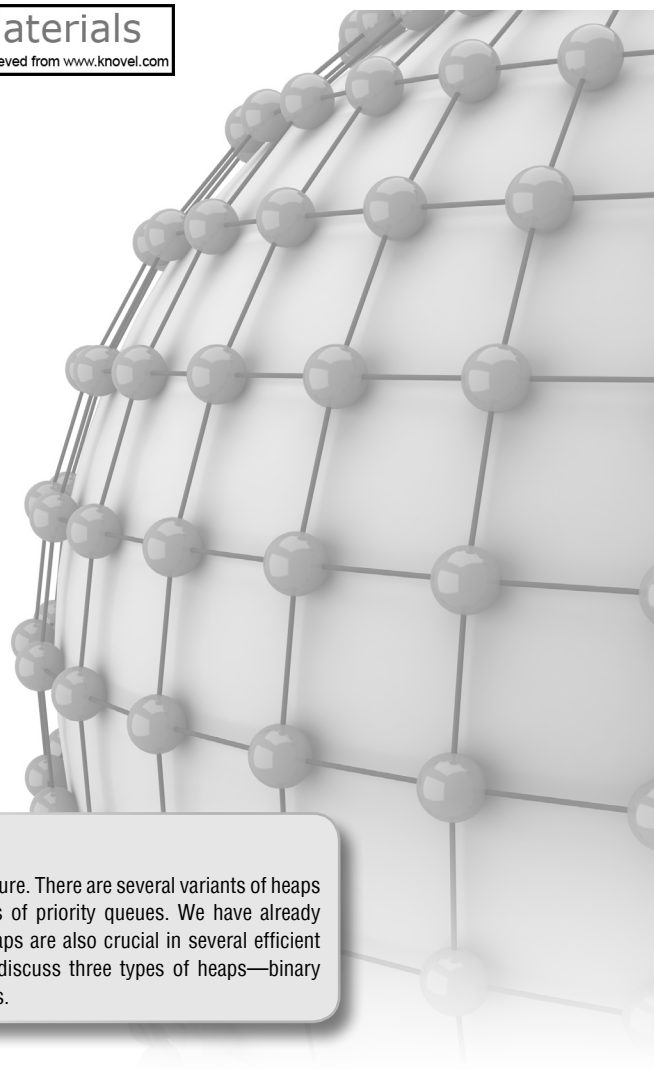
CHAPTER 12

Heaps



LEARNING OBJECTIVE

A heap is a specialized tree-based data structure. There are several variants of heaps which are the prototypical implementations of priority queues. We have already discussed priority queues in Chapter 8. Heaps are also crucial in several efficient graph algorithms. In this chapter, we will discuss three types of heaps—binary heaps, binomial heaps, and Fibonacci heaps.

12.1 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*.

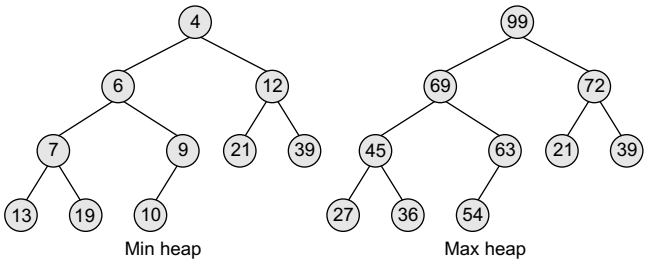


Figure 12.1 Binary heaps

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*.

Figure 12.1 shows a binary min heap and a binary max heap. 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 $\log_2 n$, 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. A binary tree is an efficient data structure, but a binary heap is more space efficient and simpler.

12.1.1 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.

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

Solution

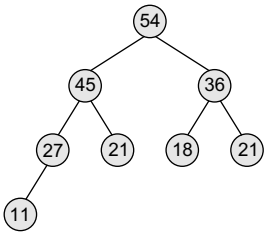


Figure 12.2 Binary heap

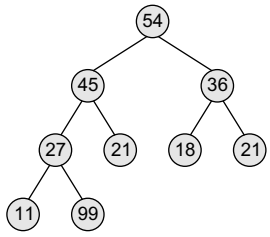


Figure 12.3 Binary heap after insertion of 99

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.

