BLM267

Chapter 12: Heaps

Data Structures Using C, Second Edition

- Binary Heaps
- Binomial Heaps

Binary Heaps

- A binary heap is a complete binary tree in which every node satisfies the heap property which states that:
- **If B is a child of A, then key(A) ≥ key(B)**
- This implies that elements at every node will be either greater than or equal to the element at its left and right child.
- Thus, the root node has the highest key value in the heap. Such a heap is commonly known as a ***max-heap.***
- Alternatively, elements at every node will be either less than or equal to the element at its left and right child.
- Thus, the root has the lowest key value. Such a heap is called a ***min-heap.***

Binary Heaps

- Figure 12.1 shows a binary min heap and a binary max heap.

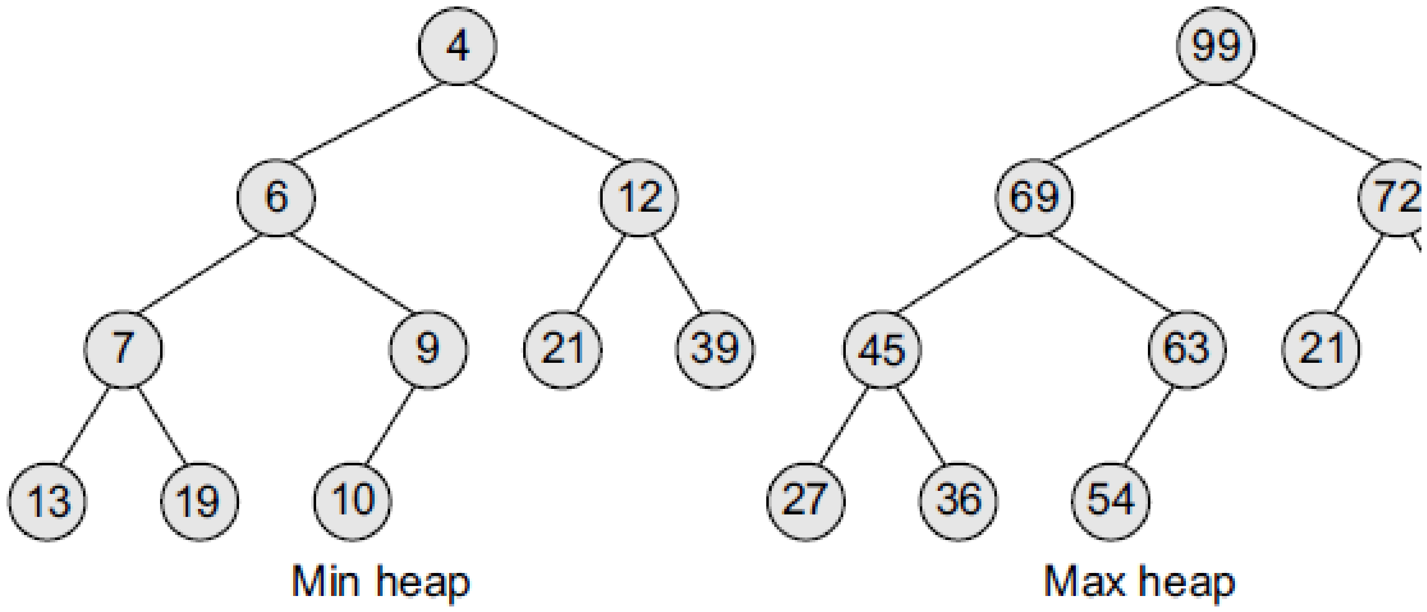


Figure 12.1 Binary heaps

Binary Heaps

- The properties of binary heaps are given as follows:
  - Since a heap is defined as a complete binary tree, all its elements can be stored sequentially in an array. It follows the same rules as that of a complete binary tree. That is, if an element is at position i in the array, then its left child is stored at position 2i and its right child at position 2i+1. Conversely, an element at position i has its parent stored at position i/2.
  - Being a complete binary tree, all the levels of the tree except the last level are completely filled.
  - The height of a binary tree is given as log2n, where n is the number of elements.
  - Heaps (also known as partially ordered trees) are a very popular data structure for implementing priority queues.
- A binary heap is a useful data structure in which elements can be added randomly but only the element with the highest value is removed in case of max heap and lowest value in case of min heap.

Binary Heaps

Inserting a New Element in a Binary Heap

- Consider a max heap H with n elements. Inserting a new value into the heap is done in the following two steps:

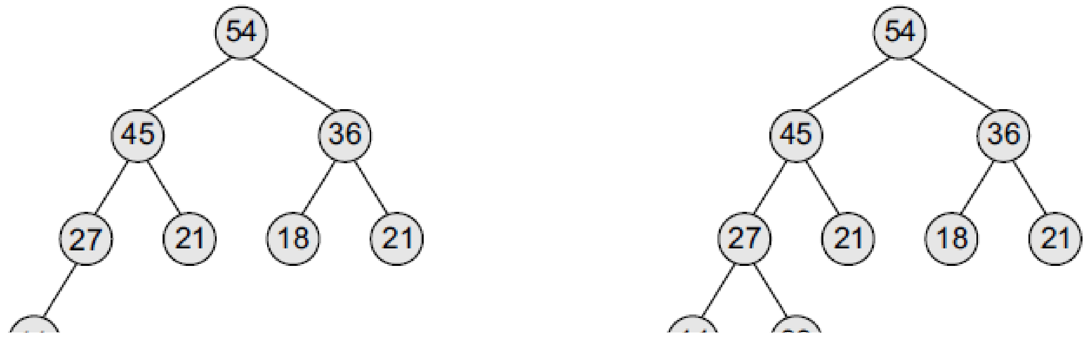
- 1. Add the new value at the bottom of H in such a way that H is still a complete binary tree but not necessarily a heap.
- 2. Let the new value rise to its appropriate place in H so that H now becomes a heap as well.

- To do this, compare the new value with its parent to check if they are in the correct order. If they are, then the procedure halts, else the new value and its parent's value are swapped and Step 2 is repeated.

Binary Heaps

Example 12.1 Consider the max heap given in Fig. 12.2 and insert 99 in it.

