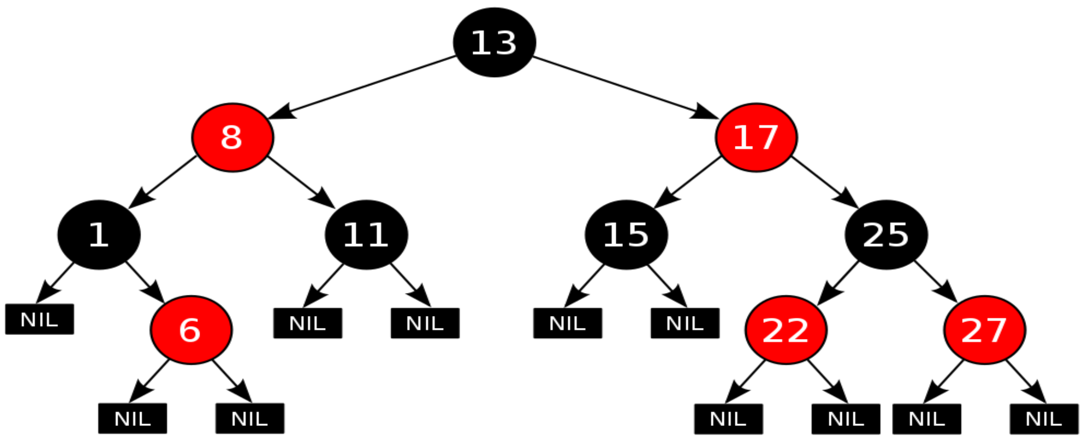
Unit 2

Red Black Trees

A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either RED or BLACK. By constraining the way nodes can be colored on any path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.

A binary search tree is a red-black tree if it satisfies the following red-black properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.



Black Height of RB Tree

The number of black nodes on any path from, but not including, a node x down to a leaf the black-height of the node, denoted bh(x). By property 5, the notion of black-height is well defined, since all descending paths from the node have the same number of black nodes.

We define the black-height of a red-black tree to be the black-height of its root.

Theorem

A red-black tree with n internal nodes has height at most 2 lg(n + 1).

Proof

We start by showing that "the subtree rooted at any node x contains at least **2bh(x) – 1"** internal nodes.

We prove this claim by induction on the height of **x**. If the height of x is 0, then x must be a leaf (nil[T]), and the subtree rooted at **x** indeed contains at least 2bh(x) - 1 = 2 0 - 1 = 0 internal nodes.

For the inductive step, consider a node x that has positive height and is an internal node with two children. Each child has a black-height of either bh(x) or bh(x) – 1, depending on whether its color is red or black, respectively. Since the height of a child of x is less than the height of x itself, we can apply the inductive hypothesis to conclude that each child has at least 2bh(x)-1 -1 internal nodes.

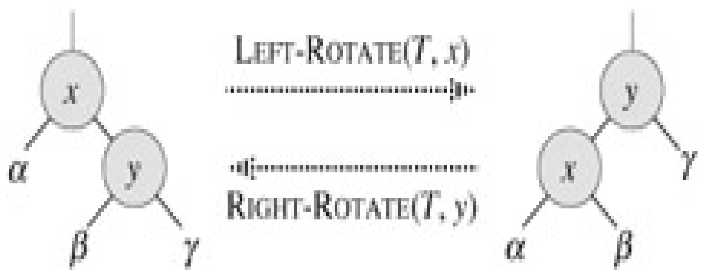
Thus, the subtree rooted at x contains at least (2 bh(x)-1 - 1) + (2 bh(x)-1 - 1) + 1 = 2 bh(x) - 1 internal nodes, which proves the claim.
To complete the proof of the lemma,

let h be the height of the tree. According to property 4, at least half the nodes on any simple path from the root to a leaf, not including the root, must be black. Consequently, the black-height of the root must be at least h/2; thus, n ≥ 2 h/2 - 1.

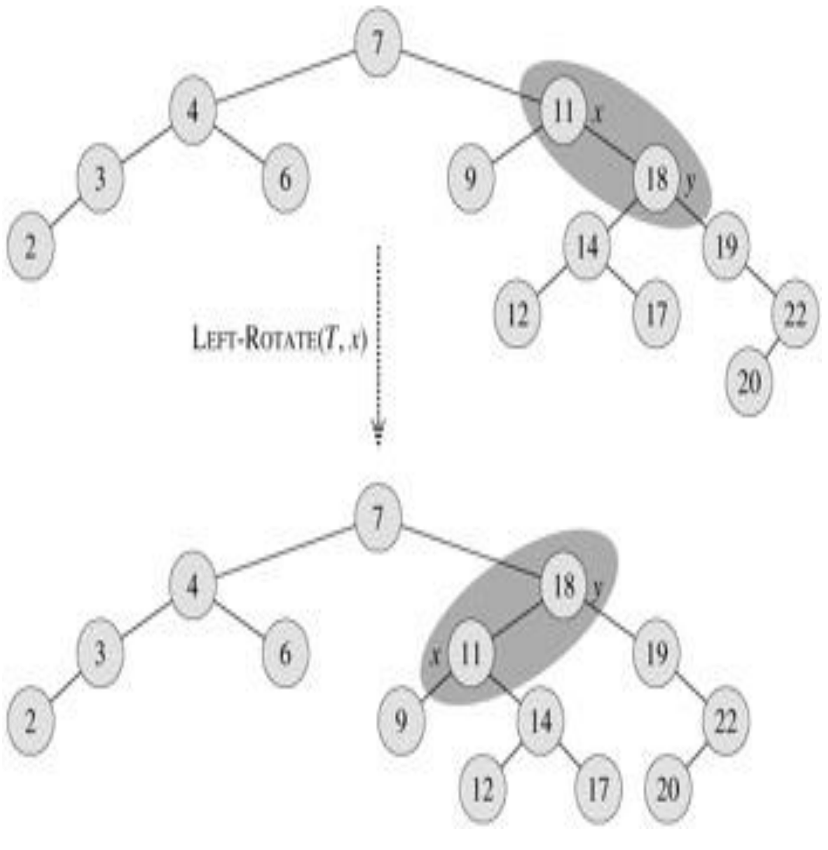
Moving the 1 to the left-hand side and taking logarithms on both sides yields lg(n + 1) ≥ h/2, or h ≤ 2 lg(n + 1).

Rotations

The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red-black tree with n keys, take O(lg n) time.



The rotation operations on a binary search tree. The operation LEFT- ROTATE(T, x) transforms the configuration of the two nodes on the left into the configuration on the right by changing a constant number of pointers. The configuration on the right can be transformed into the configuration on the left by the inverse operation RIGHT-ROTATE(T, y). The letters α, β, and γ represent arbitrary subtrees.



<u>Creation of Red Black Trees</u>

1. Similar as Binary Search Tree insertion that node z be the newly inserted node is coloured as **RED**.
2. Check the color of z's parent is RED or BLACK. If color of p[z] is black do nothing otherwise enter in the loop.
3. If p[z] is red, check p[z] is left of its parent (CASE A) or p[z] is right ot its parent (CASE B).

<u>Case A</u> If p[z] left of p[p[z]]. Calculate y as uncle of z.

Case 1: if color of y is red. Then

(i) Change color of parent of z (p[z]) and uncle y as BLACK.
(ii) Color of grand parent of z (p[p[z]]) as RED.
(iii) Change z = z’s grandparent,

```
if color[y] = RED

then { color[p[z]] ← BLACK

       color[y] ← BLACK

       color[p[p[z]]] ← RED

       z ← p[p[z]]          }
```

Case 2: If color of y is black and z is right of its parent p[z].

```
then   z ← p[z]

       LEFT-ROTATE(T, z)
```

Case 3: If color of y is black and z is left of its parent p[z].

```
then   color[p[z]] ← BLACK

       color[p[p[z]]] ← RED

       RIGHT-ROTATE(T, p[p[z]])
```

Case B : If p[z] right of p[p[z]]. calculate y as uncle of z.

Case 1: if color of y is red . Then

a) Change color of parent of z (p[z]) and uncle y as BLACK.

b) color of grand parent of z (p[p[z]]) as RED.

c) Change z = z's grandparent,

```
if color[y] = RED
then { color[p[z]] ← BLACK
         color[y] ← BLACK
        color[p[p[z]]] ← RED
         z ← p[p[z]]          }
```

Case 2: If color of y is black and z is left of its parent p[z].

```
then   z ← p[z]
       RIGHT-ROTATE(T, z)
```

Case 3: If color of y is black and z is RIGHT of its parent p[z].

```
then    color[p[z]] ← BLACK
        color[p[p[z]]] ← RED
        LEFT-ROTATE(T, p[p[z]])
```

4) At last color of root make black .

Note 1: If case 2 is applied then case 3 also applied.

<u>Algorithm of RB-INSERT-FIXUP</u>

RB-INSERT-FIXUP (T, z)

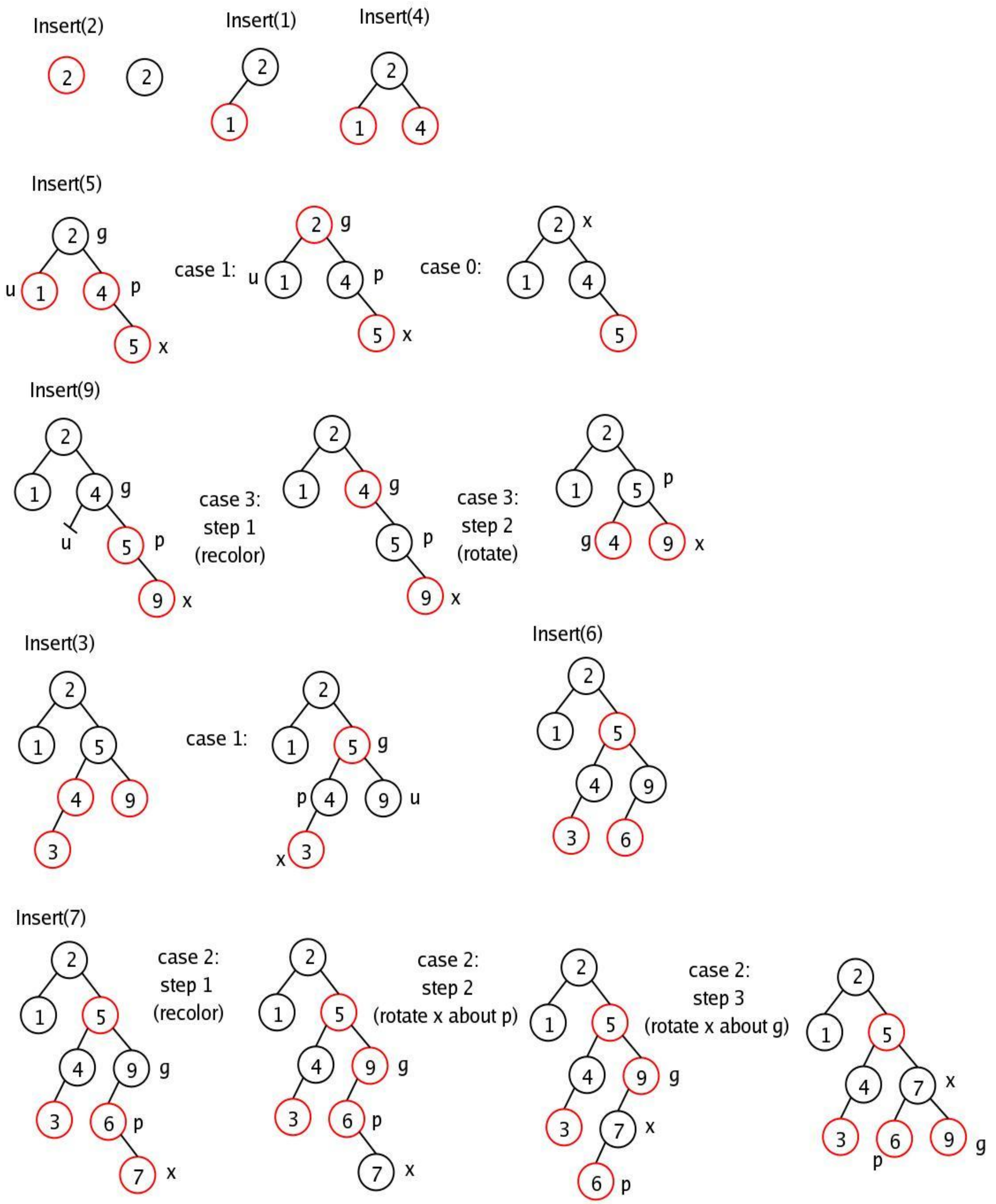
```
 1. while color [p[z]] = RED
 2. do   if p [z] = left [p[p[z]]]
 3.        then y ← right [p[p[z]]]
 4.             If color [y] = RED
 5.                then color [p[z]] ← BLACK    //Case 1
 6.                     color [y] ← BLACK         //Case 1
 7.                     color [p[z]] ← RED        //Case 1
 8.                     z  ← p[p[z]]              //Case 1
 9.             else if z= right [p[z]]
 10.                  then z ← p [z]             //Case 2
 11.                       LEFT-ROTATE (T, z)        //Case 2
 12.                  color [p[z]] ← BLACK       //Case 3
 13                   color [p [p[z]]] ← RED     //Case 3
 14.                  RIGHT-ROTATE  (T,p [p[z]])  //Case 3
```

```
15.     else (same as then clause) With "right" and "left" exchanged
16. color [root[T]] ← BLACK
```

Analysis :

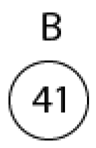
The height of a red-black tree on n nodes is O(lg n), lines 1–16 of RB-INSERT take O(lg n) time. In RB-INSERT- FIXUP, the while loop repeats only if case 1 is executed, and then the pointer z moves two levels up the tree. The total number of times the while loop can be executed is therefore O(lg n). Thus, RB-INSERT takes a total of O(lg n) time.

