5. Advanced Data Structures

Pukar Karki
Assistant Professor

Fibonacci Heaps

- The Fibonacci heap data structure serves a dual purpose.
  - **First,** it supports a set of operations that constitutes what is known as a “mergeable heap.”
  - **Second,** several Fibonacci-heap operations run in constant amortized time, which makes this data structure well suited for applications that invoke these operations frequently.

Fibonacci Heaps

Mergeable Heaps

- A mergeable heap is any data structure that supports the following five operations, in which each element has a key:

  MAKE-HEAP() creates and returns a new heap containing no elements.

  INSERT(H, x) inserts element x, whose key has already been filled in, into heap H.

  MINIMUM(H) returns a pointer to the element in heap H whose key is minimum.

  EXTRACT-MIN(H) deletes the element from heap H whose key is minimum, re-turning a pointer to the element.

  UNION(H1, H2) creates and returns a new heap that contains all the elements of heaps H1 and H2. Heaps H1 and H2 are “destroyed” by this operation.

Fibonacci Heaps

- In addition to the mergeable-heap operations above, Fibonacci heaps also support the following two operations:

  DECREASE-KEY(H, x, k) assigns to element x within heap H the new key value k, which we assume to be no greater than its current key value

  DELETE(H, x) deletes element x from heap H.

Fibonacci Heaps

Procedure	Binary heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

Fibonacci Heaps

Fibonacci Heaps in Theory and Practice

- From a theoretical standpoint, Fibonacci heaps are especially desirable when the number of EXTRACT-MIN and DELETE operations is small relative to the number of other operations performed.
- This situation arises in many applications.
- For example, some algorithms for graph problems may call DECREASE-KEY once per edge.
- For dense graphs, which have many edges, the **Θ(1)** amortized time of each call of DECREASE-KEY adds up to a big improvement over the **Θ(lg n)** worst-case time of binary heaps.

Fibonacci Heaps

Fibonacci Heaps in Theory and Practice

- From a practical point of view, however, the constant factors and programming complexity of Fibonacci heaps make them less desirable than ordinary binary (or k-ary) heaps for most applications, except for certain applications that manage large amounts of data.
- Thus, Fibonacci heaps are predominantly of theoretical interest.
- If a much simpler data structure with the same amortized time bounds as Fibonacci heaps were developed, it would be of practical use as well.

Fibonacci Heaps

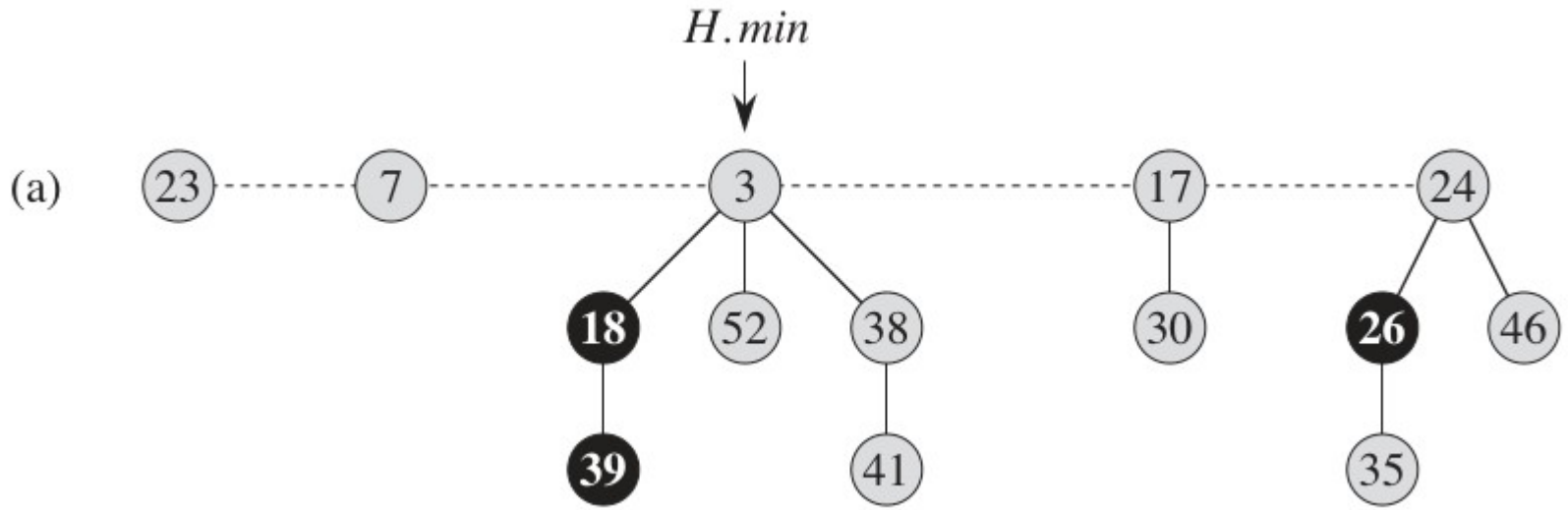
Fibonacci Heaps in Theory and Practice

- Both binary heaps and Fibonacci heaps are inefficient in how they support the operation SEARCH; it can take a while to find an element with a given key.
- For this reason, operations such as DECREASE-KEY and DELETE that refer to a given element require a pointer to that element as part of their input.

Fibonacci Heaps - Structure

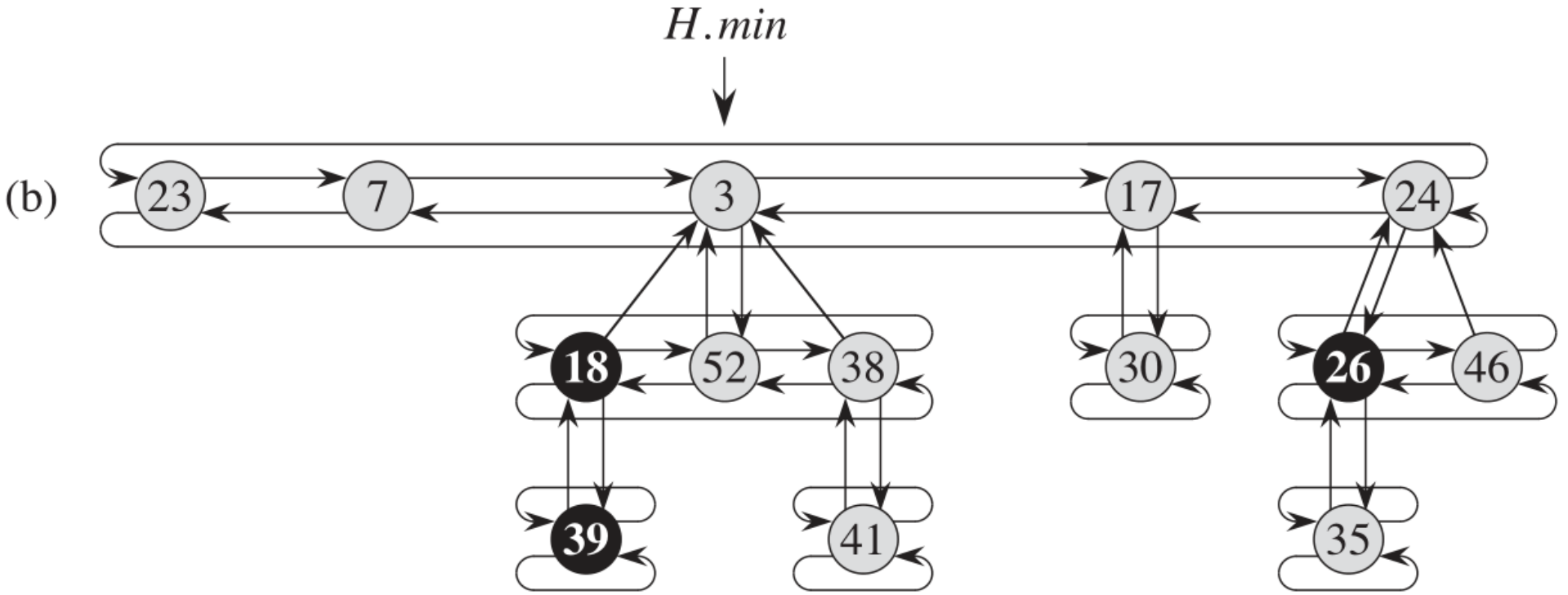
- A Fibonacci heap is a collection of rooted trees that are min-heap ordered.
- That is, each tree obeys the **min-heap property**:

  the key of a node is greater than or equal to the key of its parent.



(a) 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. Black nodes are marked.

Fibonacci Heaps - Structure



b) A more complete representation showing pointers p (up arrows), child (down arrows), and left and right (sideways arrows).

- Each node x contains a pointer x.p to its parent and a pointer x.child to any one of its children.
- The children of x are linked together in a circular, doubly linked list, which we call the child list of x.
- Each child y in a child list has pointers y.left and y.right that point to y's left and right siblings, respectively.
- If node y is an only child, then y.left = y.right = y.
- Siblings may appear in a child list in any order.

Fibonacci Heaps - Structure

- Circular, doubly linked lists have two advantages for use in Fibonacci heaps.
- First, we can insert a node into any location or remove a node from anywhere in a circular, doubly linked list in O(1) time.
- Second, given two such lists, we can concatenate them (or “splice” them together) into one circular, doubly linked list in O(1) time.

Fibonacci Heaps - Structure

- Each node has two other attributes.
- We store the number of children in the child list of node x in x.degree.
- The boolean-valued attribute x.mark indicates whether node x has lost a child since the last time x was made the child of another node.

Fibonacci Heaps - Structure

- Newly created nodes are unmarked, and a node x becomes unmarked whenever it is made the child of another node.
- Until we look at the DECREASE-KEY operation , we will just set all mark attributes to FALSE.

Fibonacci Heaps - Structure

- We access a given Fibonacci heap H by a pointer H.min to the root of a tree containing the minimum key; we call this node the minimum node of the Fibonacci heap.
- If more than one root has a key with the minimum value, then any such root may serve as the minimum node.
- When a Fibonacci heap H is empty, H.min is NIL.

Fibonacci Heaps - Structure

- The roots of all the trees in a Fibonacci heap are linked together using their left and right pointers into a circular, doubly linked list called the root list of the Fibonacci heap.
- The pointer H.min thus points to the node in the root list whose key is minimum.
- Trees may appear in any order within a root list.
- We rely on one other attribute for a Fibonacci heap H: H.n, the number of nodes currently in H.

Fibonacci Heaps - Structure

- We shall use the potential method 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.
- We then define the potential $\phi(H)$ of Fibonacci heap H by

$$\phi(H) = t(H) + 2\, m(H)$$

Mergeable-Heap operations

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 H.n = 0 and H.min = NIL; there are no trees in H.
- Because t(H) = 0 and m(H) = 0, the potential of the empty Fibonacci heap is $\phi(H) = 0$.
- The amortized cost of MAKE-FIB-HEAP is thus equal to its O(1) actual cost.

Mergeable-Heap operations

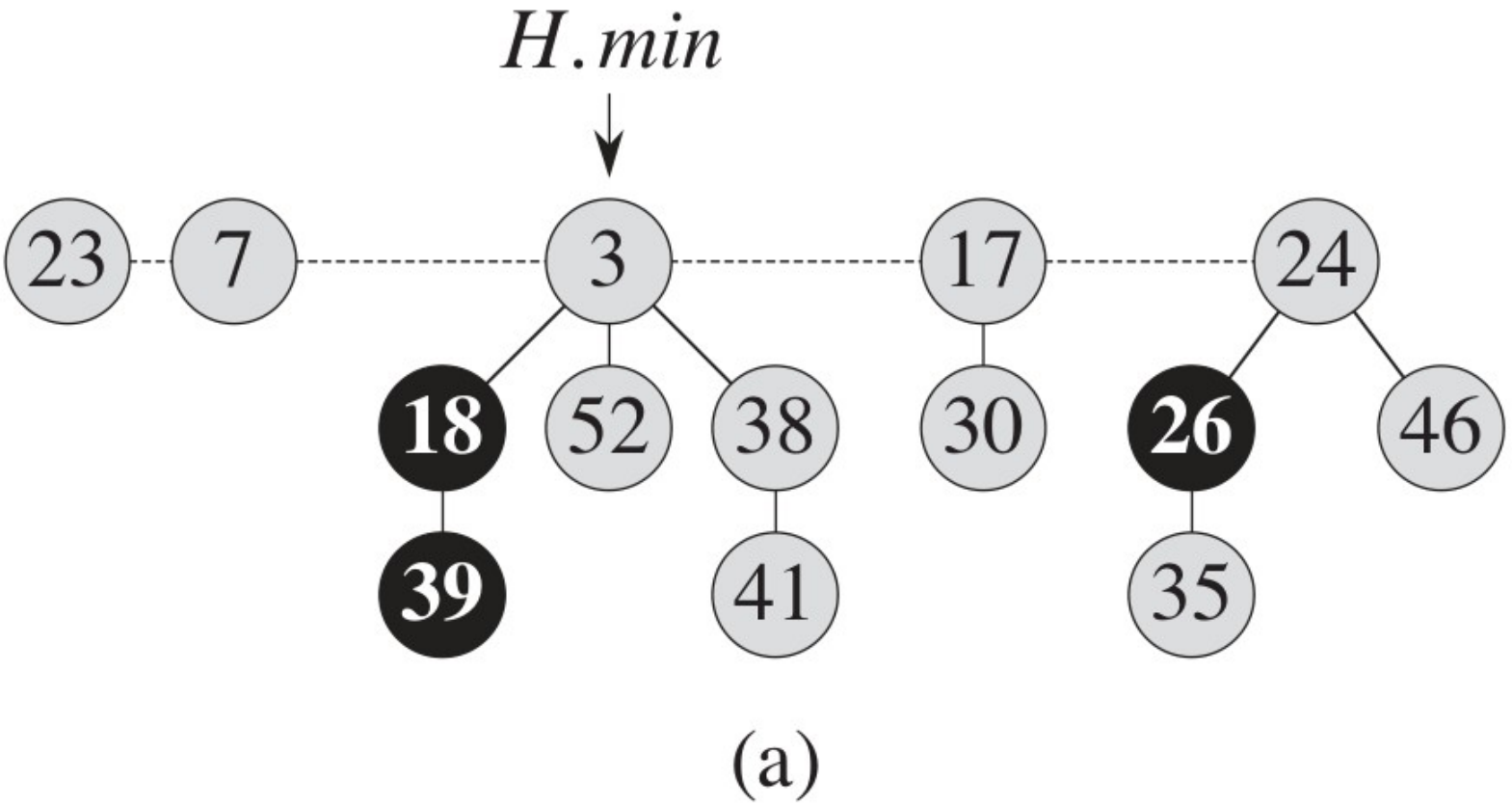
Inserting a node

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

```
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
```

Mergeable-Heap operations

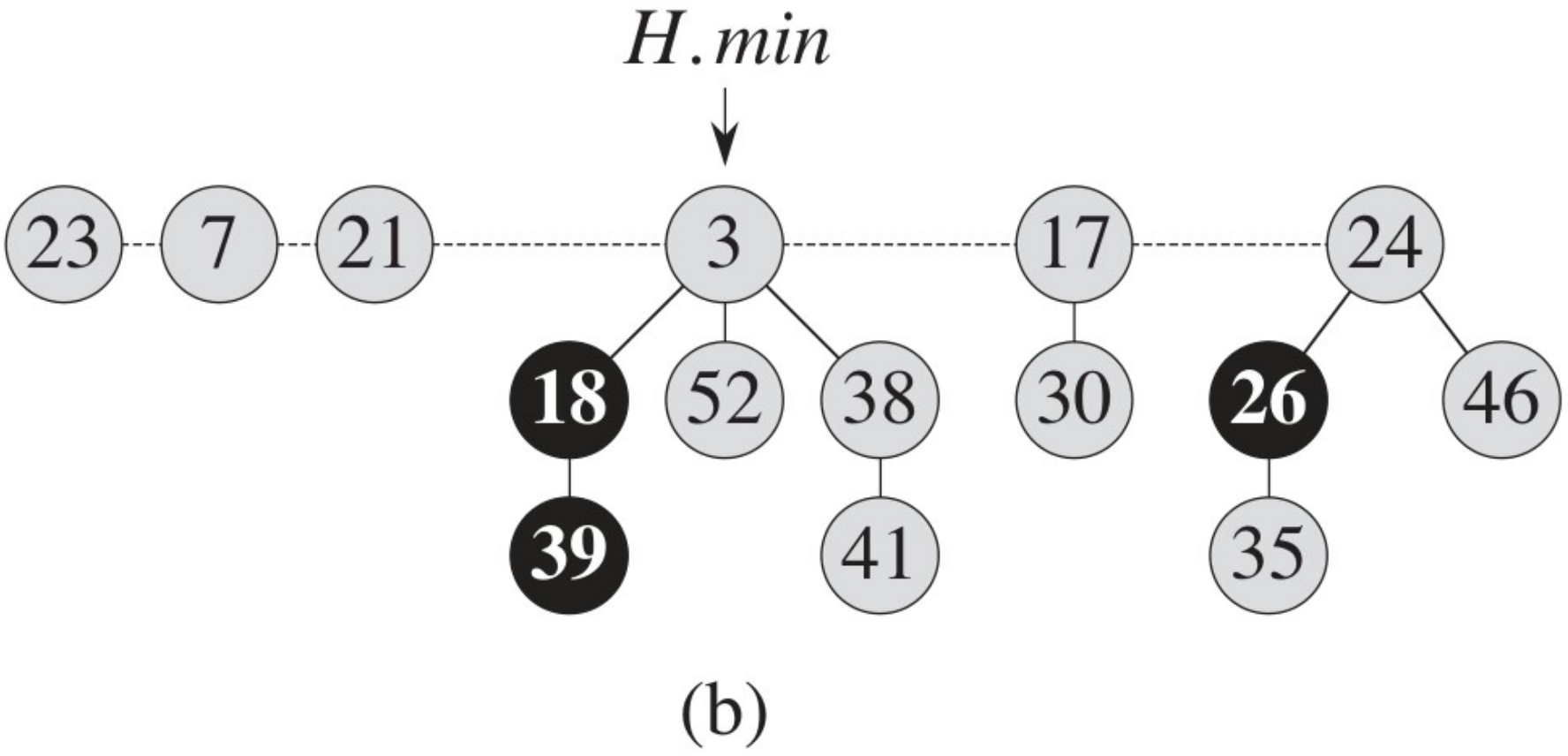
Inserting a node



(a) A Fibonacci heap H.

Mergeable-Heap operations

Inserting a node.



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

Mergeable-Heap operations

Inserting a node

```
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
```

- Lines 1–4 initialize some of the structural attributes of node x.
- Line 5 tests to see whether Fibonacci heap H is empty.
- If it is, then lines 6–7 make x be the only node in H's root list and set H.min to point to x.
- Otherwise, lines 8–10 insert x into H's root list and update H.min if necessary.
- Finally, line 11 increments H.n to reflect the addition of the new node.

Mergeable-Heap operations

Inserting a node

```
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
```

- 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) + 2 m(H)) -
  (t(H) + 2 m(H))
  = 1
- Since the actual cost is O(1), the amortized cost is O(1) + 1
  = O(1).

Mergeable-Heap operations

Finding the minimum node

- The minimum node of a Fibonacci heap H is given by the pointer H.min, 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.

Mergeable-Heap operations

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.
- Afterward, the objects representing H_1 and H_2 will never be used again.

Mergeable-Heap operations

Uniting two Fibonacci heaps

```
FIB-HEAP-UNION(H1, H2)
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
```

Mergeable-Heap operations

Uniting two Fibonacci heaps

```
FIB-HEAP-UNION(H1, H2)
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
```

- Lines 1–3 concatenate the root lists of H_1 and H_2 into a new root list H.
- Lines 2, 4, and 5 set the minimum node of H, and line 6 sets H:n to the total number of nodes.
- Line 7 returns the resulting Fibonacci heap H.

Mergeable-Heap operations

Uniting two Fibonacci heaps

```
FIB-HEAP-UNION(H1, H2)
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
```

- As in the FIB-HEAP- INSERT procedure, all roots remain roots.
- The change in potential is

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

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.