Solution



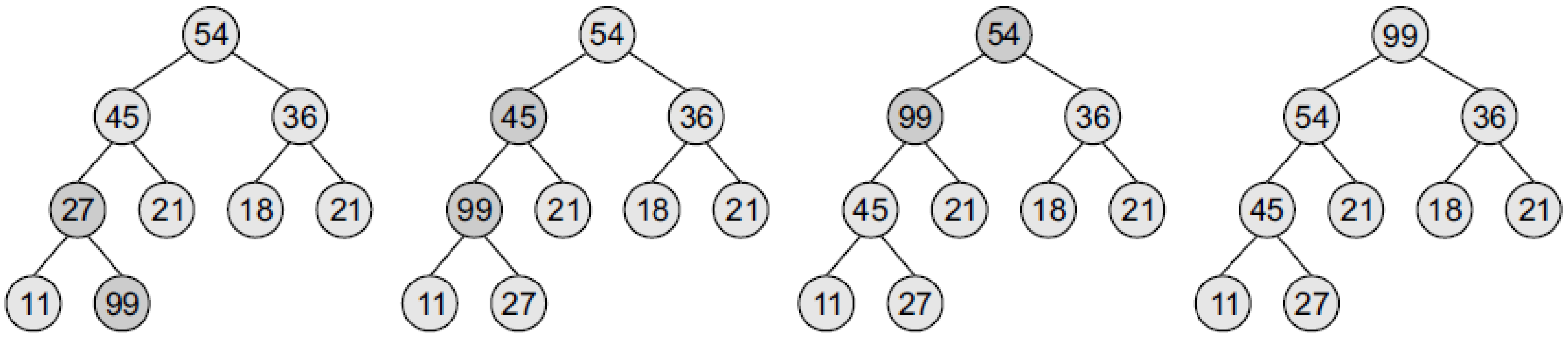


Figure 12.4 Heapify the binary heap

Binary Heaps

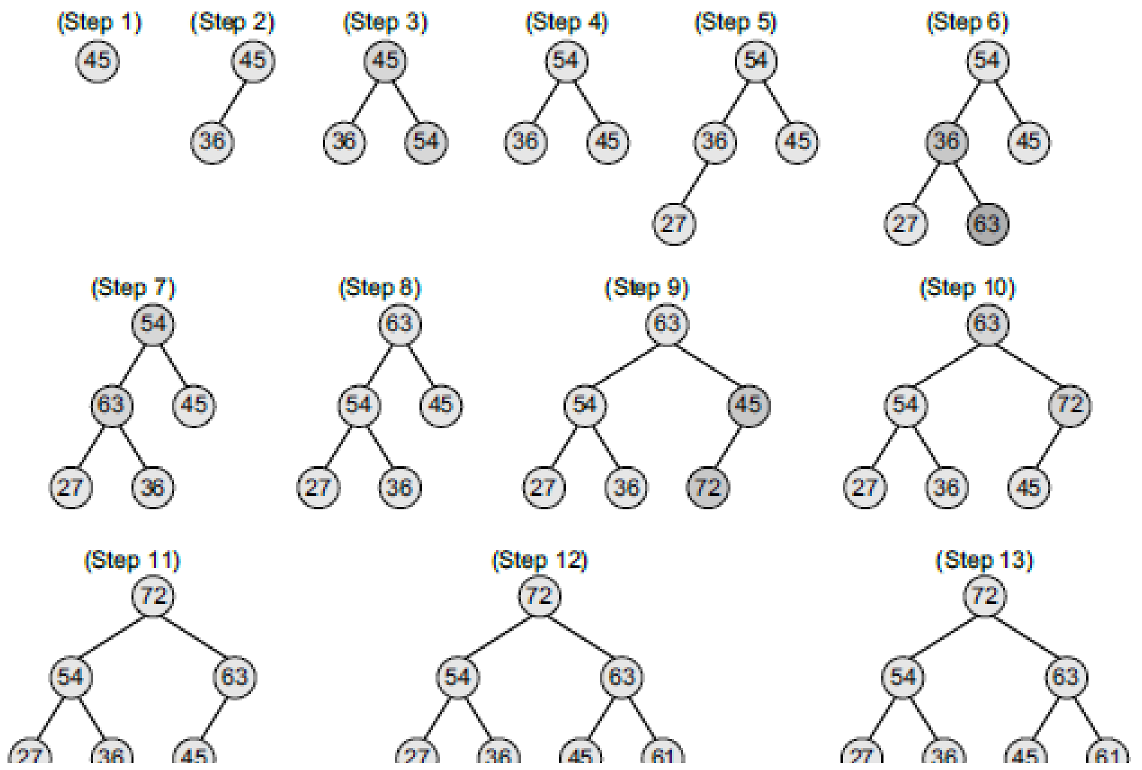
Inserting a New Element in a Binary Heap

- The first step says that insert the element in the heap so that the heap is a complete binary tree.
- So, insert the new value as the right child of node 27 in the heap. This is illustrated in Fig. 12.3.
- Now, as per the second step, let the new value rise to its appropriate place in H so that H becomes a heap as well.
- Compare 99 with its parent node value. If it is less than its parent's value, then the new node is in its appropriate place and H is a heap.
- If the new value is greater than that of its parent's node, then swap the two values.
- Repeat the whole process until H becomes a heap. This is illustrated in Fig. 12.4.

Binary Heaps

Example 12.2 Build a max heap H from the given set of numbers: 45, 36, 54, 27, 63, 72, 61, and 18. Also draw the memory representation of the heap.

Solution



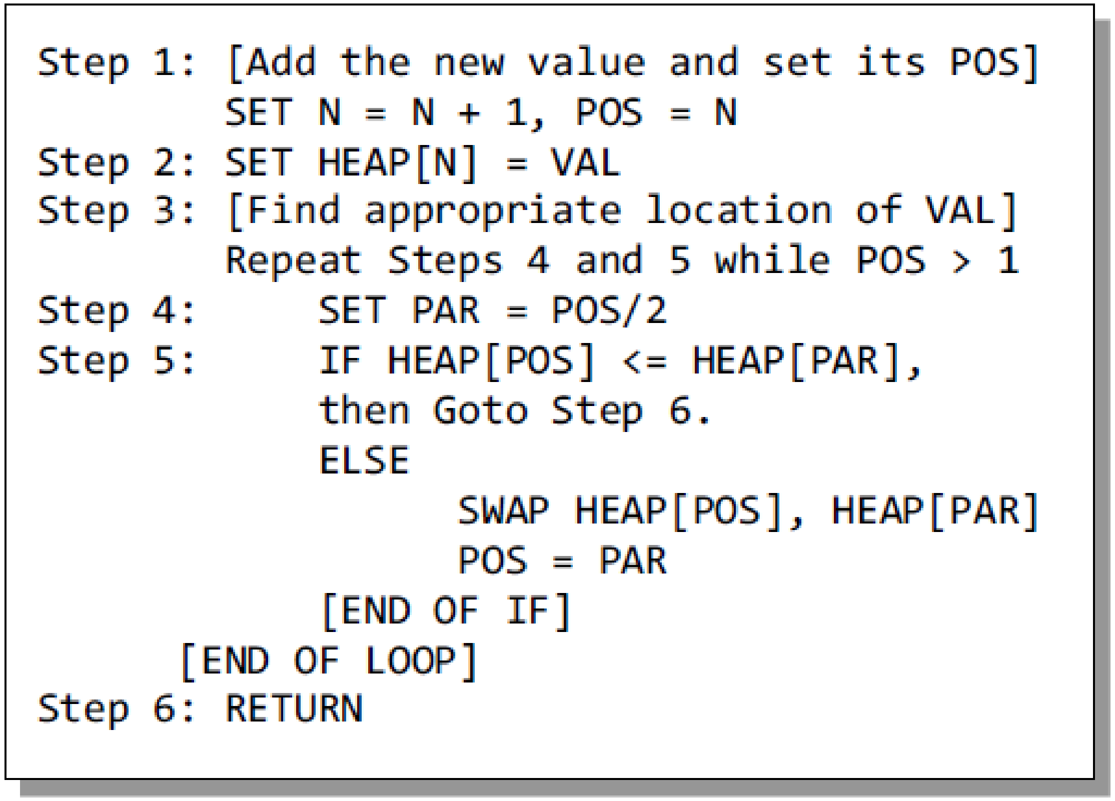
HEAP[1]	HEAP[2]	HEAP[3]	HEAP[4]	HEAP[5]	HEAP[6]	HEAP[7]	HEAP[8]	HEAP[9]	HEAP[10]
72	54	63	27	36	45	61	18		

Figure 12.6 Memory representation of binary heap H

Binary Heaps

Inserting a New Element in a Binary Heap

- After discussing the concept behind inserting a new value in the heap, let us now look at the algorithm to do so as shown in Fig. 12.7.
- We assume that H with n elements is stored in array HEAP.
- VAL has to be inserted in HEAP. The location of VAL as it rises in the heap is given by POS, and PAR denotes the location of the parent of VAL.



```
Step 1: [Add the new value and set its POS]
        SET N = N + 1, POS = N
Step 2: SET HEAP[N] = VAL
Step 3: [Find appropriate location of VAL]
        Repeat Steps 4 and 5 while POS > 1
Step 4:     SET PAR = POS/2
Step 5:     IF HEAP[POS] <= HEAP[PAR],
            then Goto Step 6.
            ELSE
                  SWAP HEAP[POS], HEAP[PAR]
                  POS = PAR
            [END OF IF]
      [END OF LOOP]
Step 6: RETURN
```

Figure 12.7 Algorithm to insert an element in a max heap

Binary Heaps

Inserting a New Element in a Binary Heap

- Note that this algorithm inserts a single value in the heap. In order to build a heap, use this algorithm in a loop.
- For example, to build a heap with 9 elements, use a for loop that executes 9 times and in each pass, a single value is inserted.
- The complexity of this algorithm in the average case is O(1).
- This is because a binary heap has O(log n) height.
- Since approximately 50% of the elements are leaves and 75% are in the bottom two levels, the new element to be inserted will only move a few levels upwards to maintain the heap.
- In the worst case, insertion of a single value may take O(log n) time and, similarly, to build a heap of n elements, the algorithm will execute in O(n log n) time.

Deleting an Element from a Binary Heap

- Consider a max heap H having n elements. An element is always deleted from the root of the heap.
- So, deleting an element from the heap is done in the following three steps:

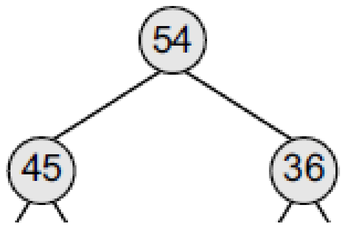
- 1. Replace the root node's value with the last node's value so that H is still a complete binary tree but not necessarily a heap.
- 2. Delete the last node.
- 3. Sink down the new root node's value so that H satisfies the heap property.
- In this step, interchange the root node's value with its child node's value (whichever is largest among its children).

Deleting an Element from a Binary Heap

- Here, the value of root node = 54 and the value of the last node = 11. So, replace 54 with 11 and delete the last node.

Example 12.3 Consider the max heap H shown in Fig. 12.8 and delete the root node's value.