Example: Construct Red Black Trees by insert 2, 1, 4, 5, 9, 3, 6, 7 keys

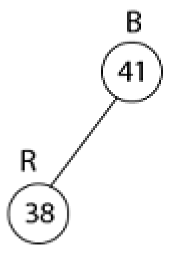


Example: Construct Red Black Trees by insert keys 41,38,31,12,19,8 into an initially empty red-black tree.

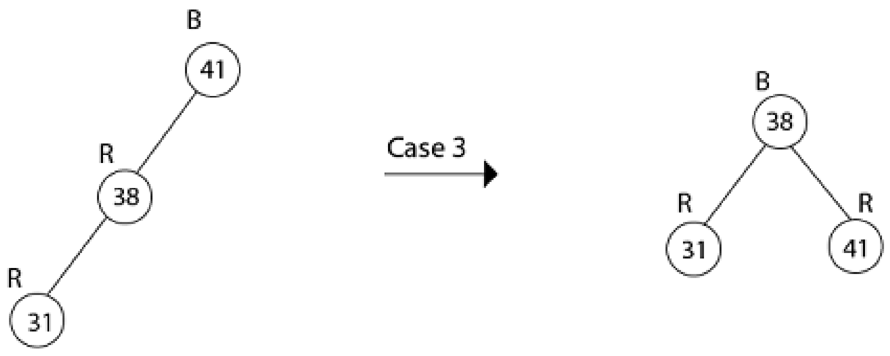
Insert 41



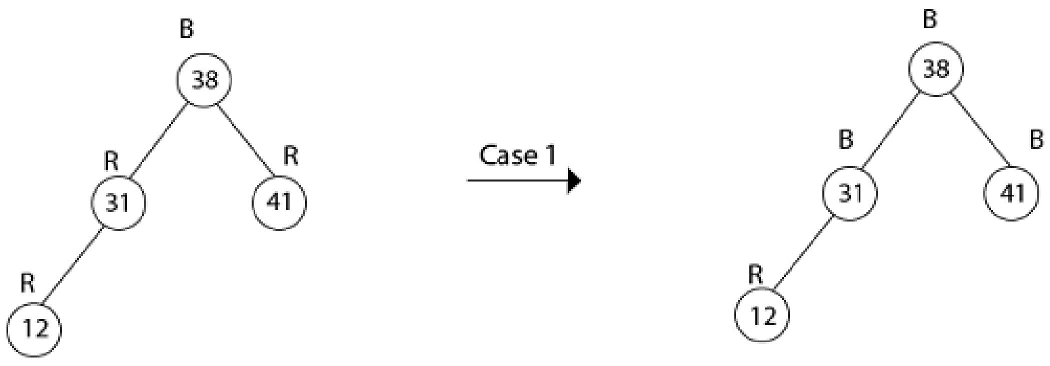
Insert 38



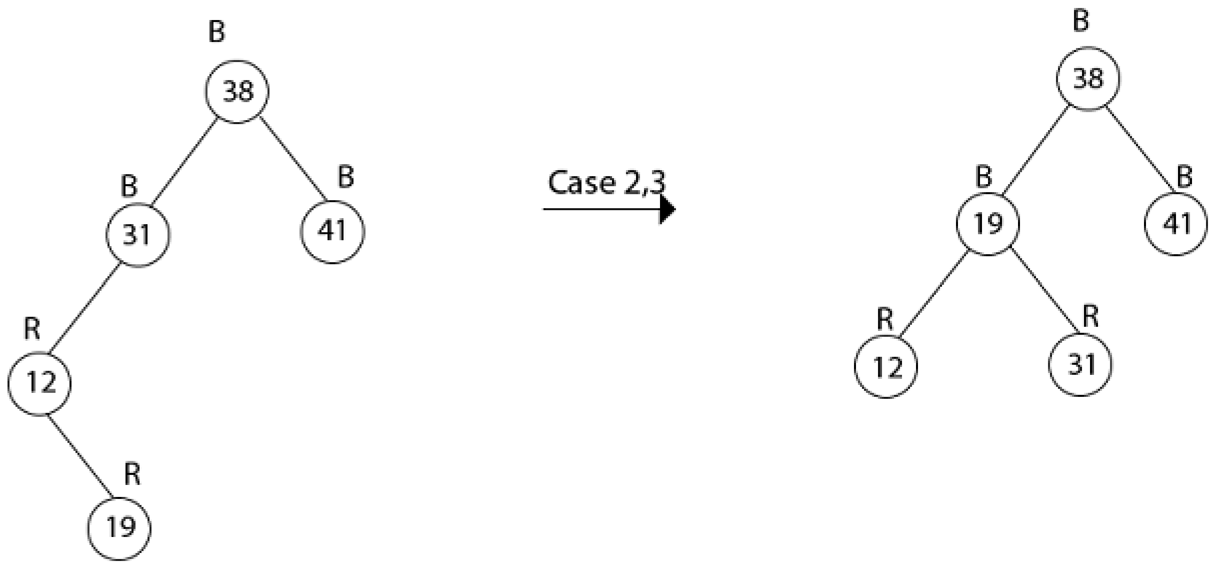
Insert 31

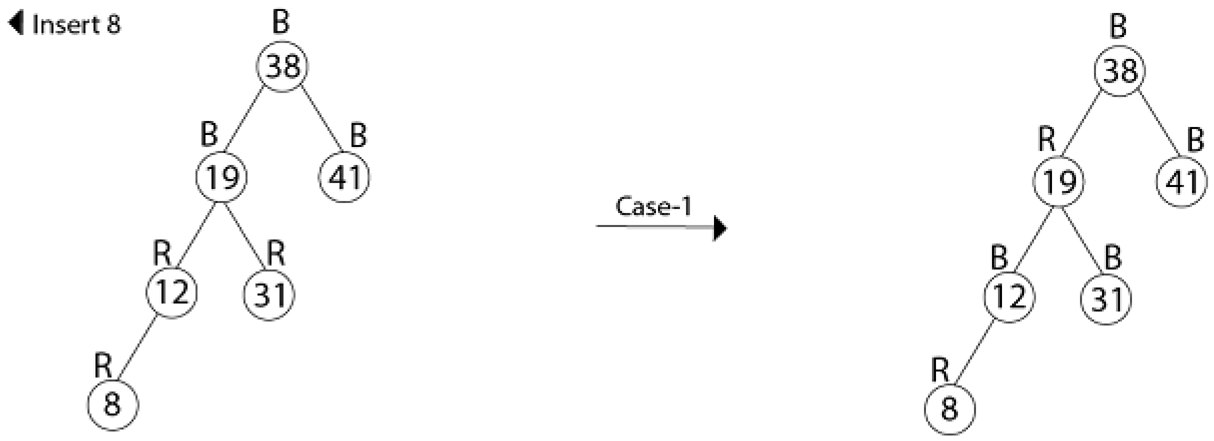


Insert 12

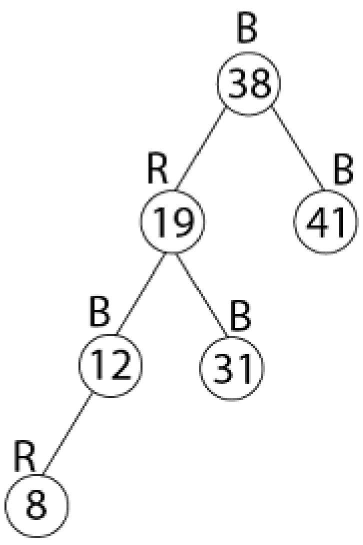


Insert 19





The final tree is



Deletion in RB Tree

First the property used in BST is used for deleting a node and then RB properties satisfied . Calculating w, y, z and x.

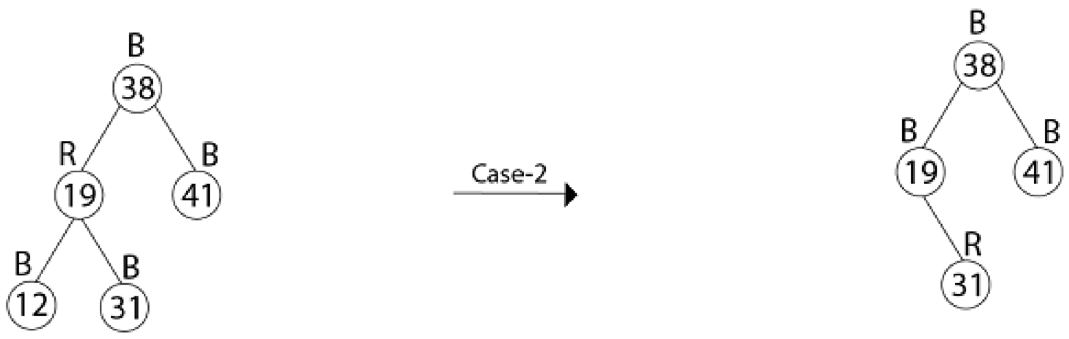
y=z(either left or right or both NIL)
y= z's successor (if both child are present)
x= left of y (if not present then right child of y)

Now for deleting if color [y]=red (no operation) otherwise call RB-DELETE-FIXUP
while(x ≠ root[T] and color[x] = BLACK) enter in loop.
There are 2 cases x is left or right of its parent. Calculate w as sibling of x.

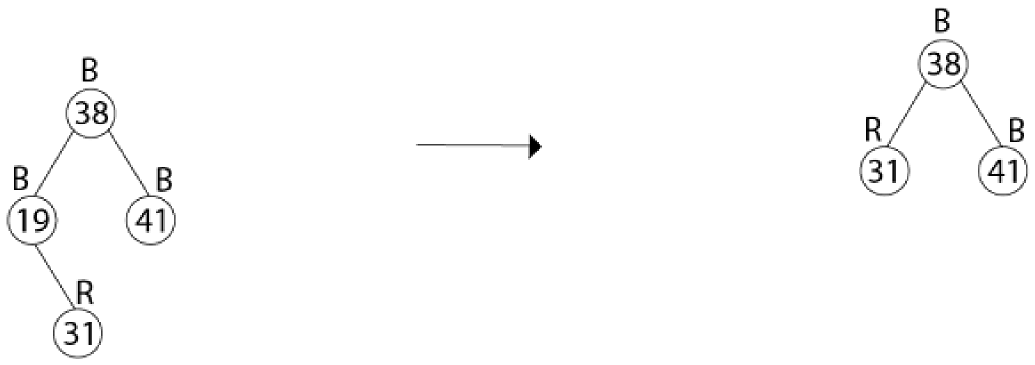
```
while( x ≠ root[T] and color[x] = BLACK) enter in loop.
 do    if x = left[p[x]]
       then w ← right[p[x]]
            if color[w] = RED
             then color[w] ← BLACK
                   color[p[x]] ← RED
                    LEFT-ROTATE(T, p[x])
                  w ← right[p[x]]
            if color[left[w]] = BLACK and color[right[w]] = BLACK
             then color[w] ← RED
                  x p[x]
                  else if color[right[w]] = BLACK ▹ Case 2
          then color[left[w]] ← BLACK
  color[w] ← RED
 RIGHT-ROTATE(T, w)
   w ← right[p[x]]

 color[w] ← color[p[x]] ▹ Case 4
 color[p[x]] ← BLACK ▹ Case 4
 color[right[w]] ← BLACK 20 LEFT-ROTATE(T, p[x]) 21 x ← root[T]
else (same as then clause with "right" and "left" exchanged)
 color[x] ← BLACK
```

Example :
In a previous example, we found that the red-black tree that results from successively inserting the keys 41,38,31,12,19,8 into an initially empty tree. Now show the red-black trees that result from the successful deletion of the keys in the order 8, 12, 19,31,38,41.

