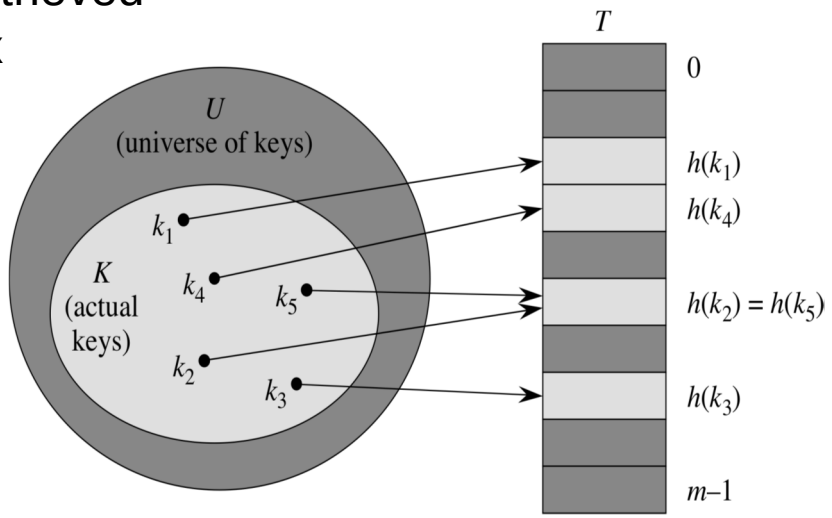
Part II

Data Structures

§ Three Types of Data Structures Considered:

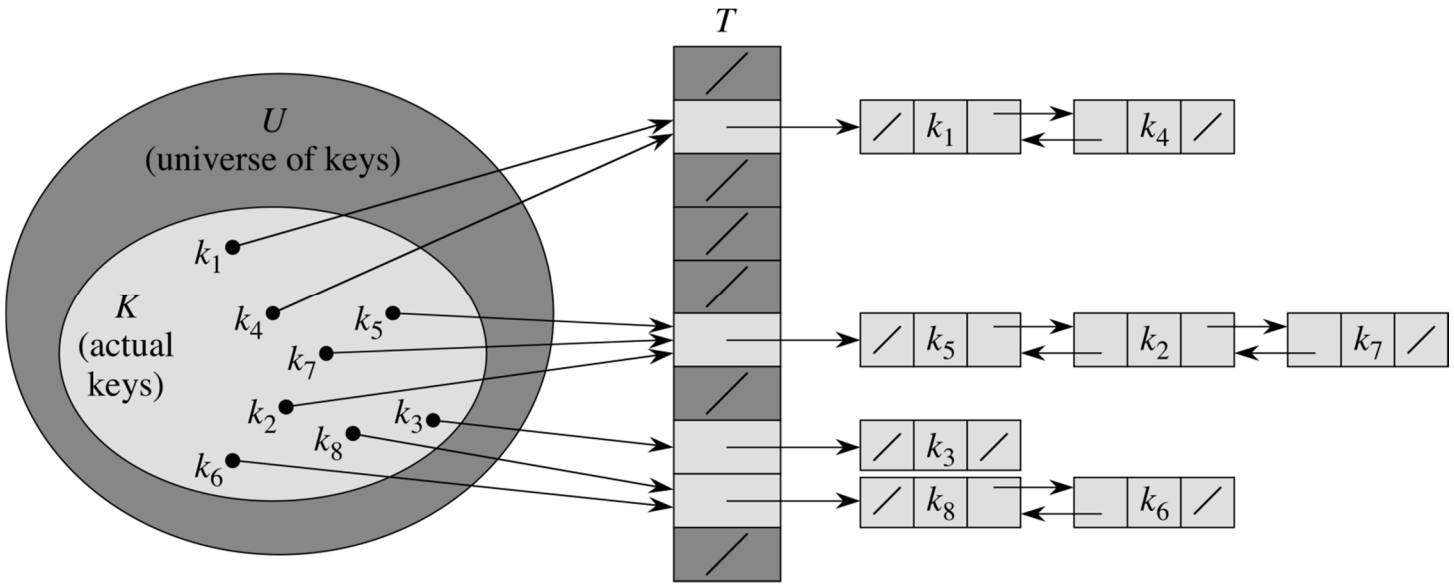
- Hash Tables
- Trees
- Heaps

- **Hash Table**
  - – suitable for large key space with small numbers of actual keys stored
  - – average search time over the hash table equal to $O(1)$
  - – collision-handling strategy required
  - – keys must be stored to ensure right items retrieved
  - – hash function maps a key to one table index



Hash Tables (continued)

- **Hash Table with Collision Resolution by Chaining**
  - multiple elements with different keys mapped to the same table entry (collisions)
  - they can be chained, with a lookup going through the chain
  - for load factor of α = n/m, where n(or m) is the number of elements (or table entries)
  - under uniform distribution, one search (be successful or not) takes Θ(1+α), according to Theorems 11.1 and 11.2
  - if hash table size (m) grows in proportion to n, then the mean search time is Θ(1)



Hash Tables (continued)

- **Hash Functions**
  - division method
  - multiplication method

1. Division method:

$h(k) = k \bmod m$, where the choice of $m = 2^p - 1$ is poor but a *prime* not close to an exact power of 2 is often good

For example, given n = 2000 character strings to be stored in a hash table with m = 701, then an unsuccessful search takes some 3 accesses

Hash Tables (continued)

2. Multiplication method:

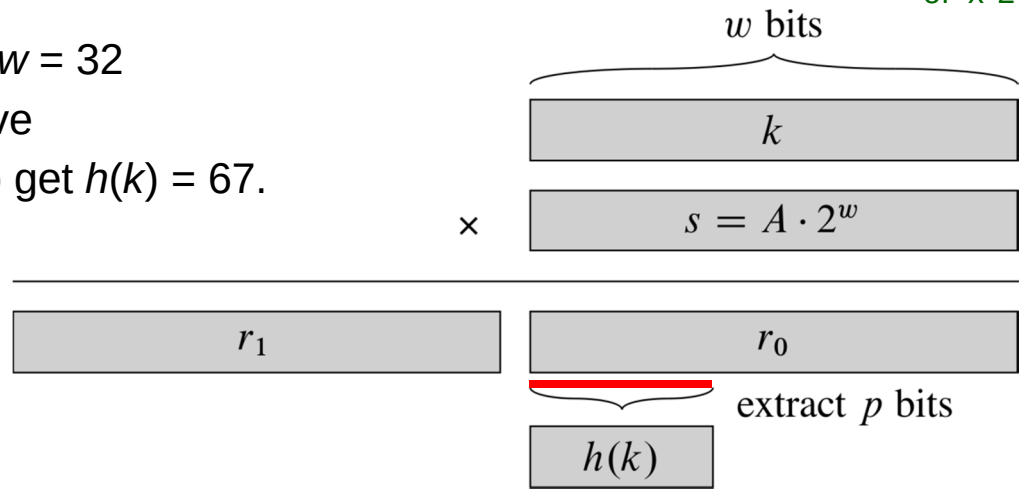
- Multiply k by a constant A, with $0 < A < 1$ and extract the fraction part of $k \cdot A$, e.g., $h(k) = \lfloor m\ (kA \mod 1) \rfloor$, where "$kA \mod 1$" equals $kA - \lfloor kA \rfloor$

  – table size (m) can be arbitrary, e.g., $m = 2^p$
  – let A be a fraction of the form $s/2^w$ as shown in the figure below
  – perform multiplication of w-bit k and w-bit s, to get a $2w$-bit product
  – product denoted by $r_1 \cdot 2^w + r_0$, then p-bit hash value is obtained from r_0

  – while arbitrary A works, $A \approx (\sqrt{5}\ -1)/2 = 0.6180339887\ \ldots$ recommended

Golden ratio: a root of x^2 + x -1 = 0

+ Let $k = 123456$, $p = 14$, $m = 2^{14}$, $w = 32$ and $A = 2654435769/2^{32}$, we have $k \cdot s = (76300 \cdot 2^{32}) + 17612864$ to get $h(k) = 67$.



Hash Tables (continued)

- **Open Addressing** (to deal with collision-chaining)
  - elements stored inside the table
  - calculating probe sequence of given key, instead of using pointers:
    $\langle h(k, 0), h(k, 1), h(k, 2), \ldots, h(k, m-1) \rangle$
  - if probe sequence is a permutation of $\langle 0, 1, 2, \ldots, m-1 \rangle$, every table entry is a candidate location for the element
  - the probe sequence is fixed for a given key
  - key has to be stored in the table entry

HASH-INSERT(T, k)

```
i = 0
repeat
    j = h(k, i)
    if T[j] == NIL
        T[j] = k
        return j
    else i = i + 1
until i == m
error "hash table overflow"
```

HASH-SEARCH(T, k)

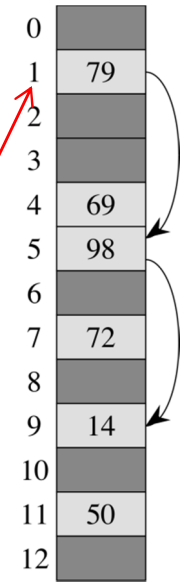
```
i = 0
repeat
    j = h(k, i)
    if T[j] == k
        return j
    i = i + 1
until T[j] == NIL or i = m
return NIL
```

Hash Tables (continued)

- **Uniform Hashing Analysis**
  - key likely to be any one of $m!$ permutations of $\langle 0, 1, 2, \ldots, m-1 \rangle$
  - probe sequences defined for open addressing
    - + linear probing: m different sequences
      - dictated by $h(k, i) = (h'(k) + i) \bmod m$
      - subject to *primary clustering*
    - + quadratic probing: m different sequences
      - dictated by $h(k, i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$
      - subject to *secondary clustering*, i.e., $h(k_1, 0) = h(k_2, 0)$ ⇒ both keys follow the same probe sequence
    - + double hashing: m^2 different sequences
      - dictated by $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$
      - many good choices: $h_2(k)$ relatively prime to m; m itself a prime; m a power of 2 and $h_2(k)$ always an odd number; or below:

        m a prime and m' slightly less than m (e.g., m-1 or m-2) with
        $h_1(k) = k \bmod m$
        $h_2(k) = 1 + (k \bmod m')$

e.g., for k =14 under m = 13, m' = 11



Hash Tables (continued)

- **Open-Address Hashing Analysis**

Theorem 1:

For an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in unsuccessful search under uniform hashing is at most $1/(1 - \alpha)$.