Mergeable-Heap operations

Extracting the Minimum Node

```
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
 8            H.min = NIL
 9       else H.min = z.right
10            CONSOLIDATE(H)
11       H.n = H.n − 1
12  return z
```

Mergeable-Heap operations

```
CONSOLIDATE(H)
 1  let A[0..D(H.n)] be a new array
 2  for i = 0 to D(H.n)
 3      A[i] = NIL
 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
 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)
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]
```

Mergeable-Heap operations

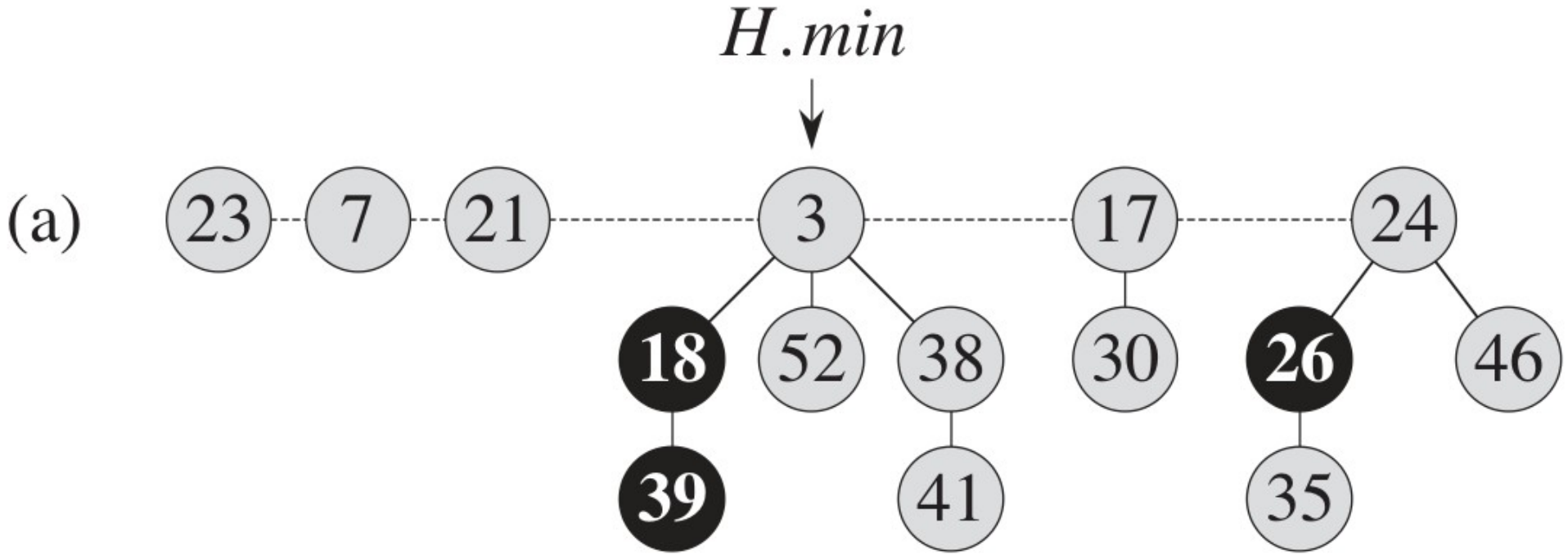
Extracting the Minimum Node

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 $y.mark = \text{FALSE}$

Mergeable-Heap operations

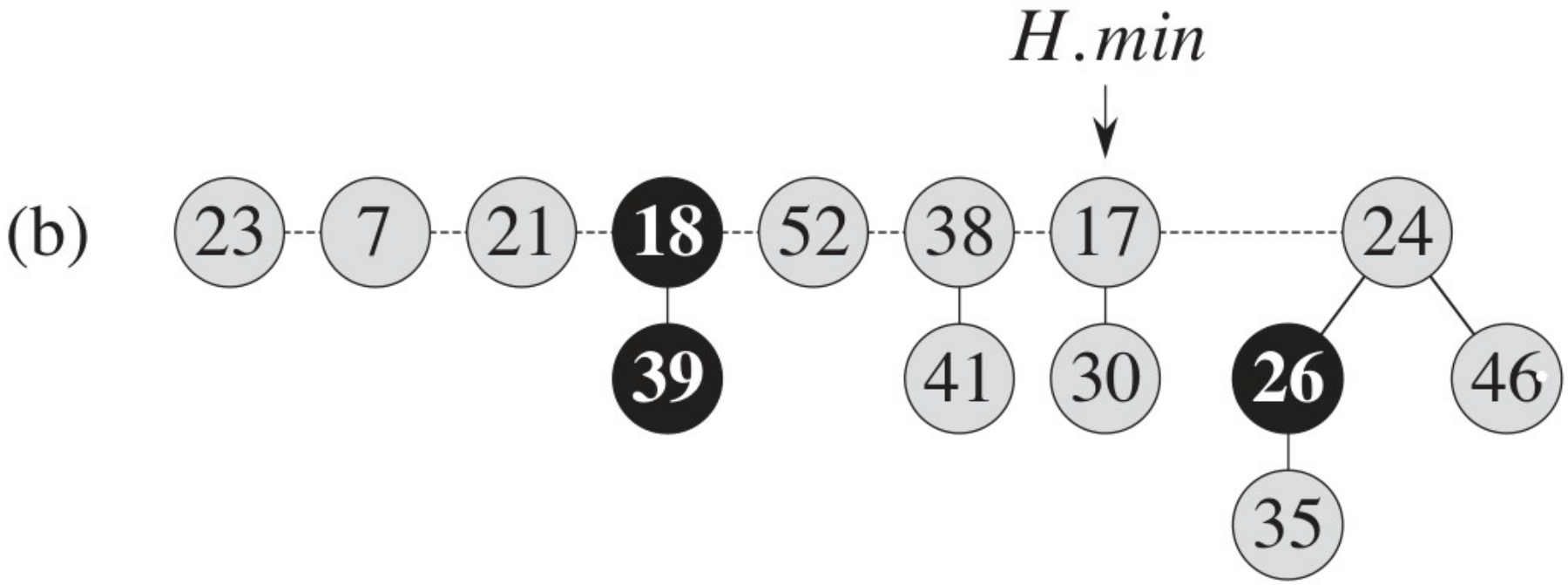
Extracting the Minimum Node



(a) A Fibonacci heap H

Mergeable-Heap operations

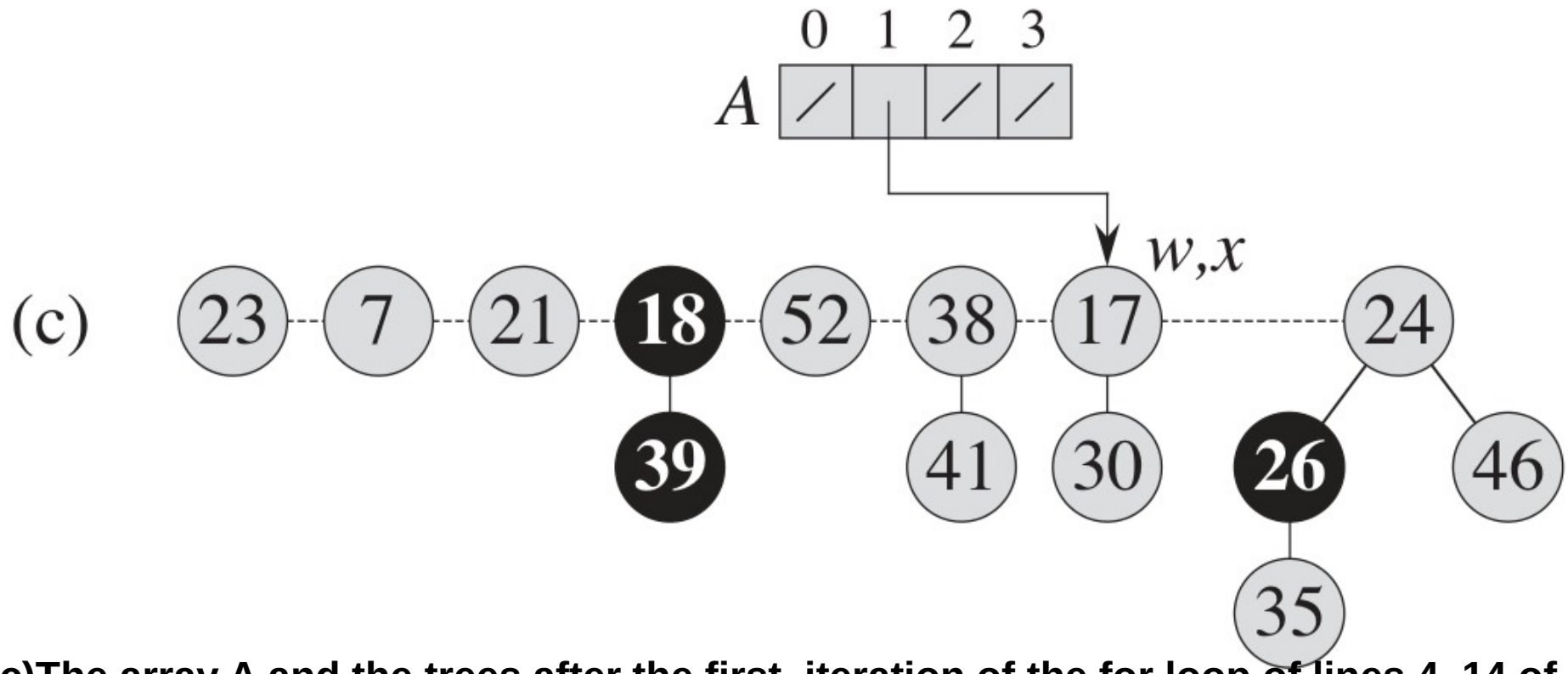
Extracting the Minimum Node



(b) The situation after removing the minimum node z from the root list and adding its children to the root list

Mergeable-Heap operations

Extracting the Minimum Node

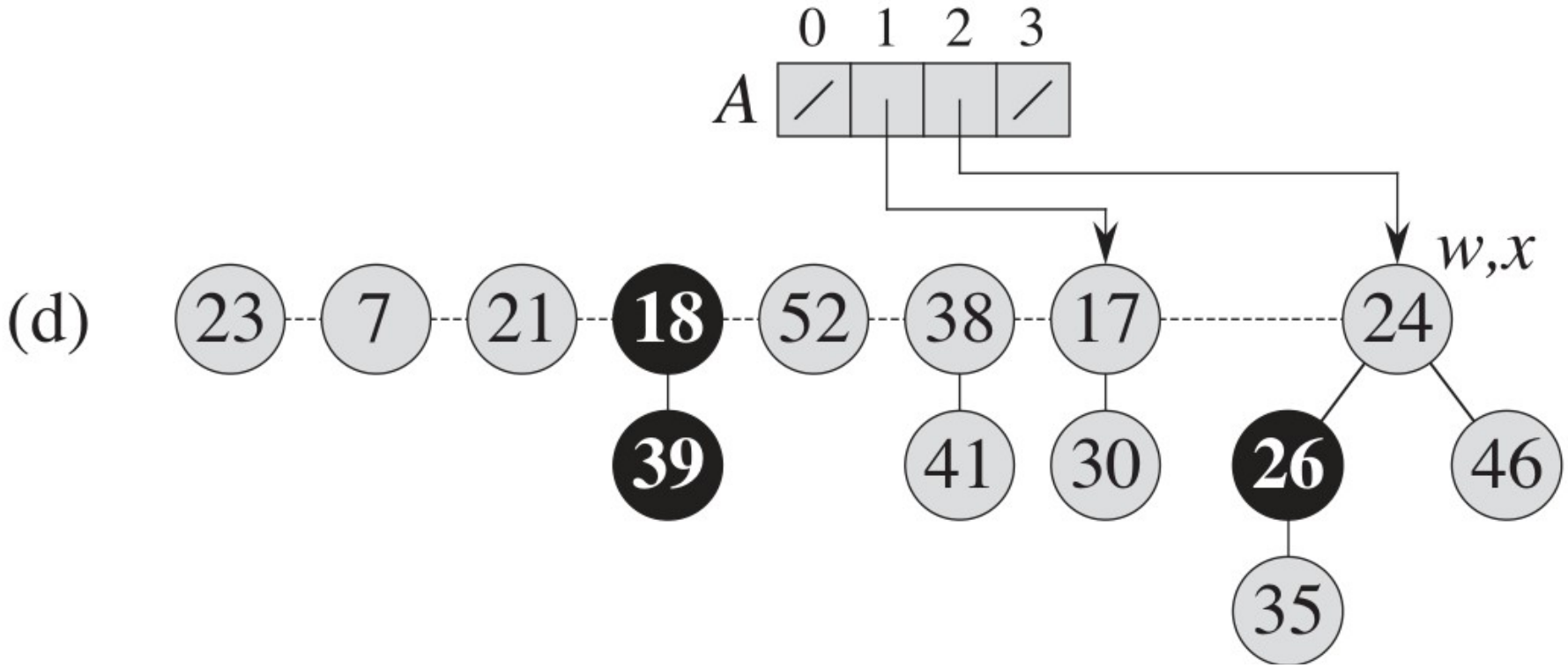


(c)The array A and the trees after the first iteration of the for loop of lines 4–14 of the procedure CONSOLIDATE.

The procedure processes the root list by starting at the node pointed to by H.min and following right pointers. Each part shows the values of w and x at the end of an iteration.

Mergeable-Heap operations

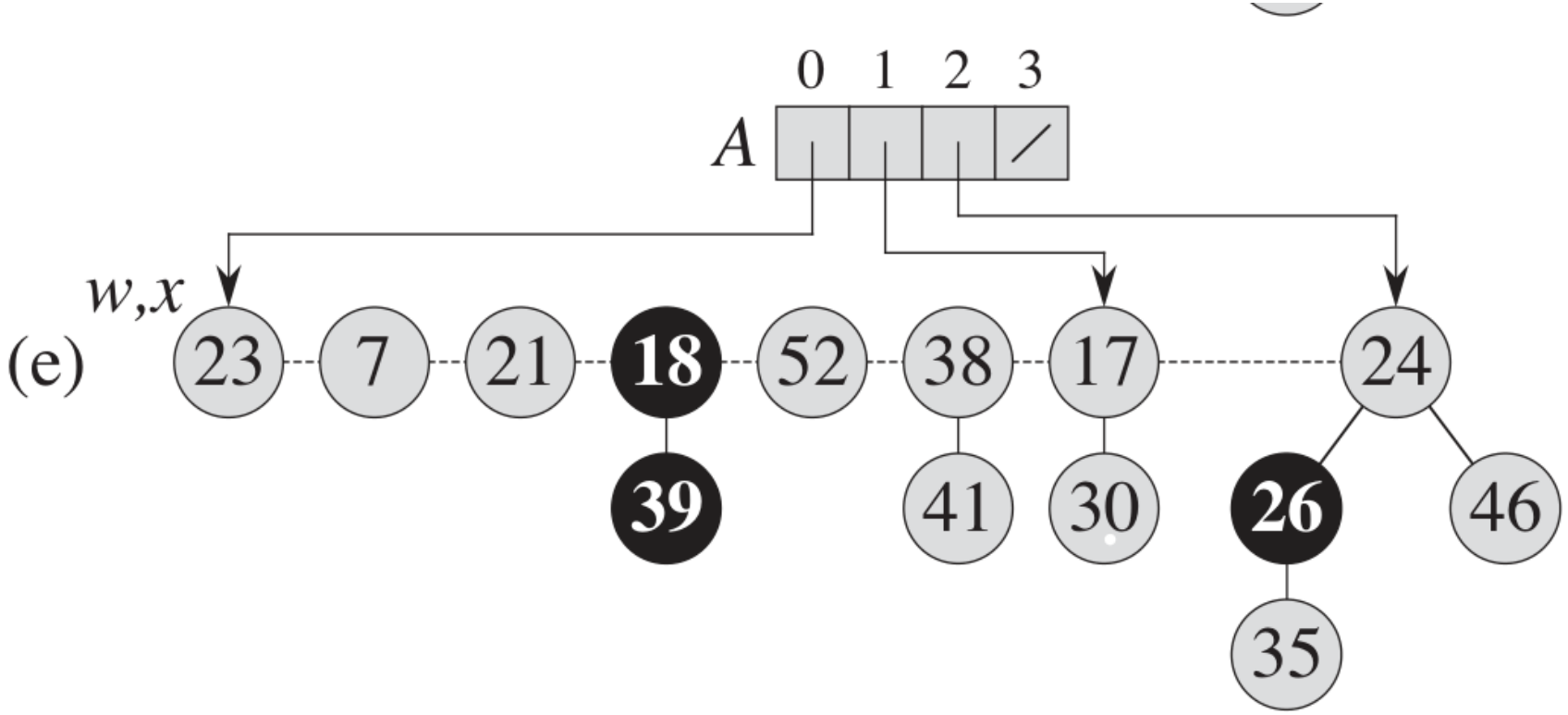
Extracting the Minimum Node



(d)The array A and the trees after the second iteration of the for loop of lines 4–14 of the procedure CONSOLIDATE.

Mergeable-Heap operations

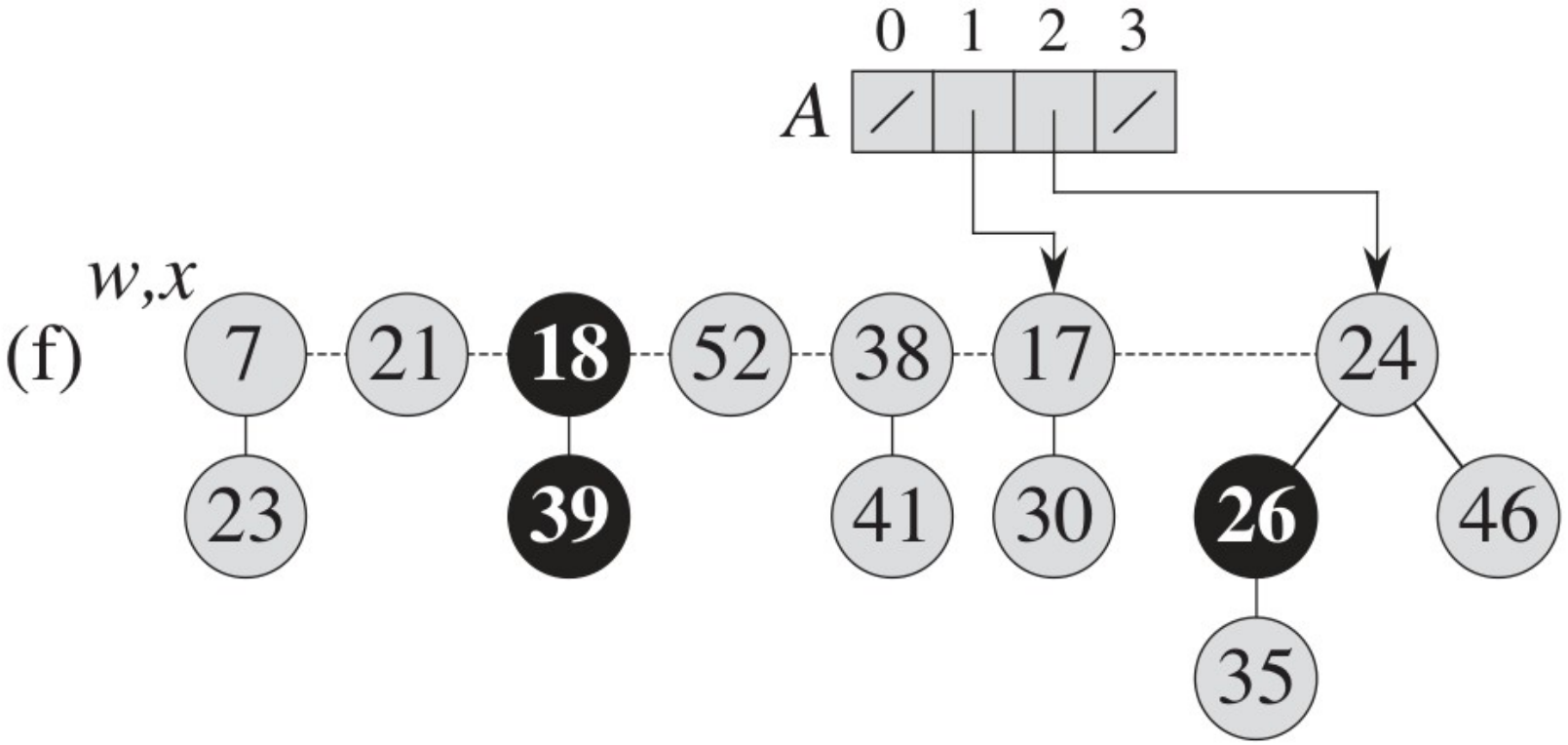
Extracting the Minimum Node



(e)The array A and the trees after the third iteration of the for loop of lines 4–14 of the procedure CONSOLIDATE.

Mergeable-Heap operations

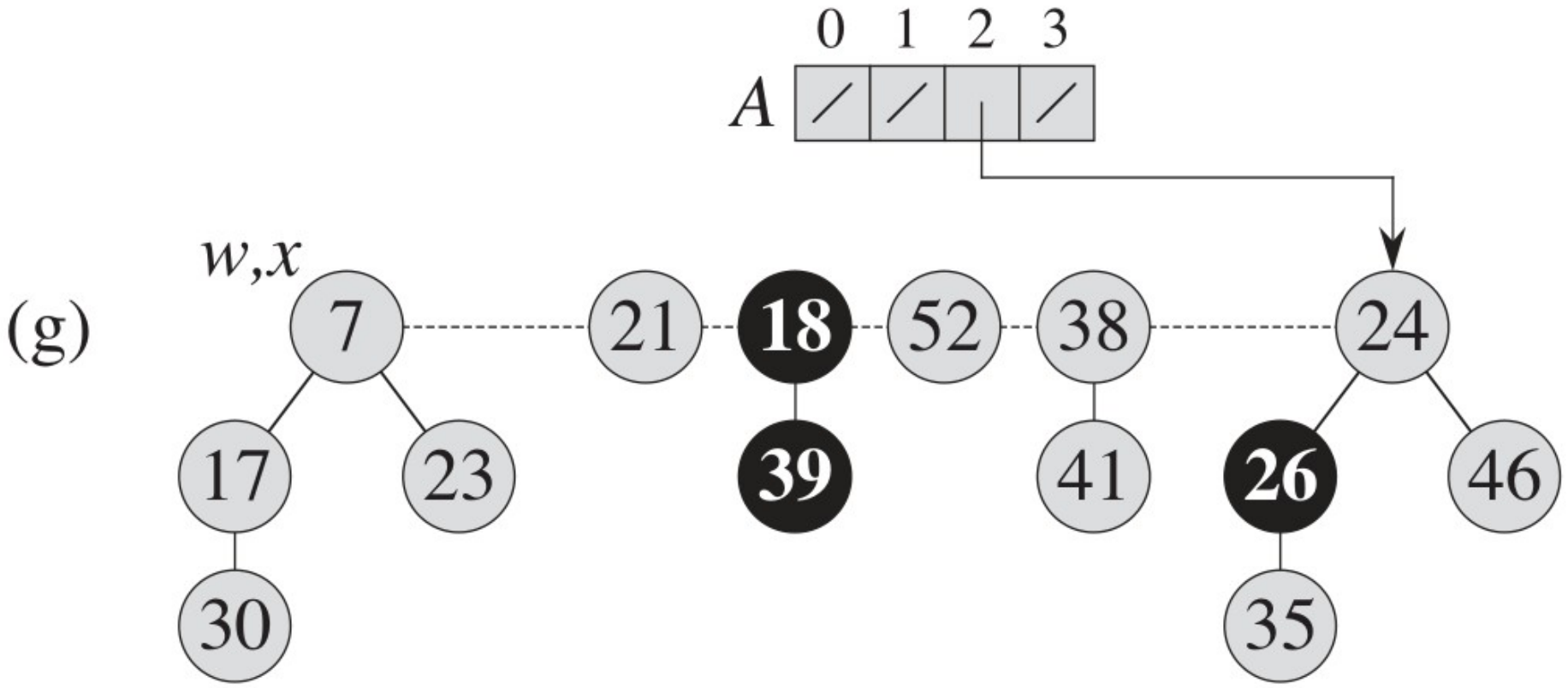
Extracting the Minimum Node



(f)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 7–13.