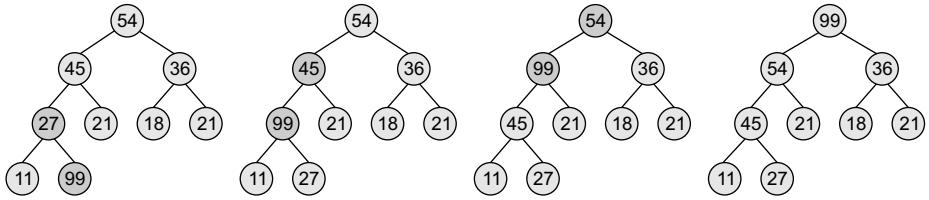


Figure 12.4 Heapify the binary heap

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

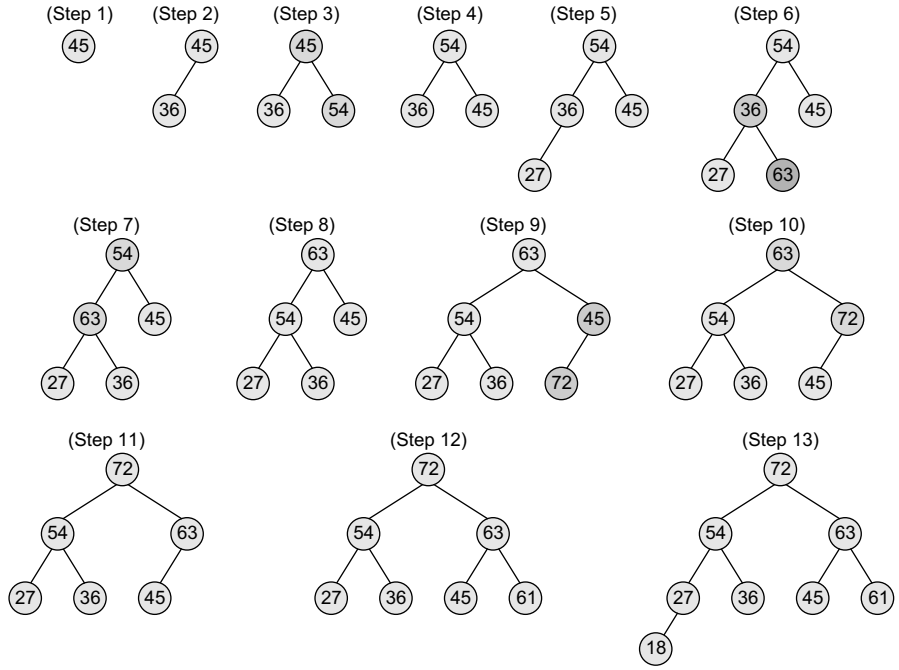


Figure 12.5

The memory representation of H can be given as shown in Fig. 12.6.

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

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

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.

12.1.2 Deleting an Element from a Binary Heap

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

Solution

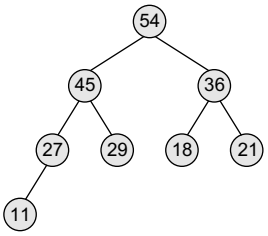


Figure 12.8 Binary heap

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).

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.

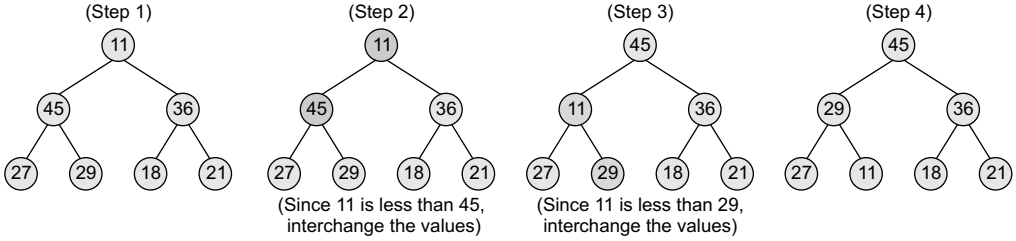


Figure 12.9 Binary heap

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

12.1.3 Applications of Binary Heaps

Binary heaps are mainly applied for

1. Sorting an array using *heapsort* algorithm. We will discuss heapsort algorithm in Chapter 14.
2. Implementing priority queues.

12.1.4 Binary Heap Implementation of Priority Queues

In Chapter 8, we learned about priority queues. We have also seen how priority queues can be implemented using linked lists. A priority queue is similar to a queue in which an item is

Figure 12.11 Priority queue visualization

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.

Conceptually, we can think of a priority queue as a bag of priorities shown in Fig. 12.11. In this bag you can insert any priority but you can take out one with the highest value.

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.

12.2 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 B_i has 2^i nodes.
- The height of a binomial tree B_i is `i`.

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

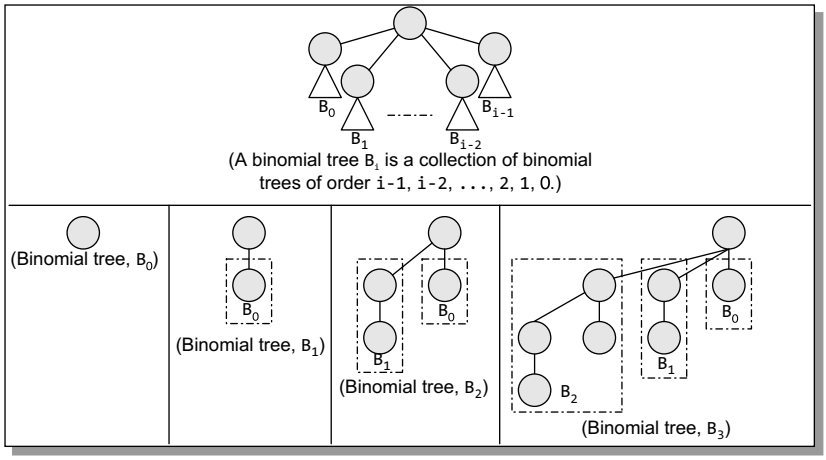


Figure 12.12 Binomial trees

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.