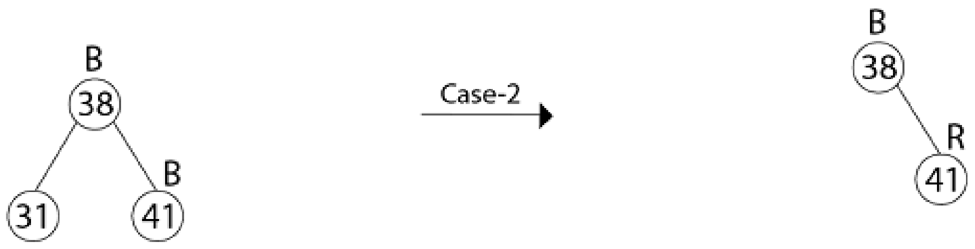
◀ Delete 12

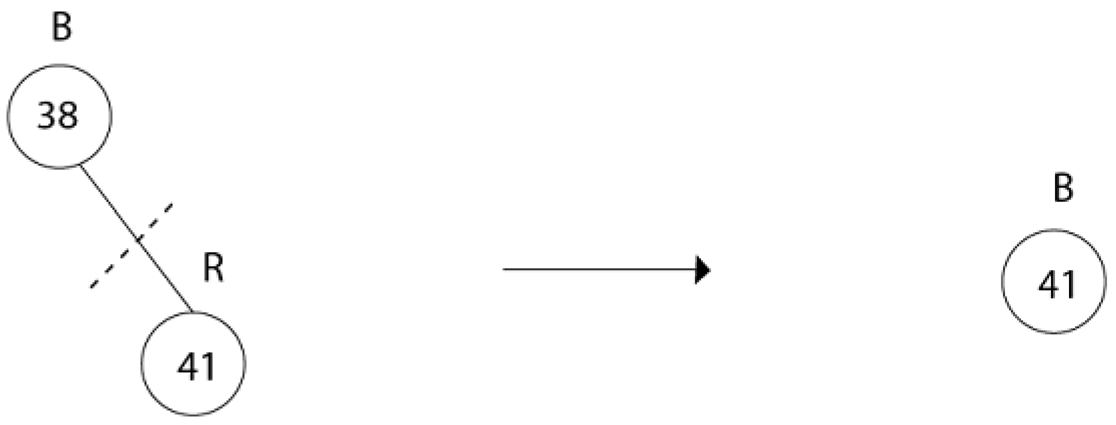


◀ Delete 19



◀ Delete 31





Delete 41   No Tree

B - Tree

B-trees are balanced search trees designed to work well on magnetic disks or other direct-access secondary storage devices.

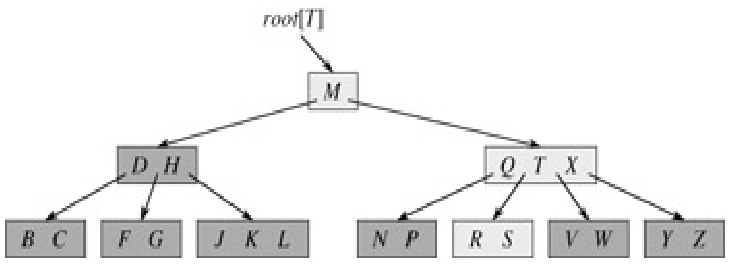
A B-tree T is a rooted tree (whose root is root[T]) having the following properties:

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 non decreasing order, so that key 1 [x] ≤key 2 [x] ≤ ··· ≤ key n[x] [x],
   c) leaf [x], a boolean value that is TRUE if x is a leaf and FALSE if x is an internal node.
2. Each internal node x also contains n[x]+ 1 pointers c 1 [x], c 2 [x], ..., 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 ≤ key 1 [x] ≤ k 2 ≤ key 2 [x] ≤··· ≤ key n[x] [x] ≤ k n[x]+1 .
4. All leaves have 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.
   b) 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.
   c) 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.



A B-tree whose keys are the consonants of english. An internal node x containing n[x] keys has n[x] + 1 children. All leaves are at the same depth in the tree. Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(log n).

Application of B tree

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.

Search: O(logtn) disk accesses. And O(t logtn) CPU time.

Create: O(1) disk accesses. And O(1) CPU time.

Insert and Delete: O(logtn) disk accesses. And O(tlogtn) CPU time.

Disk-Read: To move node from disk to memory.

Disk-Write: To move node from memory to disk.

Theorem : If $n \geq 1$, then for any n-key B-tree T of height h and minimum degree $t \geq 2$,

$$h \leq \log_t \frac{n+1}{2}$$

Proof :

Let T be of height h. The number of nodes is minimized when root has 1 key and all other nodes have t–1 keys.

If a B-tree has height h, the root contains at least one key and all other nodes contain at least **t - 1** keys. Thus, there are at least 2 nodes at depth 1, at least **2t** nodes at depth 2, at least $\mathbf{2t^2}$ nodes at depth 3, and so on, until at depth h there are at least $\mathbf{2t^{h-1}}$.

The minimum no. of keys in a B- Tree at height h

$= 1 + 2 (t-1) + 2t (t-1) + ... + 2 t^{h-1} (t-1)$

$= 1 + 2 (t-1) (1 + t + ... + t^{h-1})$

$= 1+ 2 (t-1))(t^{h} – 1)/(t - 1)$

$= 1+ 2 (t^{h} – 1)$

Since given n keys

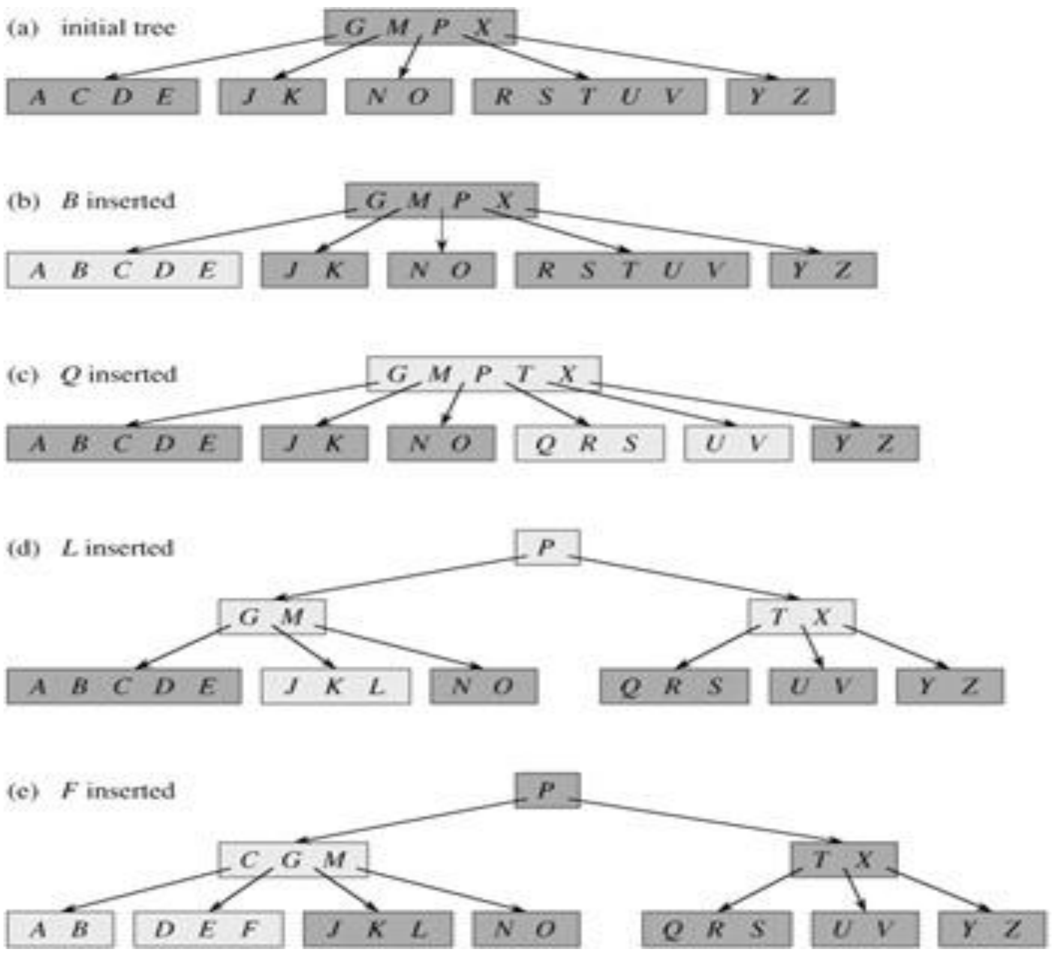
$n >= 1+ 2 (t^h – 1)$

$n +1 >= 2t^h$

$$h \leq \log_t \frac{n+1}{2}$$

<u>Creation of B Tree</u>

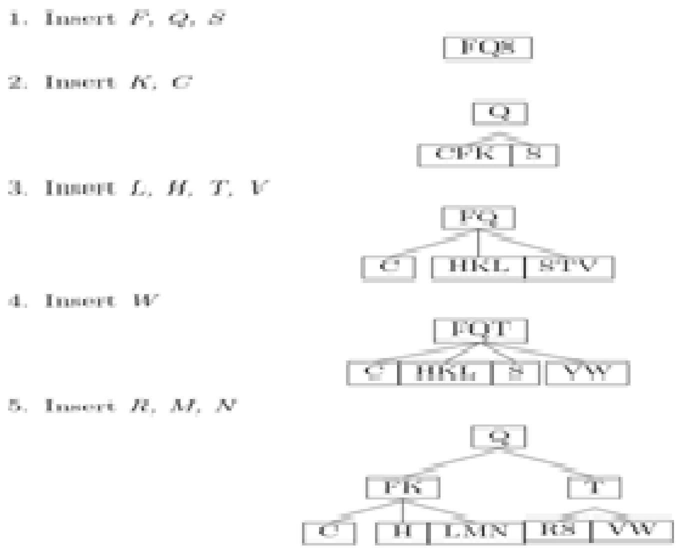
Rule: When root node is full then split if any key k insert in any leaf node. In other nodes if node has max keys and key k inert in that node must split first then insert.



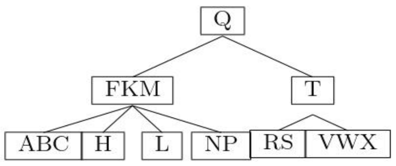
The minimum degree t for this B-tree is 3, so a node can hold at most 5 keys. Nodes that are modified by the insertion process are lightly shaded.

- **a)** The initial tree for this example.
- **b)** The result of inserting B into the initial tree; this is a simple insertion into a leaf node.
- **c)** The result of inserting Q into the previous tree. The node RSTUV is split into two nodes containing RS and UV, the key T is moved up to the root, and Q is inserted in the leftmost of the two halves (the RS node).
- **d)** The result of inserting L into the previous tree. The root is split right away, since it is full, and the B-tree grows in height by one. Then L is inserted into the leaf containing JK.
- **e)** The result of inserting F into the previous tree. The node ABCDE is split before F is inserted into the rightmost of the two halves (the DE node).

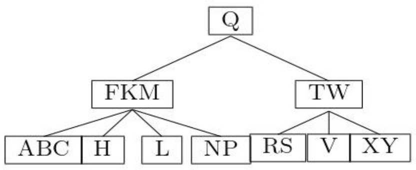
Example : Show the results of inserting the keys F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E in order to an empty B-tree with minimum degree 2.



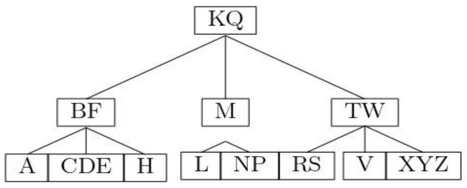
6. Insert *P, A, B, X*



7. Insert *Y*



8. Insert *D, Z, E* **FINAL CONFIGURATION**



<u>Deletion in B tree</u>

<u>Case 0:</u> Empty root -- make root’s only child the new root.
<u>Case 1:</u> k in x, x is a leaf -- delete k from x.
<u>Case 2:</u> k in x, x internal

<u>Subcase A:</u> 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.