Mergeable-Heap operations

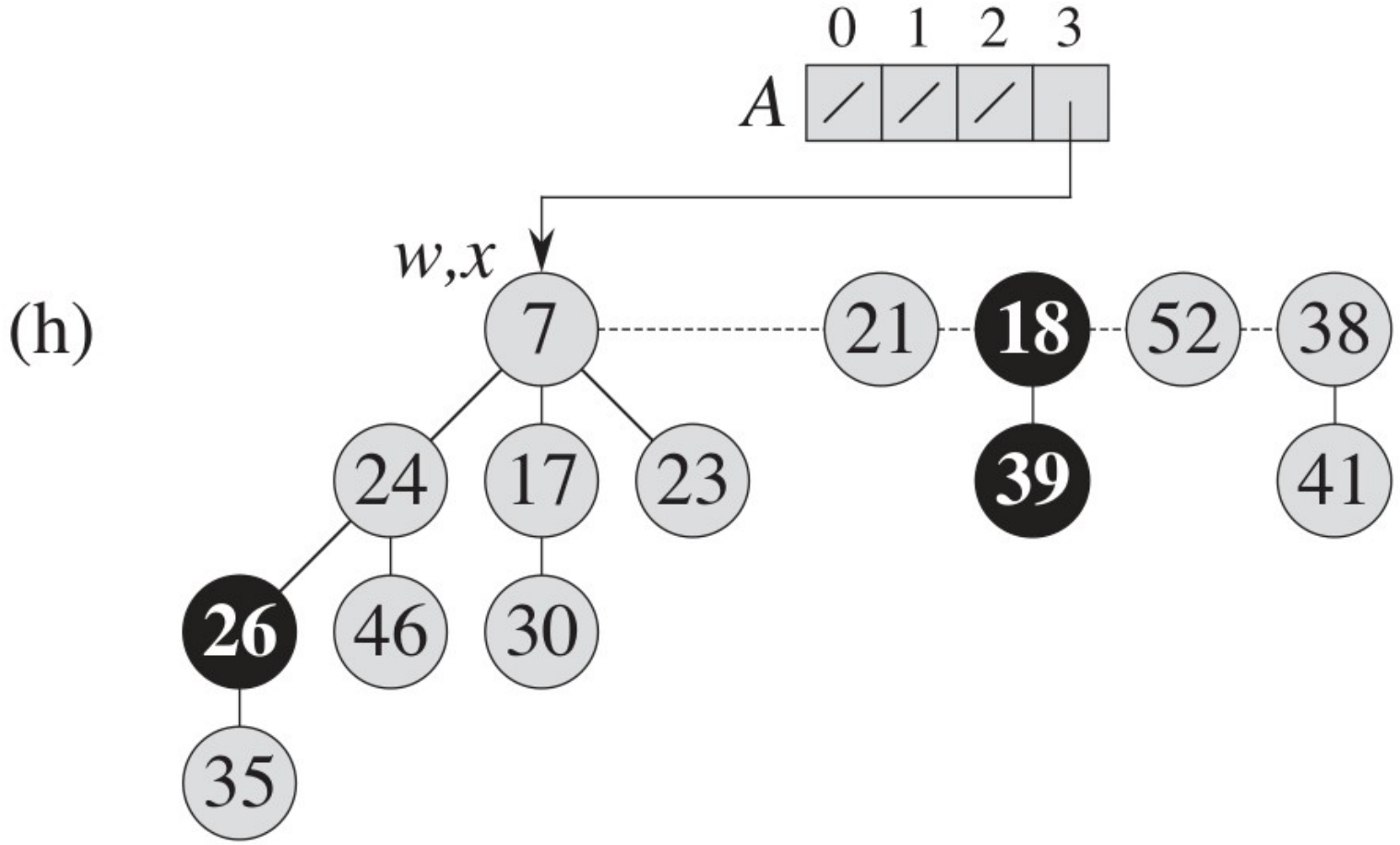
Extracting the Minimum Node



(g)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 7–13.

Mergeable-Heap operations

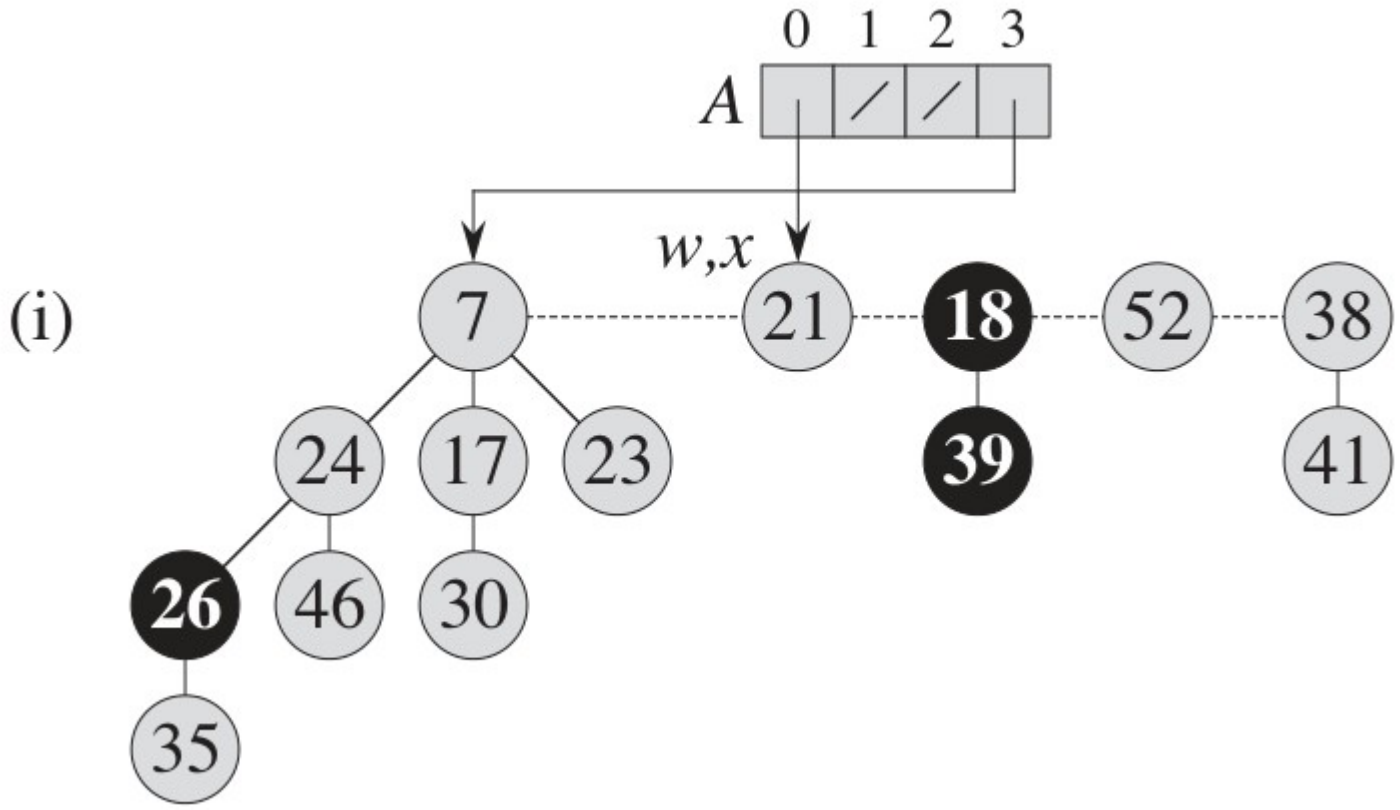
Extracting the Minimum Node



(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 7–13.

Mergeable-Heap operations

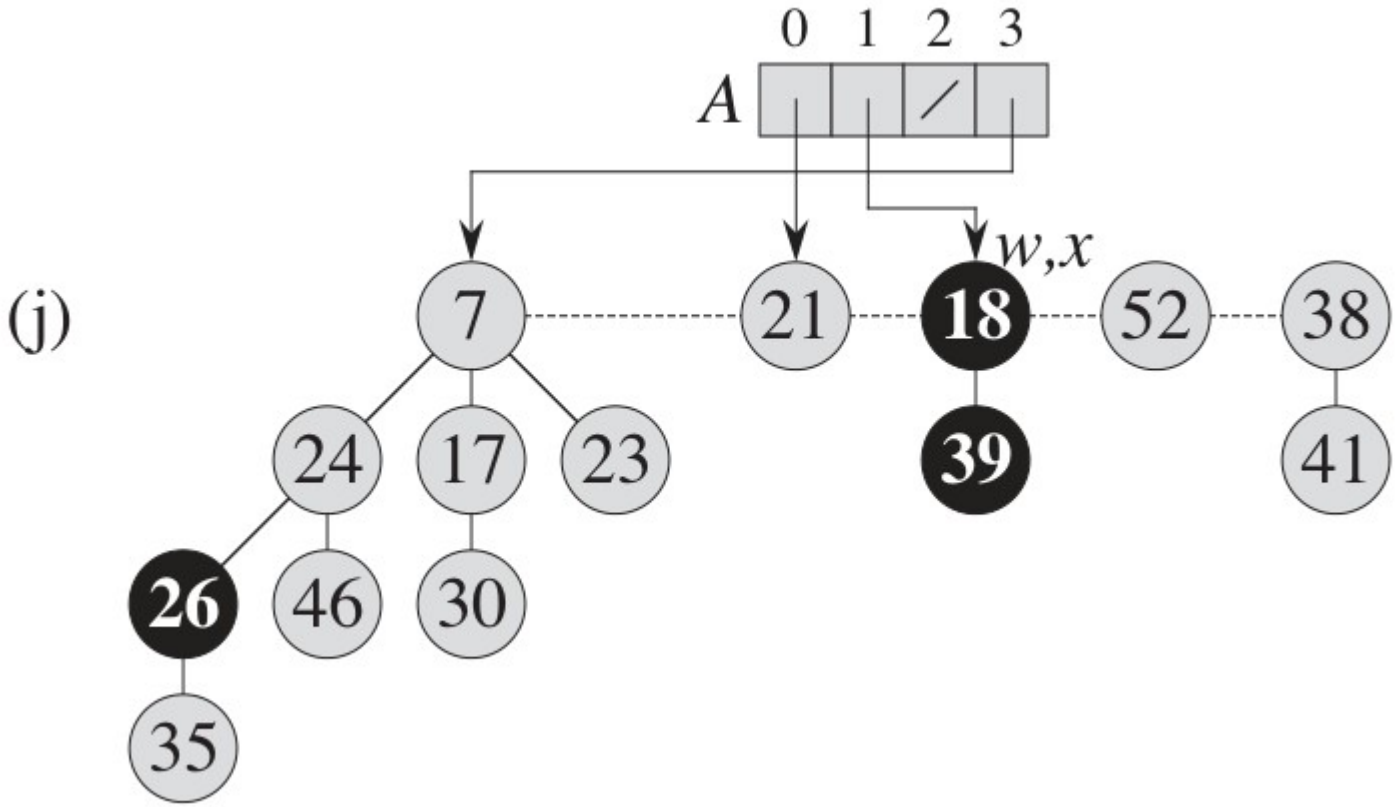
Extracting the Minimum Node



(i to l)The situation after each of the next four iterations of the for loop.

Mergeable-Heap operations

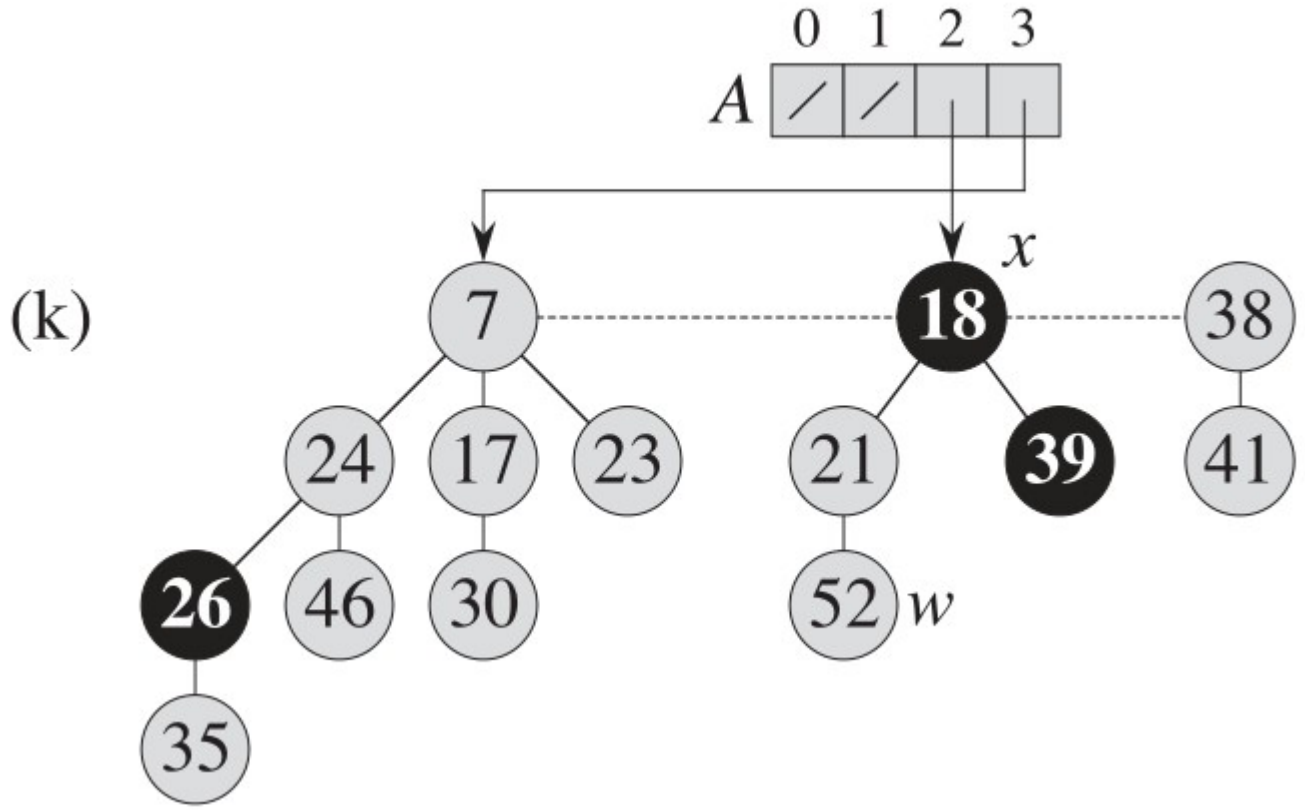
Extracting the Minimum Node



(i to l)The situation after each of the next four iterations of the for loop.

Mergeable-Heap operations

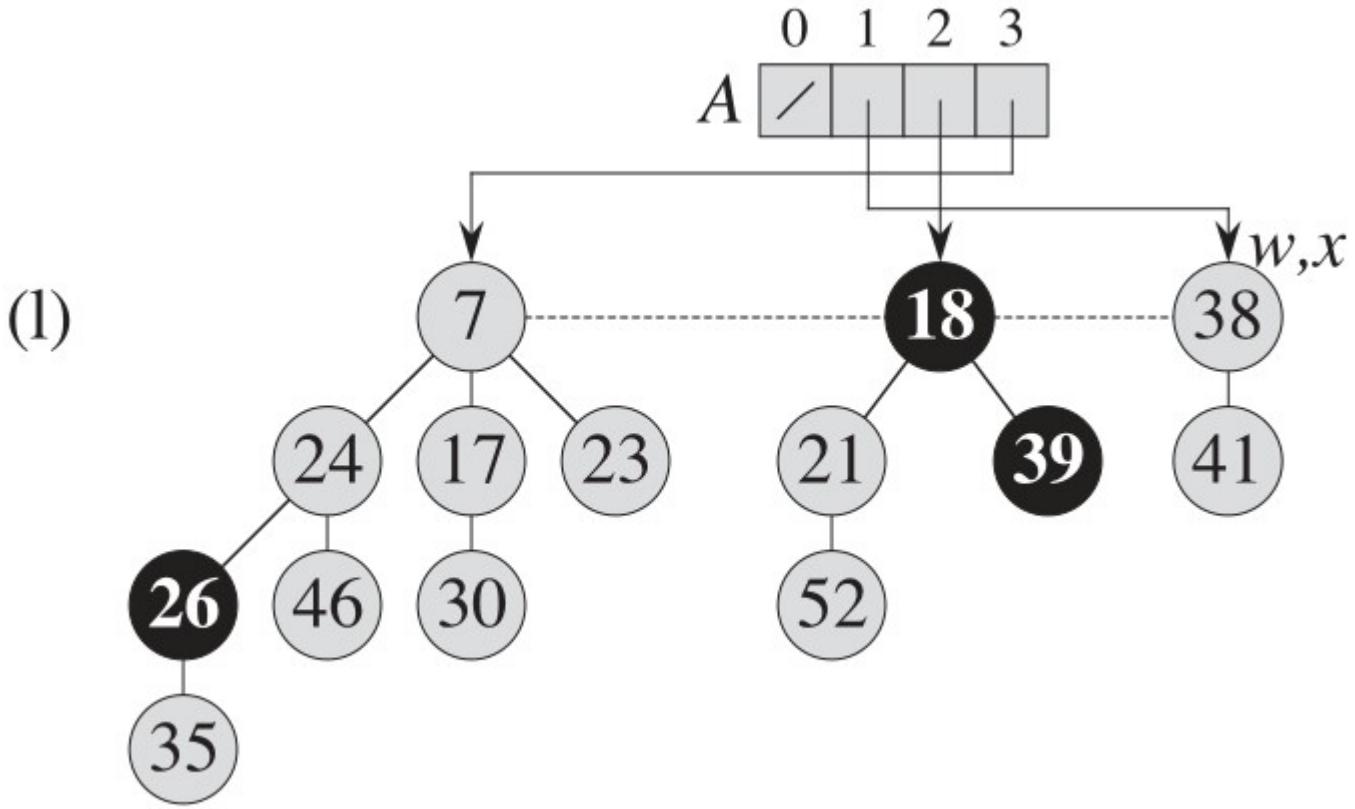
Extracting the Minimum Node



(i to l)The situation after each of the next four iterations of the for loop.

Mergeable-Heap operations

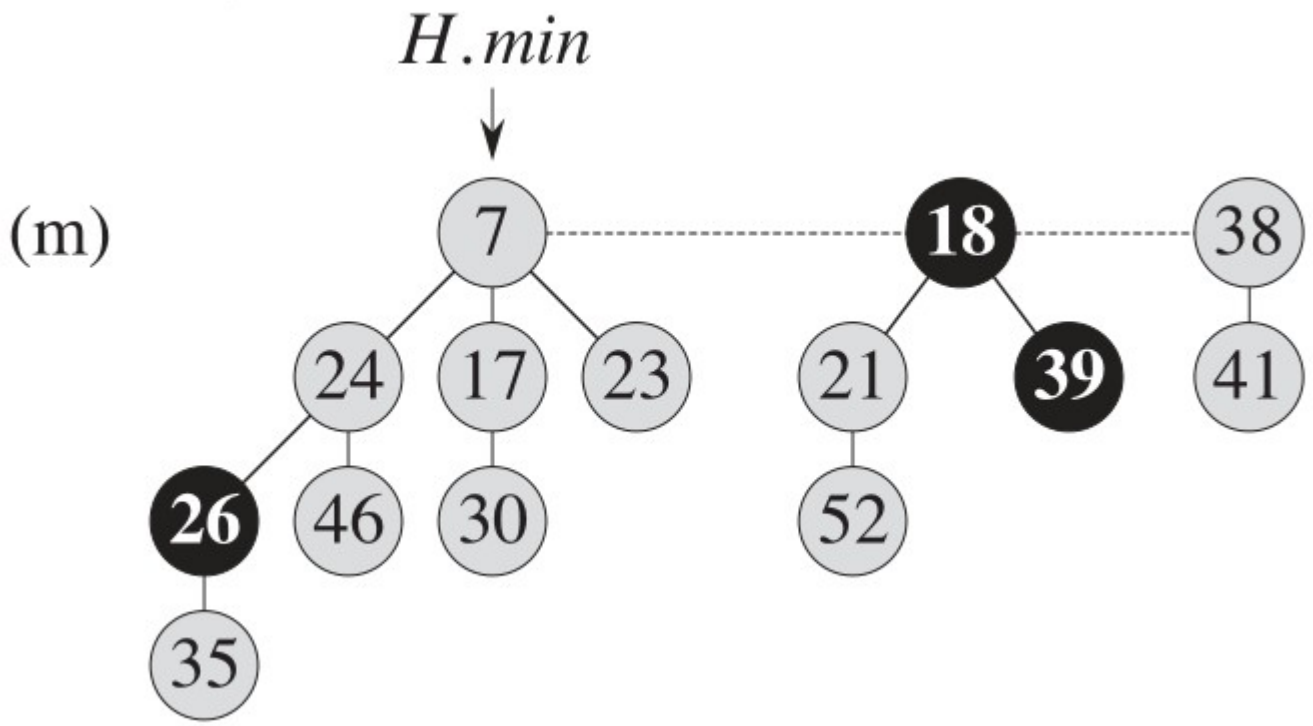
Extracting the Minimum Node



(i to l)The situation after each of the next four iterations of the for loop.

Mergeable-Heap operations

Extracting the Minimum Node



(m) Fibonacci heap H after reconstructing the root list from the array A and determining the new H.min pointer.