+ if α is a constant, an unsuccessful search runs in $O(1)$ time
+ inserting an element into an open-address hash table with load factor α takes at most $1/(1 - \alpha)$ probes on an average under uniform hashing, because it always takes 1 probe, plus prob. α to take 2 probes, plus prob. α^2 to take 3 probes, etc., yielding $1+\alpha+\alpha^2+\alpha^3+ \ldots = 1/(1 - \alpha)$

e.g., for α = 50/100, we have 2 probes.

Theorem 2:

For an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search under uniform hashing is at most $\frac{1}{\alpha} \cdot \ln\frac{1}{1-\alpha}$.

1st key takes ≤ $1/(1-1/m)$) probes, on an average, when α is $1/m$
2nd key takes ≤ $1/(1-2/m)$) probes, on an average, when α is $2/m$
:
ith key takes ≤ $1/(1-i/m)$) probes

Mean no. of probes equals results over all n keys

e.g., for α = 50/100, we have ≈ 1.39 probes, as ln(2) ~ 0.693.

Hash Tables (continued)

- **Universal Hashing**

  For the hash function of $h_{ab}(k) = ((a \cdot k + b) \bmod p) \bmod m$, where p is a large prime number, with $p > m$, $a \in \{1, 2, \ldots p\text{-}1\}$ and $b \in \{0, 1, 2, p\text{-}1\}$, the collection of such hash functions is *universal*.

  In this case, m can be any number and does not have to be a prime.
  For example, for $p = 17$ and $m = 8$, we have $h_{34}(15) = 7$.

  ((3*15+4) mod 17) mod 8 = 7

Hash Tables (continued)

- **Perfect Hashing**
  - set of keys is *static* (never changed), then excellent worst case performance exists
  - search takes $O(1)$ in memory accesses in the worst case
  - two levels of hashing, with secondary hash table S_j to hold those hashed to slot j
  - unlikely to have collisions in S_j if its size m_j equals n_j^2
  - total storage required for both primary and secondary hash tables is $O(n)$

Theorem 3:

Given n_j keys for a hash table sized $m_j = n_j^2$, the probability of having collisions $< 1/2$.

This theorem implies that one can easily (with a few trials) identify **collision-free** $h_j(k)$ with proper a_j and b_j

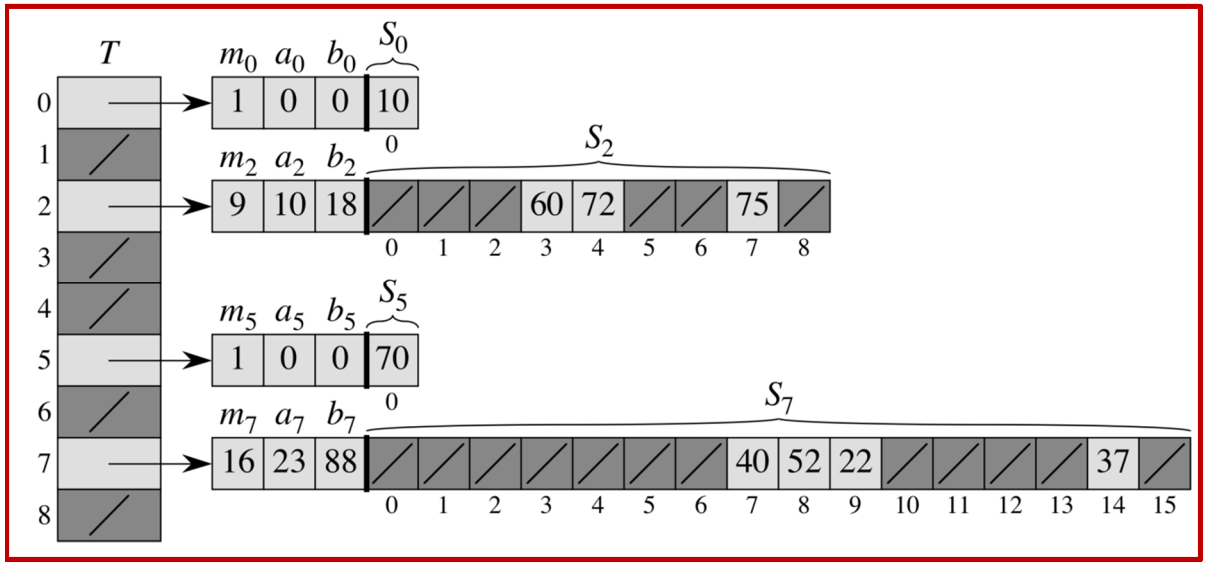


Figure 11.6 Using perfect hashing to store the set $K = \{10, 22, 37, 40, 60, 70, 75\}$. The outer hash function is $h(k) = ((a \cdot k + b) \bmod p) \bmod m$, where $a = 3$, $b = 42$, $p = 101$, and $m = 9$.

For example, $h(75) = 2$, so key 75 hashes to slot 2 of table T. A secondary hash table S_j stores all keys hashing to slot j. The size of hash table S_j is m_j, and the associated hash function is $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$. If $h_2(75) = 7$, key 75 is stored in Slot 7 of secondary hash table S_2. There are no collisions in any of the secondary hash tables, and so searching takes a constant time in the worst case.

Hash Tables (continued)

- **Perfect Hashing**
  – total storage expected for both primary and secondary hash tables is $O(n)$.

Theorem 4:

Given n keys for a hash table sized $m = n$ and n_j keys present in slot j under the outer hash, we have $E\left[\sum_{j=0}^{m-1} {n_j}^2\right] < \mathbf{2n}$. Note. This deals with the "expected" storage size.

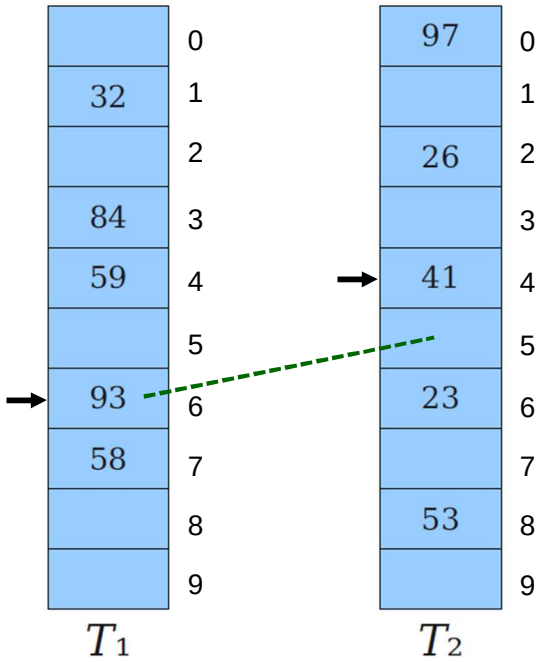
Corollary 1:

Given n keys for a hash table sized $m = n$ and n_j keys present in slot j under the outer hash, and the sizes of secondary hash tables equal to $m_j = {n_j}^2$, then the probability that total storage used for secondary hash tables $\geq 4n$ is less than ½.

This corollary implies that one can easily (with a few trials) identify **good** primary hash $h(k)$ with proper a and b that lead to a reasonable amount of total storage required.

Hash Tables (continued)

- **Cuckoo Hashing**[§]
  - – move keys around upon insertion for worst case of *O*(1) probes
    - + quick search while possibly lengthy insertion
    - + multiple hash functions required for separate hash tables
  - – delete keys in the worst-case of *O*(1)
  - – example with two hash functions below:
    - + two hash tables, each with *m* elements
    - + two hash functions, h_1 and h_2, one for a table
    - + each key *x* at either $h_1(x)$ or $h_2(x)$
    - + new key '10' with $h_1(10) = 6$ and $h_2(10) = 4$
      - * move key '93' around, if $h_2(93)=5$, for insertion

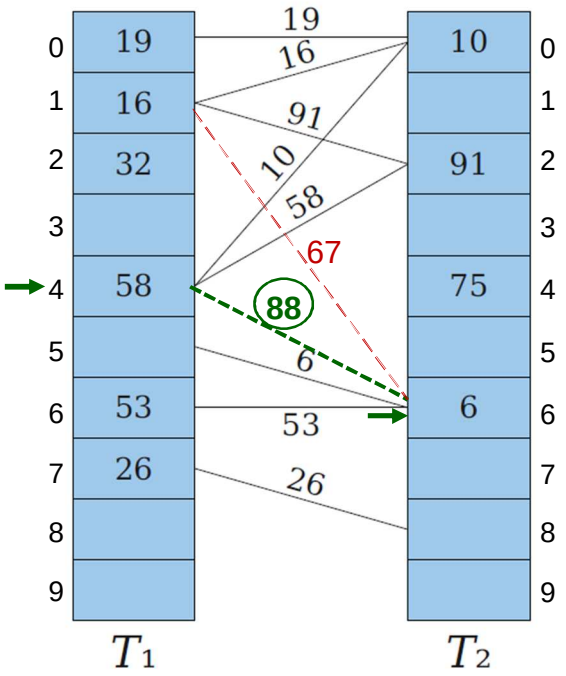


  - – Algorithm
    1. insert "new key" to T_1 located at h_1(new key), when available
    2. otherwise, "new key" displaces "existing key" in T_1; place "existing key" in T_2 located at h_2(existing key); repeat the displacement process, if needed.

§ R. Pagh and F. Rodler, "Cuckoo Hashing," *Journal of Algorithms*, Aug. 2001, pp. 121-133.