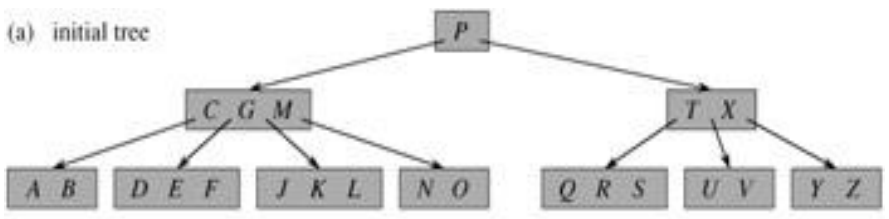
<u>Subcase B:</u> z has at least t keys -- find successor k´ of k in subtree rooted at z, recursively delete k´, replace k by k´ in x.

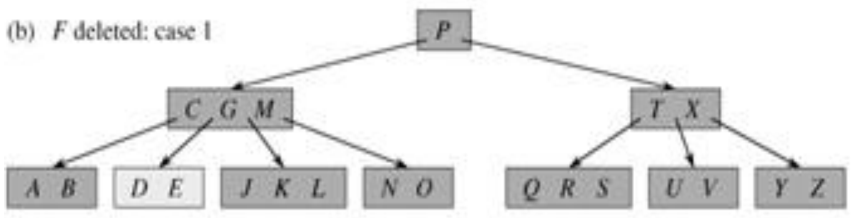
<u>Subcase C:</u> y and z both have t–1 keys -- merge k and z into y, free z, recursively delete k from y.

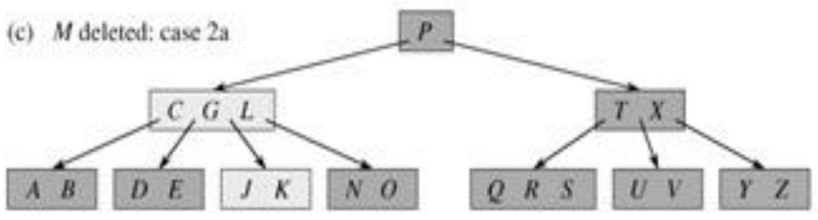
<u>Case 3:</u> k not in internal node. Let ci[x] be the root of the subtree that must contain k, if k is in the tree. If ci[x] has at least t keys, then recursively descend; otherwise, execute 3.A and 3.B as necessary.

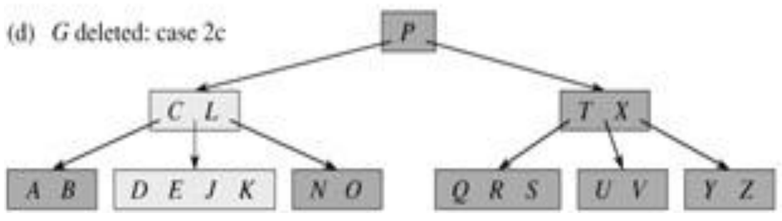
<u>Subcase A:</u> ci[x] has t–1 keys, some sibling has at least t keys.

<u>Subcase B:</u> ci[x] and sibling both have t–1 keys.

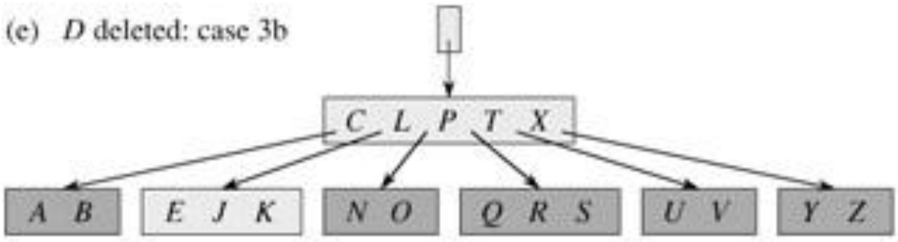

(a) initial tree


(b) *F* deleted: case 1

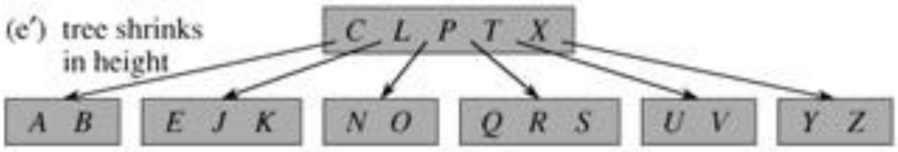

(c) *M* deleted: case 2a


(d) *G* deleted: case 2c

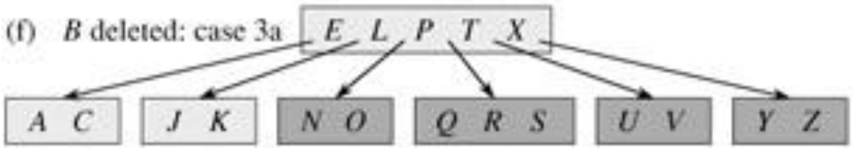
(e) *D* deleted: case 3b



(e′) tree shrinks in height



(f) *B* deleted: case 3a

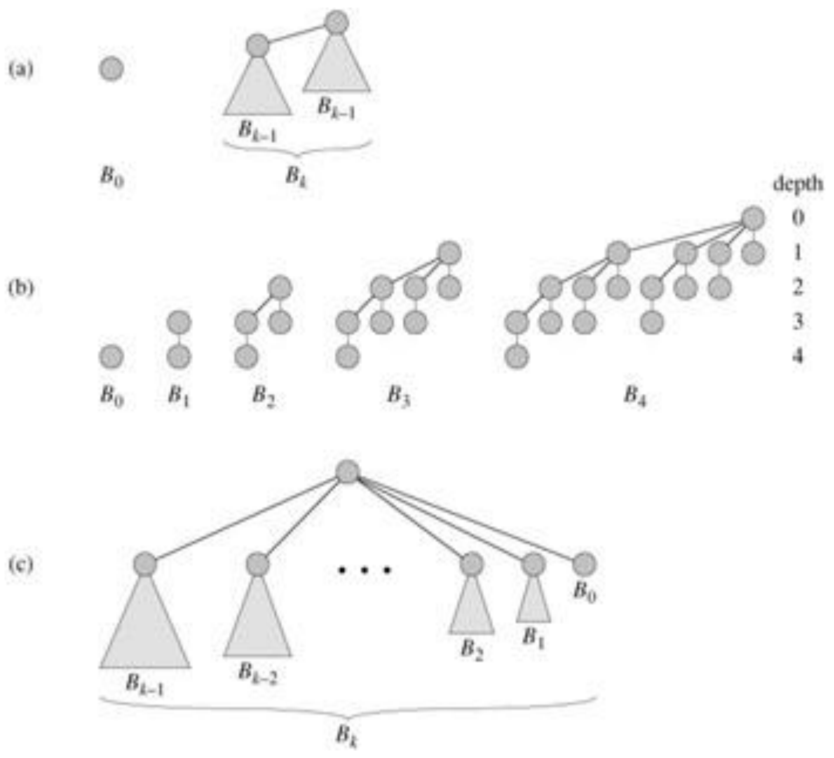


Binomial Heaps
A binomial heap is a collection of binomial trees, so this section starts by defining binomial trees and proving some key properties.

Binomial trees
The binomial tree B k is an ordered tree defined recursively, the binomial tree B_0 consists of a single node. The binomial tree B_k consists of
two binomial trees B_{k-1} that are linked together.

$B_k = B_{k-1} + B_{k-1}$

The root of one is the leftmost child of the root of the other.



(a) The recursive definition of the binomial tree B k . Triangles represent rooted subtrees.
(b) The binomial trees B_0 - B_4. Node depths in B 4 are shown.
(c) Another way of looking at the binomial tree B k .

Properties of binomial trees
For the binomial tree B_k ,

1. there are 2^k nodes,
2. the height of the tree is k,
3. there are exactly ${}^kC_i$ nodes at depth i for i = 0, 1, ..., k,
4. the root has degree k.

Proof The proof is by induction on k.
For each property, the basis is the binomial tree B_0 .
Verifying that each property holds for B_0 is trivial.
For the inductive step, we assume that the lemma holds for B_{k-1} .

1. Binomial tree B_k consists of two trees of B_{k-1}, and B_k has $2^{k-1} + 2^{k-1} = 2^k$ nodes.
2. Because of the way in which the two copies of B_{k-1} are linked to form B_k , the maximum depth of a node in B_k is one greater than the maximum depth in B k-1 .

By the inductive hypothesis, this maximum depth is (k - 1) + 1 = k.

3. Let D(k, i) be the number of nodes at depth i of binomial tree B_k . Since B_k is composed of two copies of B_{k-1} linked together, a node at depth i in B_{k-1} appears in B_k once at depth i and once at depth i + 1.

In other words, the number of nodes at depth i in B k is the number of nodes at depth i in B_{k-1} plus the number of nodes at depth i - 1 in B_{k-1}.
Thus,

$$\begin{aligned} D(k,i) &= D(k-1,i) + D(k-1,i-1) && \text{(by the inductive hypothesis)} \\ &= \binom{k-1}{i} + \binom{k-1}{i-1} && \text{(by Exercise C.1-7)} \\ &= \binom{k}{i} . \end{aligned}$$

4. The only node with greater degree in $\mathbf{B_k}$ than in $\mathbf{B_{k-1}}$ is the root, which has one more child than in $\mathbf{B_{k-1}}$. Since the root of $\mathbf{B_{k-1}}$ has degree k - 1, the root of $\mathbf{B_k}$ has degree k.

Theorem

The maximum degree of any node in an n-node binomial tree is lg n.

Proof

According to the property 4 of the binomial tree , the maximum degree will be of the root node . Let the maximum degree is k.

Hence the degree of root node becomes k, so that B_k exist.
According to property 1 , then total number of nodes in binomial tree is 2^k
Since , we are given no. of nodes is n

i.e. $n= 2^k$
k= log n

Binomial heaps

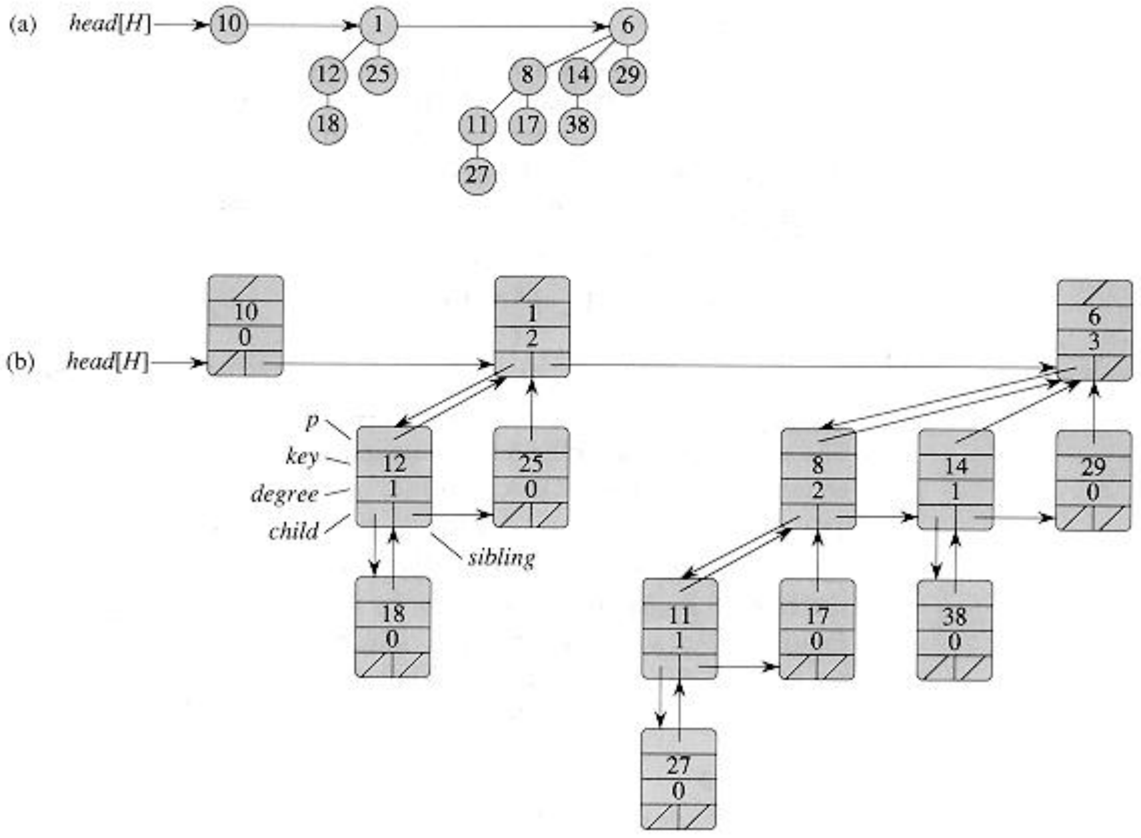
A binomial heap H is a set of binomial trees that satisfies the following binomial-heap properties.