12.2.1 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

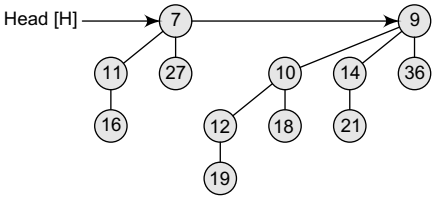


Figure 12.13 Binomial heap

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] = NIL`.

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.

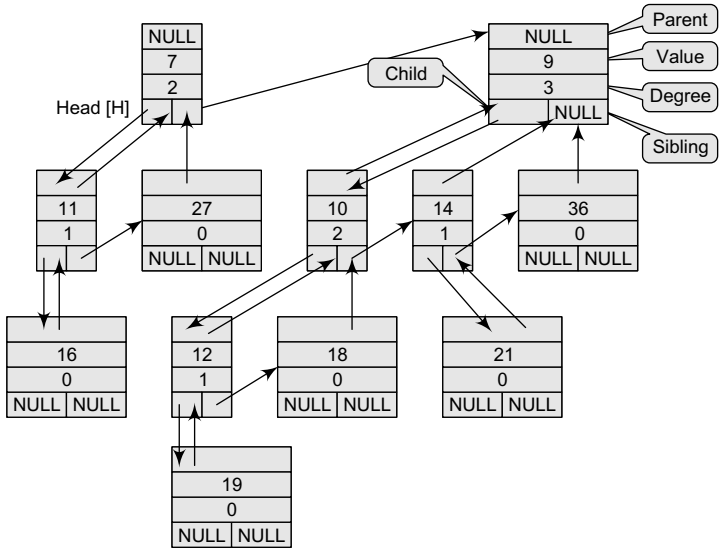


Figure 12.14 Linked representation of the binomial tree shown in Fig. 12.13

12.2.2 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.

```
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
```

Figure 12.15 Algorithm to find the node with minimum value

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)`.