Mergeable-Heap operations

Decreasing a Key

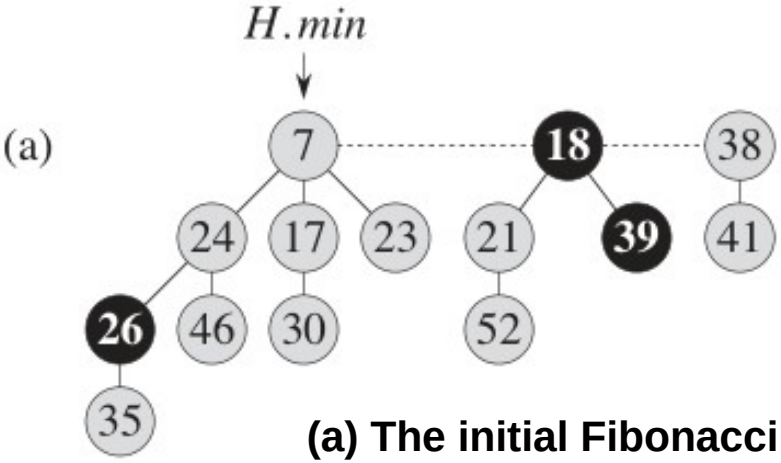
```
FIB-HEAP-DECREASE-KEY(H, x, k)
1  if k > x.key
2       error "new key is greater than current key"
3  x.key = k
4  y = x.p
5  if y ≠ NIL and x.key < y.key
6       CUT(H, x, y)
7       CASCADING-CUT(H, y)
8  if x.key < H.min.key
9       H.min = x
```

Mergeable-Heap operations

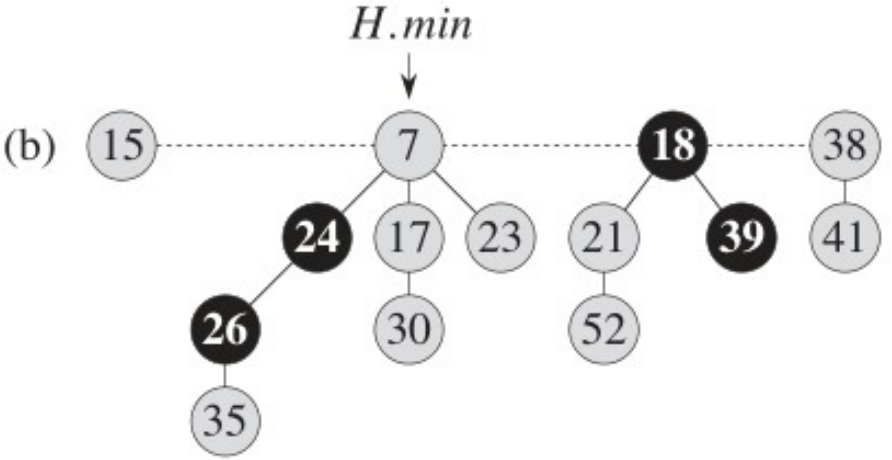
Decreasing a Key

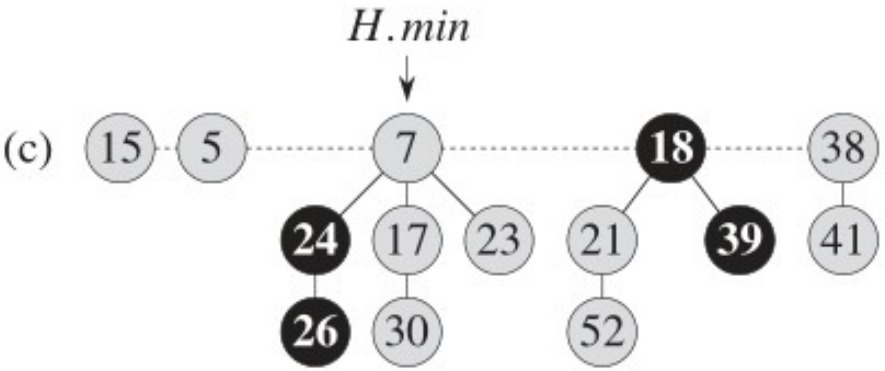
```
CUT(H, x, y)
1  remove x from the child list of y, decrementing y.degree
2  add x to the root list of H
3  x.p = NIL
4  x.mark = FALSE
```

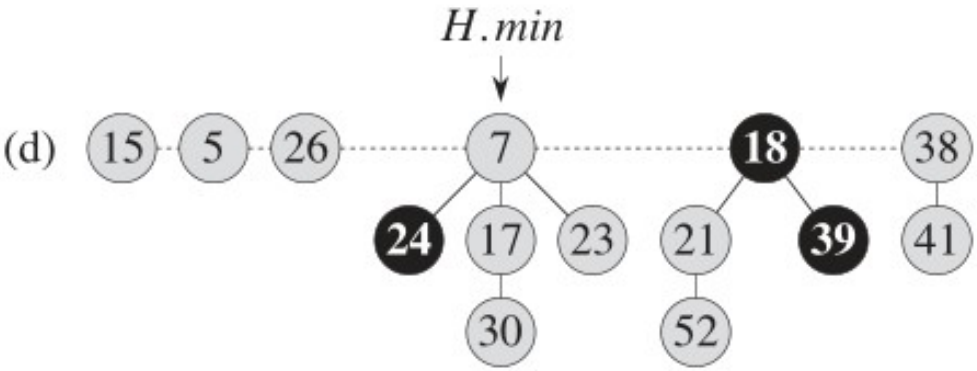
```
CASCADING-CUT(H, y)
1  z = y.p
2  if z ≠ NIL
3       if y.mark == FALSE
4            y.mark = TRUE
5       else CUT(H, y, z)
6            CASCADING-CUT(H, z)
```

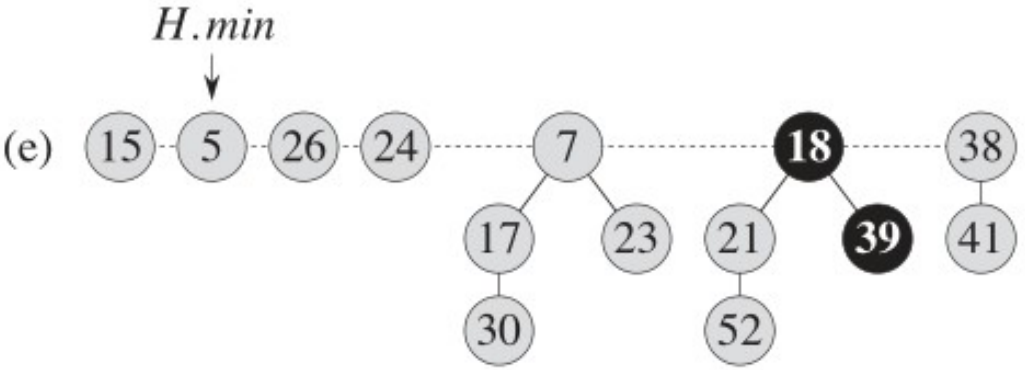


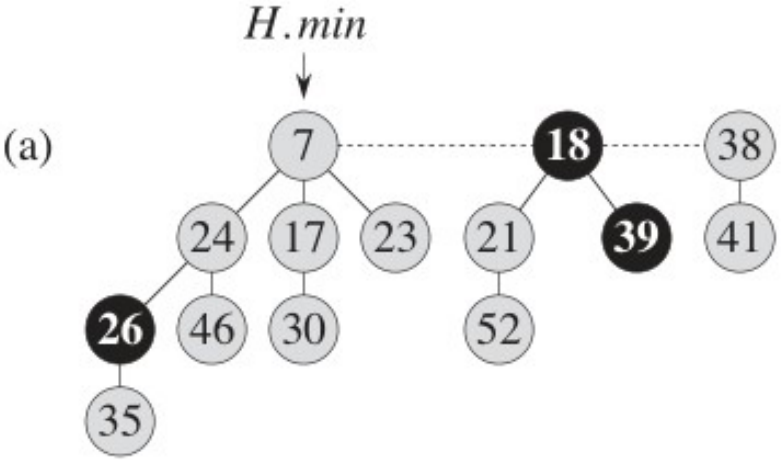
(a) The initial Fibonacci heap.

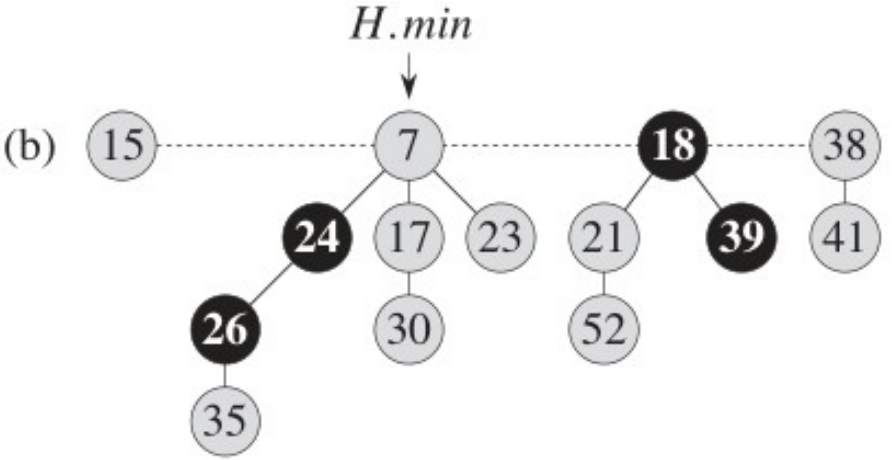


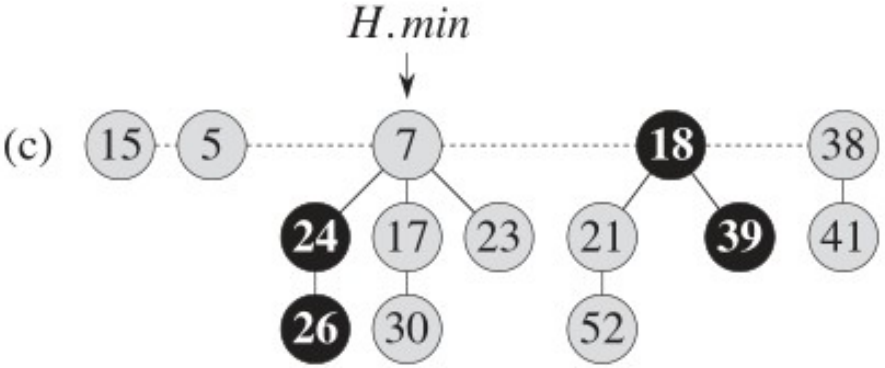


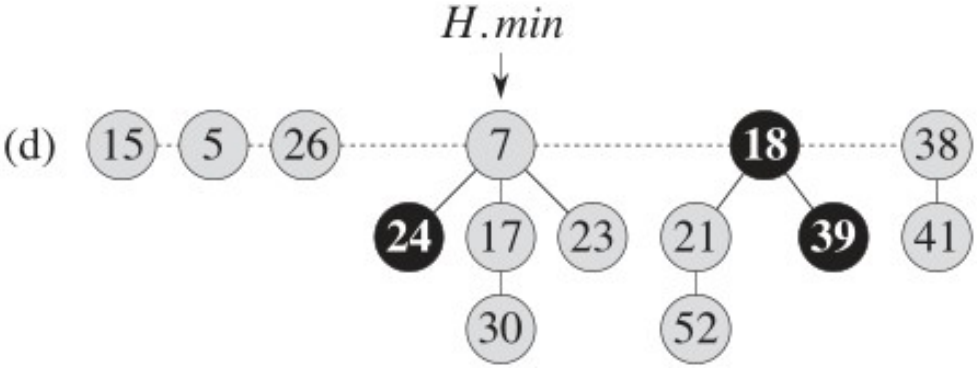


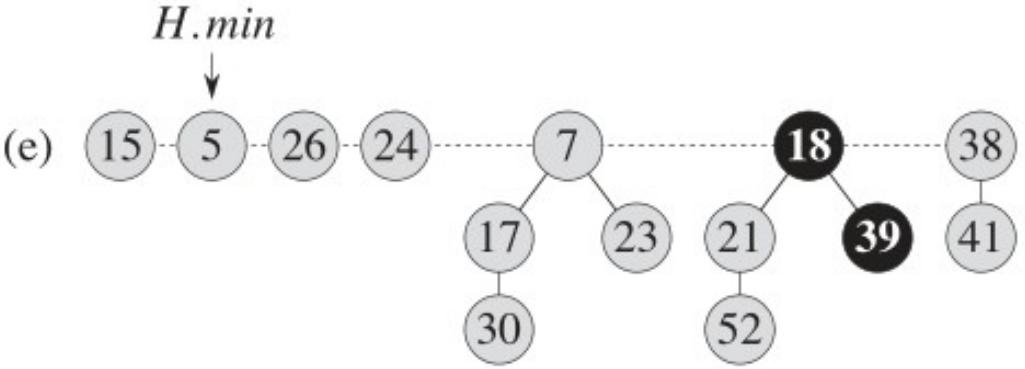




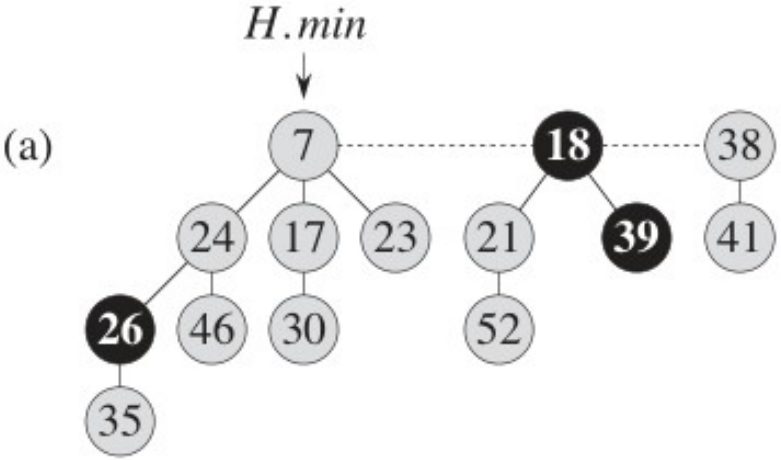


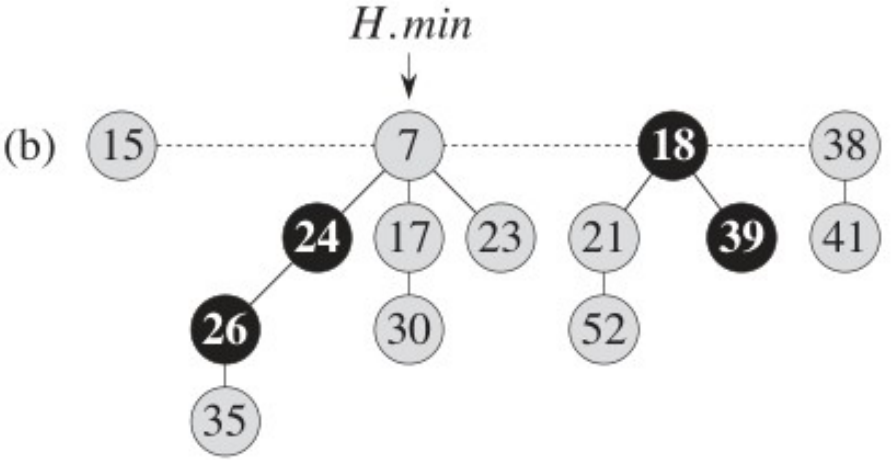


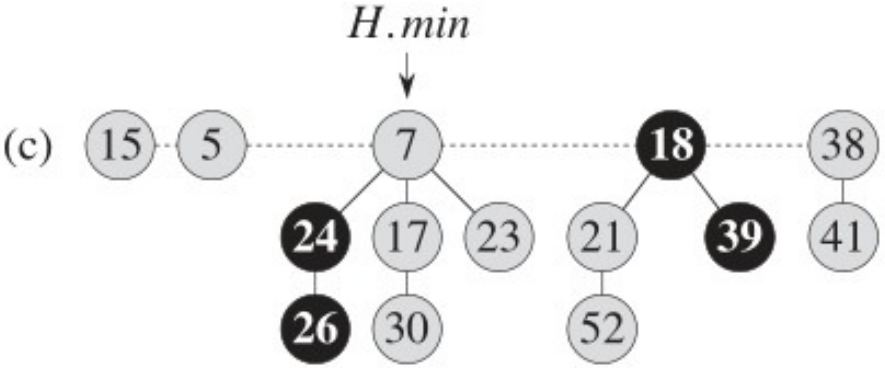


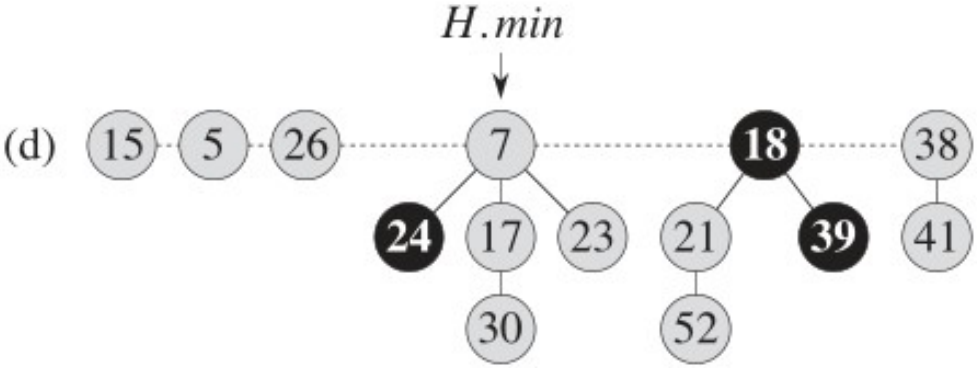


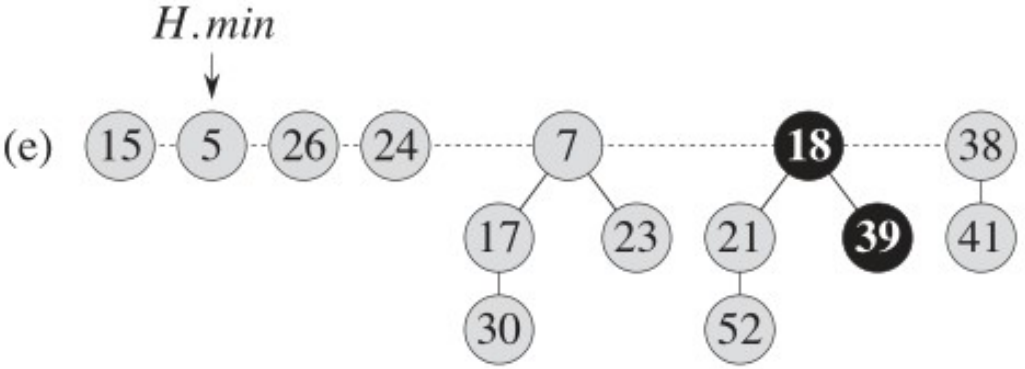
(b) The node with key 46 has its key decreased to 15.
The node becomes a root, and its parent (with key 24), which had previously been unmarked, becomes marked.











- **(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.)**
- **Part (e) shows the result of the FIB-HEAP-DECREASE-KEY operation, with H:min pointing to the new minimum node.**

Mergeable-Heap operations

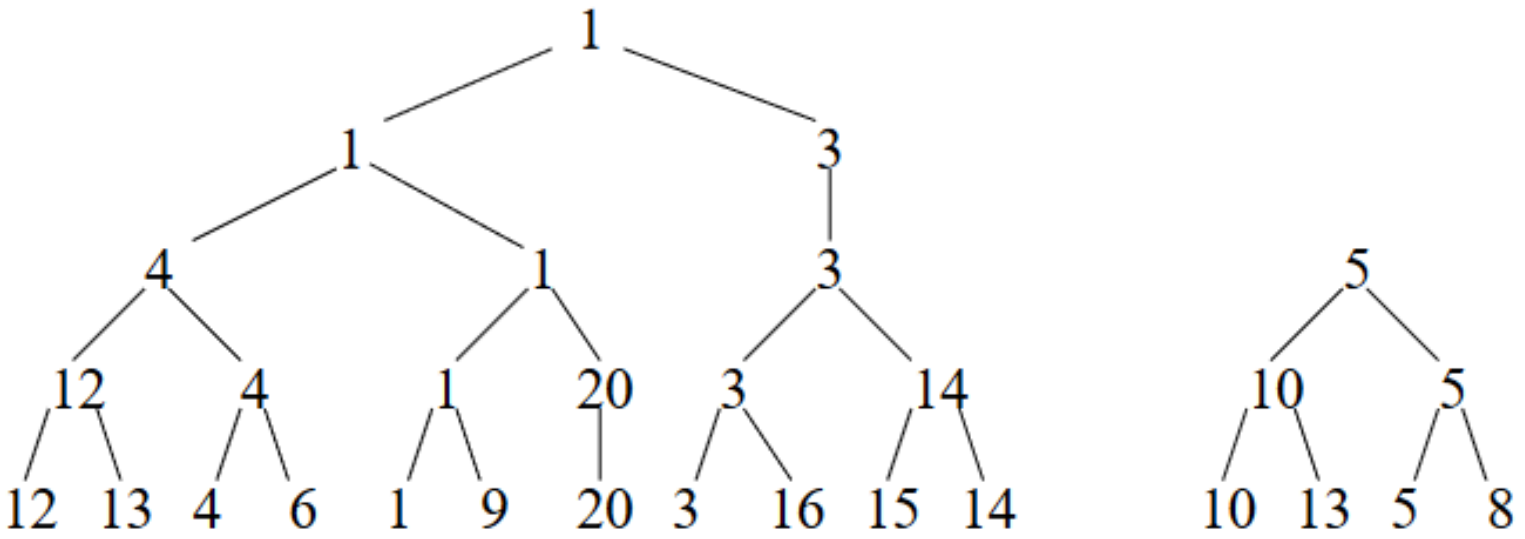
Deleting a Node

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

Quake Heaps

The approach: We will work with a collection of tournament trees, where each element in S is stored in exactly one leaf, and the element of each internal node is defined as the minimum of the elements at the children.