1. Each binomial tree in H obeys the min-heap property: the key of a node is greater than or equal to the key of its parent. We say that each such tree is min-heap-ordered.
2. For any nonnegative integer k, there is at most one binomial tree in H whose root has degree k.

The first property tells us that the root of a min-heap-ordered tree contains the smallest key in the tree.

The second property implies that an n-node binomial heap H consists of at most logn + 1.

A binomial heap H with 13 nodes. H consists of min-heap-ordered binomial trees B_3 , B_2 , and B_0 , having 8, 4, and 1 nodes respectively, for a total of 13 nodes.



Operations on binomial heaps

1. Creating a new binomial heap
To make an empty binomial heap, the MAKE-BINOMIAL-HEAP procedure simply allocates and returns an object H , where head[H] = NIL. The running time is Θ(1).

2. Finding the minimum key

The procedure BINOMIAL-HEAP-MINIMUM returns a pointer to the node with the minimum key in an n-node binomial heap H. This implementation assumes that there are no keys with value ∞. BINOMIAL-HEAP-MINIMUM returns a pointer to the node in heap H whose key is minimum.

```
BINOMIAL-HEAP-MINIMUM(H)
1 y ← NIL
2 x ← head[H]
3 min ← ∞
4 while x ≠ NIL
5  do if key[x] < min
6        then  min ← key[x]
7                 y ← x
8        x ← sibling[x]
9 return y
```

The running time of BINOMIAL-HEAP- MINIMUM is O(lg n)

3. Uniting two binomial heaps

The BINOMIAL-HEAP-UNION procedure has two phases. The first phase, performed by the call of BINOMIAL-HEAP-MERGE, merges the root lists of binomial heaps H 1 and H 2 into a single linked list H that is sorted by degree into monotonically increasing order.

Union(H1 , H2)creates and returns a new heap that contains all the nodes of heaps H 1 and H 2 into a single linked list H.

BINOMIAL-LINK

Procedure links the *Bk-1* tree rooted at node *y* to the *Bk-1* tree rooted at node z it makes z the parent of *y*

i.e. Node *z* becomes the root of a B_k tree

We maintain 3 pointers into the root list

x = points to the root currently being examined

prev-*x* = points to the root PRECEDING *x* on the root list sibling [prev-*x*] = *x*

next-*x* = points to the root FOLLOWING *x* on the root list sibling [*x*] = next-*x*

CASE 1: Occurs when degree [*x*] $\neq$ degree [next-*x*]
then move the pointers.

CASE 2: Occurs when x is the first of 3 roots of equal degree

degree [*x*] = degree [next-*x*] = degree [sibling[next-*x*]]

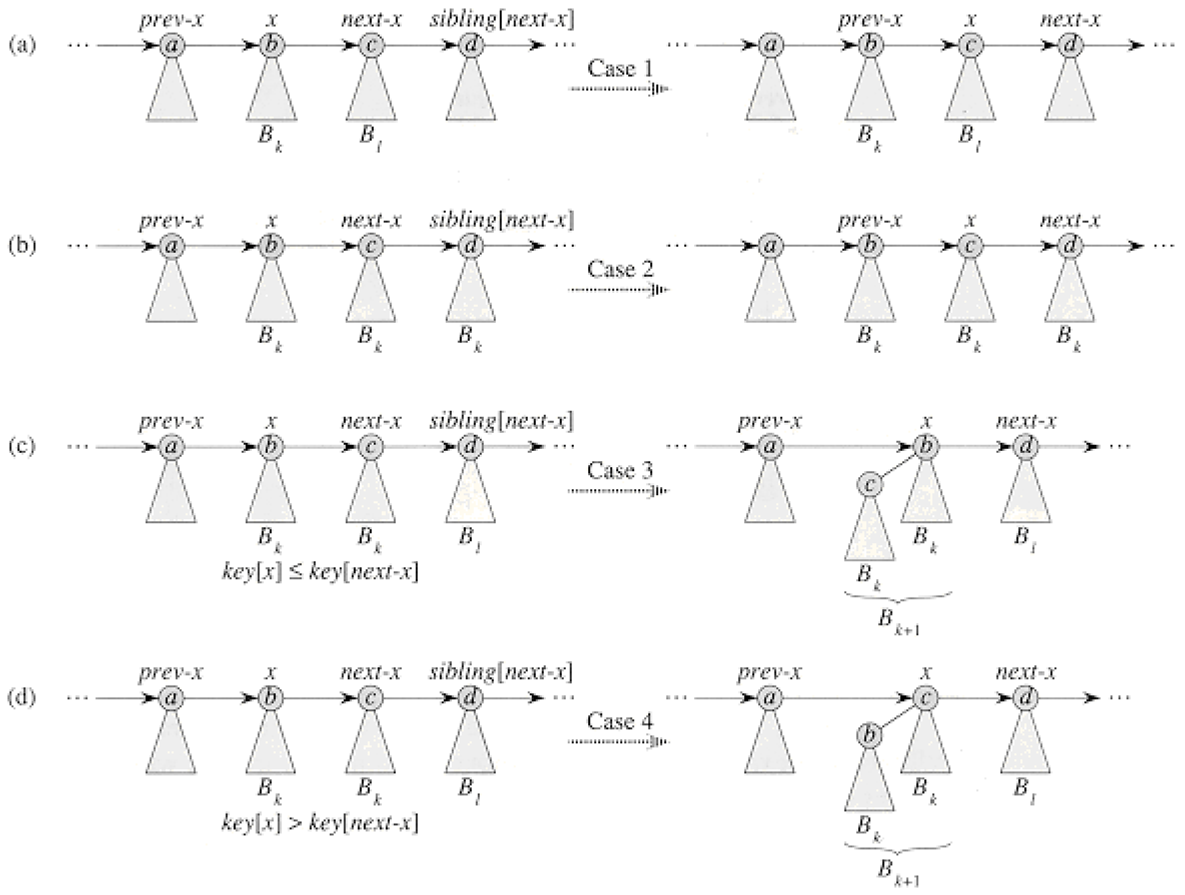
then move the pointers.

CASE 3 & 4: Occur when *x* is the first of 2 roots of equal degree

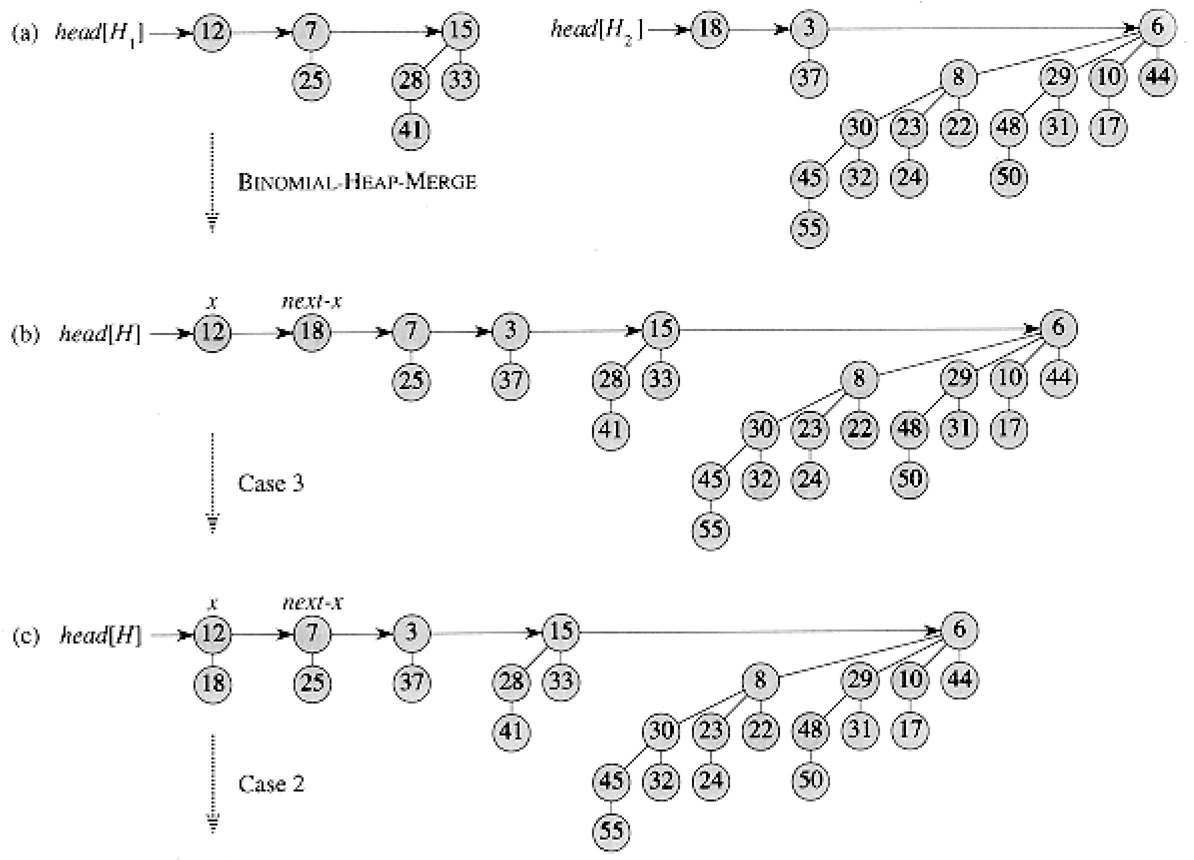
degree [*x*] = degree [next-*x*] $\neq$ degree [sibling [next-*x*]]

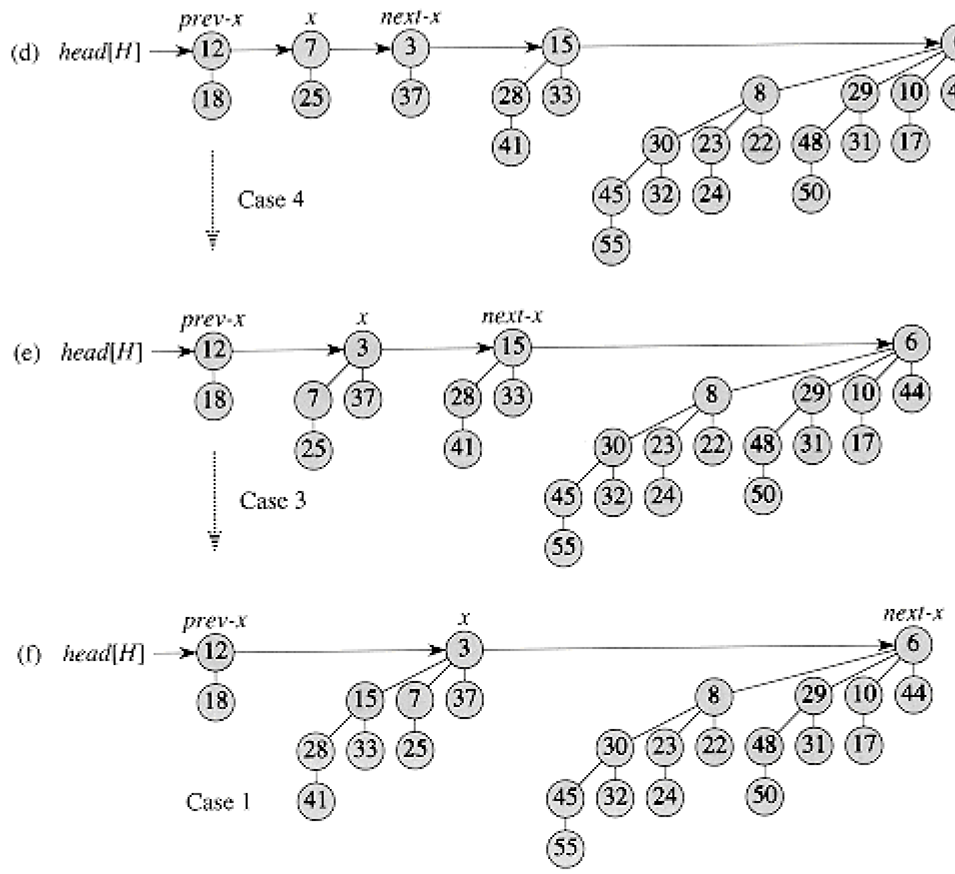
Note : Repeat these cases until next-x ≠ NIL