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

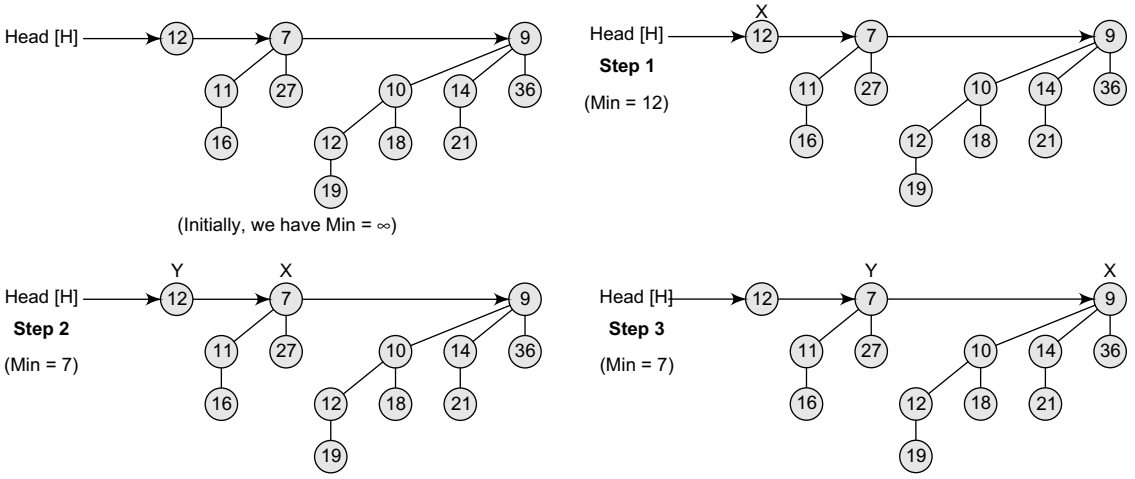


Figure 12.16 Binomial heap

Uniting Two Binomial Heaps

The procedure of uniting two binomial heaps is used as a subroutine by other operations. The `Union_Binomial-Heap()` procedure links together binomial trees whose roots have the same degree. The algorithm to link B_{i-1} tree rooted at node `Y` to the B_{i-1} tree rooted at node `Z`, making `Z` the parent of `Y`, is shown in Fig. 12.17.

```
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

The `Link_Binomial-Tree()` procedure makes `Y` the new head of the linked list of node `Z`'s children in `O(1)` time.

The algorithm to unite two binomial heaps H_1 and H_2 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

The algorithm destroys the original representations of heaps H_1 and H_2. Apart from `Link_Binomial-Tree()`, it uses another procedure `Merge_Binomial-Heap()` which is used to merge the root lists of H_1 and H_2 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 H_1 and H_2 into a single root list `H` in such a way that H_1 and H_2 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 H_1 and H_2, 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.

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]`.

In Step 7, we check for two conditions. First, if `degree[PTR]` ≠ `degree[NEXT]`, that is, when `PTR` is the root of a B_i tree and `NEXT` is the root of a B_j 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.

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 $(\log n_2 + \log n_1 + 2) \leq (2 \log n + 2) = O(\log n)$ 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 $O(\log n)$.

Example 12.5 Unite the binomial heaps given below.

Solution

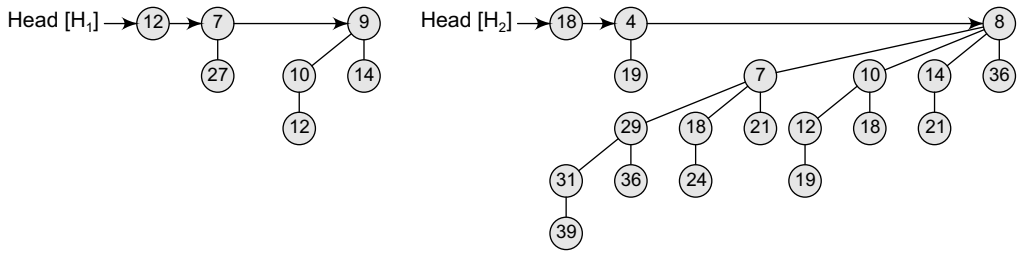


Figure 12.19(a)

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

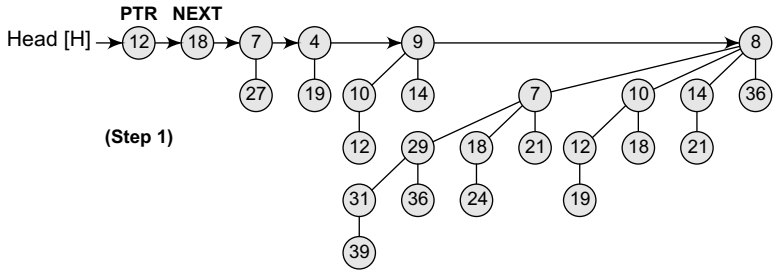


Figure 12.19(b)

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

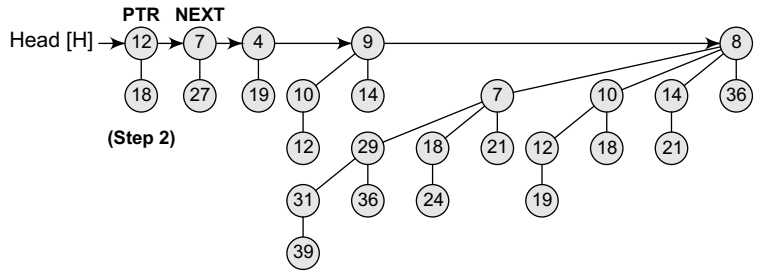


Figure 12.19(c)

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]`.

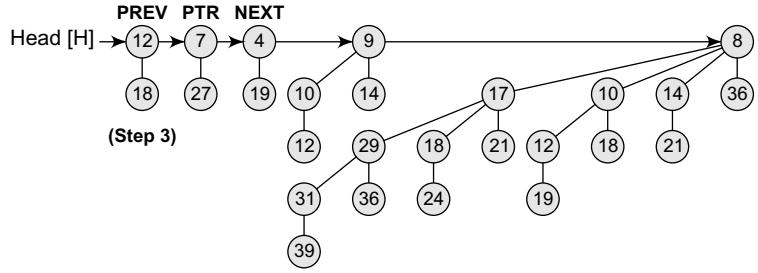


Figure 12.19(d)

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

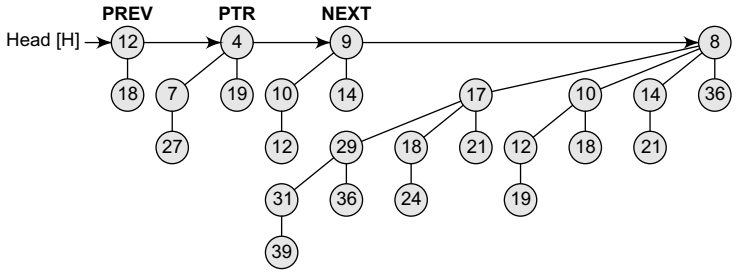


Figure 12.19(e)

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

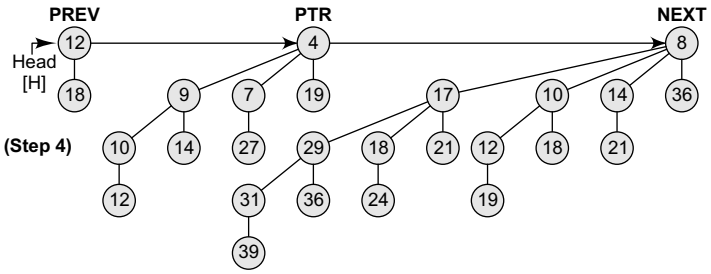


Figure 12.19(f) Binomial heap

