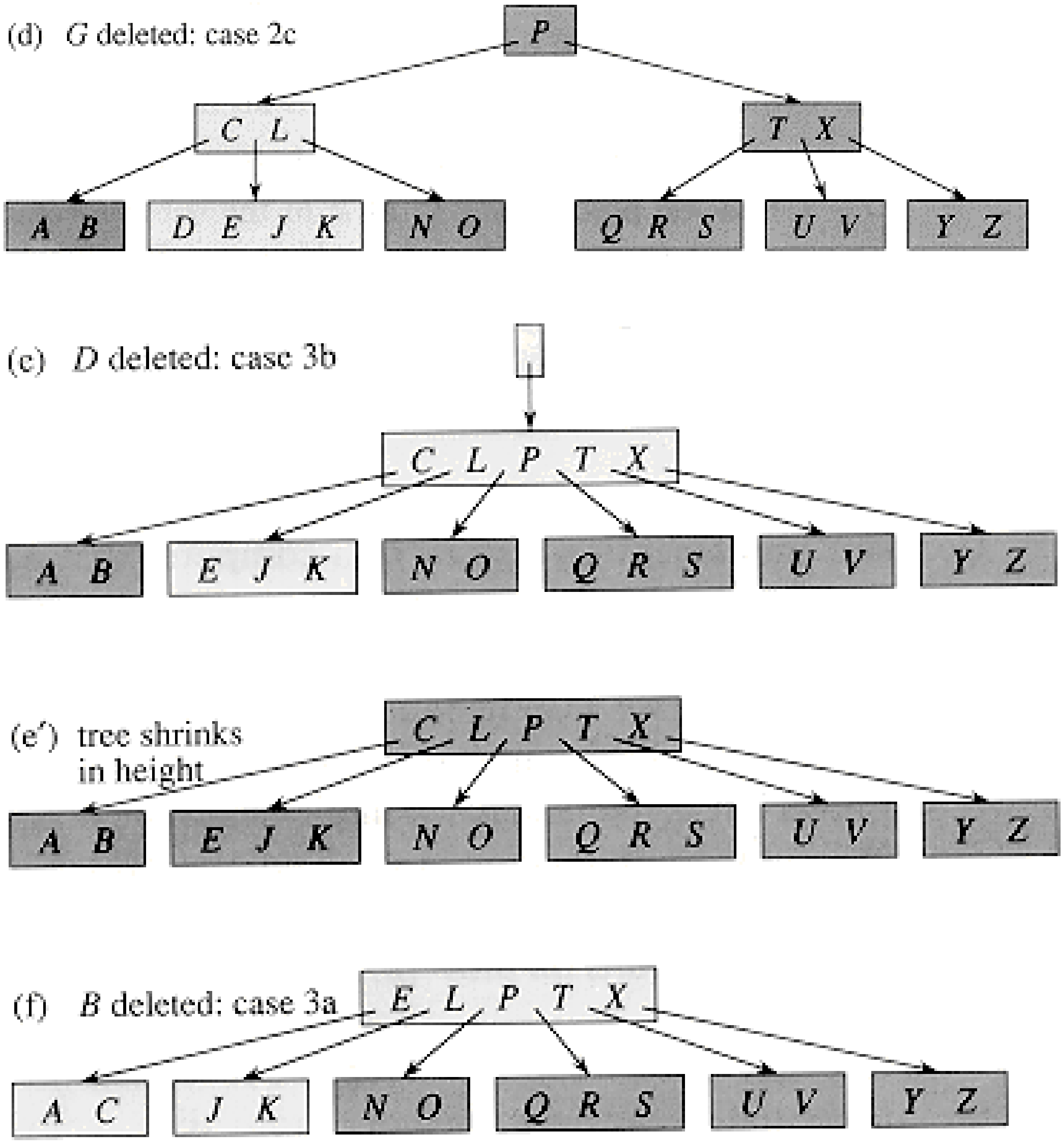

(a) initial tree


(b) *F* deleted: case 1


(c) *M* deleted: case 2a


(d) *G* deleted: case 2c

Deletion


(d) *G* deleted: case 2c

(c) *D* deleted: case 3b

(e′) tree shrinks in height

(f) *B* deleted: case 3a

Some exercises

1. Why don't we allow a minimum degree of $t = 1$?
2. For what values of t is the tree of Figure 19.1 a legal B-tree?
3. Show all legal B-trees of minimum degree 2 that represent {1, 2, 3, 4, 5}.
4. Derive a tight upper bound on the number of keys that can be stored in a B-tree of height h as a function of the minimum degree t.
5. Describe the data structure that would result if each black node in a red-black tree were to absorb its red children, incorporating their children with its own.
6. Show the results of inserting the keys
7. F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, Ein order into an empty B-tree. Only draw the configurations of the tree just before some node must split, and also draw the final configuration.
8. Explain under what circumstances, if any, redundant DISK-READ or DISK-WRITE operations are performed during the course of executing a call to B-TREE-INSERT. (A redundant DISK-READ is a DISK-READ for a page that is already in memory. A redundant DISK-WRITE writes to disk a page of information that is identical to what is already stored there.)
9. Explain how to find the minimum key stored in a B-tree and how to find the predecessor of a given key stored in a B-tree.
10. Suppose that the keys $\{1, 2, \ldots, n\}$ are inserted into an empty B-tree with minimum degree 2. How many nodes does the final B-tree have?
11. Since leaf nodes require no pointers to children, they could conceivably use a different (larger) t value than internal nodes for the same disk page size. Show how to modify the procedures for creating and inserting into a B-tree to handle this variation.
12. Suppose that B-TREE-SEARCH is implemented to use binary search rather than linear search within each node. Show that this makes the CPU time required $O(\lg n)$, independently of how t might be chosen as a function of n.
13. Suppose that disk hardware allows us to choose the size of a disk page arbitrarily, but that the time it takes to read the disk page is $a + bt$, where a and b are specified constants and t is the minimum degree for a B-tree using pages of the selected size. Describe how to choose t so as to minimize (approximately) the B-tree search time. Suggest an optimal value of t for the case in which $a = 30$ milliseconds and $b = 40$ microseconds.
14. Show the results of deleting *C, P*, and *V*, in order, from the tree of Figure (f).