then The root with the smaller key becomes the root of the linked tree

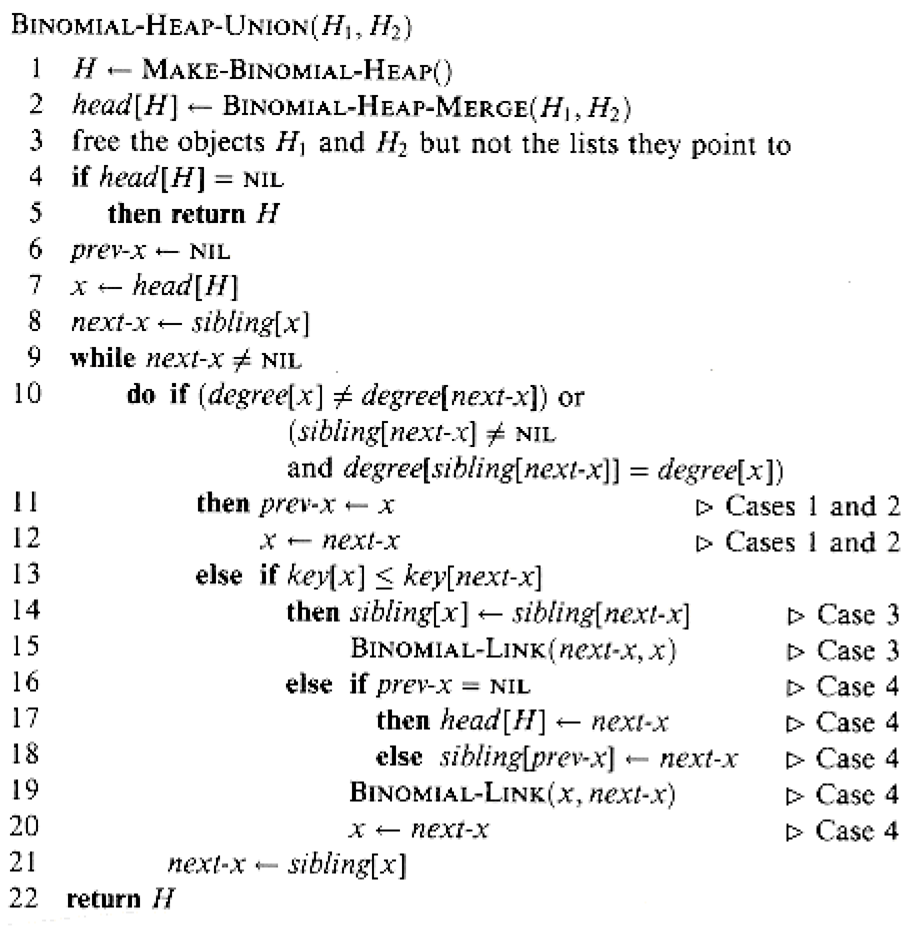


Example: Union of two Binomial Heaps


(a) head[H₁]
head[H₂]
BINOMIAL-HEAP-MERGE
(b) head[H]
x
next-x
Case 3
(c) head[H]
x
next-x
Case 2


(d) head[H]
prev-x
x
next-x
Case 4
(e) head[H]
prev-x
x
next-x
Case 3
(f) head[H]
prev-x
x
next-x
Case 1

Algorithm of Union of two Binomial Heaps



```
BINOMIAL-HEAP-UNION($H_1, H_2$)
1  $H \leftarrow$ MAKE-BINOMIAL-HEAP()
2  $head[H] \leftarrow$ BINOMIAL-HEAP-MERGE($H_1, H_2$)
3  free the objects $H_1$ and $H_2$ but not the lists they point to
4  if $head[H] =$ NIL
5     then return $H$
6  $prev\text{-}x \leftarrow$ NIL
7  $x \leftarrow head[H]$
8  $next\text{-}x \leftarrow sibling[x]$
9  while $next\text{-}x \neq$ NIL
10      do if ($degree[x] \neq degree[next\text{-}x]$) or
                  ($sibling[next\text{-}x] \neq$ NIL
                  and $degree[sibling[next\text{-}x]] = degree[x]$)
11           then $prev\text{-}x \leftarrow x$                    ▷ Cases 1 and 2
12                $x \leftarrow next\text{-}x$                    ▷ Cases 1 and 2
13           else if $key[x] \leq key[next\text{-}x]$
14                   then $sibling[x] \leftarrow sibling[next\text{-}x]$      ▷ Case 3
15                        BINOMIAL-LINK($next\text{-}x, x$)           ▷ Case 3
16                   else if $prev\text{-}x =$ NIL                ▷ Case 4
17                           then $head[H] \leftarrow next\text{-}x$        ▷ Case 4
18                           else $sibling[prev\text{-}x] \leftarrow next\text{-}x$   ▷ Case 4
19                        BINOMIAL-LINK($x, next\text{-}x$)           ▷ Case 4
20                        $x \leftarrow next\text{-}x$                 ▷ Case 4
21         $next\text{-}x \leftarrow sibling[x]$
22  return $H$
```

4. Inserting a node

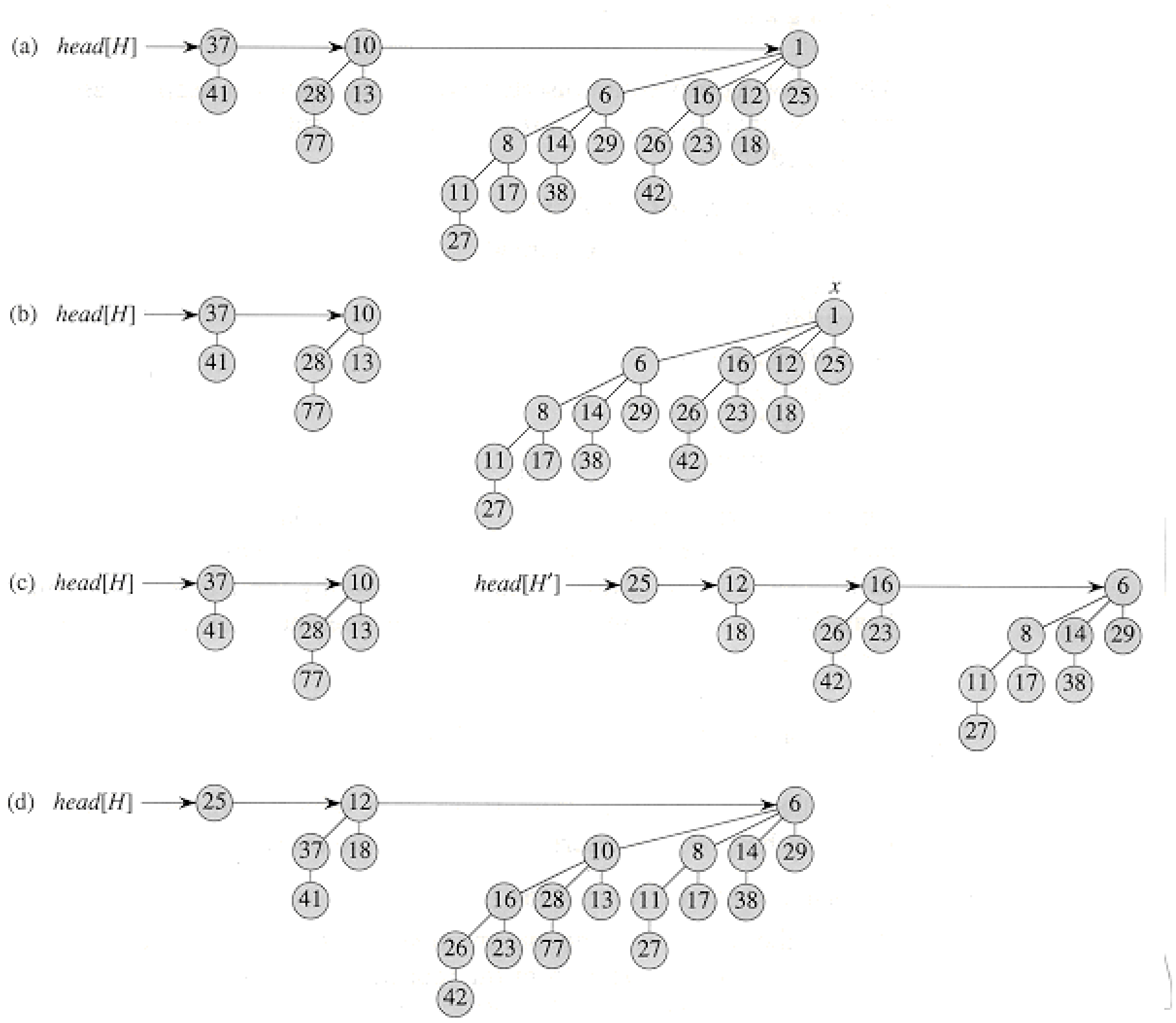
The following procedure inserts node x into binomial heap H , assuming that x has already been allocated and key[x] has already been filled in.

```
BINOMIAL-HEAP-INSERT(H, x)
1 H′ ← MAKE-BINOMIAL-HEAP()
2 p[x] ← NIL
3 child[x] ← NIL
4 sibling[x] ← NIL
5 degree[x] ← 0
6 head[H′] ← x
7 H ← BINOMIAL-HEAP-UNION(H, H′)
```

5. Extracting the node with minimum key

The following procedure extracts the node with the minimum key from binomial heap H and returns a pointer to the extracted node.

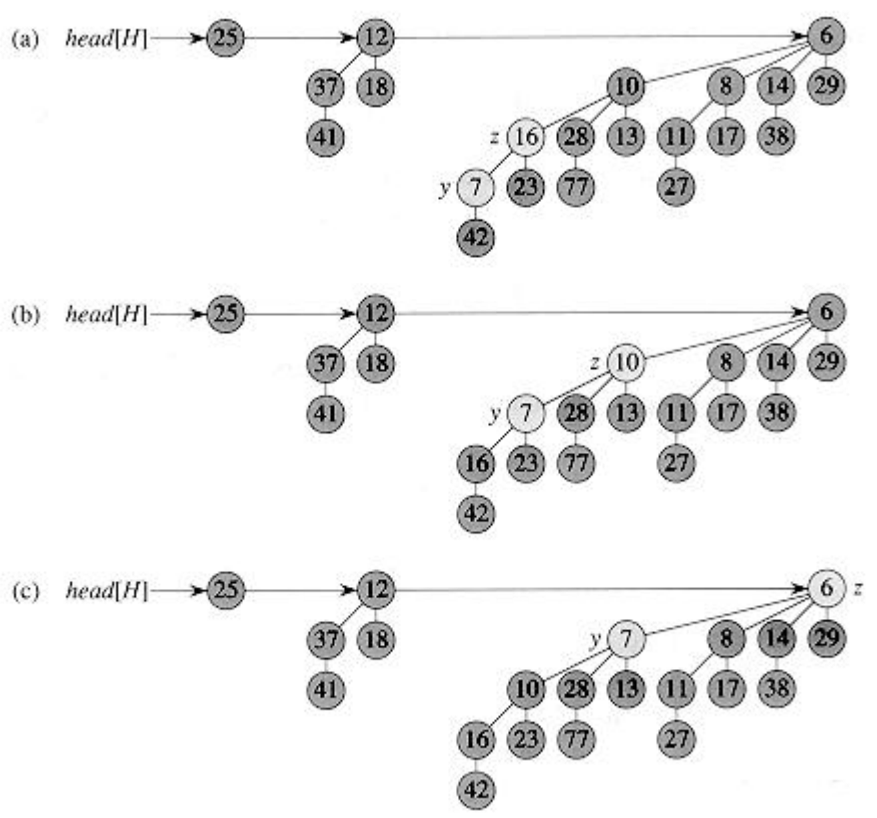
```
BINOMIAL-HEAP-EXTRACT-MIN(H)
1 find the root x with the minimum key in the root list of H, and remove x from the root list of H
2 H′ ← MAKE-BINOMIAL-HEAP()
3 reverse the order of the linked list of x's children, and set head[H′] to point to the head of the resulting list
4 H ← BINOMIAL-HEAP-UNION(H, H′)
5 return x
```



6. Decreasing a key

The following procedure decreases the key of a node x in a binomial heap H to a new value k. It signals an error if k is greater than x's current key.

```
BINOMIAL-HEAP-DECREASE-KEY(H, x, k)
1 if k > key[x]
2then error "new key is greater than current key"
3 key[x] ← k
4 y ← x
5 z ← p[y]
6 while z ≠ NIL and key[y] < key[z]
7 do exchange key[y] ↔ key[z]
8       y ← z
9       z ← p[y]
```



7. Deleting a key

It is easy to delete a node x's key and satellite information from binomial heap H in O(lg n) time. The following implementation assumes that no node currently in the binomial heap has a key of -∞.