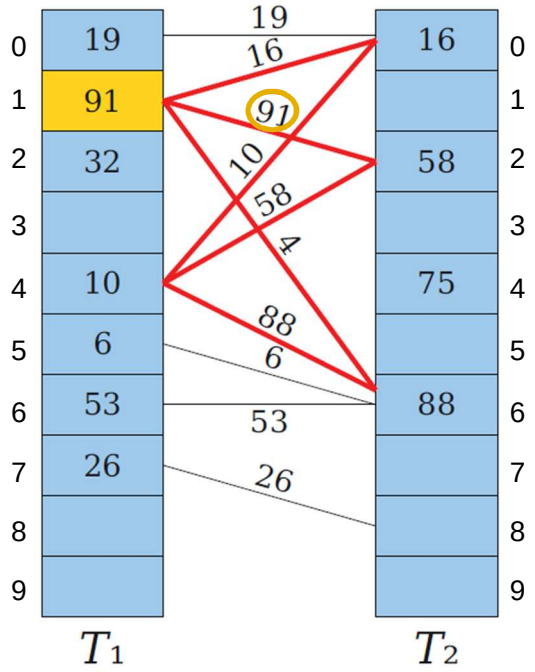
Hash Tables (continued)

- **Cuckoo Graph**
  - – derived from cuckoo hash tables
    - + each table entry is a node
    - + each key is an edge, which links the entries that can hold the key
    - + an insertion adds a new edge to the graph
      - * let $h_1(88) = 4$ and $h_2(88) = 6$
      - * displace key ‘58’ in T_1 to make room for key ‘88’
      - * displace key ‘91’ in T_2 to make room for key ‘58’
      - * displace key ‘16’ in T_1 to make room for key ‘91’
      - * displace key ‘10’ in T_2 to make room for key ‘16’
      - * repeat ….
    - + insertion done by tracing a path over the graph
    - + what about inserting next key “67” with $h_1(67) = 1$ and $h_2(67) = 6$?



Hash Tables (continued)

- **Cuckoo Graph**
  - insertion succeeds if and only if the new edge (defined by the new key) contains at most one cycle



Proof sketch:

1. each edge denotes a key and requires a table entry (i.e., node) to hold it
2. for a cycle exists, the number of its edges equals the number of its nodes involved, so that all nodes (table entries) are taken
3. if a new edge (after added) is on two cycles the edge must link two taken nodes AND all nodes on the two cycles have been used
4. the new key (edge) *cannot* be accommodated.

Binary Search Trees

§ Binary Search Tree Property

- Given tree node x, if node y is in the left (or right) subtree of x, we have $y.key \leq x.key$ (or $y.key \geq x.key$).

Theorem 1

INORDER-TREE-WALK(x) across an n-node subtree rooted at x takes $\Theta(n)$.

(because the tree height may equal n)

Proof.

The tree walk has to visit every node, and hence, the time complexity $T(n)$ is lower bounded by $\Omega(n)$.

Next, considering that INORDER-TREE-WALK(x) is called on x whose left and right subtrees have k and (n-k-1) nodes, respectively, we have

$T(n) \leq T(k) + T(n\text{-}k\text{-}1) + d$ for some $d > 0$.

By the substitution method, we prove that $T(n) \leq (c+d)n + c$, for a constant c, below:

$$\begin{aligned} T(n) &\leq T(k) + T(n\text{-}k\text{-}1) + d \\ &\leq ((c+d)k + c) + ((c+d)(n\text{-}k\text{-}1) + c) + d \\ &= (c+d)n + c - (c+d) + c + d \\ &= (c+d)n + c \end{aligned}$$

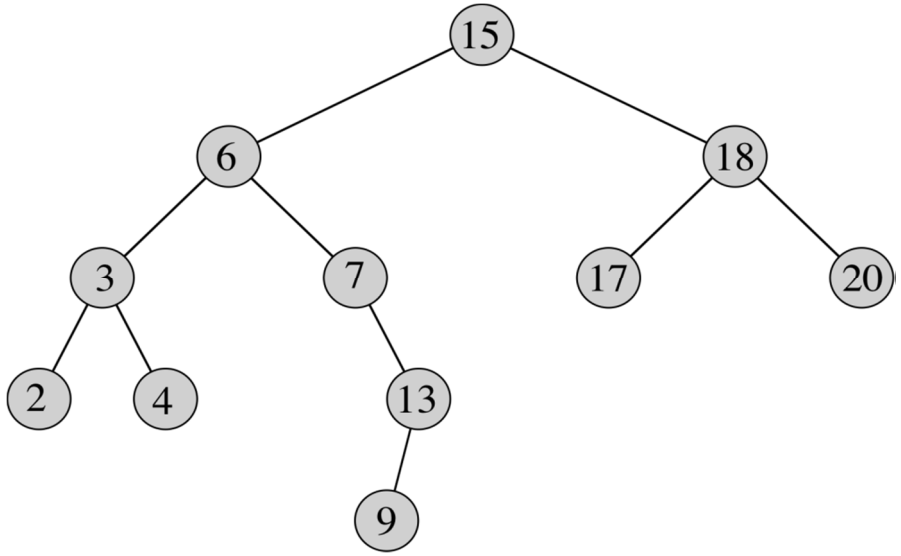
(upper bounded)

Binary Search Trees (continued)

- **Querying binary search trees**
  - searching
  - successor and predecessor
  - insertion and deletion

- Searching via TREE-SEARCH in time complexity $O(h)$

```
TREE-SEARCH(x, k)
if x == NIL or k == key[x]
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```



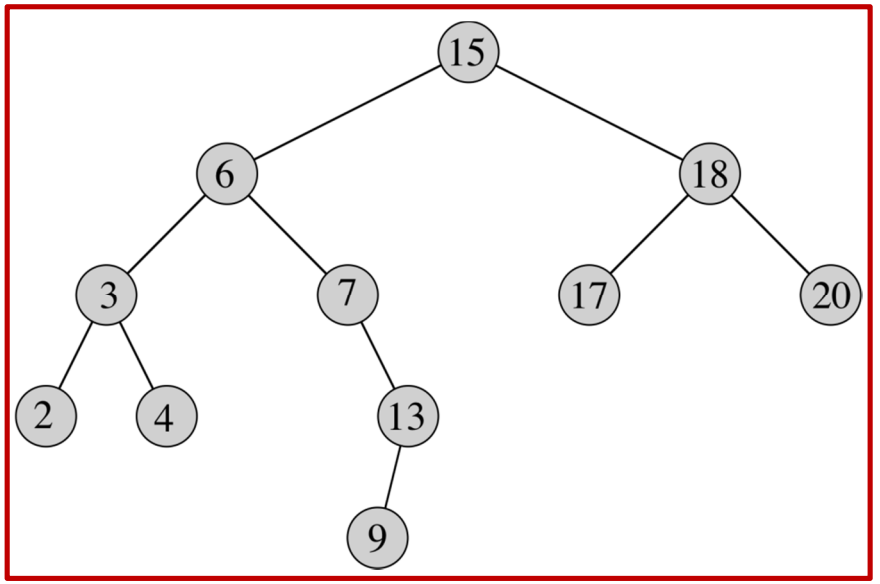
search path from root to *key* = 13 is
15 → 6 → 7 → 13

Binary Search Trees (continued)

- Successor and predecessor
  - + by inorder tree walk
  - + separate results for non-empty right subtree and for empty right subtree
  - + time complexity for a tree of height *h* is *O*(*h*)

```
TREE-SUCCESSOR(x)
  if x.right ≠ NIL
      return TREE-MINIMUM(x.right)
  y = x.p
  while y ≠ NIL and x == y.right
      x = y
      y = y.p
  return y
```

If node *x* has no right subtree, its successor is the lowest ancestor of *x* whose left child is also an ancestor of *x*.



successor of *key* = 7 is 9
successor of *key* = 13 is 15

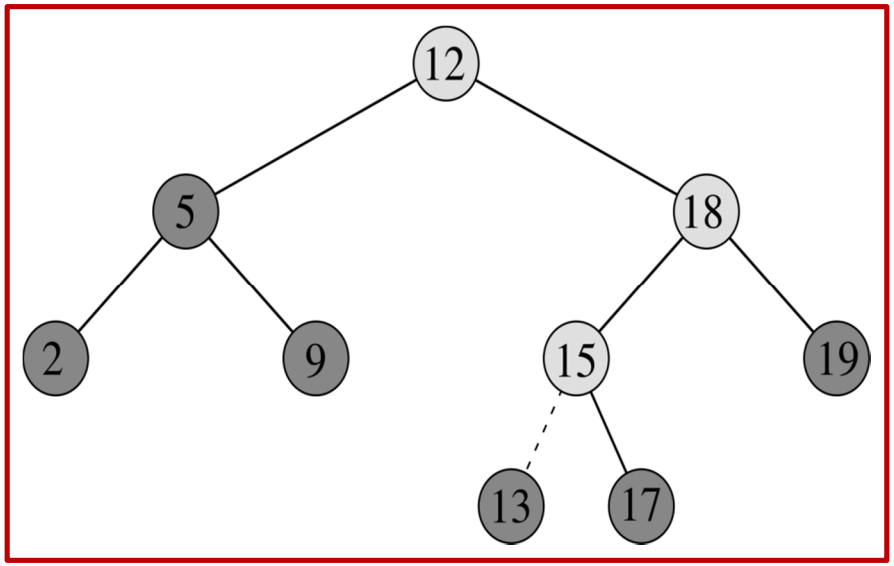
Binary Search Trees (continued)

- Insertion
  - + tree node (say, *z*) to be added, with *z.key* = *v*, *z.left* =nil, *z.right* = nil
  - + *z* added to one appropriate node with an ***absent*** left or right child
  - + tree inorder walk, with the trailing pointer *y* maintained
  - + time complexity for a tree of height *h* is *O*(*h*)

TREE-INSERT(T, z)