Solution



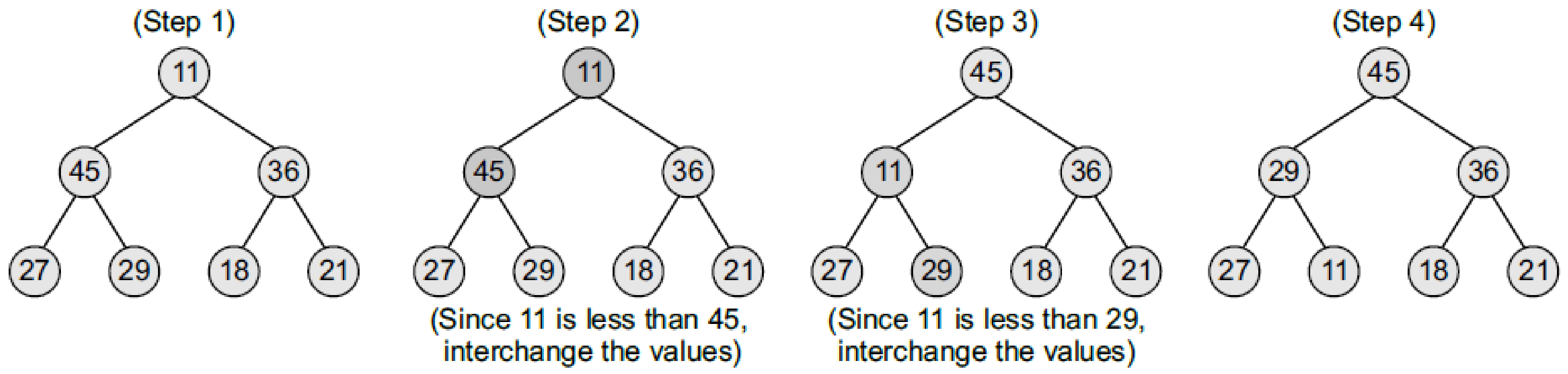

(Step 1)
11
45
36
27
29
18
21
(Step 2)
11
45
36
27
29
18
21
(Since 11 is less than 45, interchange the values)
(Step 3)
45
11
36
27
29
18
21
(Since 11 is less than 29, interchange the values)
(Step 4)
45
29
36
27
11
18
21

Figure 12.9 Binary heap

Binary Heaps

Deleting an Element from a Binary Heap

- After discussing the concept behind deleting the root element from the heap, let us look at the algorithm given in Fig. 12.10.
- We assume that heap H with n elements is stored using a sequential array called HEAP.
- LAST is the last element in the heap and PTR, LEFT, and RIGHT denote the position of LAST and its left and right children respectively as it moves down the heap.

```
Step 1: [Remove the last node from the heap]
        SET LAST = HEAP[N], SET N = N - 1
Step 2: [Initialization]
        SET PTR = 1, LEFT = 2, RIGHT = 3
Step 3: SET HEAP[PTR] = LAST
Step 4: Repeat Steps 5 to 7 while LEFT <= N
Step 5: IF HEAP[PTR] >= HEAP[LEFT] AND
         HEAP[PTR] >= HEAP[RIGHT]
              Go to Step 8
        [END OF IF]
Step 6: IF HEAP[RIGHT] <= HEAP[LEFT]
              SWAP HEAP[PTR], HEAP[LEFT]
              SET PTR = LEFT
        ELSE
              SWAP HEAP[PTR], HEAP[RIGHT]
              SET PTR = RIGHT
        [END OF IF]
Step 7: SET LEFT = 2 * PTR and RIGHT = LEFT + 1
        [END OF LOOP]
Step 8: RETURN
```

Figure 12.10 Algorithm to delete the root element from a max heap

Binary Heaps

Applications of Binary Heaps

- Binary heaps are mainly applied for
  1. Sorting an array using *heapsort algorithm.*
  2. Implementing priority queues.

Binary Heap Implementation of Priority Queues

- A priority queue is similar to a queue in which an item is dequeued (or removed) from the front.
- However, unlike a regular queue, in a priority queue the logical order of elements is determined by their priority.
- While the higher priority elements are added at the front of the queue, elements with lower priority are added at the rear.
- Though we can easily implement priority queues using a linear array, but we should first consider the time required to insert an element in the array and then sort it.
- We need O(n) time to insert an element and at least O(n log n) time to sort the array.
- Therefore, a better way to implement a priority queue is by using a binary heap which allows both enqueue and dequeue of elements in O(log n) time

Heap Sort

- Heap sort has a running time complexity of O(nlogn).
- Given an array ARR with n elements, the heap sort algorithm can be used to sort ARR in two phases:
- In phase 1, build a heap H using the elements of ARR.
- In phase 2, repeatedly delete the root element of the heap formed in phase 1.
- In a max heap, we know that the largest value in H is always present at the root node. So in phase 2, when the root element is deleted, we are actually collecting the elements

```
HEAPSORT(ARR, N)

Step 1: [Build Heap H]
        Repeat for I = 0 to N-1
                CALL Insert_Heap(ARR, N, ARR[I])
        [END OF LOOP]
Step 2: (Repeatedly delete the root element)
        Repeat while N>0
                CALL Delete_Heap(ARR, N, VAL)
                SET N = N + 1
        [END OF LOOP]
Step 3: END
```

Figure 14.12 Algorithm for heap sort

Heap Sort

- ***Complexity of Heap Sort***
- Heap sort uses two heap operations: *insertion and root deletion. Each element extracted from the* root is placed in the last empty location of the array.
- In phase 1, when we build a heap, the number of comparisons to find the right location of the new element in H cannot exceed the depth of H.
- Since H is a complete tree, its depth cannot exceed m, where m is the number of elements in heap H.
- Thus, the total number of comparisons g(n) to insert n elements of ARR in H is bounded as:

  g(n) <= n log n
- Hence, the running time of the first phase of the heap sort algorithm is O(n log n).

Heap Sort

- ***Complexity of Heap Sort***
- In phase 2, we have H which is a complete tree with m elements having left and right sub-trees as heaps.
- Assuming L to be the root of the tree, *reheaping the tree would need 4 comparisons to* move L one step down the tree H. Since the depth of H cannot exceed O(log m), reheaping the tree will require a maximum of 4 log m comparisons to find the right location of L in H.
- Since n elements will be deleted from heap H, reheaping will be done n times. Therefore, the number of comparisons to delete n elements is bounded as:

  $h(n) <= 4n \log n$
- Hence, the running time of the second phase of the heap sort algorithm is O(n log n).
- Each phase requires time proportional to O(n log n). Therefore, the running time to sort an array of n elements in the worst case is proportional to O(n log n).
- Therefore, we can conclude that heap sort is a simple, fast, and stable sorting algorithm that can be used to sort large sets of data efficiently.

Binomial Heaps

- A binomial heap H is a set of binomial trees that satisfy the binomial heap properties.
- First, let us discuss what a binomial tree is.
- A *binomial tree is an ordered tree that can be recursively defined as follows:*
  - A binomial tree of order 0 has a single node.
  - A binomial tree of order i has a root node whose children are the root nodes of binomial trees of order i–1, i–2, ..., 2, 1, and 0.
  - A binomial tree Bi has 2^I nodes.
  - The height of a binomial tree Bi is i.

Binomial Heaps

- Look at Fig. 12.12 which shows a few binomial trees of different orders. We can construct a binomial tree Bi from two binomial trees of order Bi–1 by linking them together in such a way that the root of one is the leftmost child of the root of another.

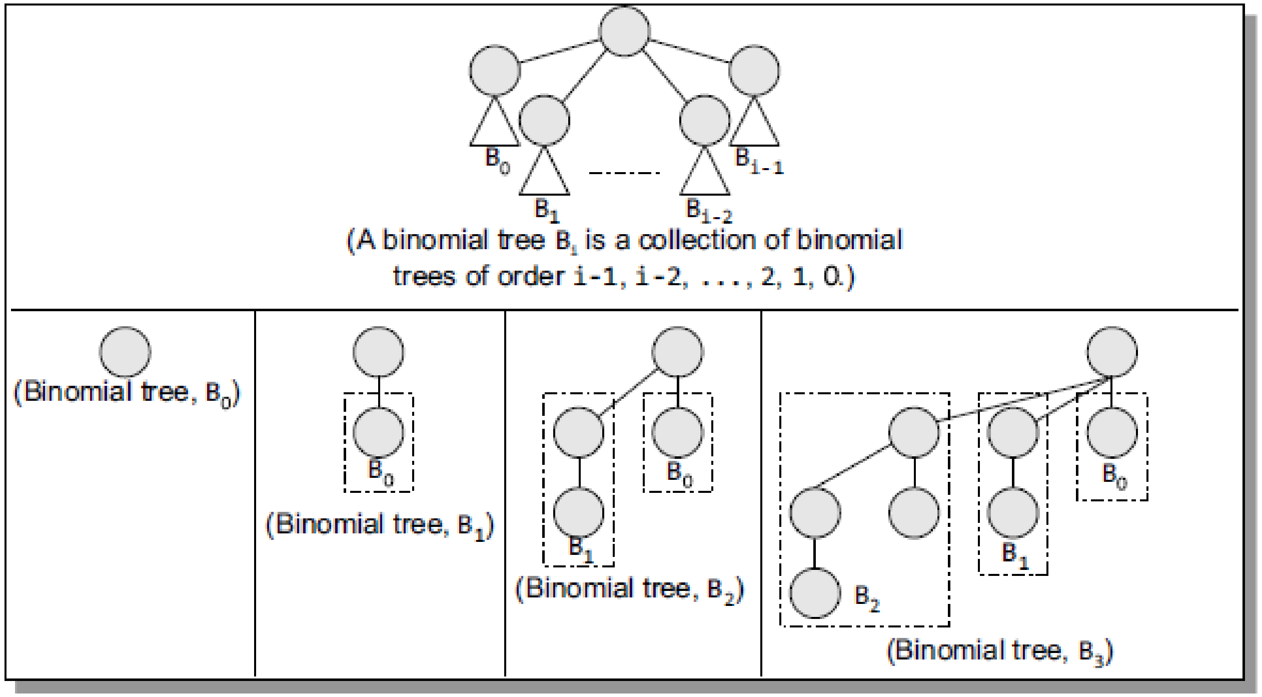


Figure 12.12 Binomial trees

Binomial Heaps

- A *binomial heap H is a collection of binomial trees that satisfy the following properties:*
  - Every binomial tree in H satisfies the minimum heap property (i.e., the key of a node is either greater than or equal to the key of its parent).
  - There can be one or zero binomial trees for each order including zero order.
- According to the first property, the root of a heap-ordered tree contains the smallest key in the tree.
- The second property, on the other hand, implies that a binomial heap H having N nodes contains at most log (N + 1) binomial trees.