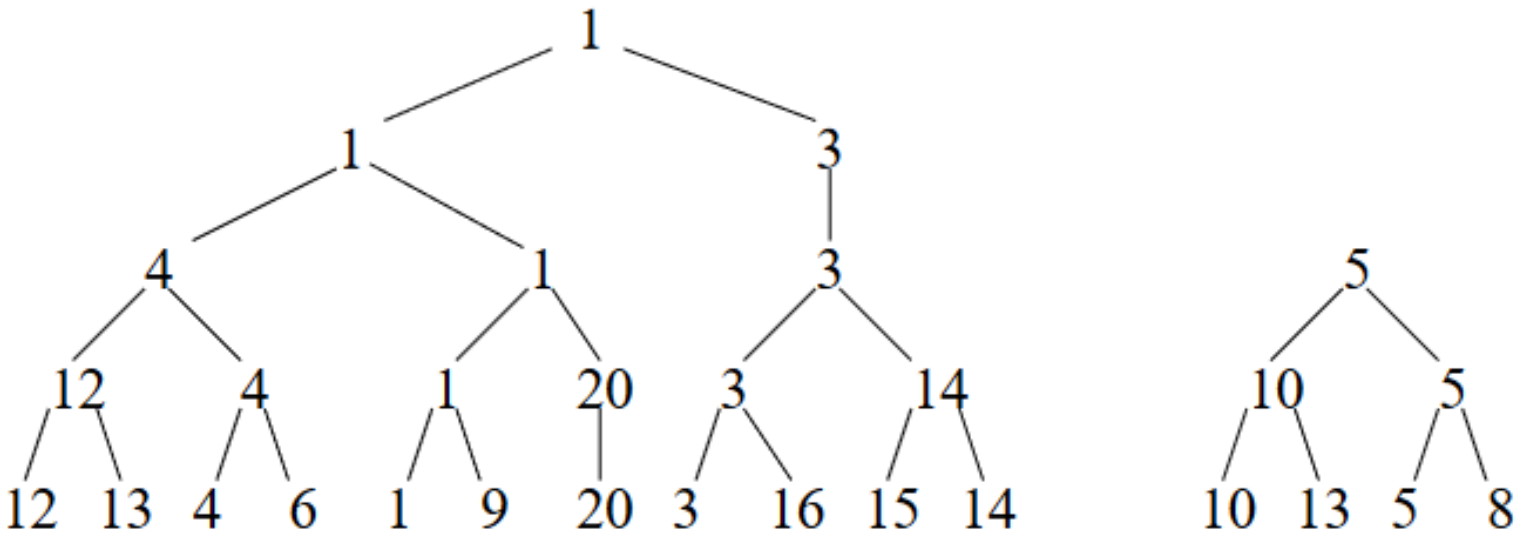
- We require that at each node x, all paths from x to a leaf have the same length; this length is referred to as the height of x.
- We also require that each internal node has degree 2 or 1.



Quake Heaps

- Two basic operations are easy to do in constant time under these requirements:

  - First, given two trees of the same height, we can link them into one, simply by creating a new root pointing to the two roots, storing the smaller element among the two roots.

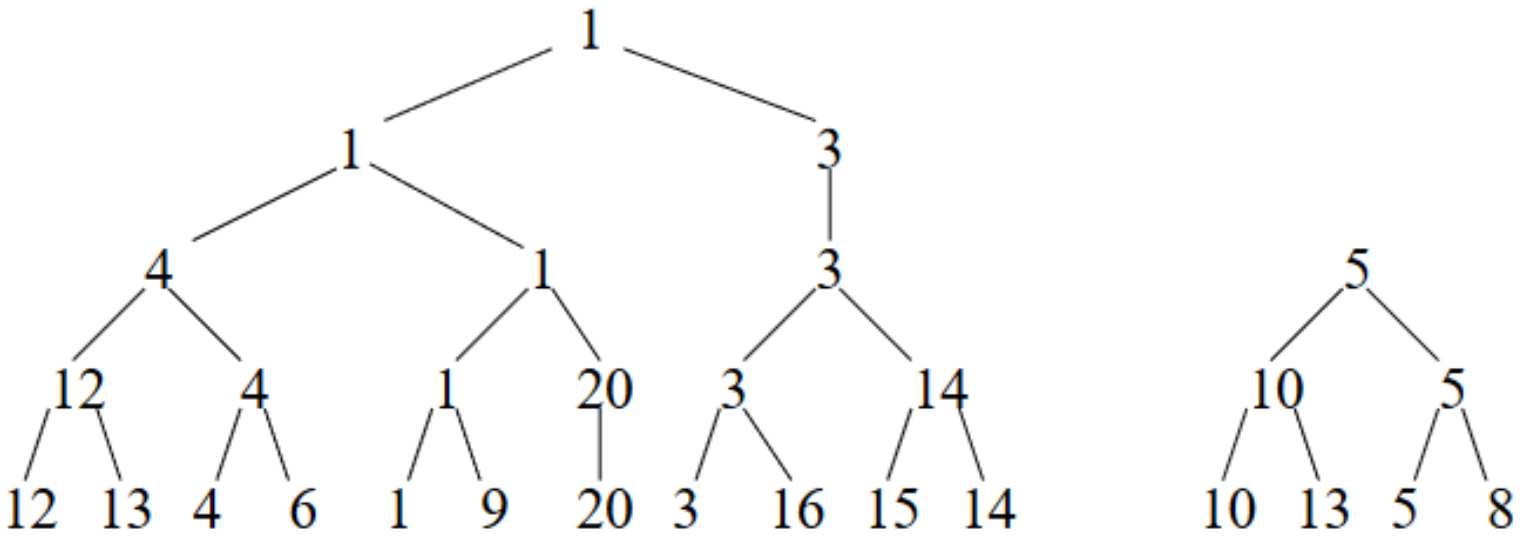


Quake Heaps

- Two basic operations are easy to do in constant time under these requirements:

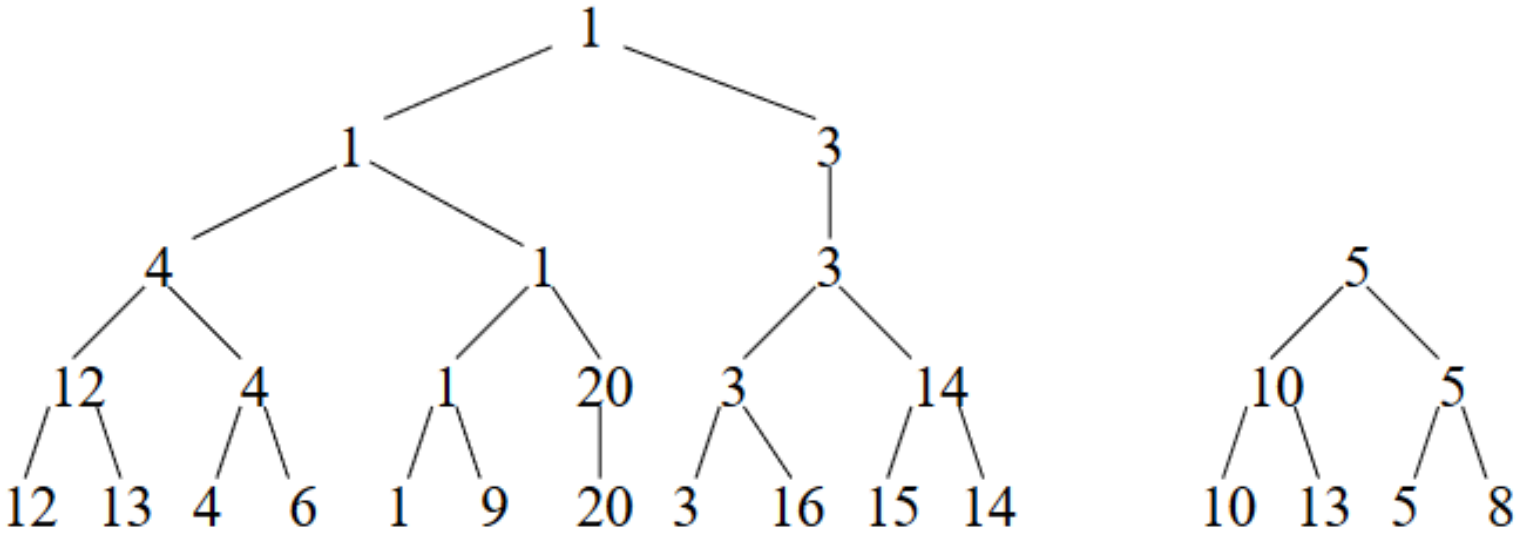
  - Secondly, given a node x whose element is different from x's parent's, we can cut out the subtree rooted at x.

  - Note that x's former parent's degree is reduced to 1, but our setup explicitly allows for degree-1 nodes.



Quake Heaps

- (To be concrete, we can set α = 3/4, for example.) The invariant clearly implies that the maximum height is at most $\log_{1/\alpha} n$.
- When the invariant is violated for some i, a “seismic” event occurs and we remove everything from height i + 1 an up, to allow rebuilding later.
- Since $n_{i+1} = n_{i+2} = \cdot \cdot \cdot = 0$ now, the invariant is restored. Intuitively, events of large “magnitude” (i.e., events at low heights I) should occur



Quake Heaps

insert(x):

1. create a new tree containing $\{x\}$

decrease-key(x, k):

1. cut the subtree rooted at the highest node storing x [yields 1 new tree]
2. change x's value to k

delete-min():

1. $x \leftarrow$ minimum of all the roots
2. remove the path of nodes storing x [yields multiple new trees]
3. while there are 2 trees of the same height:
4. link the 2 trees [reduces the number of trees by 1]
5. if $n_{i+1} > \alpha n_i$ for some i then:
6. let i be the smallest such index
7. remove all nodes at heights $> i$ [increases the number of trees]
8. return x

Quake Heaps

Analysis

- In the current data structure, let N be the number of nodes, T be the number of trees, and B be the number of degree-1 nodes (the “bad” nodes).
- Define the potential to be $N + T + \frac{1}{2\alpha - 1} B$.
- The amortized cost of an operation is the actual cost plus the change in potential.

Quake Heaps

Analysis

- For insert(), the actual cost is O(1), and N and T increase by 1.
- So, the amortized cost is O(1).

- For decrease-key(), the actual cost is O(1), and T and B increase by 1.
- So, the amortized cost is O(1).

Quake Heaps

Analysis

For delete-min(), we analyze lines 1–4 first. Let $T^{(0)}$ be the value of T just before the operation. Recall that the maximum height, and thus the length of the path in line 2, is $O(\log n)$. We can bound the actual cost by $T^{(0)} + O(\log n)$. Since after lines 3–4 there can remain at most one tree per height, T is decreased to $O(\log n)$. So, the change in T is $O(\log n) - T^{(0)}$. Since linking does not create degree-1 nodes, the change in B is nonpositive. Thus, the amortized cost is $O(\log n)$.

For lines 5–7 of delete-min(), let $n_j^{(0)}$ be the value of n_j just before these lines. We can bound the actual cost of lines 5–7 by $\sum_{j>i} n_j^{(0)}$. The change in N is at most $-\sum_{j>i} n_j^{(0)}$. The change in T is at most $+n_i^{(0)}$. Let $b_i^{(0)}$ be the number of degree-1 nodes at height i just before lines 5–7. Observe that $n_i^{(0)} \geq 2n_{i+1}^{(0)} - b_i^{(0)}$. Thus, $b_i^{(0)} \geq 2n_{i+1}^{(0)} - n_i^{(0)} \geq (2\alpha - 1)n_i^{(0)}$. Hence, the change in B is at most $-(2\alpha - 1)n_i^{(0)}$. Thus, the net change in $T + \frac{1}{2\alpha-1}B$ is nonpositive. We conclude that the amortized cost of lines 5–7 is nonpositive. Therefore, the overall amortized cost for delete-min() is $O(\log n)$.

van Emde Boas Trees

- van Emde Boas trees support the priority-queue operations, and a few others, each in O(lg lg n) worst-case time.
- The hitch is that the keys must be integers in the range 0 to n - 1, with no duplicates allowed.

van Emde Boas Trees

- Specifically, van Emde Boas trees support each of the dynamic set operations such as SEARCH, INSERT, DELETE, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR—in O(lg lg n) time.

van Emde Boas Trees

- We will use n to denote the number of elements currently in the set and u as the range of possible values.
- So each van Emde Boas tree operation runs in O(lg lg u) time.
- We call the set {0, 1, 2, ... u – 1) the universe of values that can be stored and u the universe size.
- We assume throughout this chapter that u is an exact power of 2, i.e., $u = 2k$ for some integer $k \geq 1$.

van Emde Boas Trees

Preliminary Approaches – Direct Addressing

- Direct addressing provides the simplest approach to storing a dynamic set.
- Since we are concerned only with storing keys, we can simplify the direct-addressing approach to store the dynamic set as a bit vector.
- To store a dynamic set of values from the universe {0, 1, 2, . . . u - 1}, we maintain an array A[0 .. u - 1] of u bits.
- The entry A[x] holds a 1 if the value x is in the dynamic set, and it holds a 0 otherwise.

van Emde Boas Trees

Preliminary Approaches – Direct Addressing

- Although we can perform each of the INSERT, DELETE, and MEMBER operations in O(1) time with a bit vector, the remaining operations—MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR—each take Θ(u) time in the worst case because we might have to scan through Θ(u) elements.

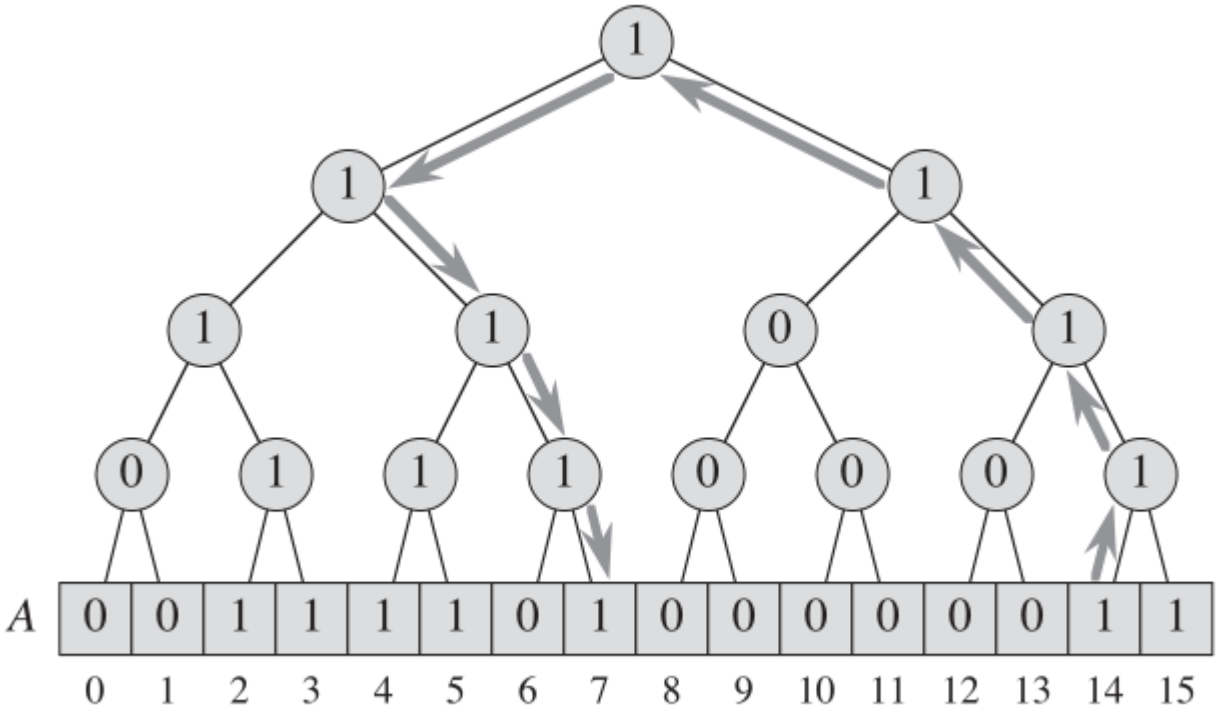
van Emde Boas Trees

Preliminary Approaches – Superimposing a binary tree structure

- We can short-cut long scans in the bit vector by superimposing a binary tree of bits on top of it.
- The entries of the bit vector form the leaves of the binary tree, and each internal node contains a 1 if and only if any leaf in its subtree contains a 1.
- In other words, the bit stored in an internal node is the logical-or of its two children.

van Emde Boas Trees

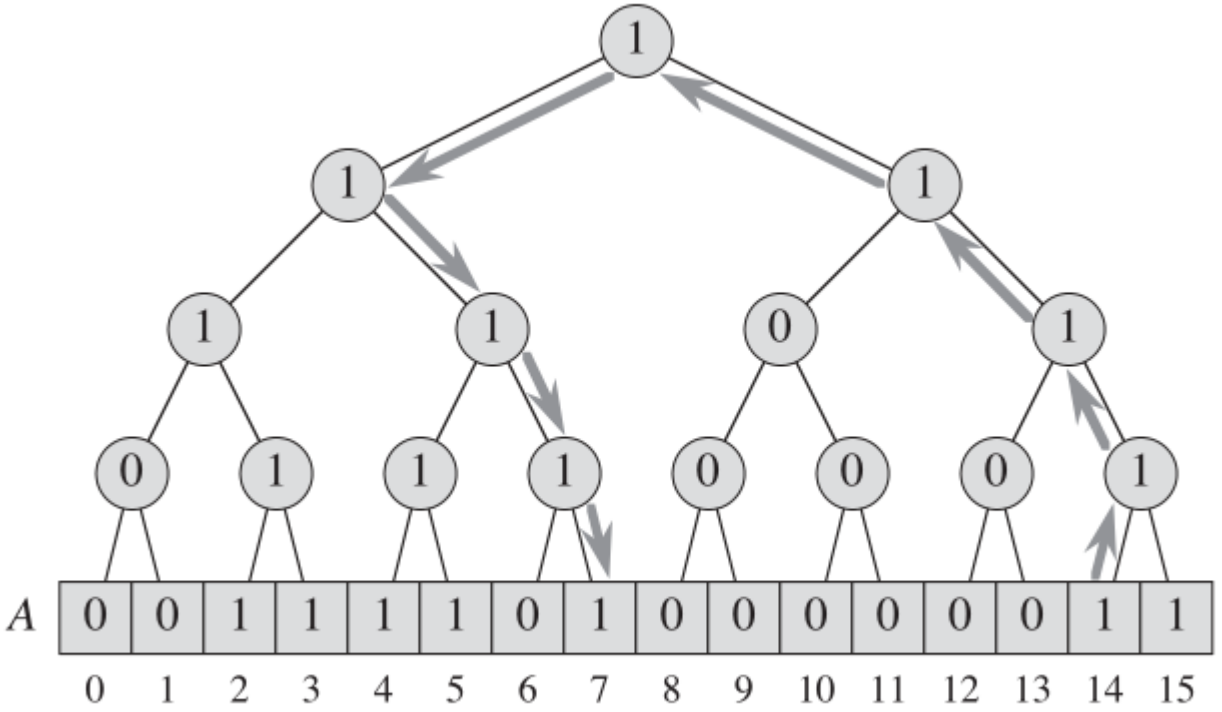
Preliminary Approaches – Superimposing a binary tree structure



- A binary tree of bits superimposed on top of a bit vector representing the set {2, 3, 4, 5, 7, 14, 15} when u = 16.
- Each internal node contains a 1 if and only if some leaf in its subtree contains a 1.
- The arrows show the path followed to determine the predecessor of 14 in the set.

van Emde Boas Trees

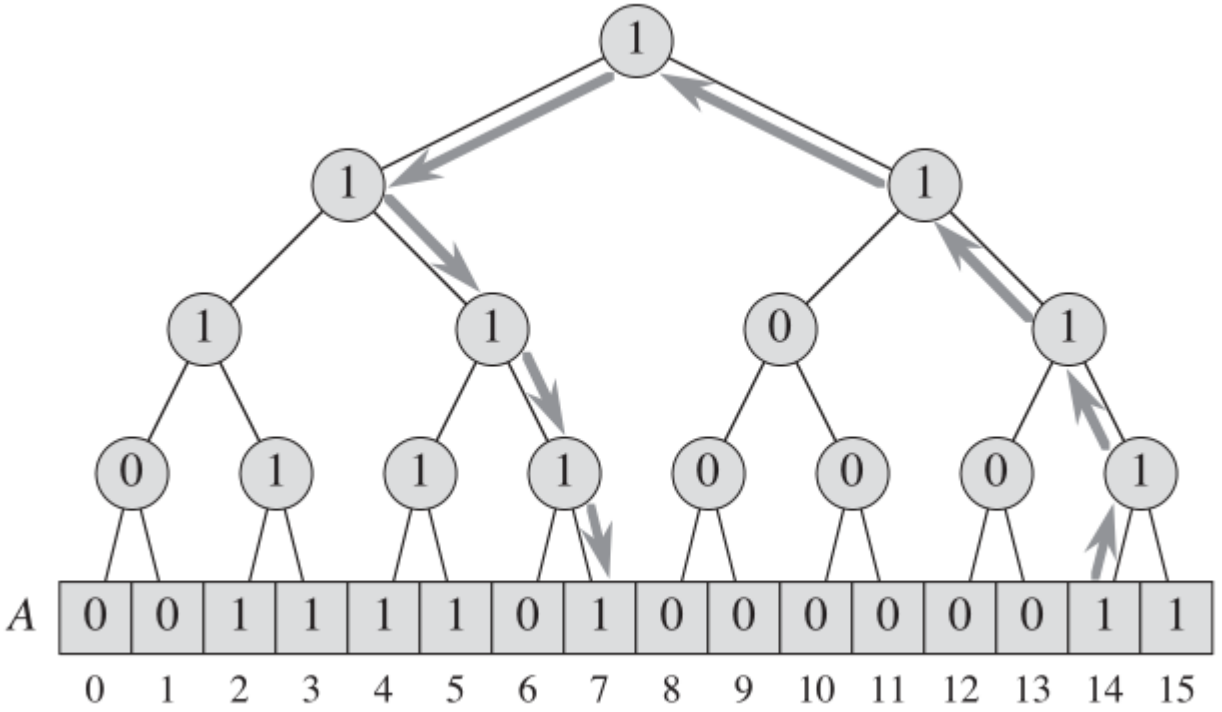
Preliminary Approaches – Superimposing a binary tree structure



- The operations that took Θ(u) worst-case time with an unadorned bit vector now use the tree structure:
  - To find the minimum value in the set, start at the root and head down toward the leaves, always taking the leftmost node containing a 1.