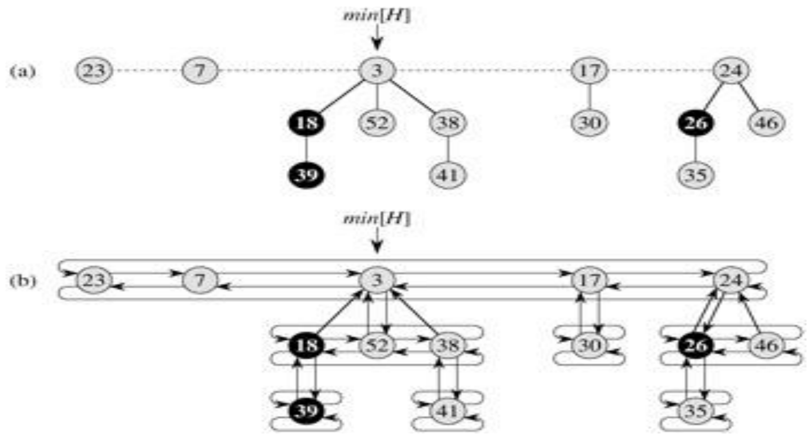
BINOMIAL-HEAP-DELETE(H, x)

```
1 BINOMIAL-HEAP-DECREASE-KEY(H, x, -∞)
2 BINOMIAL-HEAP-EXTRACT-MIN(H)
```

Fibonacci Heaps

Structure of Fibonacci heaps

Like a binomial heap, a Fibonacci heap is a collection of min-heap-ordered trees. The trees in a Fibonacci heap are not constrained to be binomial trees, example of a Fibonacci heap.



A Fibonacci heap consisting of five min-heap-ordered trees and 14 nodes.
The dashed line indicates the root list. The minimum node of the heap is the node containing the key 3. The three marked nodes are blackened.

The potential of this particular Fibonacci heap is 5+2*3 = 11.

Unlike trees within binomial heaps, which are ordered, trees within Fibonacci heaps are rooted but unordered. The children of x are linked together in
a circular, doubly linked list, which we call the child list of x.
Circular, doubly linked lists have two advantages for use in Fibonacci heaps. First, we can remove a node from a circular, doubly linked list in O(1) time. Second, given two such lists, we can concatenate them into one circular, doubly linked list in O(1) time.
The boolean-valued field mark[x] indicates whether node x has lost a child since the last time x was made the child of another node.

A given Fibonacci heap H is accessed by a pointer min[H] to the root of a tree containing a minimum key; this node is called the minimum node of the Fibonacci heap. If a Fibonacci heap H is empty, then min[H] = NIL.

Potential Function

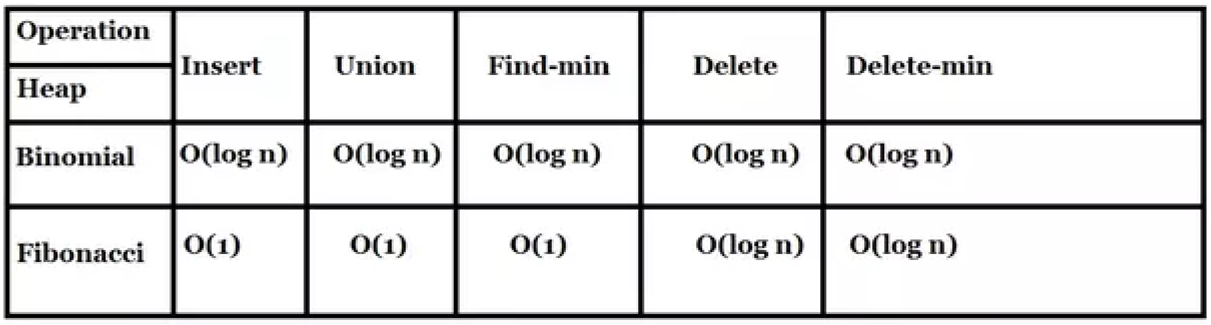
to analyze the performance of Fibonacci heap operations. For a given Fibonacci heap H, we indicate by t(H) the number of trees in the root list of H and by m(H) the number of marked nodes in H. The potential of Fibonacci heap H is then defined by

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

Maximum degree

The amortized analyses we shall perform in the remaining sections of this chapter 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).

Difference between binomial heap and fibonacci heap



Operation / Heap	Insert	Union	Find-min	Delete	Delete-min
Binomial	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
Fibonacci	O(1)	O(1)	O(1)	O(log n)	O(log n)

Binomial heaps use a singly linked circular link list and Fibonacci heaps use a doubly linked circular linked list.

- Every Binomial heap is a Fibonacci heap but every Fibonacci heap isn't Binomial heap.
- Binomial heap use head[H] pointer while fibonacci heap use min [H] pointer.
- Binomial heap is ordered while fibonacci heap is unordered.

- Fibonacci heaps is uses amortized cost for analysis but binomial heap does not uses.
- Data members in a node for Binomial heaps are data, link, degree and child. Whereas in Fibonacci heaps it is data, parent, mark, child, link and degree.

Mergeable-heap operations

We describe and analyze the mergeable-heap operations as implemented for Fibonacci heaps. If only these operations-MAKE-HEAP, INSERT, MINIMUM, EXTRACT-MIN, and UNION are to be supported, each Fibonacci heap is simply a collection of "unordered" binomial trees. An unordered binomial tree is like a binomial tree, and it, too, is defined recursively. The unordered binomial tree U 0 consists of a single node, and an unordered binomial tree U k consists of two unordered binomial trees U k-1 for which the root of one is made into any child of the root of the other.

For the unordered binomial tree U_k , 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 , . . . , U_{k-1} in some order.
Thus, if an n-node Fibonacci heap is a collection of unordered binomial trees, then D(n) = lg n.

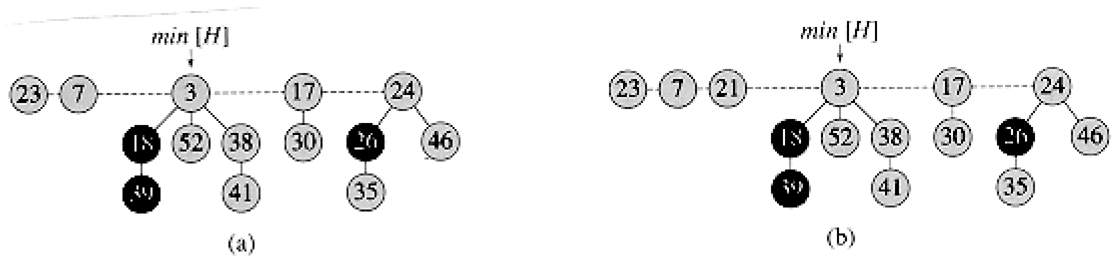
1.Creating a new Fibonacci heap

To make an empty Fibonacci heap, the MAKE-FIB-HEAP procedure allocates and returns the Fibonacci heap object H , where n[H] = 0 and min[H] = NIL; there are no trees in H . Because t(H) = 0 and m(H) = 0, the potential of the empty Fibonacci heap is Φ(H) = 0. The amortized cost of MAKE-FIB-HEAP is thus equal to its O(1) actual cost.

2.Inserting a node

The following procedure inserts node x into Fibonacci heap H , assuming that the node has already been allocated and that key[x] has already been filled in.

```
FIB-HEAP-INSERT(H, x)
1 degree[x] ← 0
2 p[x] ← NIL
3 child[x] ← NIL
4 left[x] ← x
5 right[x] ← x
6 mark[x] ← FALSE
7 concatenate the root list containing x with root list H
8 if min[H] = NIL or key[x] < key[min[H]]
9 then min[H] ← x
10 n[H] ← n[H] + 1
```



Fibonacci heap H after the node with key 21 has been inserted. The node becomes its own heap-ordered tree and is then added to the root list, becoming the left sibling of the root.

To determine the amortized cost of FIB-HEAP-INSERT, let H be the input Fibonacci heap and H' be the resulting Fibonacci heap. Then, $t(H') = t(H) + 1$ and $m(H') = m(H)$, and the increase in potential is

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

Since the actual cost is $O(1)$, the amortized cost is $O(1) + 1 = O(1)$.

3. Finding the minimum node

The minimum node of a Fibonacci heap H is given by the pointer min[H], so we can find the minimum node in O(1) actual time. Because the potential of H does not change, the amortized cost of this operation is equal to its O(1) actual cost.

4.Uniting two Fibonacci heaps

The following procedure unites Fibonacci heaps H 1 and H 2 , destroying H 1 and H 2 in the process. It simply concatenates the root lists of H 1 and H 2 and then determines the new minimum node.

```
FIB-HEAP-UNION(H 1 , H 2 )
1 H ← MAKE-FIB-HEAP()
2 min[H] ← min[H 1 ]
3 concatenate the root list of H 2 with the root list of H
4 if (min[H 1 ] = NIL) or (min[H 2 ] ≠ NIL and min[H 2 ] < min[H 1 ])
5   then min[H] ← min[H 2 ]
6  n[H] ← n[H 1 ] + n[H 2 ]
7 free the objects H 1 and H 2
8 return H
```

The change in potential is

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

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

$= 0,$

because $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$. The amortized cost of FIB-HEAP-UNION is therefore equal to its O(1) actual cost.

5.Extracting the minimum node

The process of extracting the minimum node is the most complicated of the operations presented in this section.

```
FIB-HEAP-EXTRACT-MIN(H)
1 z ← min[H]
2 if z ≠ NIL
3 then for each child x of z
4do add x to the root list of H
5p[x] ← NIL
6remove z from the root list of H
```

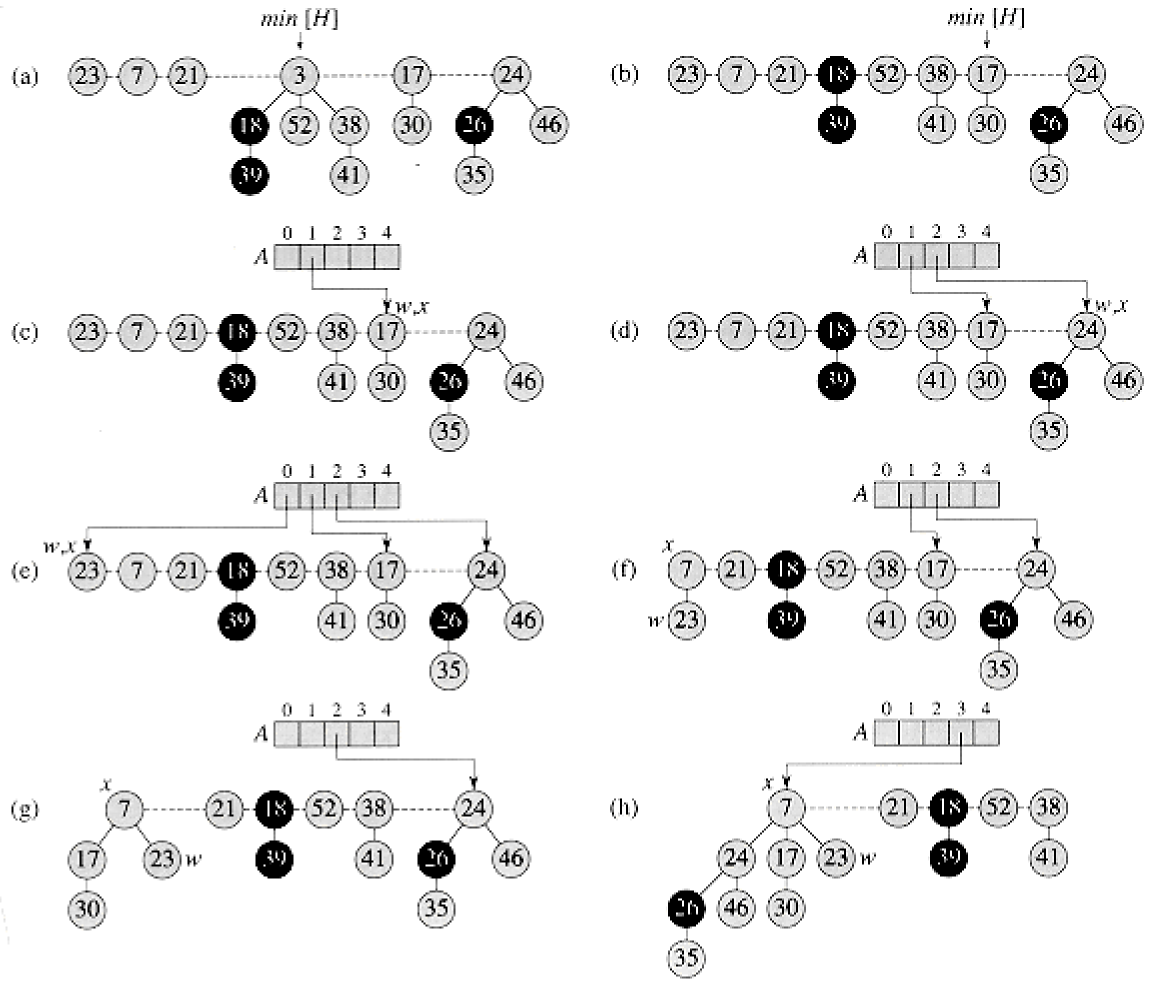
```
7        if z = right[z]
8         then min[H] ← NIL
9         else min[H] ← right[z]
10              CONSOLIDATE(H)
11     n[H] ← n[H] - 1
12 return z
```