Binomial Heaps

- **Linked Representation of Binomial Heaps**
- Each node in a binomial heap H has a val field that stores its value. In addition, each node N has following pointers:
  - P(N) that points to the parent of N
  - Child(N) that points to the leftmost child
  - Sibling(N) that points to the sibling of N which is immediately to its right
- If N is the root node, then P(N) = NULL. If N has no children, then Child(N) = NULL, and if N is the rightmost child of its parent, then Sibling(N) = NULL.
- In addition to this, every node N has a degree field which stores the number of children of N. Look at the binomial heap shown in Fig. 12.13.
- Figure 12.14 shows its corresponding linked representation.

Binomial Heaps

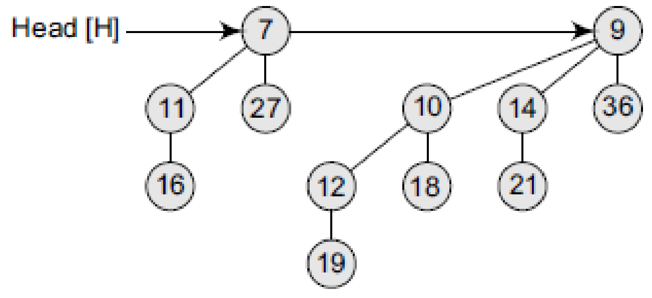


Figure 12.13 Binomial heap

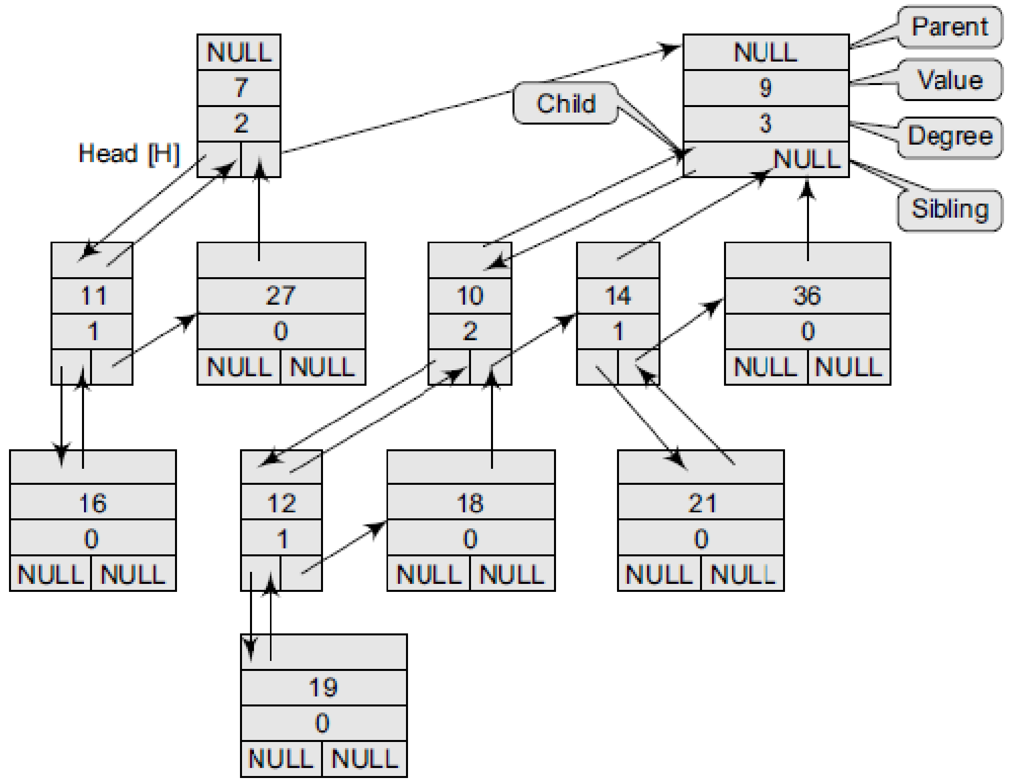


Figure 12.14 Linked representation of the binomial tree shown in Fig. 12.13

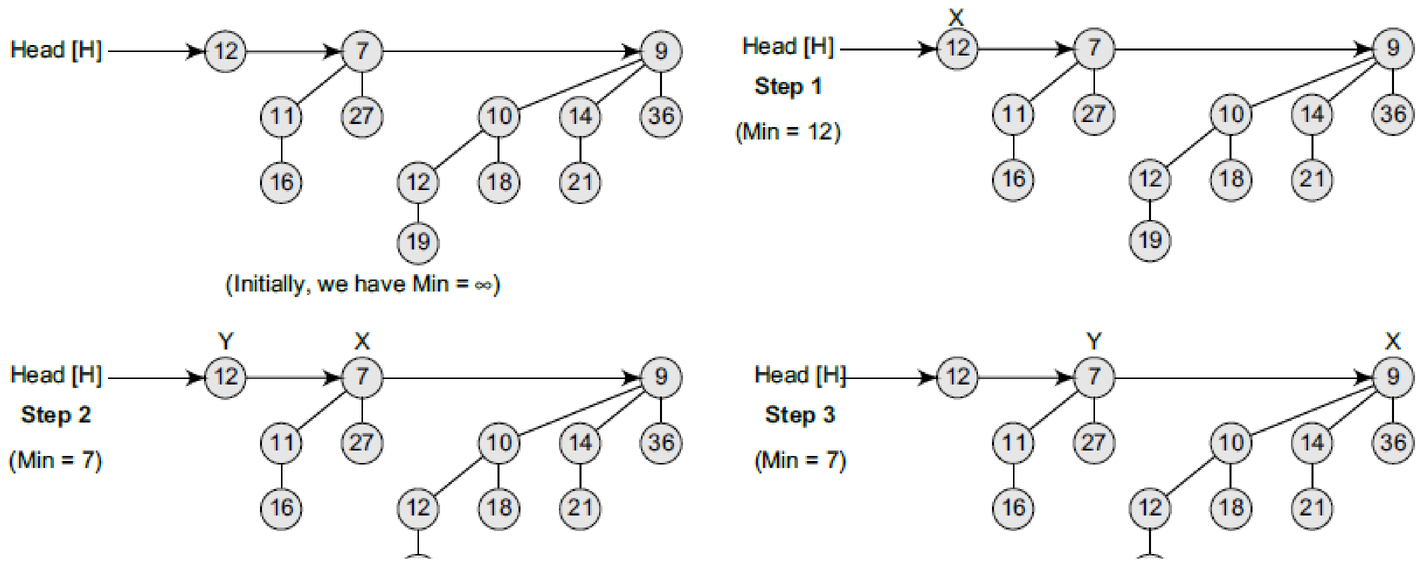
Binomial Heaps

- **Operations on Binomial Heaps**
- In this section, we will discuss the different operations that can be performed on binomial heaps.
- ***Creating a New Binomial Heap***
- The procedure Create_Binomial–Heap() allocates and returns an object H, where Head(H) is set to NULL. The running time of this procedure can be given as O(1).
- ***Finding the Node with Minimum Key***
- The procedure Min_Binomial–Heap() returns a pointer to the node which has the minimum value in the binomial heap H. The algorithm for Min_Binomial–Heap() is shown in Fig. 12.15.
- We have already discussed that a binomial heap is heap-ordered; therefore, the node with the minimum value in a particular binomial tree will appear as a root node in the binomial heap.
- Thus, the Min_ Binomial–Heap() procedure checks all roots. Since there are at most log (n + 1) roots to check, the running time of this procedure is O(log n).

Binomial Heaps

```
Min_Binomial-Heap(H)

Step 1: [INITIALIZATION] SET Y = NULL, X = Head[H] and Min = ∞
Step 2: REPEAT Steps 3 and 4 While X ≠ NULL
Step 3:     IF Val[X] < Min
                  SET Min = Val[X]
                  SET Y = X
            [END OF IF]
Step 4:     SET X = Sibling[X]
        [END OF LOOP]
Step 5: RETURN Y
```

Example 12.4 Consider the binomial heap given below and see how the procedure works in this case.