van Emde Boas Trees

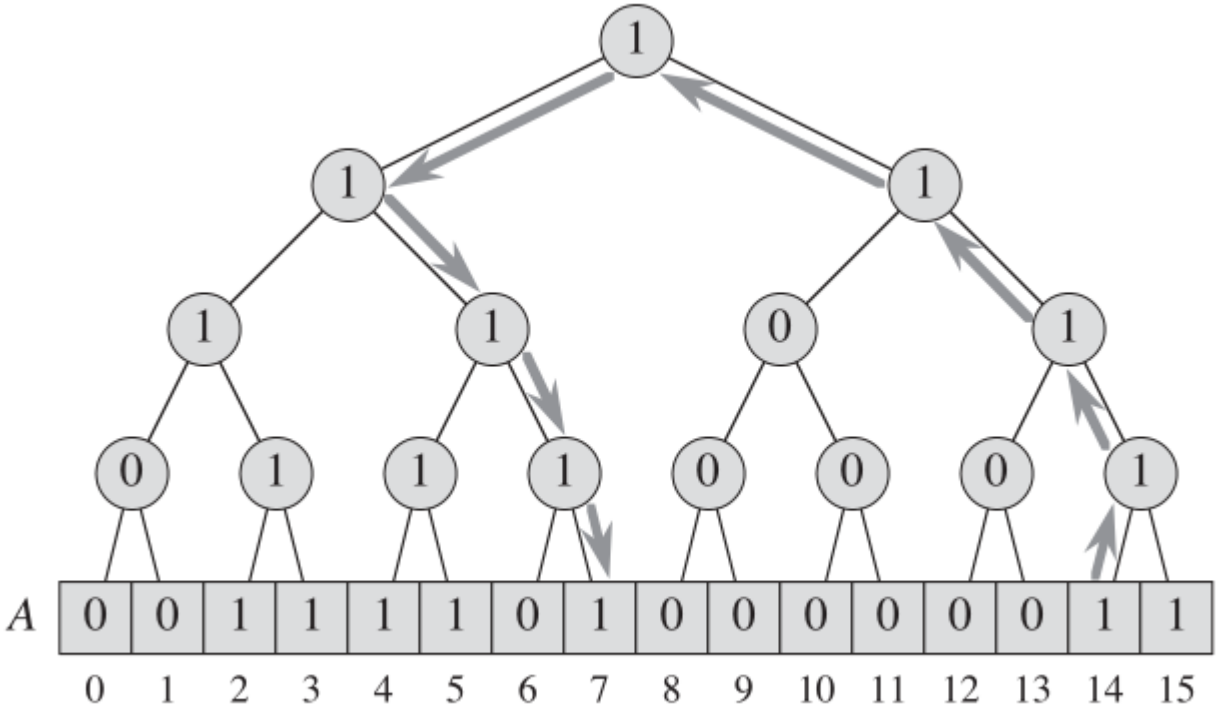
Preliminary Approaches – Superimposing a binary tree structure



- The operations that took Θ(u) worst-case time with an unadorned bit vector now use the tree structure:
  - To find the maximum value in the set, start at the root and head down toward the leaves, always taking the rightmost node containing a 1.

van Emde Boas Trees

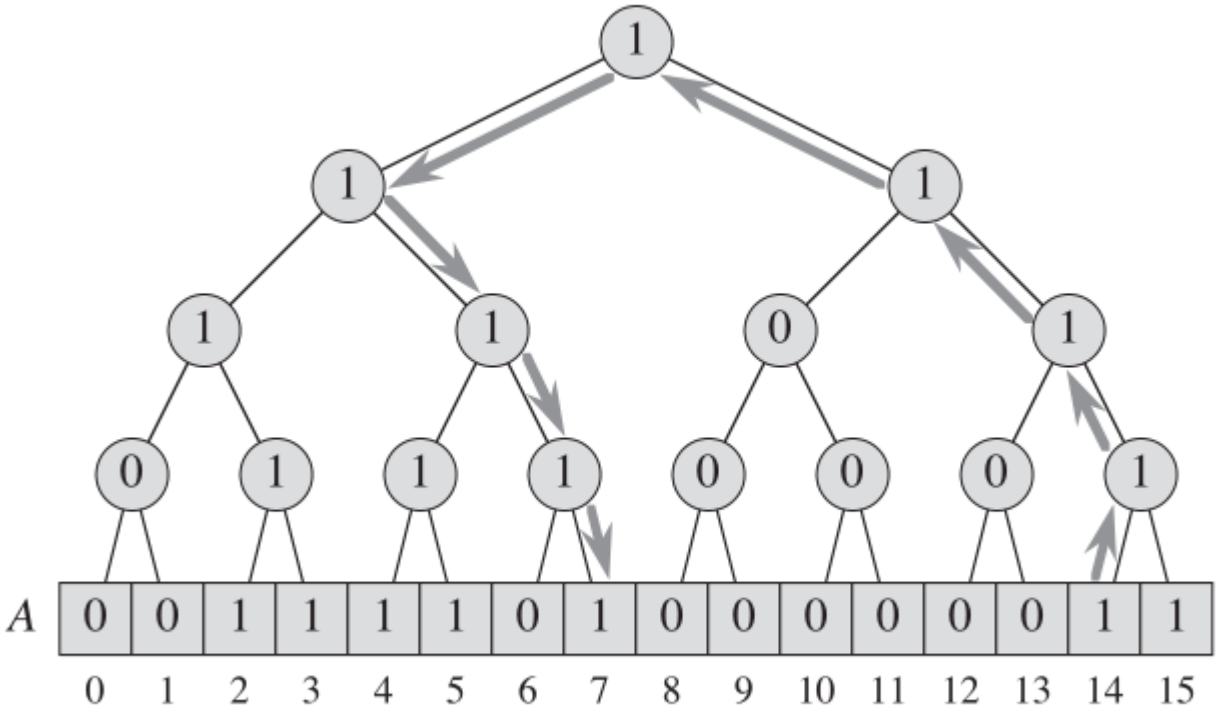
Preliminary Approaches – Superimposing a binary tree structure



- To find the successor of x, start at the leaf indexed by x, and head up toward the root until we enter a node from the left and this node has a 1 in its right child z.
- Then head down through node z, always taking the leftmost node containing a 1 (i.e., find the minimum value in the subtree rooted at the right child z).

van Emde Boas Trees

Preliminary Approaches – Superimposing a binary tree structure



- Since the height of the tree is lg u and each of the above operations makes at most one pass up the tree and at most one pass down, each operation takes O(lg u) time in the worst case.

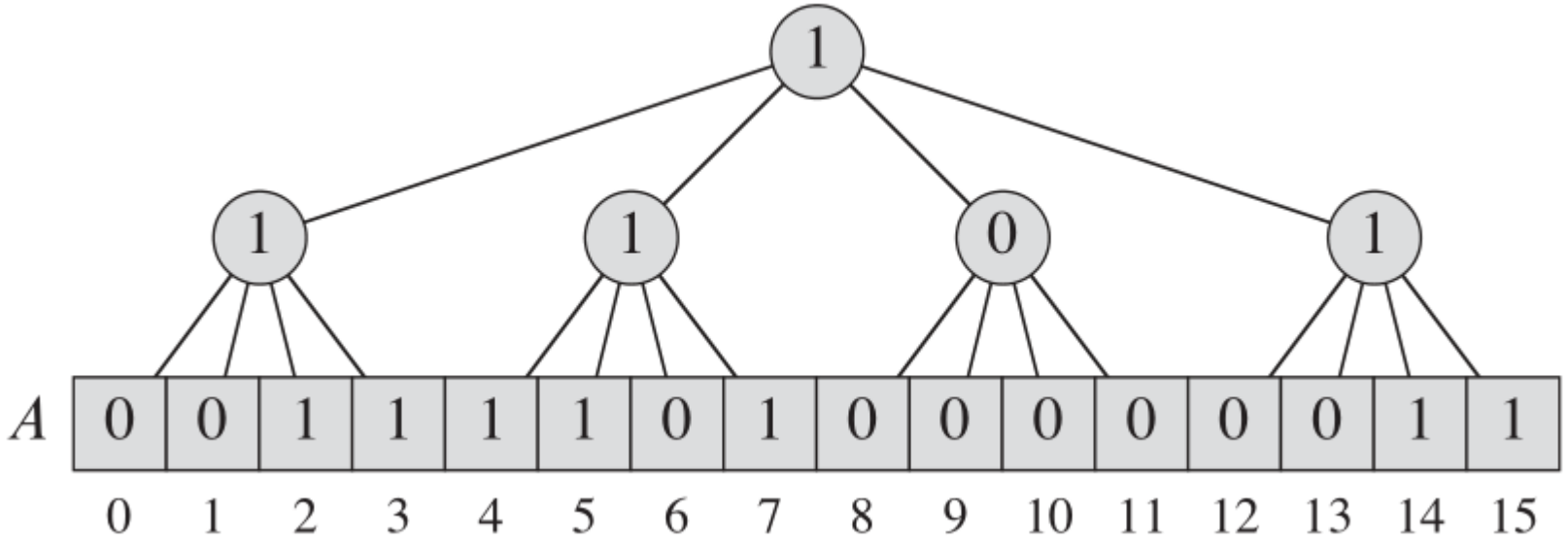
van Emde Boas Trees

Preliminary Approaches – Superimposing a tree of constant height

- What happens if we superimpose a tree with greater degree?
- Let us assume that the size of the universe is $u = 2^{2k}$ for some integer k, so that $\sqrt{u}$ is an integer.
- Instead of superimposing a binary tree on top of the bit vector, we superimpose a tree of degree $\sqrt{u}$.

van Emde Boas Trees

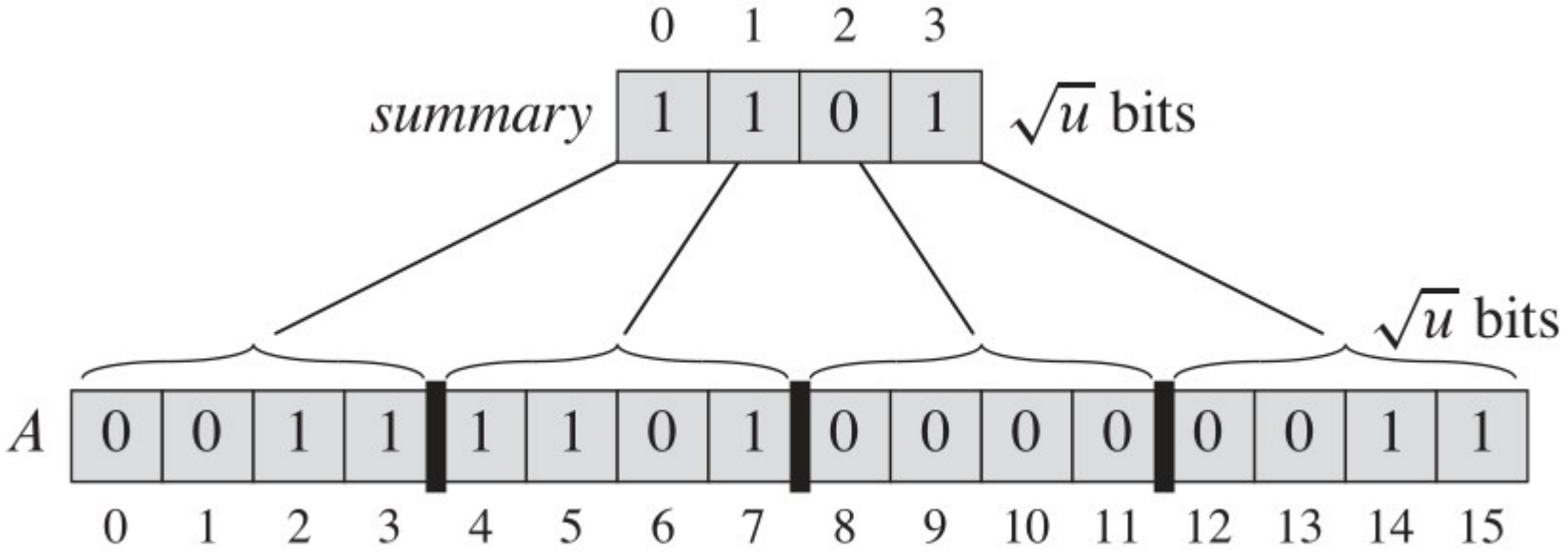
Preliminary Approaches – Superimposing a tree of constant height



- As before, each internal node stores the logical-or of the bits within its sub-tree, so that the √u internal nodes at depth 1 summarize each group of √u values.

van Emde Boas Trees

Preliminary Approaches – Superimposing a tree of constant height

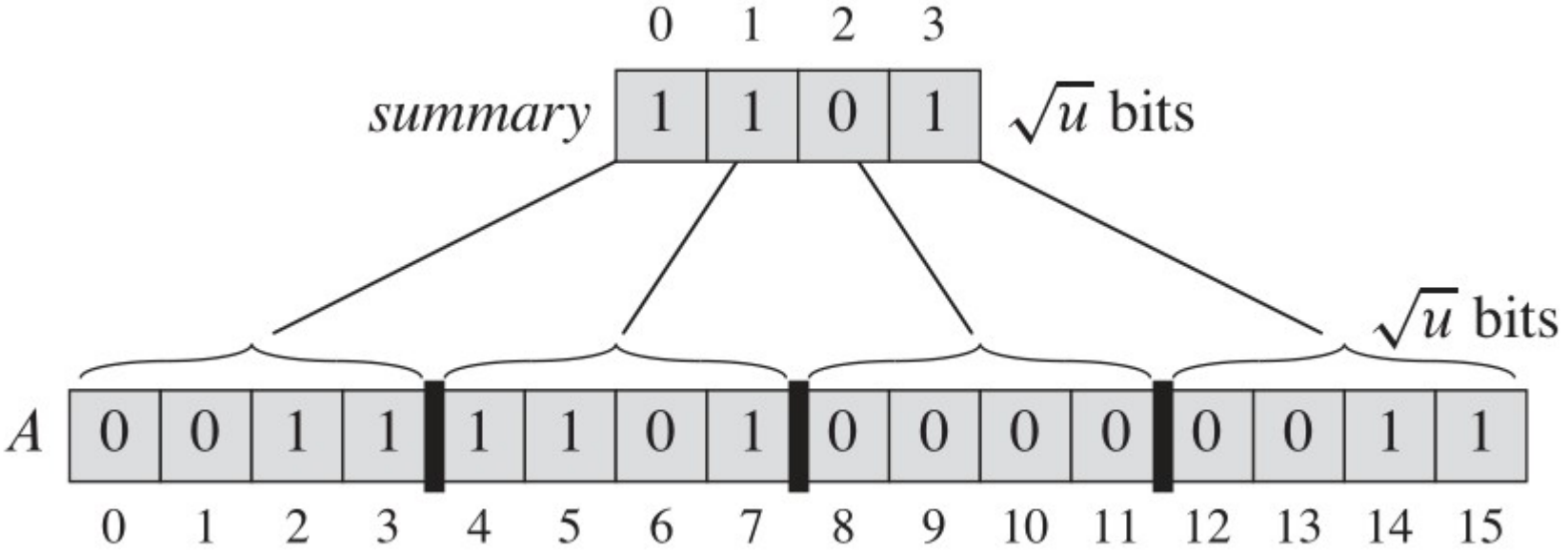


- We can think of these nodes as an array summary[0 . . √u - 1], where summary[i] contains a 1 if and only if the subarray A[i√u .. (i+1) √u – 1] contains a 1.

- We call this √u-bit subarray of A the i^{th} cluster.

van Emde Boas Trees

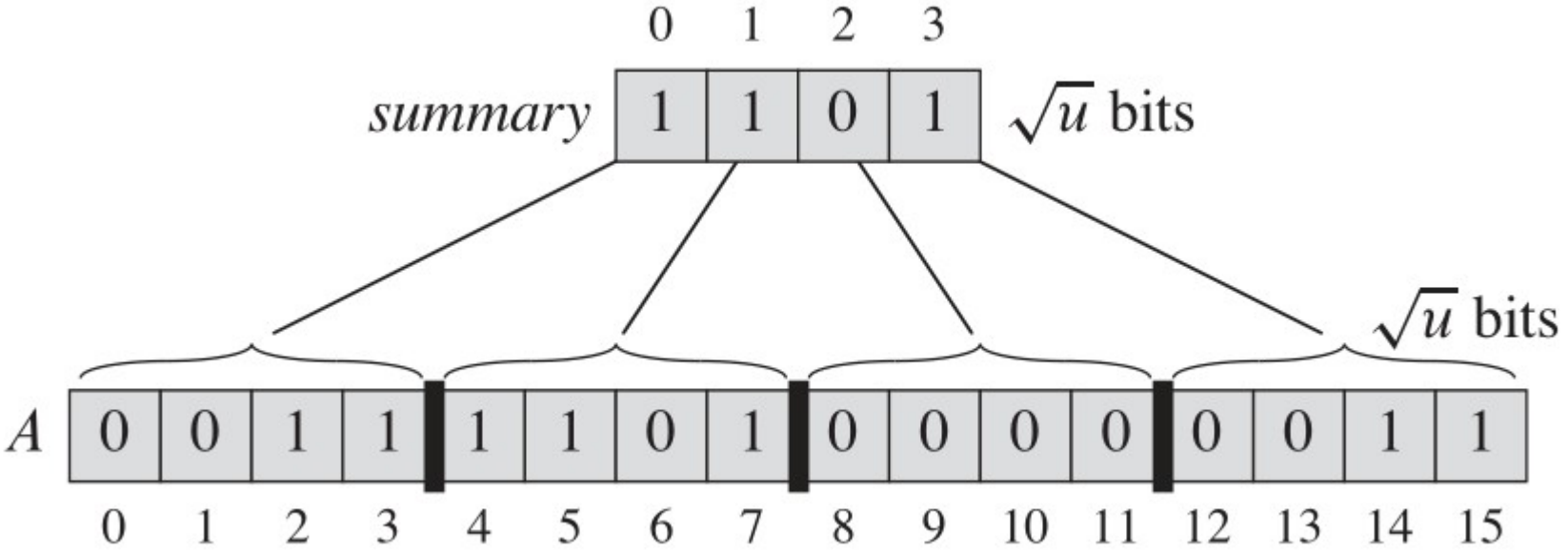
Preliminary Approaches – Superimposing a tree of constant height



- For a given value of x, the bit A[x] appears in cluster number floor(x/√u).
- Now INSERT becomes an O(1) time operation: to insert x, set both A[x] and summary[floor(x/√u)] to 1.

van Emde Boas Trees

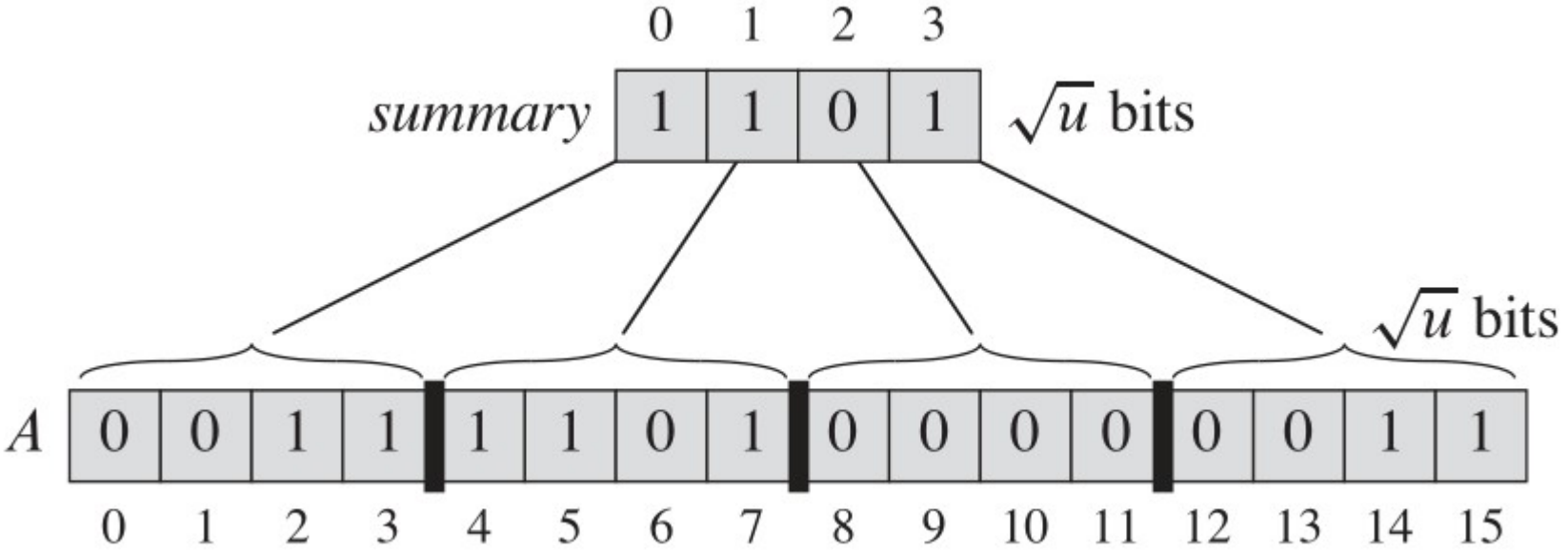
Preliminary Approaches – Superimposing a tree of constant height



- We can use the summary array to perform each of the operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and DELETE in O(√u) time.