CONSOLIDATE(H)

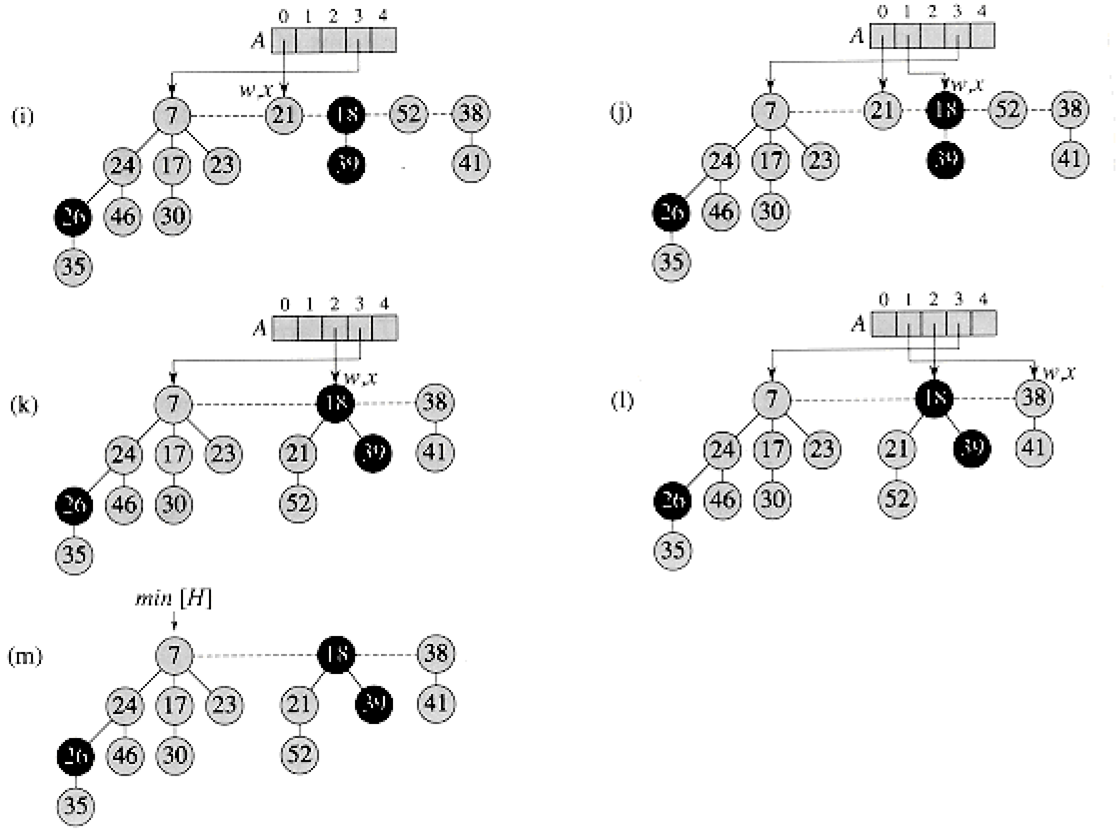
```
1 for i ← 0 to D(n[H])
2 do A[i] ← NIL
3 for each node w in the root list of H
4  do x ← w
5       d ← degree[x]
6     while A[d] ≠ NIL
7         do y ← A[d]
8         if key[x] > key[y]
9           then exchange x ↔ y
10           FIB-HEAP-LINK(H, y, x)
11            A[d] ← NIL
12           d ← d + 1
13          A[d] ← x
14  min[H] ← NIL
15 for i ← 0 to D(n[H])
16 do if A[i] ≠ NIL
17 then add A[i] to the root list of H
18   if min[H] = NIL or key[A[i]] < key[min[H]]
19            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] ← FALSE
```



(a) A Fibonacci heap H. (b) The situation after the minimum node z is removed from the root list and its children are added to the root list. (c)-(e) The array A and the trees after each of the first three iterations of the for loop of lines 3-13 of the procedure CONSOLIDATE. The root list is processed by starting at the minimum node and following right pointers. Each part shows the values of w and x at the end of an iteration. (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. In part (h), the node with key 24 has been linked to the node with key 7. Since no node was previously pointed to by A[3], at the end of the for loop iteration, A[3] is set to point to the root of the resulting tree. (i)-(l) The situation after each of the next four iterations of the while loop. (m) Fibonacci heap H after reconstruction of the root list from the array A and determination of the new min[H] pointer.



We are now ready to show that the amortized cost of extracting the minimum node of an *n*-node Fibonacci heap is *O*(*D*(*n*)). Let *H* denote the Fibonacci heap just prior to the `FIB-HEAP-EXTRACT-MIN` operation.

The actual cost of extracting the minimum node can be accounted for as follows. An *O(D(n))* contribution comes from there being at most *D*(*n*) children of the minimum node that are processed in FIB-HEAP-EXTRACT-MIN and from the work in lines 1-2 and 14-19 of CONSOLIDATE. It remains to analyze the contribution from the **for** loop of lines 3-13. The size of the root list upon calling CONSOLIDATE is at most *D*(*n*) + *t*(*H*) - 1, since it consists of the original *t*(*H*) root-list nodes, minus the extracted root node, plus the children of the extracted node, which number at most *D*(*n*). Every time through the **while** loop of lines 6-12, one of the roots is linked to another, and thus the total amount of work performed in the **for** loop is at most proportional to *D*(*n*) + *t*(*H*). Thus, the total actual work is *O*(*D*(*n*) + *t*(*H*)).

The potential before extracting the minimum node is *t*(*H*) + 2*m*(*H*), and the potential afterward is at most (*D*(*n*) + 1) + 2*m*(*H*), since at most *D*(*n*) + 1 roots remain and no nodes become marked during the operation. The amortized cost is thus at most

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

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

$$= O(D(n))$$

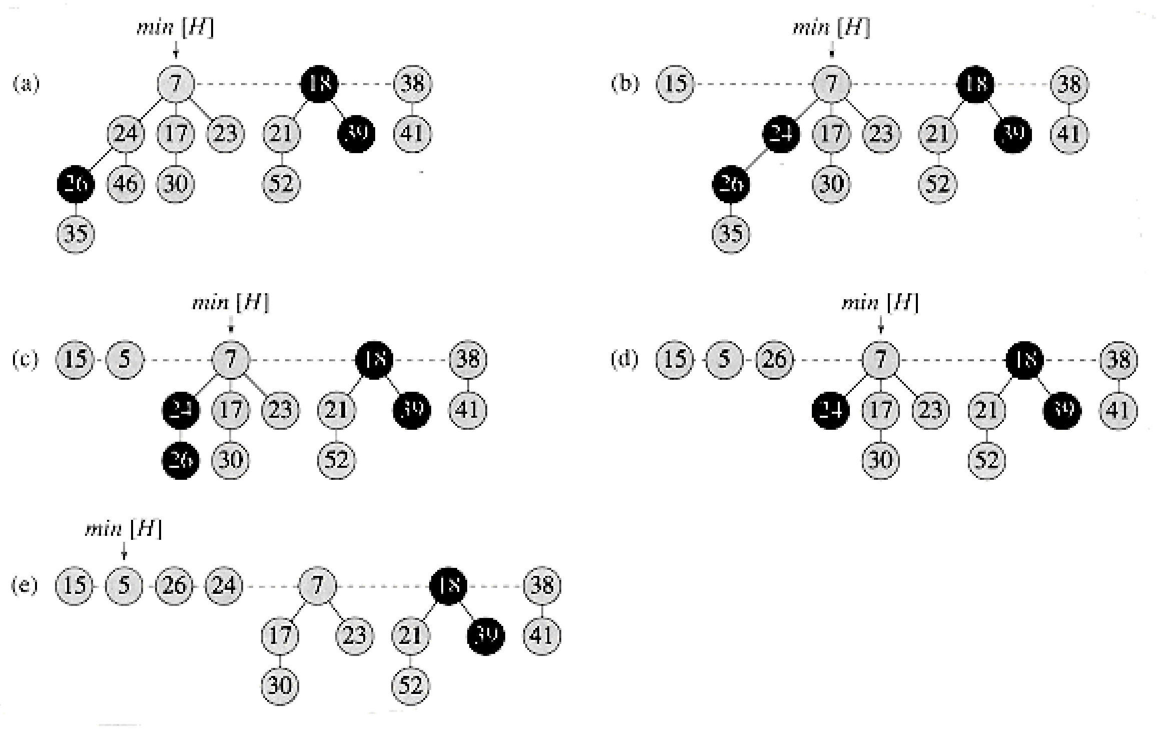
6.Decreasing a key and deleting a node

In this section, we show how to decrease the key of a node in a Fibonacci heap in O(1) amortized time and how to delete any node from an n-node Fibonacci heap in O(D(n)) amortized time.

Decreasing a key

In the following pseudo code for the operation FIB-HEAP-DECREASE-KEY, we assume as before that removing a node from a linked list does not change any of the structural fields in the removed node.

```
FIB-HEAP-DECREASE-KEY(H, x, k)
1 if k > key[x]
2 then error "new key is greater than current key"
3 key[x] ← k
4 y ← p[x]
5 if y ≠ NIL and key[x] < key[y]
6 then CUT(H, x, y)
7         CASCADING-CUT(H, y)
8 if key[x] < key[min[H]]
9then min[H] ← x

CUT(H, x, y)
1 remove x from the child list of y, decrementing 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
3then if mark[y] = FALSE
4then mark[y] ← TRUE
5else CUT(H, y, z)
6CASCADING-CUT(H, z)
```



(a) The initial Fibonacci heap. (b) The node with key 46 has its key decreased to 15. The node becomes a root, and its parent (with key 24), which had previously been unmarked, becomes marked. (c)-(e) The node with key 35 has its key decreased to 5. In part (c), the node, now with key 5, becomes a root. Its parent, with key 26, is marked, so a cascading cut occurs. The node with key 26 is cut from its parent and made an unmarked root in (d). Another cascading cut occurs, since the node with key 24 is marked as well. This node is cut from its parent and made an unmarked root in part (e). The cascading cuts stop at this point, since the node with key 7 is a root. (Even if this node were not a root, the cascading cuts would stop, since it is unmarked.) The result of the `FIB-HEAP-DECREASE-KEY` operation is shown in part (e), with min[H] pointing to the new minimum node.

Deleting a node

It is easy to delete a node from an *n*-node Fibonacci heap in $O(D(n))$ amortized time, as is done by the following pseudo code. We assume that there is no key value of $-\infty$ currently in the Fibonacci heap.

```
FIB-HEAP-DELETE(H, x)
1  FIB-HEAP-DECREASE-KEY(H, x, -∞)
2  FIB-HEAP-EXTRACT-MIN(H)
```

FIB-HEAP-DELETE is analogous to BINOMIAL-HEAP-DELETE. It makes *x* become the minimum node in the Fibonacci heap by giving it a uniquely small key of $-\infty$. Node *x* is then removed from the Fibonacci heap by the FIB-HEAP-EXTRACT-MIN procedure. The amortized time of FIB-HEAP-DELETE is the sum of the $O(1)$ amortized time of FIB-HEAP-DECREASE-KEY and the $O(D(n))$ amortized time of FIB-HEAP-EXTRACT-MIN.