Binomial Heaps

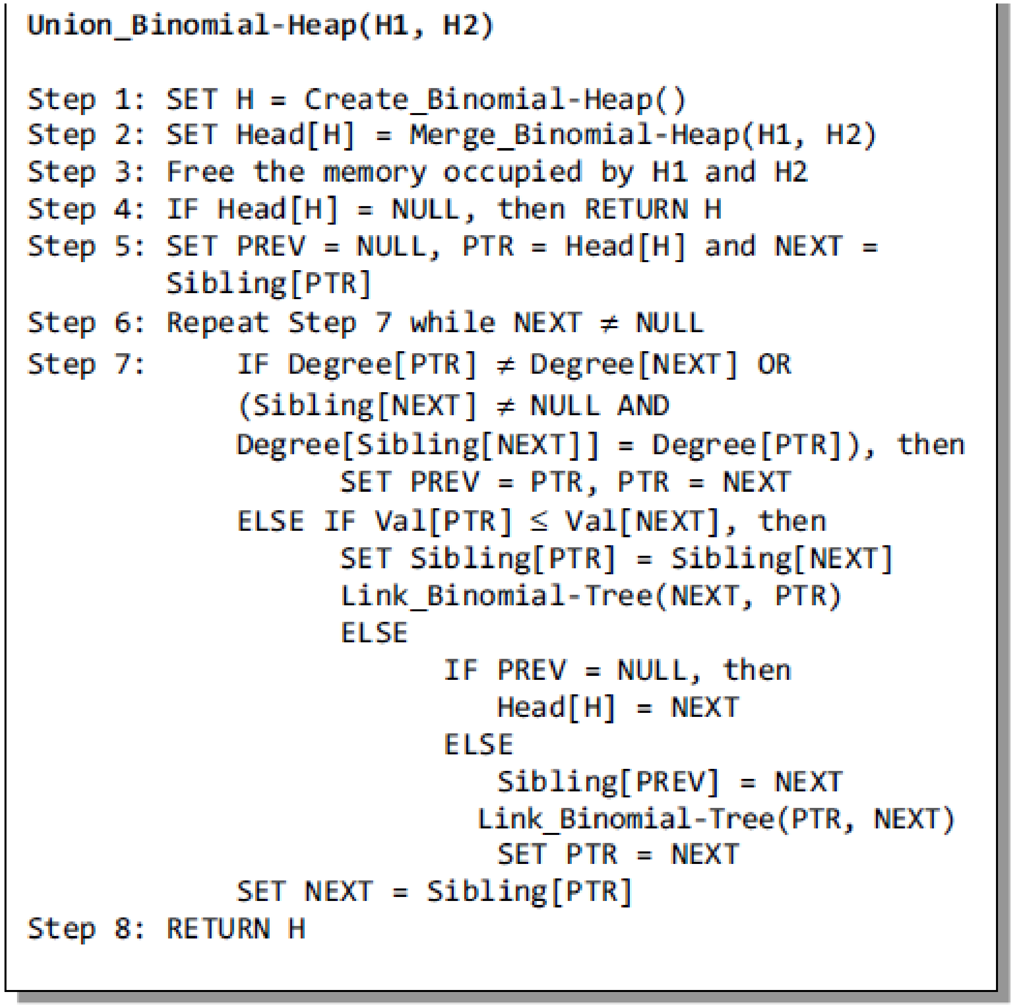
- ***Uniting Two Binomial Heaps***
- The procedure of uniting two binomial heaps is used as a subroutine by other operations.
- The Link_Binomial–Tree() procedure links together binomial trees whose roots have the same degree.
- The algorithm to link Bi–1 tree rooted at node Y to the Bi–1 tree rooted at node Z, making Z the parent of Y, is shown in Fig. 12.17.
- The Link_Binomial–Tree() procedure makes Y the new head of the linked list of node Z's children in O(1) time.

```
Link_Binomial-Tree(Y, Z)

Step 1: SET Parent[Y] = Z
Step 2: SET Sibling[Y] = Child[Z]
Step 3: SET Child[Z] = Y
Step 4: Set Degree[Z] = Degree[Z]+ 1
Step 5: END
```

Figure 12.17 Algorithm to link two binomial trees

Binomial Heaps

- ***Uniting Two Binomial Heaps***
- The algorithm to unite two binomial heaps H1 and H2 is given in Fig. 12.18.



```
Union_Binomial-Heap(H1, H2)

Step 1: SET H = Create_Binomial-Heap()
Step 2: SET Head[H] = Merge_Binomial-Heap(H1, H2)
Step 3: Free the memory occupied by H1 and H2
Step 4: IF Head[H] = NULL, then RETURN H
Step 5: SET PREV = NULL, PTR = Head[H] and NEXT =
        Sibling[PTR]
Step 6: Repeat Step 7 while NEXT ≠ NULL
Step 7:     IF Degree[PTR] ≠ Degree[NEXT] OR
            (Sibling[NEXT] ≠ NULL AND
            Degree[Sibling[NEXT]] = Degree[PTR]), then
                  SET PREV = PTR, PTR = NEXT
            ELSE IF Val[PTR] ≤ Val[NEXT], then
                  SET Sibling[PTR] = Sibling[NEXT]
                  Link_Binomial-Tree(NEXT, PTR)
                  ELSE
                        IF PREV = NULL, then
                           Head[H] = NEXT
                        ELSE
                           Sibling[PREV] = NEXT
                          Link_Binomial-Tree(PTR, NEXT)
                           SET PTR = NEXT
            SET NEXT = Sibling[PTR]
Step 8: RETURN H
```

Figure 12.18 Algorithm to unite two binomial heaps

Binomial Heaps

- ***Uniting Two Binomial Heaps***
- The algorithm destroys the original representations of heaps H1 and H2.
- Apart from Link_Binomial–Tree(), it uses another procedure Merge_Binomial–Heap() which is used to merge the root lists of H1 and H2 into a single linked list that is sorted by degree into a monotonically increasing order.
- In the algorithm, Steps 1 to 3 merge the root lists of binomial heaps H1 and H2 into a single root list H in such a way that H1 and H2 are sorted strictly by increasing degree.
- Merge_Binomial–Heap() returns a root list H that is sorted by monotonically increasing degree.
- If there are m roots in the root lists of H1 and H2, then Merge_Binomial–Heap() runs in O(m) time.
- This procedure repeatedly examines the roots at the heads of the two root lists and appends the root with the lower degree to the output root list, while removing it from its input root list.

Binomial Heaps

- ***Uniting Two Binomial Heaps***
- Step 4 of the algorithm checks if there is at least one root in the heap H.
- The algorithm proceeds only if H has at least one root.
- In Step 5, we initialize three pointers: PTR which points to the root that is currently being examined, PREV which points to the root preceding PTR on the root list, and NEXT which points to the root following PTR on the root list.
- In Step 6, we have a while loop in which at each iteration, we decide whether to link PTR to NEXT or NEXT to PTR depending on their degrees and possibly the degree of sibling(NEXT).

Binomial Heaps

- ***Uniting Two Binomial Heaps***
- In Step 7, we check for two conditions. First, if degree(PTR) ≠ degree(NEXT), that is, when PTR is the root of a Bi tree and NEXT is the root of a Bj tree for some j > i, then PTR and NEXT are not linked to each other, but we move the pointers one position further down the list.
- Second, we check if PTR is the first of three roots of equal degree, that is,

  degree(PTR) = degree(NEXT) = degree(Sibling(NEXT))
- In this case also, we just move the pointers one position further down the list by writing PREV= PTR, PTR = NEXT.
- However, if the above IF conditions do not satisfy, then the case that pops up is that PTR is the first of two roots of equal degree, that is,

  degree(PTR) = degree(NEXT) ≠ degree(Sibling(NEXT))
- In this case, we link either PTR with NEXT or NEXT with PTR depending on whichever has the smaller key.
- Of course, the node with the smaller key will be the root after the two nodes are linked.

Binomial Heaps

- ***Uniting Two Binomial Heaps***
- The running time of Union_Binomial–Heap() can be given as O(log n), where n is the total number of nodes in binomial heaps H_1 and H_2.
- If H_1 contains n_1 nodes and H_2 contains n_2 nodes, then H_1 contains at most $\log(n_1 + 1)$ roots and H_2 contains at most $\log(n_2 + 1)$ roots, so H contains at most $(\log n_2 + \log n_1 + 2) \leqslant (2 \log n + 2) = O(\log n)$ roots when we call Merge_Binomial–Heap().
- Since, $n = n_1 + n_2$, the Merge_Binomial–Heap() takes O(log n) to execute.
- Each iteration of the while loop takes O(1) time, and because there are at most $(\log n_1 + \log n_2 + 2)$ iterations, the total time is thus O(log n).

Binomial Heaps

- *Uniting Two Binomial Heaps*

Example 12.5 Unite the binomial heaps given below.

Solution

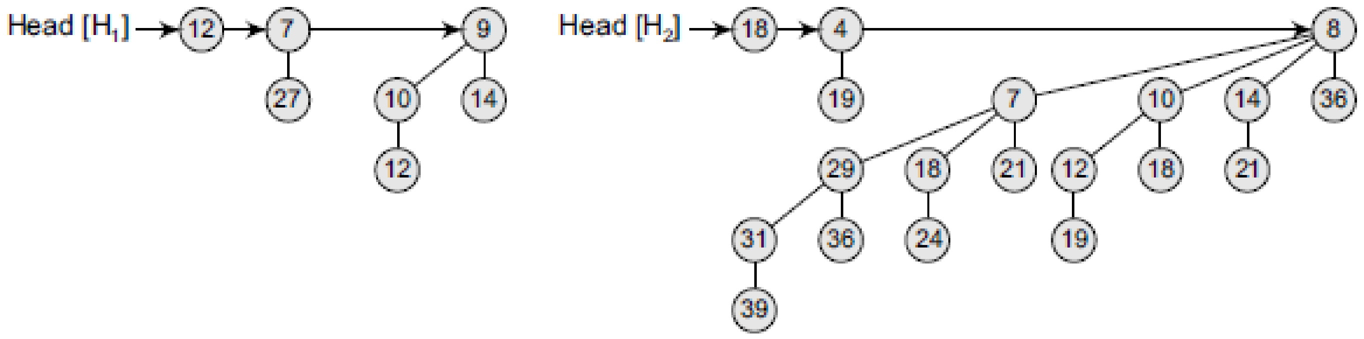
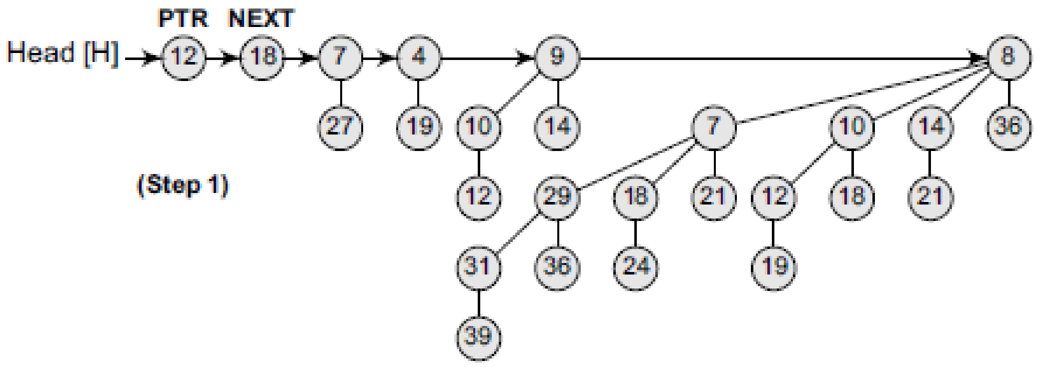