```
y = NIL
x = T.root
while x ≠ NIL
      y = x
      if z.key < x.key
            x = x.left
      else x = x.right
z.p = y
if y == NIL
      T.root = z          // tree T was empty
elseif z.key < y.key
      y.left = z
else y.right = z
```

tree traversal downward until the insertion point, with the added node a **leaf node**



Inserting *key* = 13 into binary search tree, with affected path marked.

Binary Search Trees (continued)

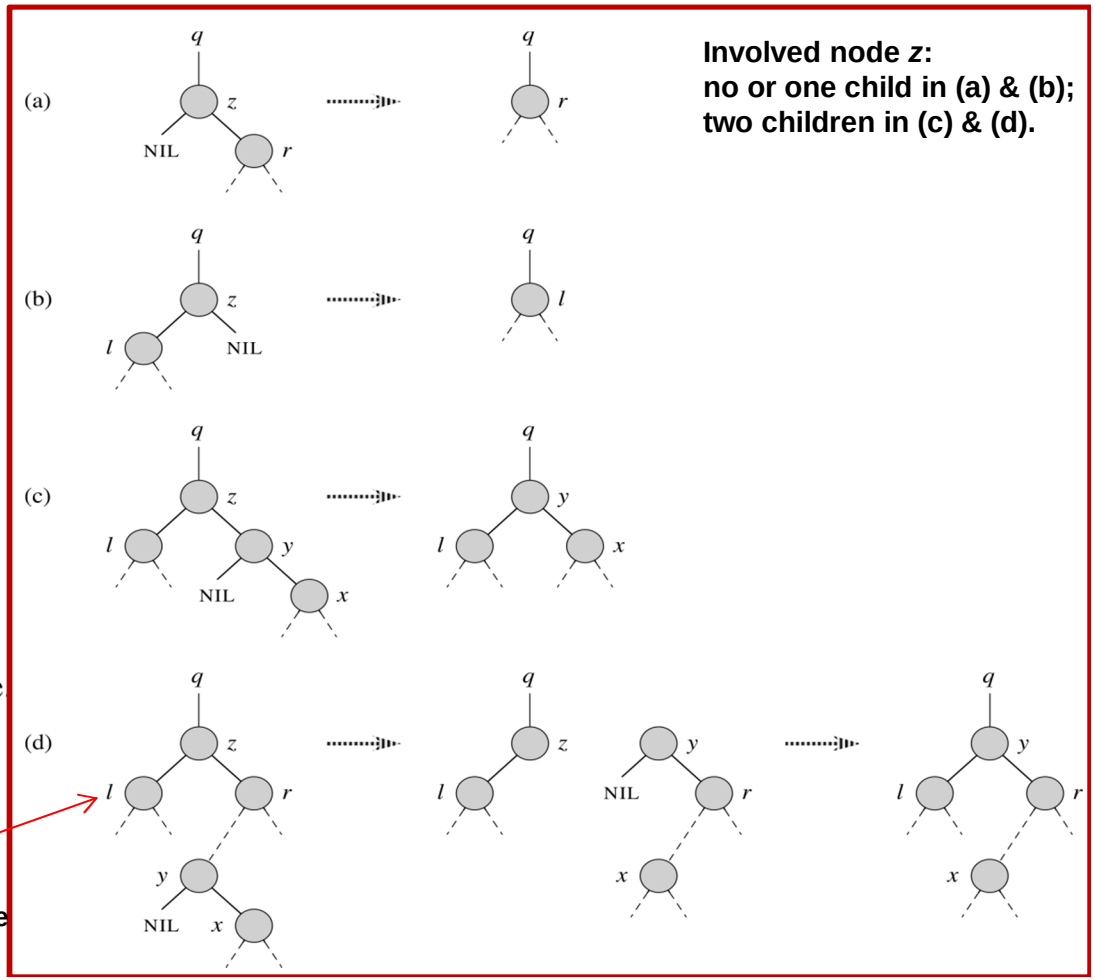
- Deletion
  - + three basic cases involved: *z* having no child, exactly one child, and two children
  - + based on TRANSPLANT for replacing sibling subtree with another subtree
  - + time complexity for a tree of height *h* is *O*(*h*)

```
TRANSPLANT(T, u, v)
  if u.p == NIL
      T.root = v
  elseif u == u.p.left
      u.p.left = v
  else u.p.right = v
  if v ≠ NIL
      v.p = u.p
```

```
TREE-DELETE(T, z)
  if z.left == NIL
      TRANSPLANT(T, z, z.right)          // z has no left child
  elseif z.right == NIL
      TRANSPLANT(T, z, z.left)           // z has just a left child
  else // z has two children.
      y = TREE-MINIMUM(z.right)          // y is z's successor
      if y.p ≠ z   ← This is case (d) shown in the figure.
          // y lies within z's right subtree but is not the root of this subtree.
          TRANSPLANT(T, y, y.right)
          y.right = z.right
          y.right.p = y
      // Replace z by y.
      TRANSPLANT(T, z, y)
      y.left = z.left
      y.left.p = y
```

Prepare a subtree rooted at ***y*** whose left subtree is empty.

Alternatively, one may cut the left-hand subtree of *z* and move it to become the left-hand subtree of *y*.



Involved node *z*:
no or one child in (a) & (b);
two children in (c) & (d).

Binary Search Trees (continued)

– Randomly build binary search trees
  + worst-case tree height (h) being n-1
  + can be shown that $h \geq \lfloor \lg n \rfloor$
  + like quicksort, average case proven to be much closer to best case than worst case
  + special case of creating binary search trees (via insertion alone) with random keys, we have following theorem:

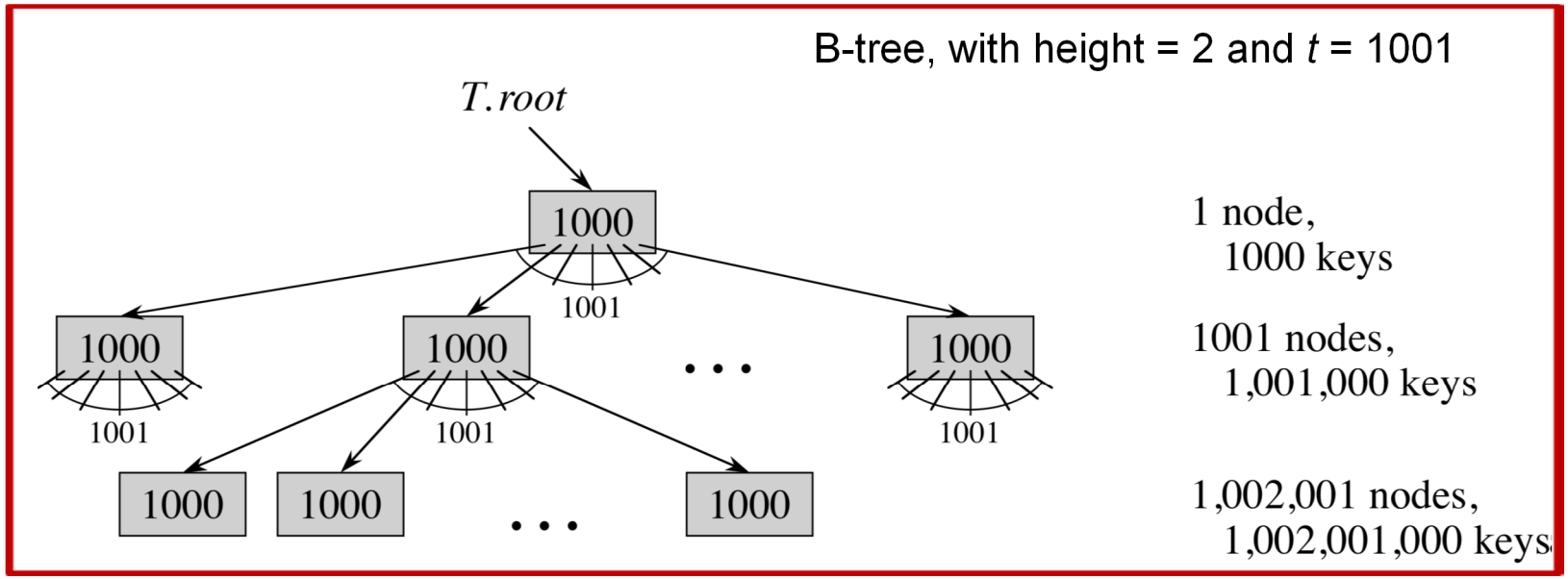
Theorem 2. ← This theorem refers to the special case that the tree is created by insertion alone (and no deletion).

The expected height of a randomly built binary search tree on n distinct keys is $\boldsymbol{O(\lg n)}$.

B-Trees

§ Balanced Search Trees (B-Trees)

+ balance achieved by permitting multiple (i.e., ≥ t-1) keys in any non-root node
+ node has at most 2t-1 keys, called **full node** (with degree = 2t)
+ keys stored in a node in non-decreasing order
+ node x (with $x.n$ keys) has $x.n$+1 children, pointed by $x.c_1$, $x.c_2$, …., $x.c_{x.n+1}$, then
  $k_1 \le x.key_1 \le k_2 \le x.key_2 \le \ldots \le x.key_{x.n} \le k_{x.n+1}$, where k_i is a key stored in subtree rooted at $x.c_i$ and $x.key_i$ is a key stored in node x
+ all leaves have the same height
+ simplest B-tree is for t = 2, called 2-3-4 tree (each internal node has 2, 3, or 4 children)



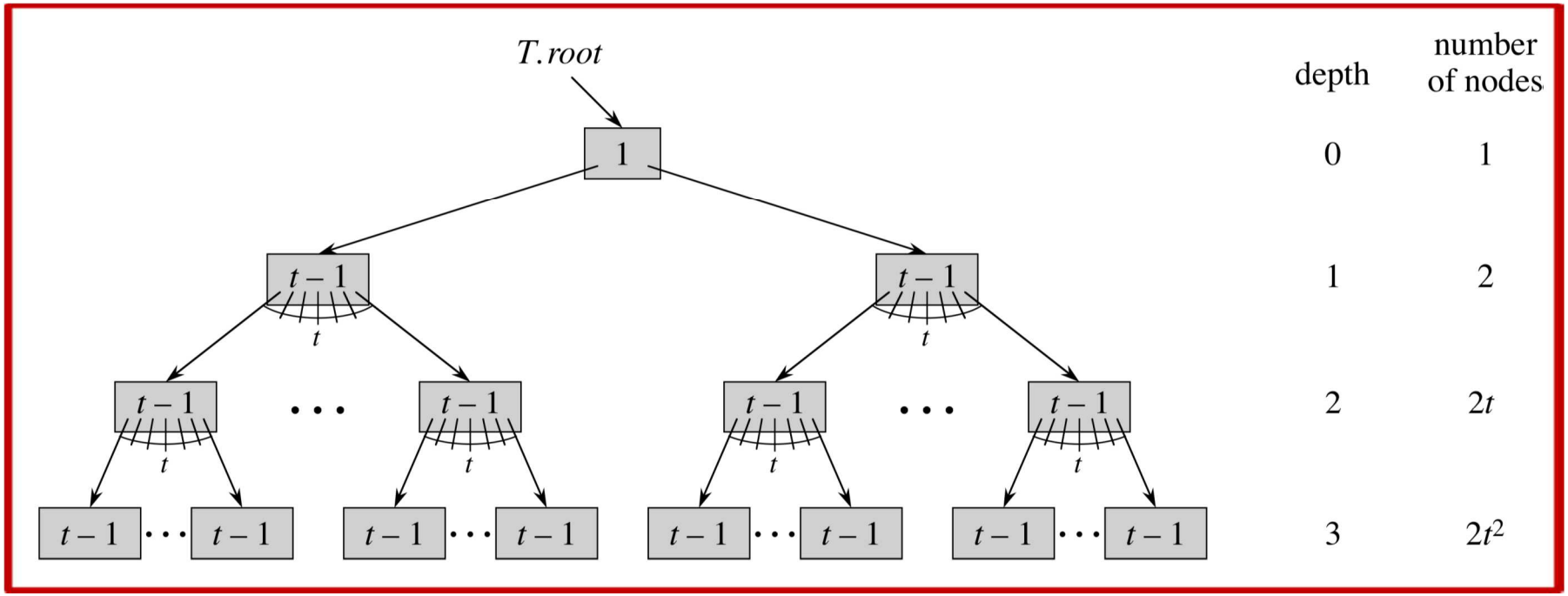
B-Trees (continued)

- **Theorem**

  For any n-key B-tree T of height h and minimum node degree $t \geq 2$ (i.e., number of keys in each non-root node $\geq t$-1), we have $h \leq \log_t \frac{n+1}{2}$.

  + Proof follows the fact of $n \geq 1 + (t-1) \cdot \sum_{i=1}^{h} 2 \cdot t^{i-1}$ illustrated in the figure below

  (to show the minimum possible number of keys in a B-tree with height = 3).



B-Trees (continued)

- **Basic operations**
  - + Searching for record with key = *k*: each internal node ***x*** makes an ($x.n$ +1)-way branching decision; returning (y, i) → node ***y*** and its index i with $y.key_i = k$

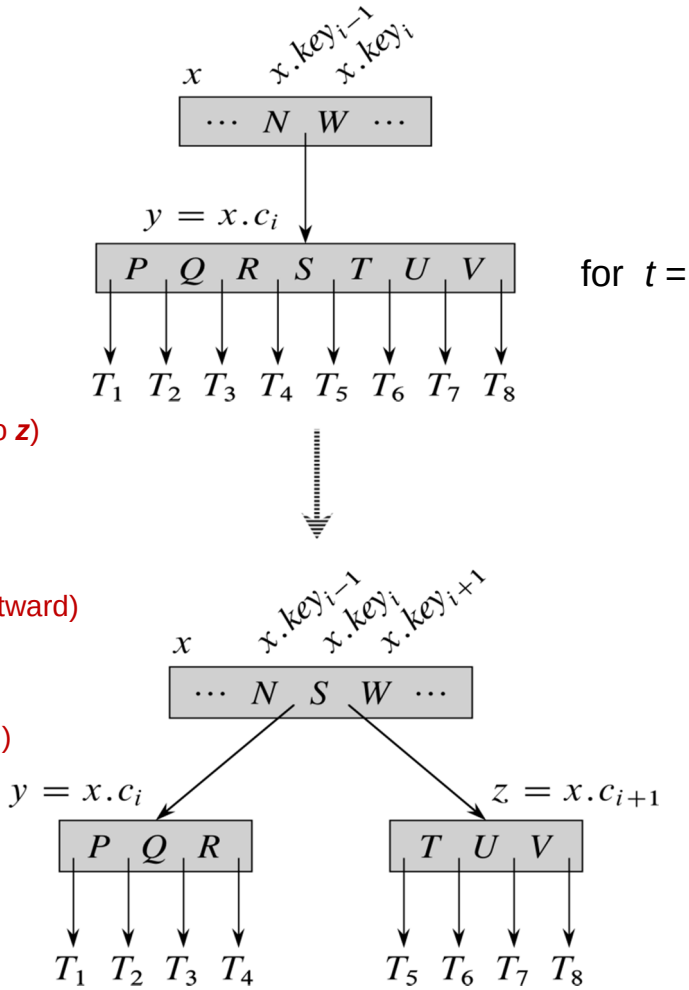
```
B-TREE-SEARCH(x, k)
1  i = 1
2  while i ≤ x.n and k > x.key_i
3       i = i + 1
4  if i ≤ x.n and k == x.key_i
5       return (x, i)
6  elseif x.leaf
7       return NIL
8  else DISK-READ(x.c_i)
9       return B-TREE-SEARCH(x.c_i, k)
```

x.leaf: Boolean variable to indicate if it is a leaf node

B-Trees (continued)

- **Basic operations**
  - + Splitting a full node: for *nonfull internal node* ***x*** and a *full child* of ***x*** (say, ***y*** = $x.c_i$), ***y*** is split at median key *S*, which moves up into ***x***, with those > *S* forming a new subtree