van Emde Boas Trees

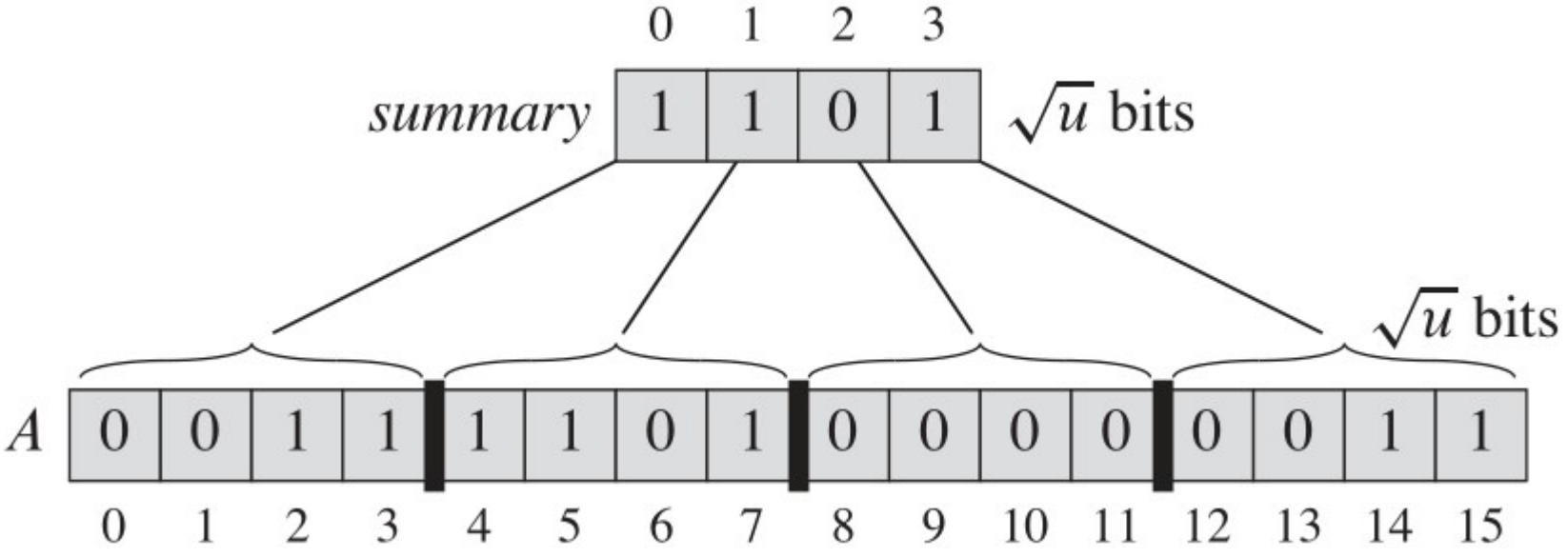
Preliminary Approaches – Superimposing a tree of constant height



- To find the minimum (maximum) value, find the leftmost (rightmost) entry in summary that contains a 1, say summary[i], and then do a linear search within the ith cluster for the leftmost (rightmost) 1.

van Emde Boas Trees

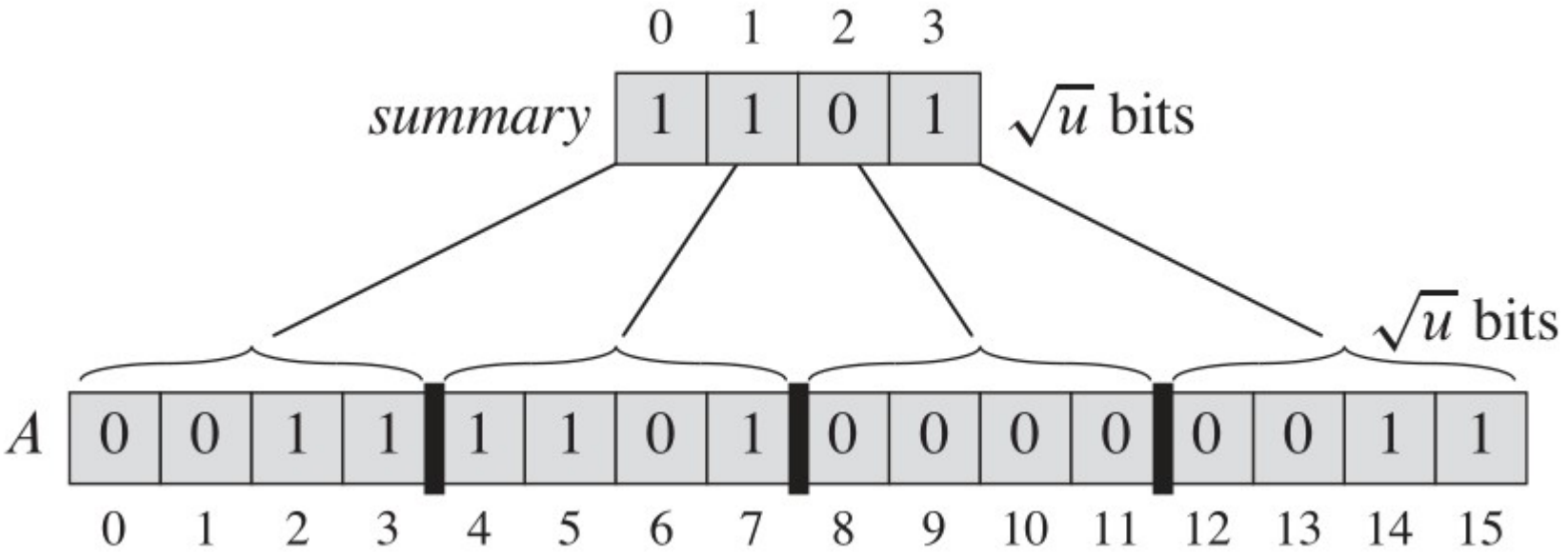
Preliminary Approaches – Superimposing a tree of constant height



- To find the successor (predecessor) of x, first search to the right (left) within its cluster. If we find a 1, that position gives the result.
- Otherwise, let i = floor(x/√u) and search to the right (left) within the summary array from index i. The first position that holds a 1 gives the index of a cluster. Search within that cluster for the leftmost (rightmost) 1.
- That position holds the successor (predecessor).

van Emde Boas Trees

Preliminary Approaches – Superimposing a tree of constant height



- To delete the value x, let i = floor(x/√u).
- Set A[x] to 0 and then set summary[i] to the logical-or of the bits in the ith cluster.

van Emde Boas Trees

Preliminary Approaches – Superimposing a tree of constant height

- At first glance, it seems as though we have made negative progress.
- Superimposing a binary tree gave us O(lg u) time operations, which are asymptotically faster than O(√u) time.
- Using a tree of degree √u will turn out to be a key idea of van Emde Boas trees, however.

van Emde Boas Trees

Proto van Emde Boas structures

- For the universe {0, 1, 2, . . . u - 1}, we define a proto van Emde Boas structure, or proto-vEB structure, which we denote as proto-VEB(u), recursively as follows.
- Each proto-VEB(u) structure contains an attribute u giving its universe size.
- In addition, it contains the following:

van Emde Boas Trees

Proto van Emde Boas structures

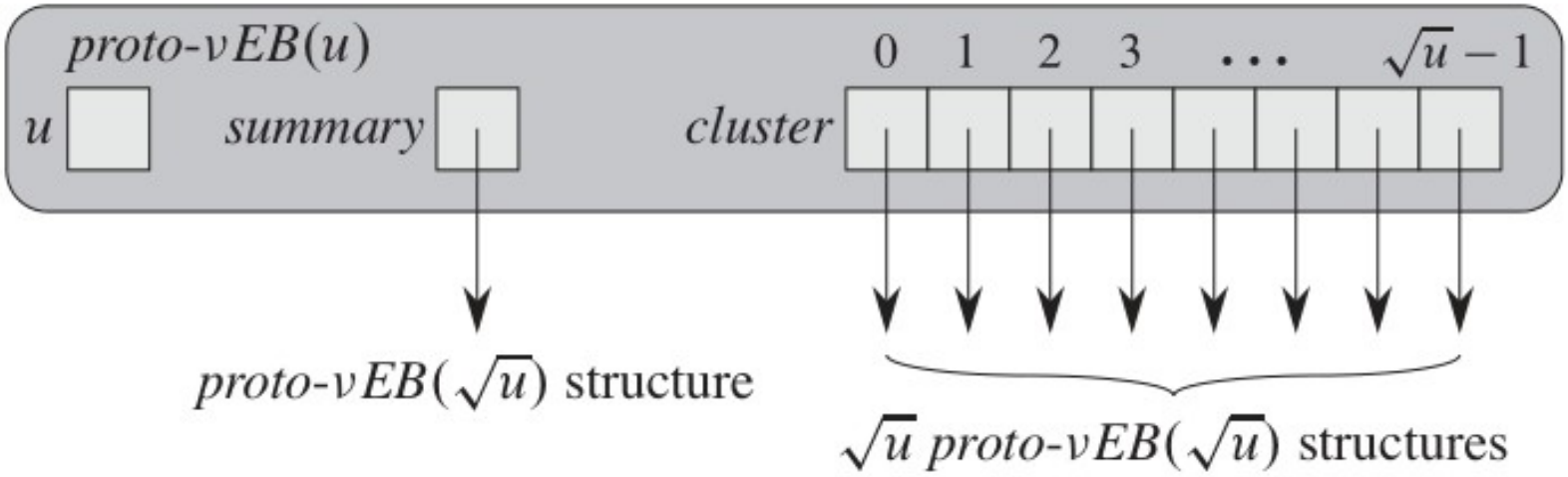
- If u = 2, then it is the base size, and it contains an array A[0 .. 1] of two bits.
- Otherwise, $u = 2^{2^{\wedge k}}$ for some integer $k \geq 1$, so that $u \geq 4$. In addition to the universe size u, the data structure proto-VEB(u) contains the following attributes

  - a pointer named summary to a proto-VEB($\sqrt{u}$) structure

  - and an array cluster[0 . . $\sqrt{u}$ - 1] of $\sqrt{u}$ pointers, each to a proto-VEB($\sqrt{u}$) structure.

van Emde Boas Trees

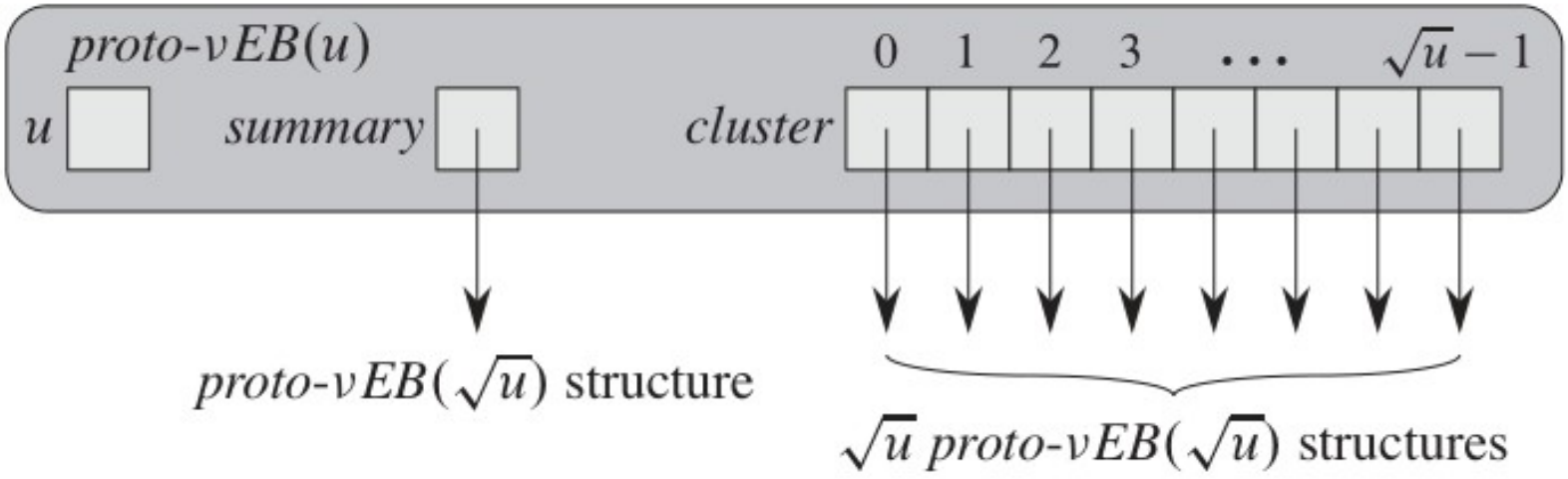
Proto van Emde Boas structures



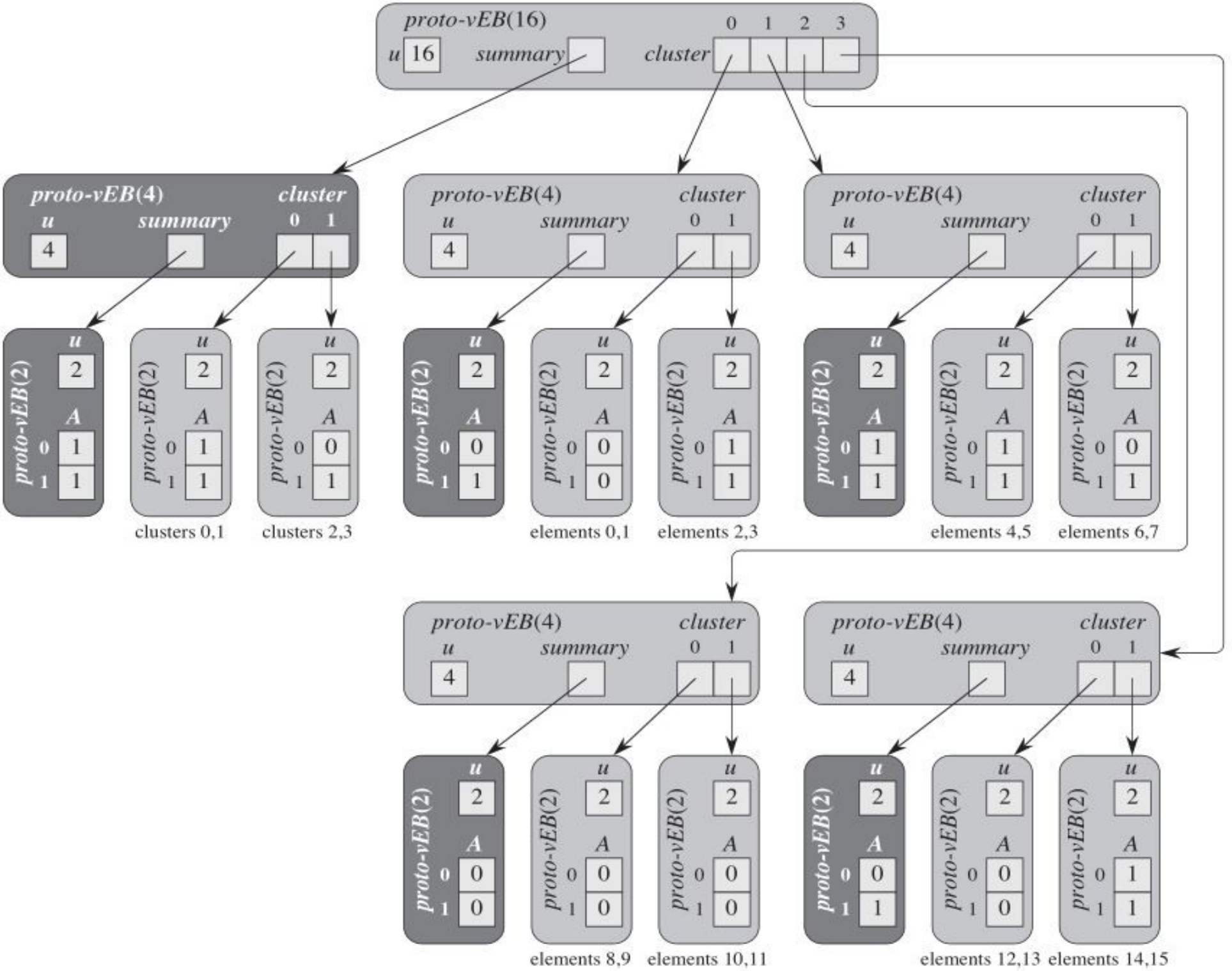
- The information in a proto-VEB(u) structure when u ≥ 4.
- The structure contains the universe size u, a pointer summary to a proto-VEB(√u) structure, and an array cluster[0 . . √u – 1) of √u pointers to proto-VEB(√u) structures.

van Emde Boas Trees

Proto van Emde Boas structures



- The element x, where $0 \leq x < u$, is recursively stored in the cluster numbered high(x) as element low(x) within that cluster.
  Where high(x) = floor(x/√u)
  low(x) = x mod √u


proto-vEB(16)
u
16
summary
cluster
0
1
2
3
proto-vEB(4)
u
4
summary
cluster
0
1
proto-vEB(2)
u
2
A
0
1
1
clusters 0,1
clusters 2,3
elements 0,1
elements 2,3
elements 4,5
elements 6,7
elements 8,9
elements 10,11
elements 12,13
elements 14,15

van Emde Boas Trees

- The proto-vEB structure of the previous section is close to what we need to achieve O(lg lg u) running times.
- It falls short because we have to recurse too many times in most of the operations.

van Emde Boas Trees

- Now, we shall design a data structure that is similar to the proto-vEB structure but stores a little more information, thereby removing the need for some of the recursion.
- We also will allow the universe size u to be any exact power of 2 and when $\sqrt{u}$ is not an integer that is, if u is an odd power of 2 ($u = 2^{2k+1}$ for some integer $k \geq 0$), then we will divide the lg u bits of a number into the most significant **ceil((lg u)/2)** bits and the least significant **floor((lg u)/2) bits.**

van Emde Boas Trees

- For convenience, we denote $2^{\text{ceil}((\lg u)/2)}$ (the "up- per square root" of u) by $\sqrt[\uparrow]{u}$
- And, we denote $2^{\text{floor}((\lg u)/2)}$(the "lower square root" of u) by $\sqrt[\downarrow]{u}$
- So that,

$$u = \sqrt[\uparrow]{u} \cdot \sqrt[\downarrow]{u}$$

- When u is an even power of 2 ($u = 2^{2k}$ for some integer k),

$$\sqrt[\uparrow]{u} = \sqrt[\downarrow]{u} = \sqrt{u}.$$

van Emde Boas Trees

- Because we now allow u to be an odd power of 2, we must redefine our helpful functions

$$\begin{aligned} \text{high}(x) &= \left\lfloor x / \sqrt[\downarrow]{u} \right\rfloor , \\ \text{low}(x) &= x \bmod \sqrt[\downarrow]{u} , \\ \text{index}(x, y) &= x \sqrt[\downarrow]{u} + y . \end{aligned}$$

van Emde Boas Trees

The ***van Emde Boas tree***, or ***vEB tree***, modifies the proto-vEB structure. We denote a vEB tree with a universe size of u as $vEB(u)$ and, unless u equals the base size of 2, the attribute *summary* points to a $vEB(\uparrow\sqrt{u})$ tree and the array $cluster[0\,..\,\uparrow\sqrt{u} - 1]$ points to $\uparrow\sqrt{u}$ $vEB(\downarrow\sqrt{u})$ trees.

vEB tree contains two attributes not found in a proto-vEB structure:

- min stores the minimum element in the vEB tree, and
- max stores the maximum element in the vEB tree.

van Emde Boas Trees

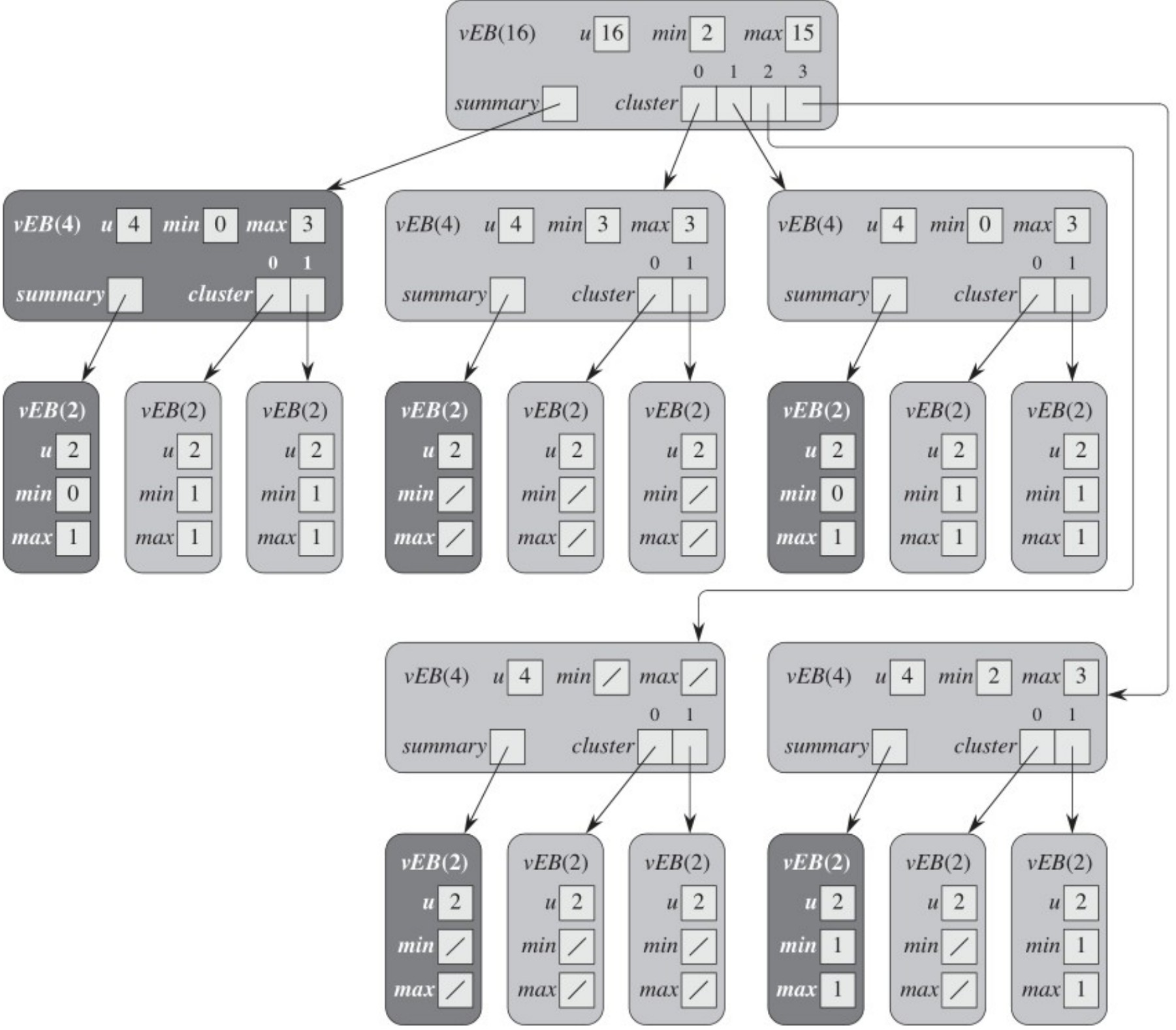
- Furthermore, the element stored in min does not appear in any of the recursive $vEB(\sqrt[\downarrow]{u})$ trees that the cluster array points to.