```
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

```
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

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.

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.

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

Solution

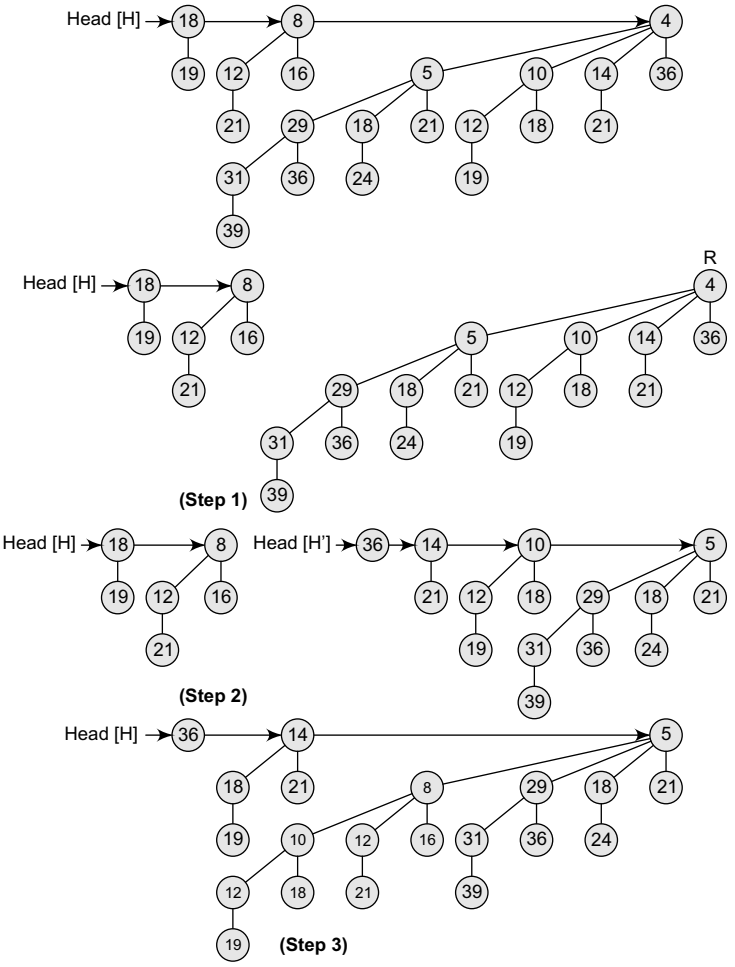


Figure 12.22 Binomial heap

```
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 binomial heap H

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.

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.