```
B-TREE-SPLIT-CHILD(x, i)                    (split i-th child of x)
 1  z = ALLOCATE-NODE()
 2  y = x.c_i                               (node x's i-th child is full)
 3  z.leaf = y.leaf
 4  z.n = t - 1
 5  for j = 1 to t - 1
 6      z.key_j = y.key_{j+t}
 7  if not y.leaf
 8      for j = 1 to t                      (move child pointers from y to z)
 9          z.c_j = y.c_{j+t}
10  y.n = t - 1
11  for j = x.n + 1 downto i + 1
12      x.c_{j+1} = x.c_j                   (shift child pointers in x rightward)
13  x.c_{i+1} = z
14  for j = x.n downto i
15      x.key_{j+1} = x.key_j               (shift keys in x rightward)
16  x.key_i = y.key_t                       (move middle key in y up)
17  x.n = x.n + 1
18  DISK-WRITE(y)
19  DISK-WRITE(z)
20  DISK-WRITE(x)
```



B-Trees (continued)

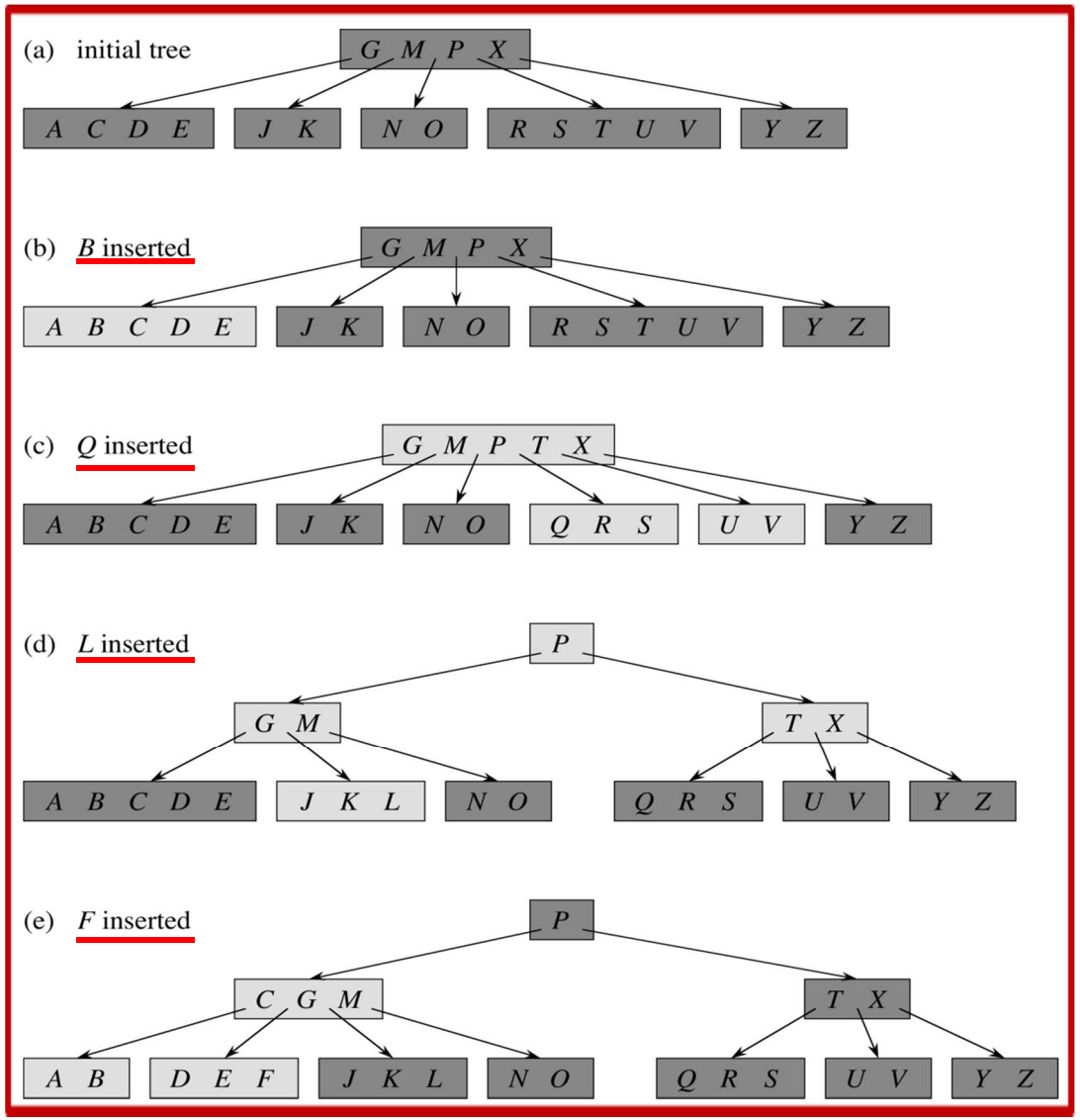
- **Basic operations**
  - + Inserting a key: B-tree ***T*** of height *h*, takes *O*(*h*) disk accesses (a node is kept as a disk page)
    - – *can't* insert the key into a newly created node, as it requires at least *t*-1 keys
    - – **never** descend through a **full node**, achieved by B-TREE-SPLIT-CHILD
    - – done with no back-tracking
    - – root split to increase height by 1
    - – insert only to a **leaf node**

```
B-TREE-INSERT(T, k)
 1  r = T.root
 2  if r.n == 2t − 1          // splitting the root?
 3      s = ALLOCATE-NODE()
 4      T.root = s
 5      s.leaf = FALSE
 6      s.n = 0
 7      s.c_1 = r
 8      B-TREE-SPLIT-CHILD(s, 1)        (split 1st child of S)
 9      B-TREE-INSERT-NONFULL(s, k)
10  else B-TREE-INSERT-NONFULL(r, k)
```