Figure 12.19(a)

After `Merge_Binomial-Heap()`, the resultant heap can be given as follows:



Binomial Heaps

- ***Uniting Two Binomial Heaps***

Link NEXT to PTR, making PTR the parent of the node pointed by NEXT.

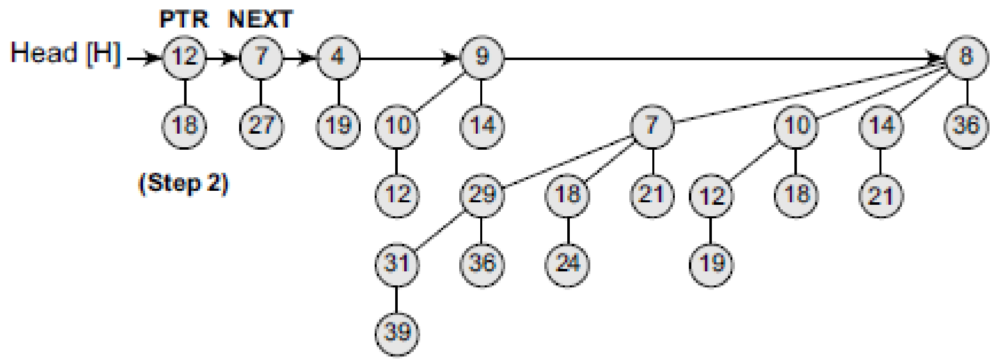


Figure 12.19(c)

Now PTR is the first of the three roots of equal degree, that is, degree[PTR] = degree[NEXT] = degree[sibling[NEXT]]. Therefore, move the pointers one position further down the list by writing PREV = PTR, PTR = NEXT, and NEXT = sibling[PTR].

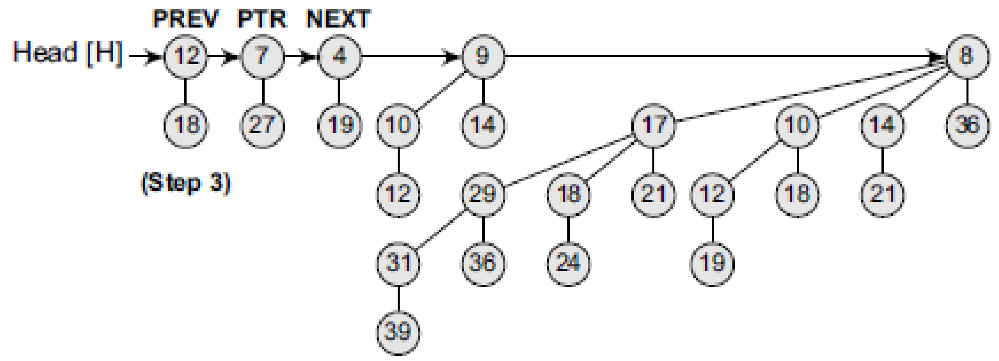


Figure 12.19(d)

Binomial Heaps

- ***Uniting Two Binomial Heaps***

Link PTR to NEXT, making NEXT the parent of the node pointed by PTR.

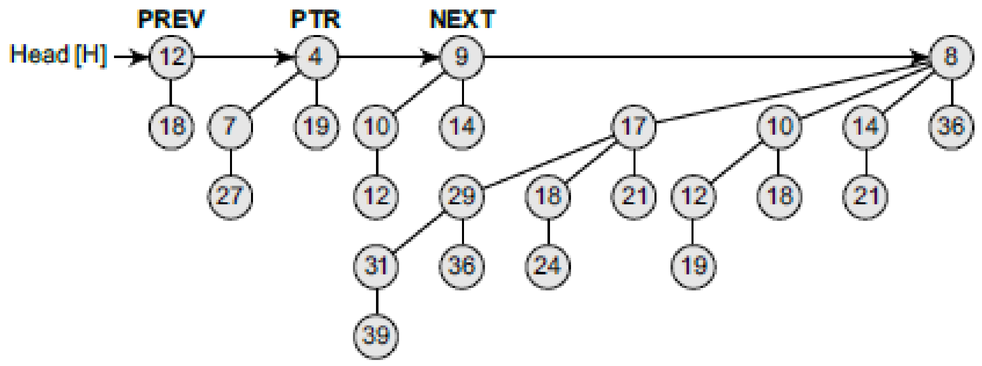


Figure 12.19(e)

Link NEXT to PTR, making PTR the parent of the node pointed by NEXT.

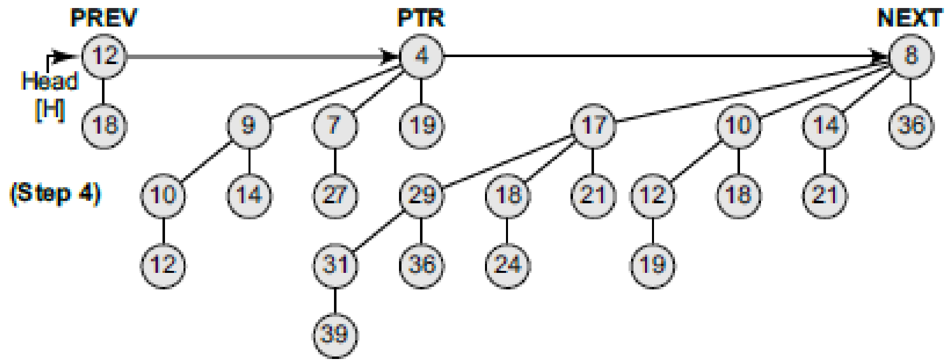
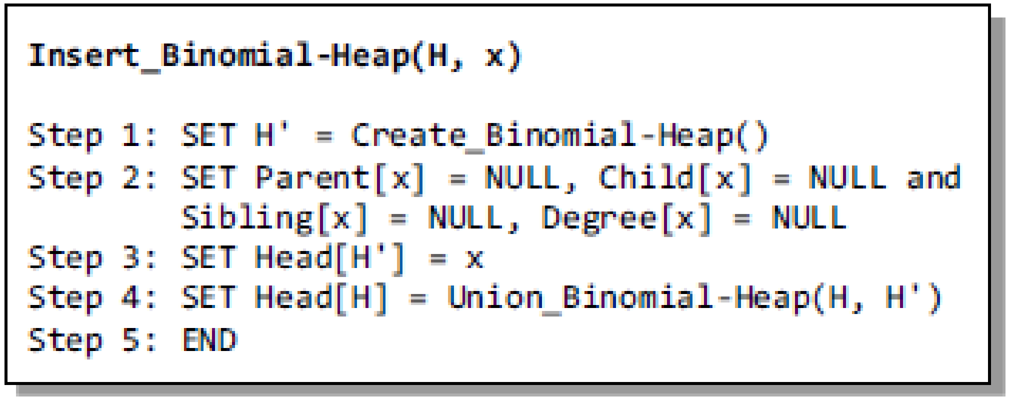


Figure 12.19(f) Binomial heap

Binomial Heaps

- ***Inserting a New Node***
- The Insert_Binomial–Heap() procedure is used to insert a node x into the binomial heap H.
- *The* pre-condition of this procedure is that x has already been allocated space and val(x) has already been filled in.
- The algorithm shown in Fig. 12.20 simply makes a binomial heap H′ in O(1) time.
- H′ contains just one node which is x.
- Finally, the algorithm unites H′ with the n-node binomial heap H in O(log n) time.
- Note that the memory occupied by H′ is freed in the Union_Binomial–Heap(H, H′) procedure.