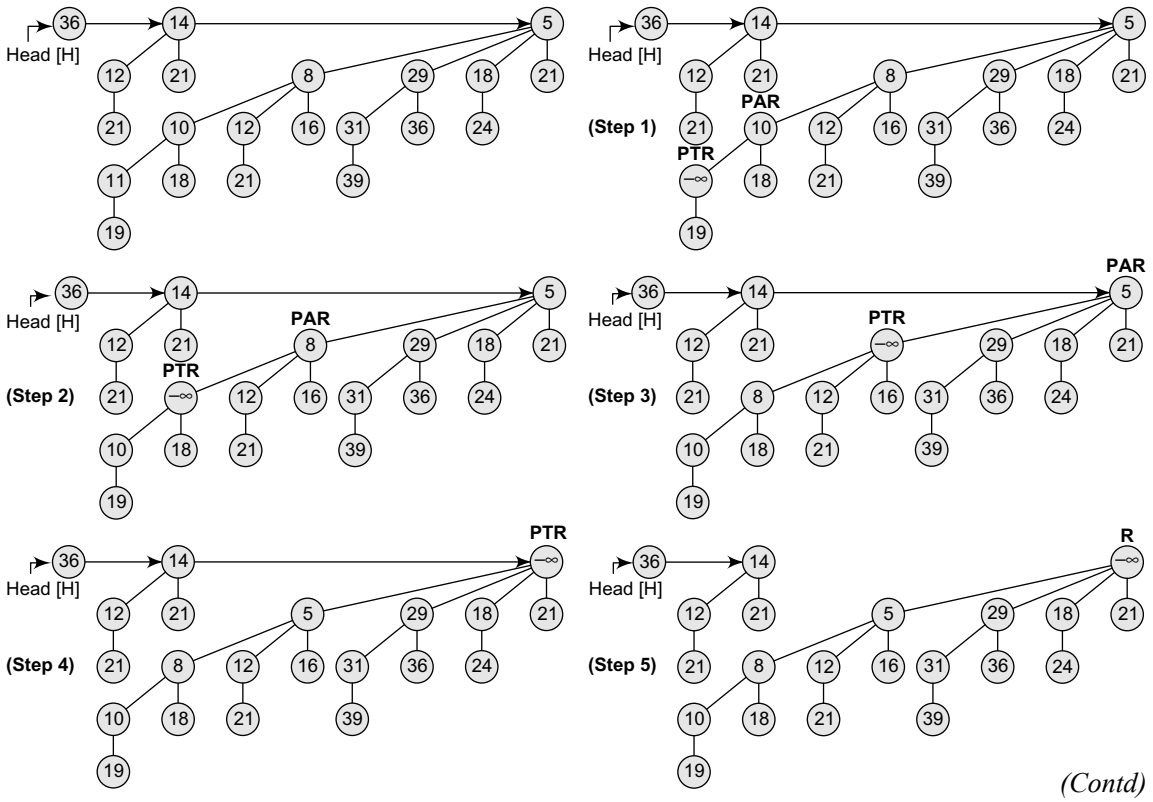
```
Binomial-Heap_Delete-Node(H, x)

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

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 upto a root and then this root is removed from the heap by making a call to the Min-Extract_Binomial-Heap procedure. The Binomial-Heap_Delete-Node procedure takes O(log n) time.

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

Solution



(Contd)

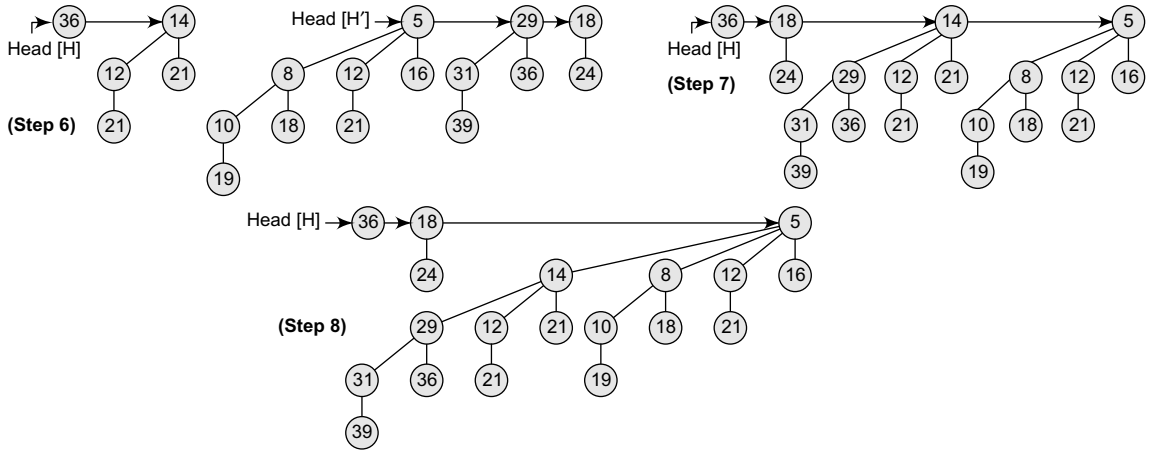


Figure 12.25 (Contd) Binomial heap

12.3 FIBONACCI HEAPS

In the last section, we have seen that binomial heaps support operations such as insert, extract-minimum, decrease-value, delete, and union in `O(log n)` worst-case time. In this section, we will discuss Fibonacci heaps which support the same operations but have the advantage that operations that do not involve deleting an element run in `O(1)` amortized time. So, theoretically, Fibonacci heaps are especially desirable when the number of extract-minimum and delete operations is small relative to the number of other operations performed. This situation arises in many applications, where algorithms for graph problems may call the decrease-value once per edge. However, the programming complexity of Fibonacci heaps makes them less desirable to use.

A Fibonacci heap is a collection of trees. It is loosely based on binomial heaps. If neither the decrease-value nor the delete operation is performed, each tree in the heap is like a binomial tree. Fibonacci heaps differ from binomial heaps as they have a more relaxed structure, allowing improved asymptotic time bounds.

12.3.1 Structure of Fibonacci Heaps

Although a Fibonacci heap is a collection of heap-ordered trees, the trees in a Fibonacci heap are not constrained to be binomial trees. That is, while the trees in a binomial heap are ordered, those within Fibonacci heaps are rooted but unordered.

Look at the Fibonacci heap given in Fig. 12.26. The figure shows that each node in the Fibonacci heap contains the following pointers:

- a pointer to its parent, and
- a pointer to any one of its children.

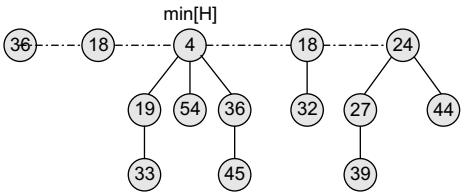


Figure 12.26 Fibonacci heap

Note that the children of each node are linked together in a circular doubly linked list which is known as the child list of that node. Each child `x` in a child list contains pointers to its left and right siblings. If node `x` is the only child of its parent, then `left[x] = right[x] = x` (refer Fig. 12.25).

Circular doubly linked lists provide an added advantage, as they allow a node to be removed in `O(1)` time. Also, given two circular doubly linked lists, the lists can be concatenated to form one list in `O(1)` time.

Apart from this information, every node will store two other fields. First, the number of children in the child list of node `x` is stored in `degree[x]`. Second, a boolean value `mark[x]` indicates whether node `x` has lost a child since the last time `x` was made the child of another node. Of course, the newly created nodes are unmarked. Also, when the node `x` is made the child of another node, it becomes unmarked.

Fibonacci heap `H` is generally accessed by a pointer called `min[H]` which points to the root that has a minimum value. If the Fibonacci heap `H` is empty, then `min[H] = NULL`.

As seen in Fig. 12.27, 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. Also note that the order of the trees within a root list is arbitrary.

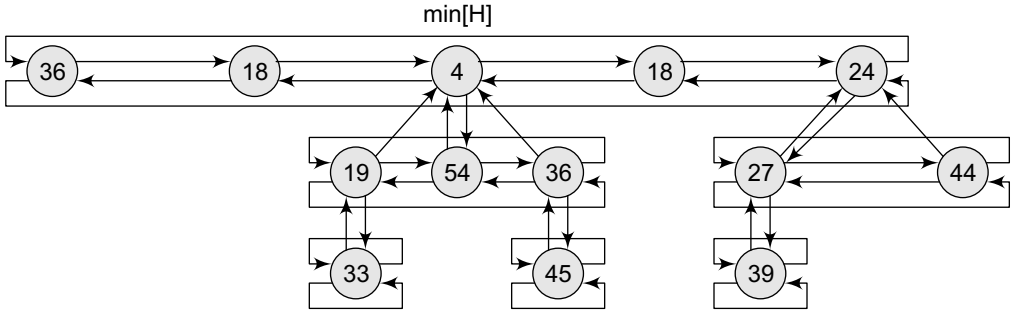


Figure 12.27 Linked representation of the Fibonacci heap shown in Fig. 12.24

In a Fibonacci heap `H`, the number of nodes in `H` is stored in `n[H]` and the degree of nodes is stored in `D(n)`.

12.3.2 Operations on Fibonacci Heaps

In this section, we will discuss the operations that can be implemented on Fibonacci heaps. If we perform operations such as create-heap, insert, find extract-minimum, and union, then each Fibonacci heap is simply a collection of unordered binomial trees. An *unordered binomial tree* U_0 consists of a single node, and an unordered binomial tree U_i consists of two unordered binomial trees U_{i-1} for which the root of one is made into a child of the root of another. All the properties of a binomial tree also hold for unordered binomial trees but for an unordered binomial tree U_i, the root has degree `i`, which is greater than that of any other node. The children of the root are roots of sub-trees $U_0, U_1, \ldots, U_{i-1}$ in some order. Thus, if an `n`-node Fibonacci heap is a collection of unordered binomial trees, then `D(n) = log n`. The underlying principle of operations on Fibonacci heaps is to delay the work as long as possible.

Creating a New Fibonacci Heap

To create an empty Fibonacci heap, the `Create_Fib-Heap` procedure allocates and returns the Fibonacci heap object `H`, where `n[H] = 0` and `min[H] = NULL`. The amortized cost of `Create_Fib-Heap` is equal to `O(1)`.

Inserting a New Node

The algorithm to insert a new node in an already existing Fibonacci heap is shown in Fig. 12.28.

In Steps 1 and 2, we first initialize the structural fields of node `x`, making it its own circular doubly linked list. Step 3 adds `x` to the root list of `H` in `O(1)` actual time. Now, `x` becomes an unordered binomial tree in the Fibonacci heap. In Step 4, the pointer to the minimum node of Fibonacci heap `H` is updated. Finally, we increment the number of nodes in `H` to reflect the addition of the new node.

Note that unlike the insert operation in the case of a binomial heap, when we insert a node in a Fibonacci heap, no attempt is made to consolidate the trees within the Fibonacci heap. So, even if k consecutive insert operations are performed, then k single-node trees are added to the root list.