```
B-TREE-INSERT-NONFULL(x, k)
 1  i = x.n
 2  if x.leaf                  (upon a leaf, key is inserted therein)
 3      while i ≥ 1 and k < x.key_i
 4          x.key_{i+1} = x.key_i      (shift key in x rightward)
 5          i = i − 1
 6      x.key_{i+1} = k
 7      x.n = x.n + 1
 8      DISK-WRITE(x)
 9  else while i ≥ 1 and k < x.key_i
10          i = i − 1
11      i = i + 1
12      DISK-READ(x.c_i)           (descend one level)
13      if x.c_i.n == 2t − 1       // full child?
14          B-TREE-SPLIT-CHILD(x, i)
15          if k > x.key_i
16              i = i + 1
17      B-TREE-INSERT-NONFULL(x.c_i, k)
```

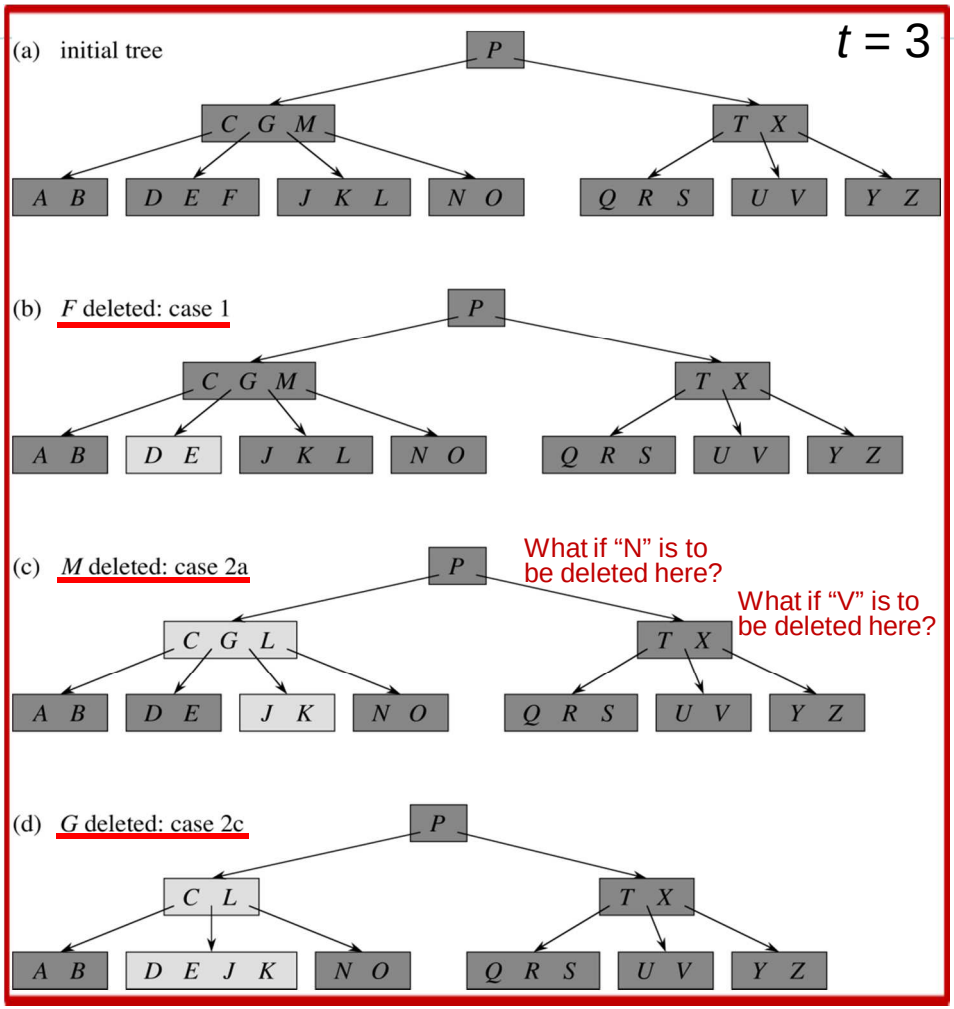
B-Trees (continued)

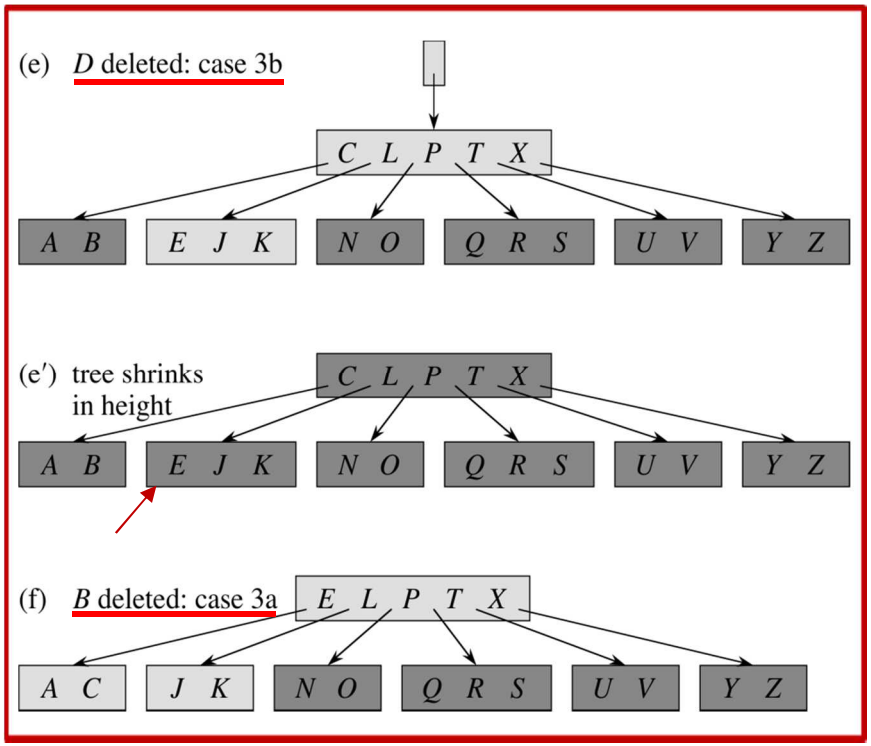
+ Inserting a key: B-tree ***T*** with $t = 3$ and modified nodes lightly shaded



B-Trees (continued)

- **Basic operations**
  - + Deleting a key: delete key *k* from the subtree rooted at ***x*** in B-tree ***T*** with the minimum degree of *t* (where the key may be in any node, not just at a leaf node)
    - – number of keys kept in any **internal node** upon descending through would be **at least *t***





B-Trees (continued)

+ Deleting key *k* from the subtree rooted at ***x*** in B-tree ***T*** with the minimum degree of *t*
  – **Deletion Procedure** sketched below:

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. (We can find k' and delete it in a single downward pass.)
   b. If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x. If z 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. (We can find k' and delete it 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 $x.c_i$ of the appropriate subtree that must contain k, if k is in the tree at all. If $x.c_i$ 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 $x.c_i$ has only $t-1$ keys but has an immediate sibling with at least t keys, give $x.c_i$ an extra key by moving a key from x down into $x.c_i$, moving a key from $x.c_i$'s immediate left or right sibling up into x, and moving the appropriate child pointer from the sibling into $x.c_i$.
   b. If $x.c_i$ and both of $x.c_i$'s immediate siblings have $t-1$ keys, merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

Targeted node with the key

This case is explained next, as Rule 2:
While at internal node *x* contains the key (say, *k*) to be deleted, it checks if prededecessor or successor of *k* can be borrowed to replace *k* (and in this case, only the borrowed key is moved; and its associated pointer stays).

Prepare to walk down the tree

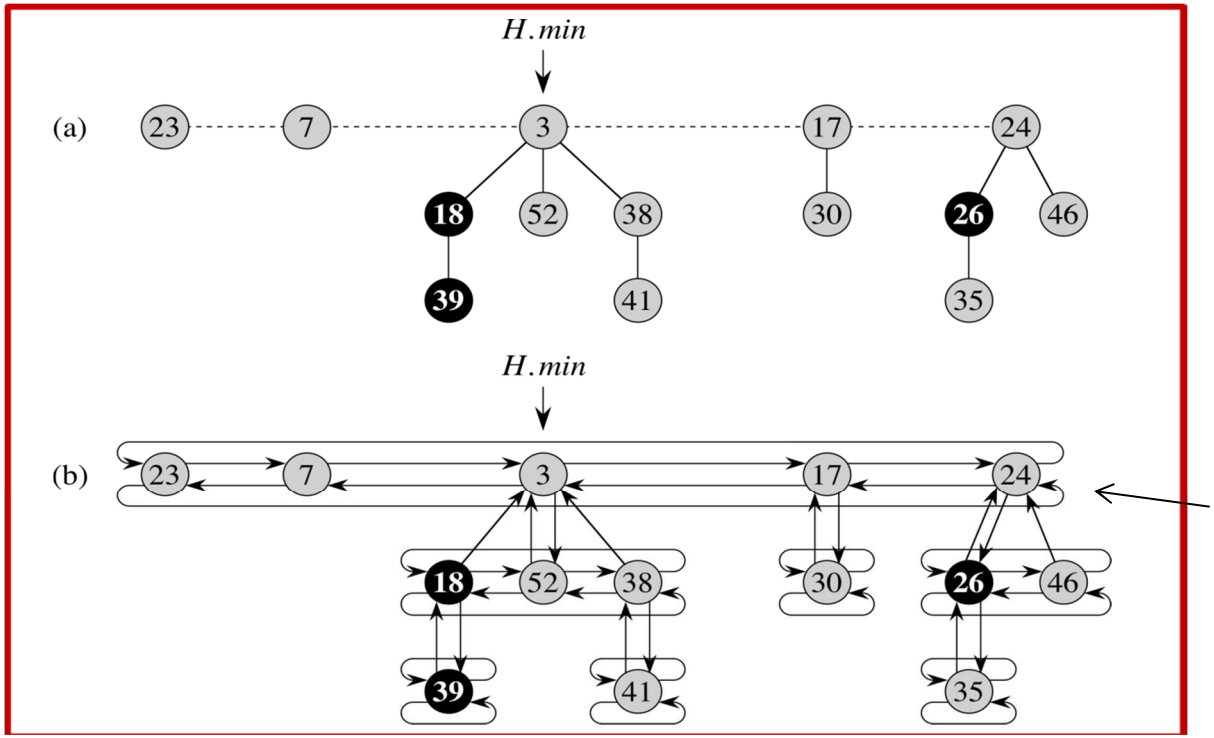
visited

This case is explained first, as Rule 1:
While at internal node *x*, it checks if the root of the target child has at least *t* key
If not, it prepares that root node by borrowing a key (plus associated pointer, i.e., a subtree) from its left or right sibling, if possible.

Fibonacci Heaps

§ Fibonacci Heap

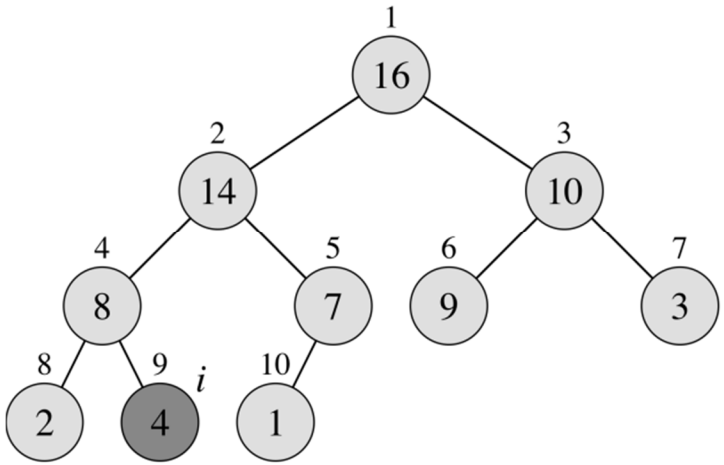
+ a collection of min-heap trees, with their roots doubly linked
+ belongs to the “mergeable heap” that supports five operations efficiently:
  MAKE-HEAP(); INSERT(H, x); MINIMUM(H); EXTRACT-MIN(H); UNION(H_1, H_2)
+ additionally, the Fibonacci heap supports:
  DECREASE-KEY(H, x, k); DELETE(H, x)
+ several preceding operations run in constant amortized time



Four pointers exist for each node: ***p***, ***child***, ***left***, ***right,*** permitting children of a node to be doubly linked as the child list.

Procedure	Binary heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT (to Root List)	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION (i.e., Merging)	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

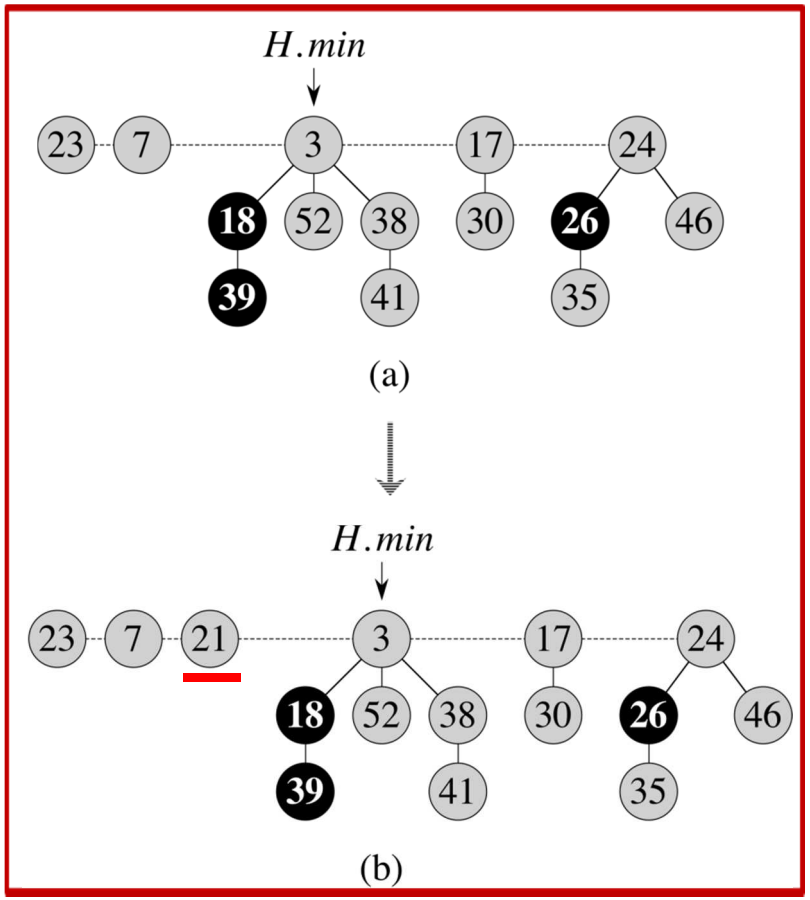
Max Binary Heap



Fibonacci Heaps (continued)

- **Mergeable heap operations**
  + inserting a node, with time complexity $O(1)$, by adding the node to the root list