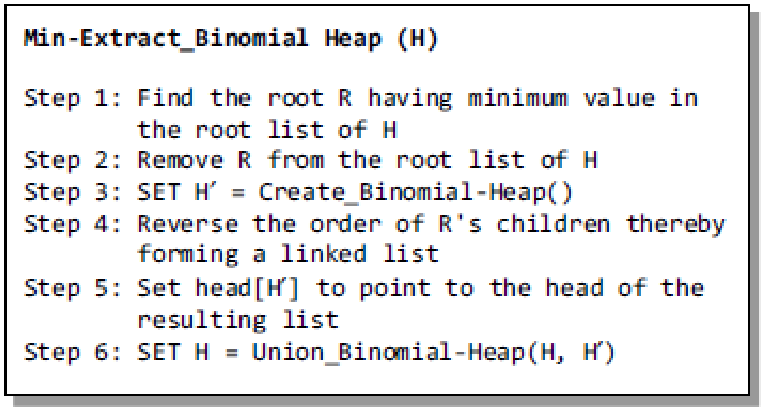


```
Insert_Binomial-Heap(H, x)

Step 1: SET H' = Create_Binomial-Heap()
Step 2: SET Parent[x] = NULL, Child[x] = NULL and
        Sibling[x] = NULL, Degree[x] = NULL
Step 3: SET Head[H'] = x
Step 4: SET Head[H] = Union_Binomial-Heap(H, H')
Step 5: END
```

Figure 12.20 Algorithm to insert a new element in a binomial heap

Binomial Heaps

- ***Extracting the Node with Minimum Key***
- The algorithm to extract the node with minimum key from a binomial heap H is shown in Fig. 12.21.
- The Min–Extract_Binomial–Heap procedure accepts a heap H as a parameter and returns a pointer to the extracted node.
- In the first step, it finds a root node R with the minimum value and removes it from the root list of H.
- Then, the order of R's children is reversed and they are all added to the root list of
- H'.
- Finally, Union_Binomial–Heap (H, H') is called to unite the two heaps and R is returned.
- The algorithm Min–Extract_Binomial–Heap() runs in O(log n) time, where n is the number of nodes in H.



```
Min-Extract_Binomial Heap (H)

Step 1: Find the root R having minimum value in
        the root list of H
Step 2: Remove R from the root list of H
Step 3: SET H' = Create_Binomial-Heap()
Step 4: Reverse the order of R's children thereby
        forming a linked list
Step 5: Set head[H'] to point to the head of the
        resulting list
Step 6: SET H = Union_Binomial-Heap(H, H')
```

Figure 12.21 Algorithm to extract the node with minimum key from a binomial heap

Binomial Heaps

- ***Extracting the Node with Minimum Key***

Example 12.6 Extract the node with the minimum value from the given binomial heap.

Solution

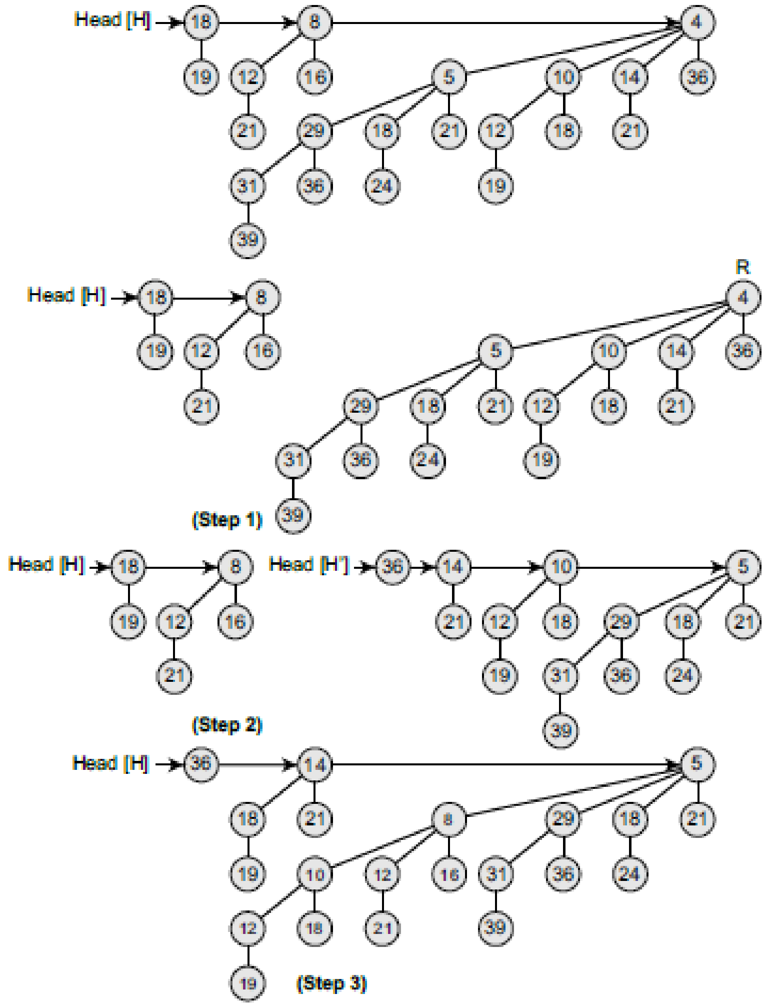


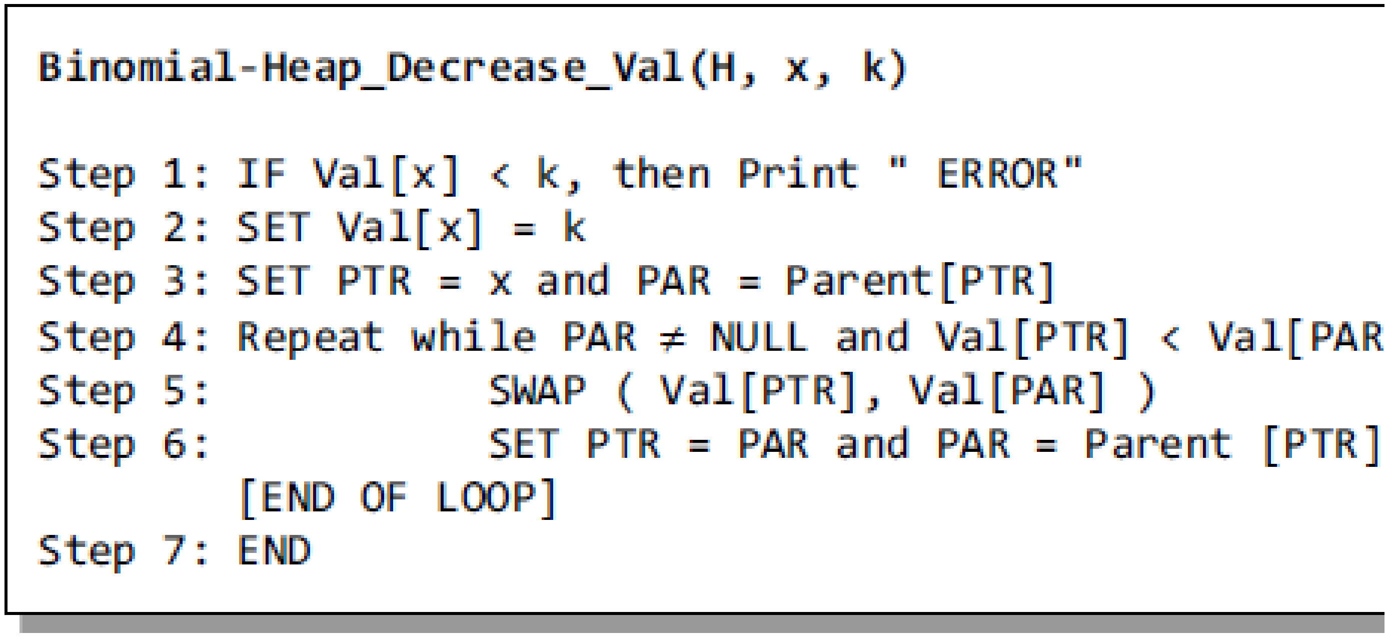
Figure 12.22 Binomial heap

Binomial Heaps

- ***Decreasing the Value of a Node***
- The algorithm to decrease the value of a node x in a binomial heap H is given in Fig. 12.23.
- In the algorithm, the value of the node is overwritten with a new value k, *which is less than the current* value of the node.
- In the algorithm, we first ensure that the new value is not greater than the current value and then assign the new value to the node.
- We then go up the tree with PTR initially pointing to node x. In each iteration of the while loop, val(PTR) is compared with the value of its parent PAR.
- However, if either PTR is the root or key(PTR) ≥ key(PAR), then the binomial tree is heap-ordered.
- Otherwise, node PTR violates heap-ordering, so its key is exchanged with that of its parent. We set PTR = PAR and PAR = Parent(PTR) to move up one level in the tree and continue the process.
- The Binomial–Heap_Decrease_Val procedure takes O(log n) time as the maximum depth of node x is log n, so the while loop will iterate at most log n times.