```
Insert_Fib-Heap(H, x)

Step 1: [INITIALIZATION] SET Degree[x] = 0, Parent[x] = NULL,
        Child[x] = NULL, mark[x] = False
Step 2: SET Left[x] = x and Right[x] = x
Step 3: Concatenate the root list containing x with the
        root list of H
Step 4: IF min[H] = NULL OR Val[x] < Val[min[H]], then
            SET min[H] = x
        [END OF IF]
Step 5: SET n[H] = n[H]+ 1
Step 6: END
```

Figure 12.28 Algorithm to insert a new node in a Fibonacci heap

Example 12.8 Insert node 16 in the Fibonacci heap given below.

Solution

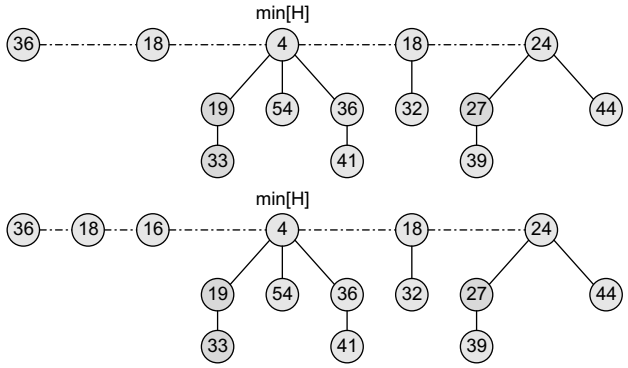


Figure 12.29 Fibonacci heap

Finding the Node with Minimum Key

Fibonacci heaps maintain a pointer min[H] that points to the root having the minimum value. Therefore, finding the minimum node is a straightforward task that can be performed in just O(1) time.