```
FIB-HEAP-INSERT(H, x)
 1  x.degree = 0
 2  x.p = NIL
 3  x.child = NIL
 4  x.mark = FALSE
 5  if H.min == NIL
 6       create a root list for H containing just x
 7       H.min = x
 8  else insert x into H's root list
 9       if x.key < H.min.key
10            H.min = x
11  H.n = H.n + 1
```



Fibonacci Heaps (continued)

- **Uniting two Fibonacci heaps**
  + concatenating the root lists of H_1 and H_2
  + determining the new minimum node

FIB-HEAP-UNION(H_1, H_2)

```
1  H = MAKE-FIB-HEAP()
2  H.min = H1.min
3  concatenate the root list of H2 with the root list of H
4  if (H1.min == NIL) or (H2.min ≠ NIL and H2.min.key < H1.min.key)
5        H.min = H2.min
6  H.n = H1.n + H2.n
7  return H
```

Fibonacci Heaps (continued)

- **Extracting minimum node**
  + doing consolidation of trees on the root list, so that one root is left for each degree
  + calling CONSOLIDATE(*H*) to link trees of the same degree

Doing delayed work of consolidating trees in the root list

```
FIB-HEAP-EXTRACT-MIN(H)
 1  z = H.min
 2  if z ≠ NIL
 3       for each child x of z
 4            add x to the root list of H
 5            x.p = NIL
 6       remove z from the root list of H
 7       if z == z.right                  // z being the root of the only tree and having no child (since any child, if existing,
 8            H.min = NIL                      would then have been added to the root list)
 9       else H.min = z.right
10            CONSOLIDATE(H)
11       H.n = H.n − 1
12  return z
```

Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

Fibonacci Heaps (continued)

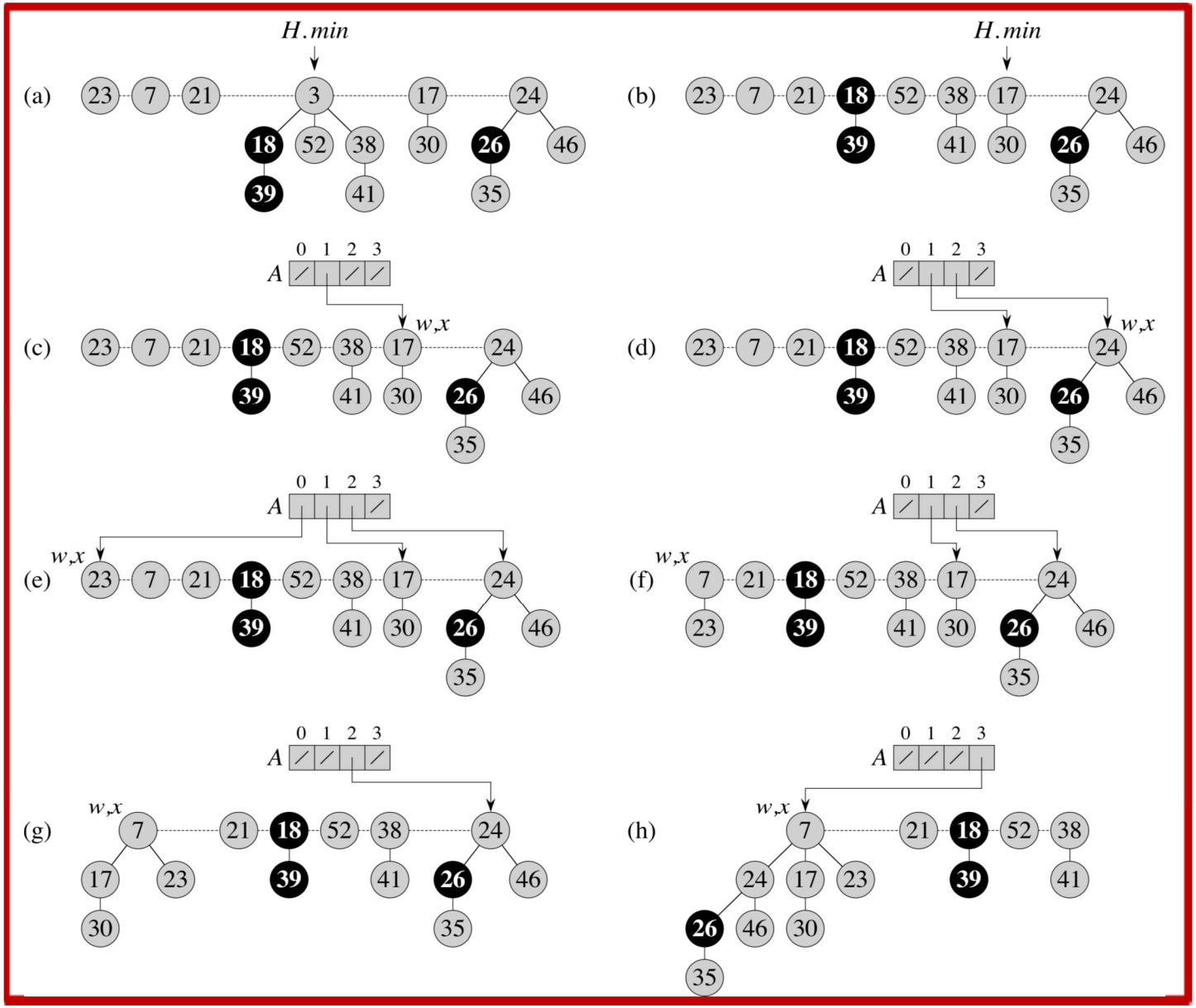
- **CONSOLIDATE(*H*)** for consolidating trees on the root list