- The elements stored in a vEB(u) tree V , therefore, are V.min plus all the elements recursively stored in the $vEB(\sqrt[\downarrow]{u})$ trees pointed to by $V.cluster[0\,..\,\sqrt[\uparrow]{u}-1]$.

van Emde Boas Trees

- Note that when a vEB tree contains two or more elements, we treat min and max differently: the element stored in min does not appear in any of the clusters, but the element stored in max does.

- Since the base size is 2, a vEB(2) tree does not need the array A that the corresponding proto-vEB(2) structure has.
- Instead, we can determine its elements from its min and max attributes.
- In a vEB tree with no elements, regardless of its universe size u, both min and max are NIL.


vEB(16) u 16 min 2 max 15
0 1 2 3
summary cluster
vEB(4) u 4 min 0 max 3
0 1
summary cluster
vEB(4) u 4 min 3 max 3
0 1
summary cluster
vEB(4) u 4 min 0 max 3
0 1
summary cluster
vEB(2) u 2 min 0 max 1
vEB(2) u 2 min 1 max 1
vEB(2) u 2 min 1 max 1
vEB(2) u 2 min / max /
vEB(2) u 2 min / max /
vEB(2) u 2 min / max /
vEB(2) u 2 min 0 max 1
vEB(2) u 2 min 1 max 1
vEB(2) u 2 min 1 max 1
vEB(4) u 4 min / max /
0 1
summary cluster
vEB(4) u 4 min 2 max 3
0 1
summary cluster
vEB(2) u 2 min / max /
vEB(2) u 2 min / max /
vEB(2) u 2 min / max /
vEB(2) u 2 min 1 max 1
vEB(2) u 2 min / max /
vEB(2) u 2 min 1 max 1

van Emde Boas Trees

- Figure shows a vEB(16) tree V holding the set {2, 3, 4, 5, 7, 14, 15}.

- Because the smallest element is 2, V.min equals 2, and even though high(2) = 0, the element 2 does not appear in the vEB(4) tree pointed to by V.cluster[0]: notice that V.cluster[0].min equals 3, and so 2 is not in this vEB tree.

- Similarly, since V.cluster[0].min equals 3, and 2 and 3 are the only elements in V.cluster[0], the vEB(2) clusters within V.cluster[0] are empty.

van Emde Boas Trees

The min and max attributes will turn out to be key to reducing the number of recursive calls within the operations on vEB trees. These attributes will help us in four ways:

1. The MINIMUM and MAXIMUM operations do not even need to recurse, for they can just return the values of min or max.

van Emde Boas Trees

The min and max attributes will turn out to be key to reducing the number of recursive calls within the operations on vEB trees. These attributes will help us in four ways:

2. The SUCCESSOR operation can avoid making a recursive call to determine whether the successor of a value x lies within high(x). That is because x' successor lies within its cluster if and only if x is strictly less than the ma attribute of its cluster. A symmetric argument holds for PREDECESSOR and min.

van Emde Boas Trees

The min and max attributes will turn out to be key to reducing the number of recursive calls within the operations on vEB trees. These attributes will help us in four ways:

3. We can tell whether a vEB tree has no elements, exactly one element, or at least two elements in constant time from its min and max values. This ability will help in the INSERT and DELETE operations. If min and max are both NIL, then the vEB tree has no elements. If min and max are non-NIL but are equal to each other, then the vEB tree has exactly one element. Otherwise, both min and max are non-NIL but are unequal, and the vEB tree has two or more elements.

van Emde Boas Trees

The min and max attributes will turn out to be key to reducing the number of recursive calls within the operations on vEB trees. These attributes will help us in four ways:

4. If we know that a vEB tree is empty, we can insert an element into it by updating only its min and max attributes. Hence, we can insert into an empty vEB tree in constant time. Similarly, if we know that a vEB tree has only one element, we can delete that element in constant time by updating only min and max. These properties will allow us to cut short the chain of recursive calls.

van Emde Boas Trees

Finding the minimum and maximum elements

- Because we store the minimum and maximum in the attributes min and max, two of the operations are one-liners, taking constant time:

```
vEB-Tree-Minimum(V)
1  return V.min

vEB-Tree-Maximum(V)
1  return V.max
```

van Emde Boas Trees

Determining whether a value is in the set

```
vEB-TREE-MEMBER(V, x)
1  if x == V.min or x == V.max
2        return TRUE
3  elseif V.u == 2
4        return FALSE
5  else return vEB-TREE-MEMBER(V.cluster[high(x)], low(x))
```

- Line 1 checks to see whether x equals either the minimum or maximum element.
- If it does, line 2 returns TRUE. Otherwise, line 3 tests for the base case.
- Since a vEB(2) tree has no elements other than those in min and max, if it is the base case, line 4 returns FALSE.

- The other possibility—it is not a base case and x equals neither min nor max—is handled by the recursive call in line 5.

- This procedure takes O(lg lg u) time.

van Emde Boas Trees

Finding the Successor

```
VEB-TREE-SUCCESSOR(V, x)
 1  if V.u == 2
 2       if x == 0 and V.max == 1
 3            return 1
 4       else return NIL
 5  elseif V.min ≠ NIL and x < V.min
 6       return V.min
 7  else max-low = VEB-TREE-MAXIMUM(V.cluster[high(x)])
 8       if max-low ≠ NIL and low(x) < max-low
 9            offset = VEB-TREE-SUCCESSOR(V.cluster[high(x)], low(x))
10            return index(high(x), offset)
11       else succ-cluster = VEB-TREE-SUCCESSOR(V.summary, high(x))
12            if succ-cluster == NIL
13                 return NIL
14            else offset = VEB-TREE-MINIMUM(V.cluster[succ-cluster])
15                 return index(succ-cluster, offset)
```

van Emde Boas Trees

Finding the Predecessor

```
VEB-TREE-PREDECESSOR(V, x)
 1  if V.u == 2
 2       if x == 1 and V.min == 0
 3            return 0
 4       else return NIL
 5  elseif V.max ≠ NIL and x > V.max
 6       return V.max
 7  else min-low = VEB-TREE-MINIMUM(V.cluster[high(x)])
 8       if min-low ≠ NIL and low(x) > min-low
 9            offset = VEB-TREE-PREDECESSOR(V.cluster[high(x)], low(x))
10            return index(high(x), offset)
11       else pred-cluster = VEB-TREE-PREDECESSOR(V.summary, high(x))
12            if pred-cluster == NIL
13                 if V.min ≠ NIL and x > V.min
14                      return V.min
15                 else return NIL
16            else offset = VEB-TREE-MAXIMUM(V.cluster[pred-cluster])
17                 return index(pred-cluster, offset)
```

van Emde Boas Trees

Inserting an Element

```
VEB-TREE-INSERT(V, x)
 1  if V.min == NIL
 2      VEB-EMPTY-TREE-INSERT(V, x)
 3  else if x < V.min
 4          exchange x with V.min
 5      if V.u > 2
 6          if VEB-TREE-MINIMUM(V.cluster[high(x)]) == NIL
 7              VEB-TREE-INSERT(V.summary, high(x))
 8              VEB-EMPTY-TREE-INSERT(V.cluster[high(x)], low(x))
 9          else VEB-TREE-INSERT(V.cluster[high(x)], low(x))
10      if x > V.max
11          V.max = x
```

```
VEB-EMPTY-TREE-INSERT(V, x)
1  V.min = x
2  V.max = x
```

van Emde Boas Trees

Deleting an element

```
vEB-Tree-Delete(V, x)
 1  if V.min == V.max
 2      V.min = NIL
 3      V.max = NIL
 4  elseif V.u == 2
 5      if x == 0
 6          V.min = 1
 7      else V.min = 0
 8      V.max = V.min
 9  else if x == V.min
10          first-cluster = vEB-Tree-Minimum(V.summary)
11          x = index(first-cluster,
                vEB-Tree-Minimum(V.cluster[first-cluster]))
12          V.min = x
13      vEB-Tree-Delete(V.cluster[high(x)], low(x))
14      if vEB-Tree-Minimum(V.cluster[high(x)]) == NIL
15          vEB-Tree-Delete(V.summary, high(x))
16          if x == V.max
17              summary-max = vEB-Tree-Maximum(V.summary)
18              if summary-max == NIL
19                  V.max = V.min
20              else V.max = index(summary-max,
                        vEB-Tree-Maximum(V.cluster[summary-max]))
21      elseif x == V.max
22          V.max = index(high(x),
                vEB-Tree-Maximum(V.cluster[high(x)]))
```

Data Structures for Disjoints Sets

- A disjoint-set data structure maintains a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets.
- We identify each set by a representative, which is some member of the set.
- In some applications, it doesn't matter which member is used as the representative; we care only that if we ask for the representative of a dynamic set twice without modifying the set between the requests, we get the same answer both times.
- Other applications may require a prespecified rule for choosing the representative, such as choosing the smallest member in the set (assuming, of course, that the elements can be ordered).

Data Structures for Disjoints Sets

- We wish to support the following operations:

1) MAKE-SET(x) creates a new set whose only member (and thus representative) is x. Since the sets are disjoint, we require that x not already be in some other set.

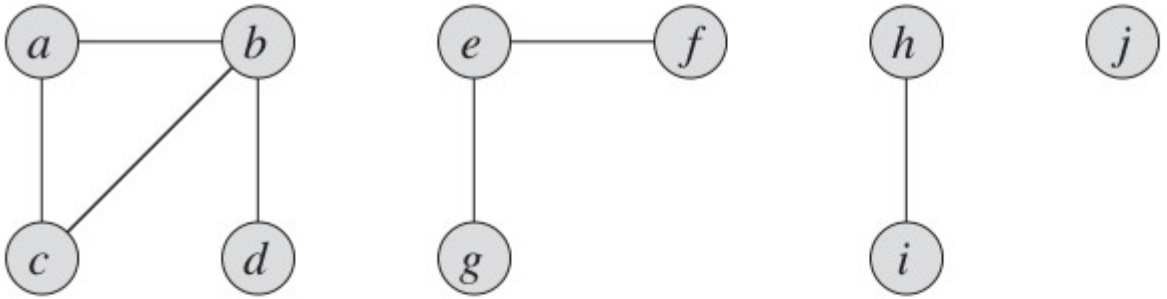
2) UNION(x, y) unites the dynamic sets that contain x and y, say Sx and Sy, into a new set that is the union of these two sets. We assume that the two sets are disjoint prior to the operation. The representative of the resulting set is any member of $S_x \cup S_y$, although many implementations of UNION specifically choose the representative of either S_x or S_y as the new representative. Since we require the sets in the collection to be disjoint, conceptually we destroy sets S_x and S_y,removing them from the collection S. In practice, we often absorb the elements of one of the sets into the other set.

3) FIND-SET(x) returns a pointer to the representative of the (unique) set containing x.

Data Structures for Disjoints Sets

- We shall analyze the running times of disjoint-set data structures in terms of two parameters: n, the number of MAKE-SET operations, and m, the total number of MAKE-SET, UNION, and FIND-SET operations.
- Since the sets are disjoint, each UNION operation reduces the number of sets by one.
- After n - 1 UNION operations, therefore, only one set remains.
- The number of UNION operations is thus at most n - 1.
- Note also that since the MAKE-SET operations are included in the total number of operations m, we have $m \geq n$.
- We assume that the n MAKE-SET operations are the first n operations performed.

Data Structures for Disjoints Sets



(a)

Edge processed	Collection of disjoint sets									
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

(b)

Disjoint-set data structures can be used in determining the connected components of an undirected graph

Data Structures for Disjoints Sets

```
CONNECTED-COMPONENTS(G)
1  for each vertex v ∈ G.V
2      MAKE-SET(v)
3  for each edge (u, v) ∈ G.E
4      if FIND-SET(u) ≠ FIND-SET(v)
5          UNION(u, v)
```

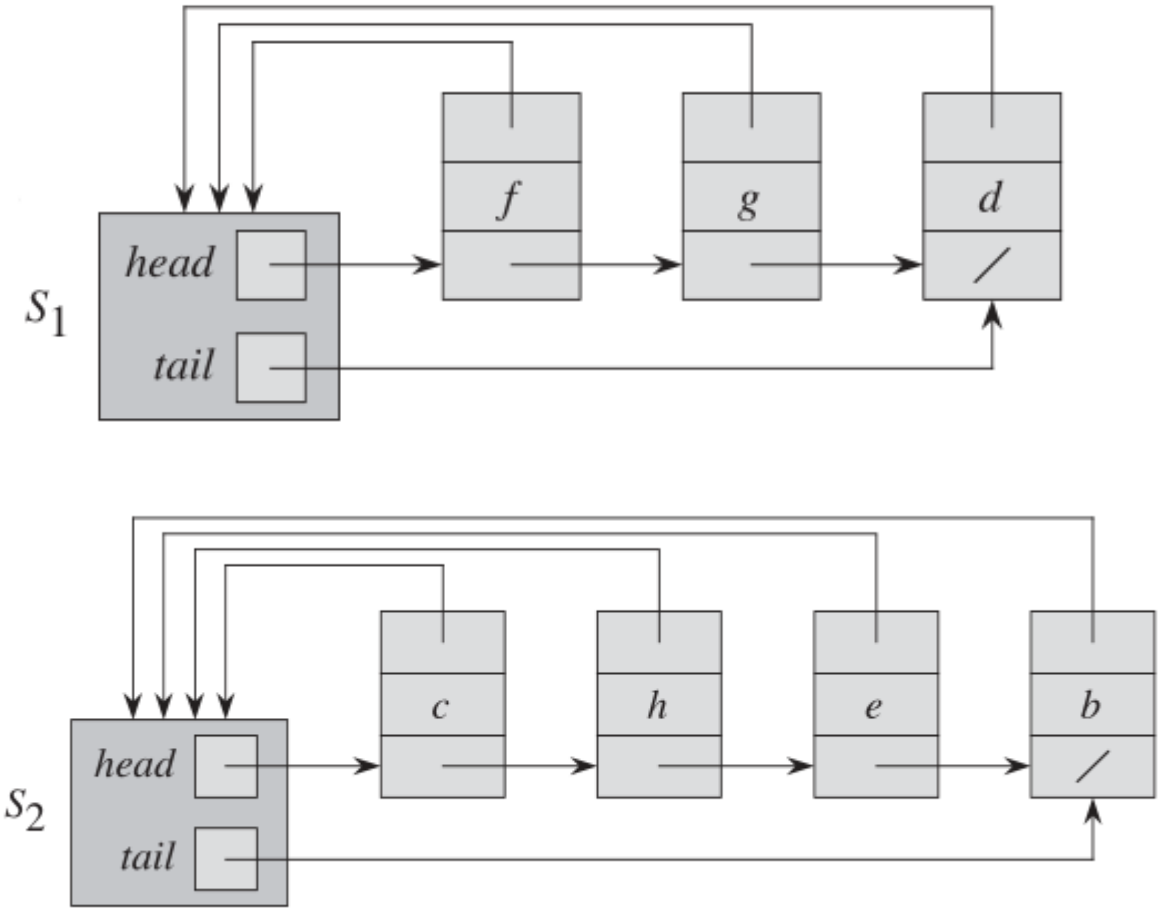
```
SAME-COMPONENT(u, v)
1  if FIND-SET(u) == FIND-SET(v)
2      return TRUE
3  else return FALSE
```

Data Structures for Disjoints Sets

Linked-List Representation of Disjoint sets

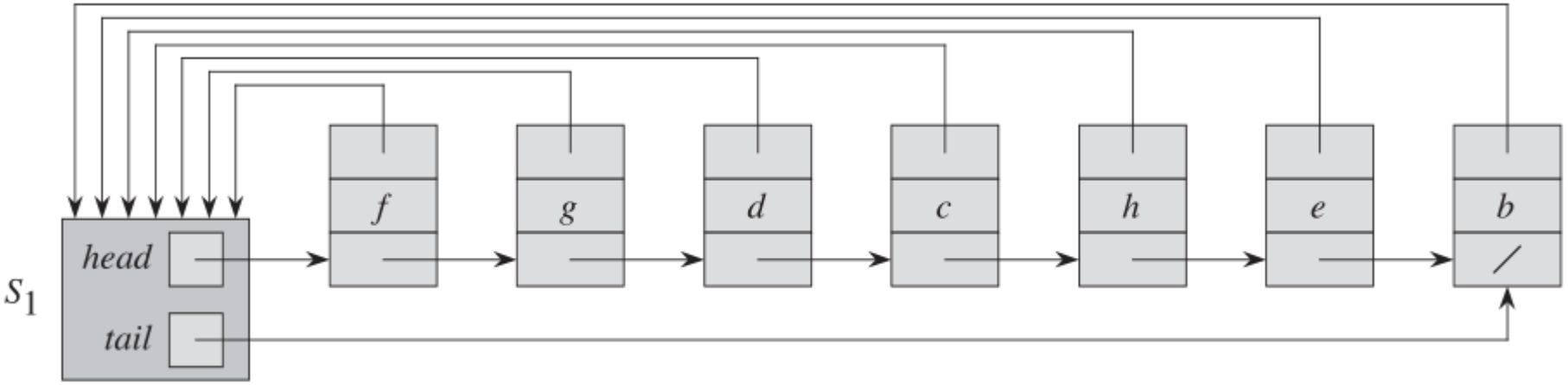
- A simple way to implement a disjoint-set data structure is representing each set by its own linked list.
- The object for each set has attributes head, pointing to the first object in the list, and tail, pointing to the last object.
- Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object.
- Within each linked list, the objects may appear in any order. The representative is the set member in the first object in the list.

Data Structures for Disjoints Sets



- Linked-list representations of two sets. Set S_1 contains members d, f , and g, with representative f , and set S_2 contains members b, c, e, and h, with representative c.
- Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object.
- Each set object has pointers head and tail to the first and last objects, respectively.

Data Structures for Disjoints Sets



- The result of UNION(g, e), which appends the linked list containing e to the linked list containing g.
- The representative of the resulting set is f . The set object for e's list, S_2, is destroyed.

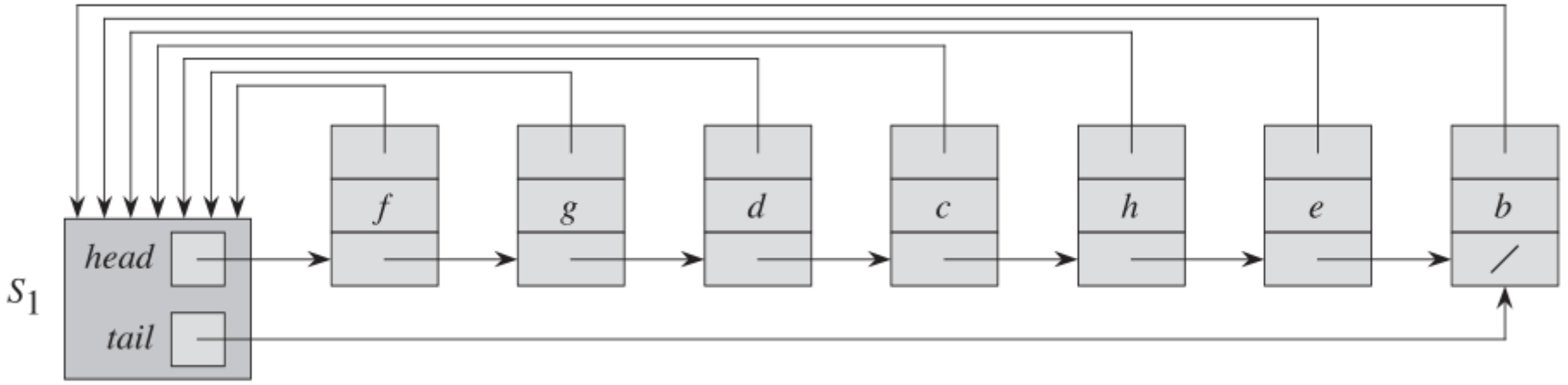
Data Structures for Disjoints Sets

- With this linked-list representation, both MAKE-SET and FIND-SET are easy, requiring O(1) time.
- To carry out MAKE-SET(x), we create a new linked list whose only object is x.
- For FIND-SET(x), we just follow the pointer from x back to its set object and then return the member in the object that head points to.

Data Structures for Disjoints Sets

- We perform UNION(x, y) by appending y's list onto the end of x's list.
- The representative of x's list becomes the representative of the resulting set.
- We use the tail pointer for x's list to quickly find where to append y's list.
- Because all members of y's list join x's list, we can destroy the set object for y's list.
- Unfortunately, we must update the pointer to the set object for each object originally on y's list, which takes time linear in the length of y's list.

Data Structures for Disjoints Sets



- In above figure, for example, the operation UNION(g, e) causes pointers to be updated in the objects for b, c, e, and h.

Data Structures for Disjoints Sets

- In fact, we can easily construct a sequence of m operations on n objects that requires $\Theta(n^2)$ time.
- Suppose that we have objects $x_1, x_2 \ldots . x_n$.
- We execute the sequence of n MAKE-SET operations followed by n - 1 UNION operations shown in Figure on next slide, so that m = 2n – 1.
- We spend Θ(n) time performing the n MAKE-SET operations.
- Because the ith UNION operation updates i objects, the total number of objects updated by all n - 1 UNION operations is

$$\sum_{i=1}^{n-1} i = \Theta(n^2) .$$

- The total number of operations is 2n - 1, and so each operation on average requires Θ(n) time. That is, the amortized time of an operation is Θ(n).

Data Structures for Disjoints Sets

Operation	Number of objects updated
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
⋮	⋮
MAKE-SET(x_n)	1
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
⋮	⋮
UNION(x_n, x_{n-1})	$n - 1$

A sequence of 2n - 1 operations on n objects that takes $\Theta(n^2)$ time, or $\Theta(n)$ time per operation on average, using the linked-list set representation and the simple implementation of UNION.

Review Questions

1) Compare Fibonacci Heaps and Quake Heaps.
2) Explain the concept of van Emde Boas Trees.
3) How can disjoint sets be represented using linked lists? Explain.
4) Explain and analyze MAKE-SET, FIND-SET, and UNION operations on disjoint sets using the linked-list representation.