Binomial Heaps

- ***Decreasing the Value of a Node***



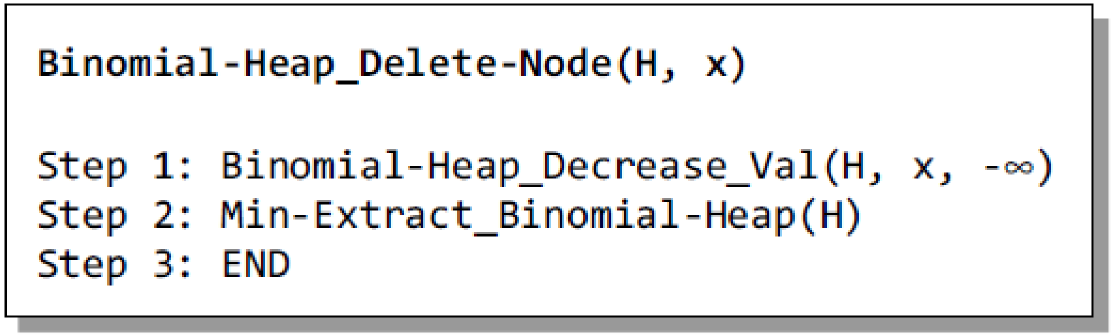
```
Binomial-Heap_Decrease_Val(H, x, k)

Step 1: IF Val[x] < k, then Print " ERROR"
Step 2: SET Val[x] = k
Step 3: SET PTR = x and PAR = Parent[PTR]
Step 4: Repeat while PAR ≠ NULL and Val[PTR] < Val[PAR
Step 5:             SWAP ( Val[PTR], Val[PAR] )
Step 6:             SET PTR = PAR and PAR = Parent [PTR]
        [END OF LOOP]
Step 7: END
```

Figure 12.23 Algorithm to decrease the value of a node x in a binoı heap H

Binomial Heaps

- ***Deleting a Node***
- Once we have understood the Binomial–Heap_Decrease_Val procedure, it becomes easy to delete a node x's value from the binomial heap H in O(log n) time.
- To start with the algorithm, we set the value of x to –∞. Assuming that there is no node in the heap that has a value less than –∞, the algorithm to delete a node from a binomial heap can be given as shown in Fig. 12.24.
- The Binomial–Heap_Delete–Node procedure sets the value of x to –∞, which is a unique minimum value in the entire binomial heap.
- The Binomial–Heap_Decrease_Val algorithm bubbles this key up to a root and then this root is removed from the heap by making a call to the Min–Extract_Binomial–Heap procedure

es O(log n) time.



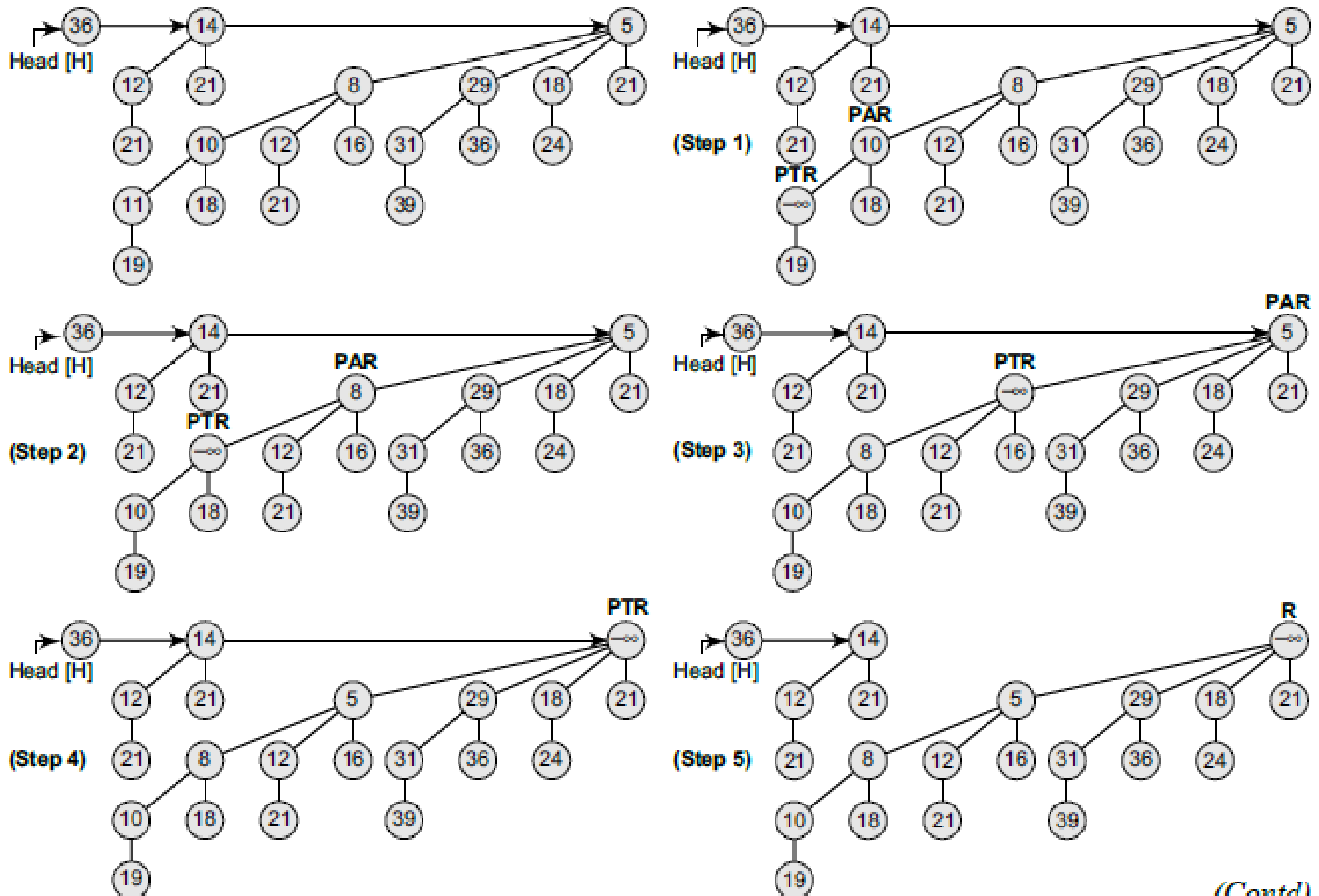
```
Binomial-Heap_Delete-Node(H, x)

Step 1: Binomial-Heap_Decrease_Val(H, x, -∞)
Step 2: Min-Extract_Binomial-Heap(H)
Step 3: END
```

Figure 12.24 Algorithm to delete a node from a bionomial heap

Binomial Heaps

Example 12.7 Delete the node with the value 11 from the binomial heap H.

Solution



(Contd)

Binomial Heaps

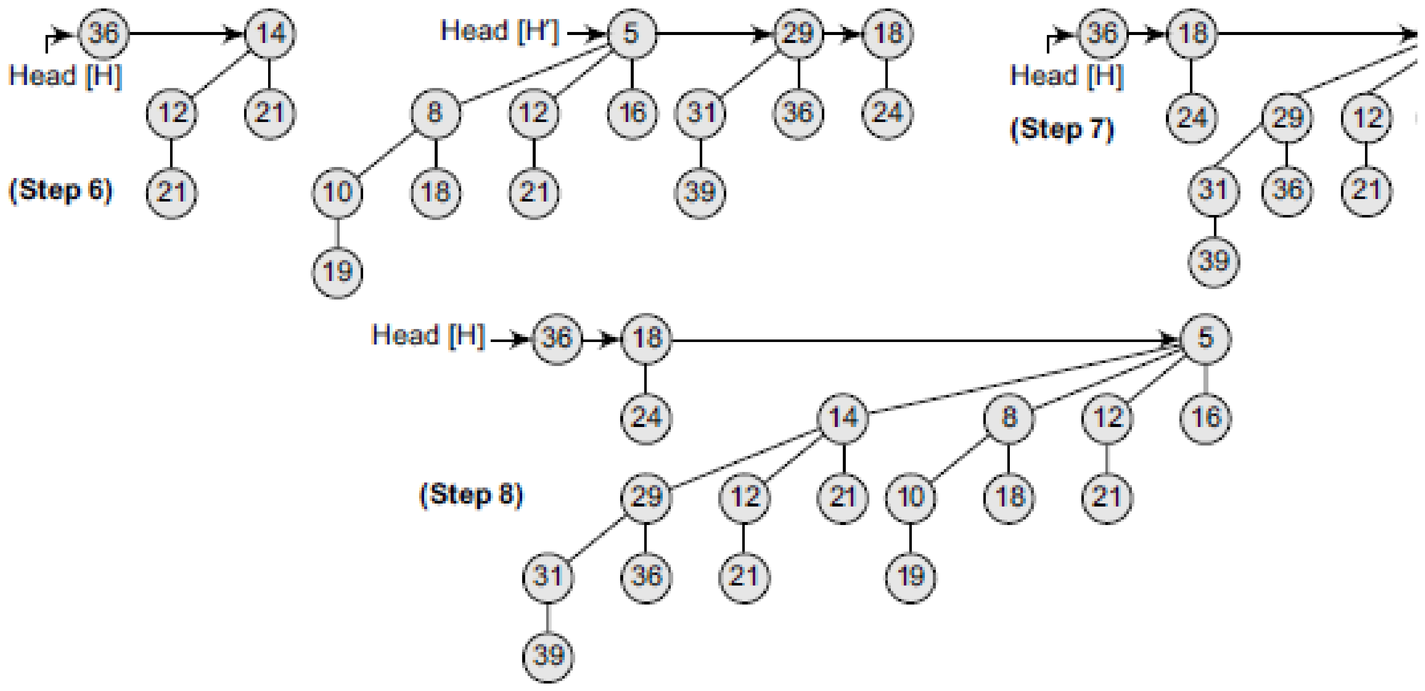


Figure 12.25 (Contd) Binomial heap