```
CONSOLIDATE(H)
 1  let A[0..D(H.n)] be a new array      // D(H, n) being the largest degree of a constituent MIN heap
 2  for i = 0 to D(H.n)
 3      A[i] = NIL                       // auxiliary array initialization
 4  for each node w in the root list of H
 5      x = w
 6      d = x.degree
 7      while A[d] ≠ NIL
 8          y = A[d]                     // another node with the same degree as x existing
 9          if x.key > y.key
10              exchange x with y
11          FIB-HEAP-LINK(H, y, x)
12          A[d] = NIL
13          d = d + 1
14      A[d] = x
15  H.min = NIL
16  for i = 0 to D(H.n)                  // establish the root list and find H.min after consolidation
17      if A[i] ≠ NIL
18          if H.min == NIL
19              create a root list for H containing just A[i]
20              H.min = A[i]
21          else insert A[i] into H's root list
22              if A[i].key < H.min.key
23                  H.min = 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 x.degree
3  y.mark = FALSE
```

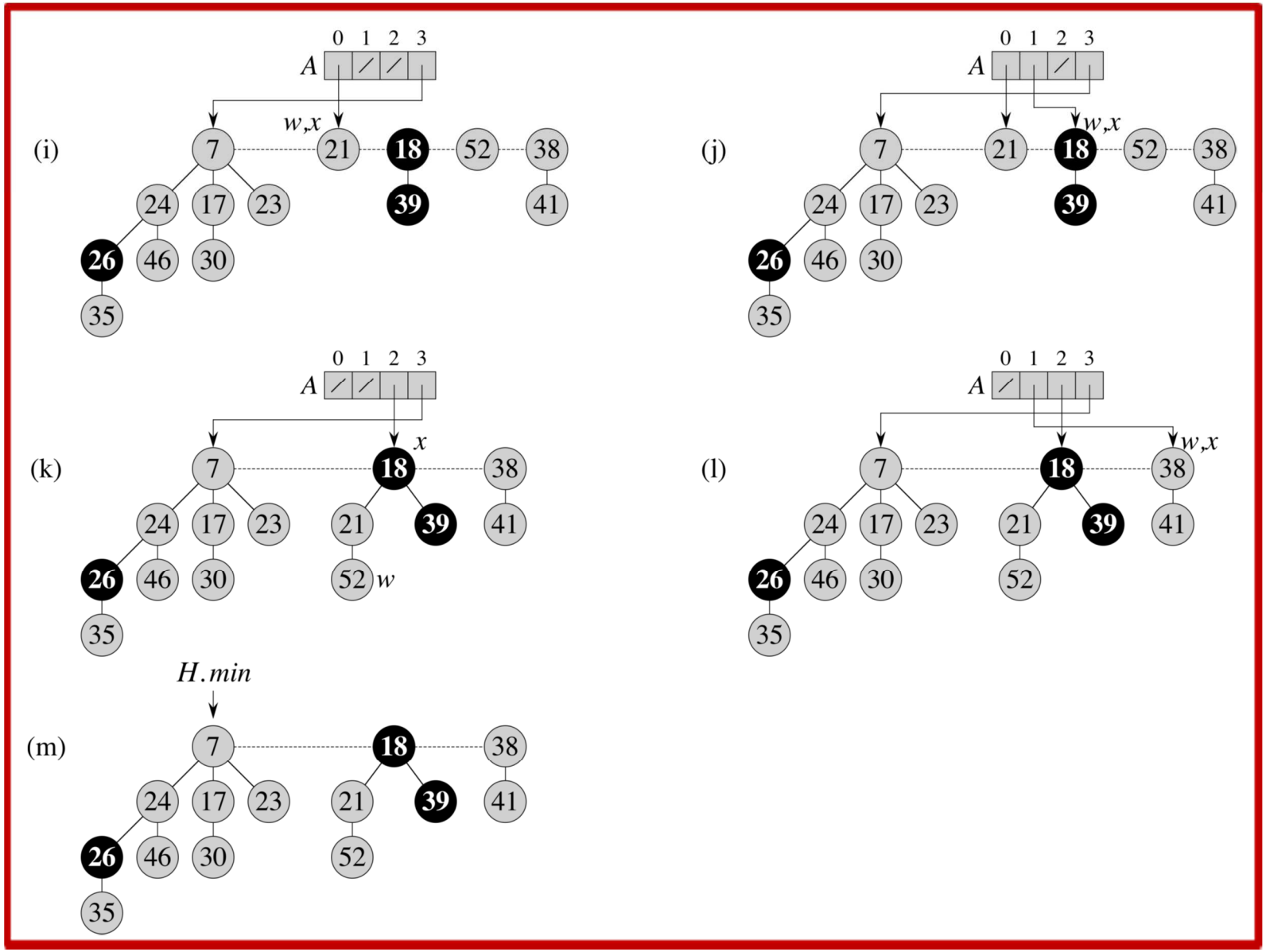
Fibonacci Heaps (continued)

- **Extracting minimum node**



Fibonacci Heaps (continued)

- **Extracting minimum node**



Fibonacci Heaps (continued)

- **Decreasing a key**

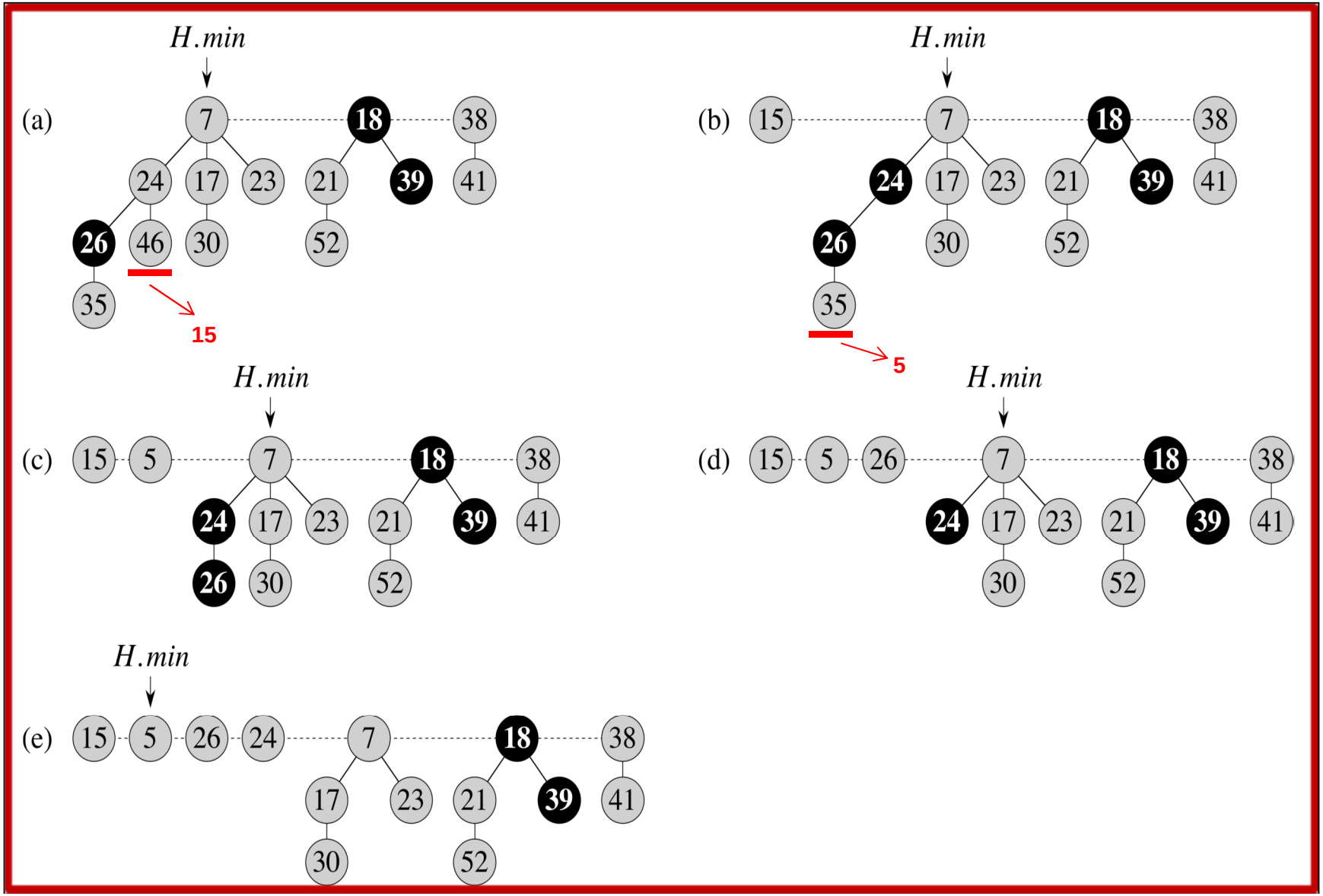
```
FIB-HEAP-DECREASE-KEY(H, x, k)
1  if k > x.key
2      error "new key is greater than current key"
3  x.key = k
4  y = x.p
5  if y ≠ NIL and x.key < y.key
6      CUT(H, x, y)
7      CASCADING-CUT(H, y)
8  if x.key < H.min.key
9      H.min = x

CUT(H, x, y)
1  remove x from the child list of y, decrementing y.degree
2  add x to the root list of H
3  x.p = NIL
4  x.mark = FALSE

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

Fibonacci Heaps (continued)

- **Decreasing a key**



Fibonacci Heaps (continued)

- **Deleting a key**

FIB-HEAP-DELETE(H, x)

FIB-HEAP-DECREASE-KEY(H, x, $-\infty$)
FIB-HEAP-EXTRACT-MIN(H)

Fibonacci Heaps (continued)

- **Heap Bounds**

Lemma 1.

Let x be any node in a Fibonacci heap and $k = x.degree$. We have size(x) $\geq F_{k+2} \geq \phi^k$, where $\phi = (1 + \sqrt{5})/2$.

Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …
D(k) ≥ 1, 2, 4, 8, 16, 32, 64, 128, ….

Lemma 2.

The maximum degree $D(x)$ of any node in an n-node Fibonacci heap is $O(\lg n)$.