```
Union_Fib-Heap(H1, H2)

Step 1: H = Create_Fib-Heap()
Step 2: SET min[H] = min[H1]
Step 3: Concatenate root list of H2 with that of H
Step 4: IF (min[H1] = NULL) OR (min[H2] != NULL
         and min[H2] < min[H1]), then
                    SET min[H] = min[H2]
        [END OF IF]
Step 5: SET n[H] = n[H1] + n[H2]
Step 6: Free H1 and H2
Step 7: RETURN H
```

Figure 12.30 Algorithm to unite two Fibonacci heaps

Uniting Two Fibonacci Heaps

The algorithm given in Fig. 12.30 unites two Fibonacci heaps H_1 and H_2.

In the algorithm, we first concatenate the root lists of H_1 and H_2 into a new root list H. Then, the minimum node of H is set and the total number of nodes in H is updated. Finally, the memory occupied by H_1 and H_2 is freed and the resultant heap H is returned.

Extracting the Node with Minimum Key

The process of extracting the node with minimum value from a Fibonacci heap is the most complicated operation of all the operations that we have discussed so far. Till now, we had been delaying the work of consolidating the trees, but in this operation, we will finally implement the consolidation process. The algorithm to extract the node with minimum value is given in Fig. 12.31.

In the `Extract-Min_Fib-Heap` algorithm, we first make a root out of each of the minimum node's children and then remove the minimum node from the root list of `H`. Finally, the root list of the resultant Fibonacci heap `H` is consolidated by linking the roots of equal degree until at most one root remains of each degree.

Note that in Step 1, we save a pointer `x` to the minimum node; this pointer is returned at the end. However, if `x = NULL`, then the heap is already empty. Otherwise, the node `x` is deleted from `H` by making all its children the roots of `H` and then removing `x` from the root list (as done in Step 2). If `x = right[x]`, then `x` is the only node on the root list, so now `H` is empty. However, if `x != Right[x]`, then we set the pointer `min[H]` to the node whose address is stored in the right field of `x`.

```
Extract-Min_Fib-Heap(H)

Step 1: SET x = min[H]
Step 2: IF x != NULL, then
            For each child PTR of x
                  Add PTR to the root list of H and
                  Parent[PTR] = NULL
            Remove x from the root list of H
        [END OF IF]
Step 3: IF x = Right[x], then
            SET min[H] = NULL
        ELSE
            SET min[H] = Right[x]
            Consolidate(H)
        [END OF IF]
Step 4: SET n[H] = n[H] - 1
Step 5: RETURN x
```

Figure 12.31 Algorithm to extract the node with minimum key

Consolidating a Heap

A Fibonacci heap is consolidated to reduce the number of trees in the heap. While consolidating the root list of `H`, the following steps are repeatedly executed until every root in the root list has a distinct *degree* value.

- Find two roots `x` and `y` in the root list that has the same degree and where `Val[x]` ≤ `Val[y]`.
- Link `y` to `x`. That is, remove `y` from the root list of `H` and make it a child of `x`. This operation is actually done in the `Link_Fib-Heap` procedure. Finally, `degree[x]` is incremented and the mark on `y`, if any, is cleared.

In the consolidate algorithm shown in Fig. 12.32, we have used an auxiliary array `A[0… D(n[H])]`, such that if `A[i] = x`, then `x` is currently a node in the root list of `H` and `degree[x]= i`.

In Step 1, we set every entry in the array `A` to `NULL`. When Step 1 is over, we get a tree that is rooted at some node `x`. Initially, the array entry `A[degree[x]]` is set to point to `x`. In the `for` loop, each root node in `H` is examined. In each iteration of the `while` loop, `A[d]` points to some root `TEMP` because `d = degree[PTR] = degree[TEMP]`, so these two

```
Consolidate(H)

Step 1: Repeat for i=0 to D(n[H]), SET A[i] = NULL
Step 2: Repeat Steps 3 to 12 for each node x in the
        root list of H
Step 3:     SET PTR = x
Step 4:     SET deg = Degree[PTR]
Step 5:     Repeat Steps 6 to 10 while A[deg] != NULL
Step 6:           SET TEMP = A[deg]
Step 7:           IF Val[PTR] > Val[TEMP], then
Step 8:                 EXCHANGE PTR and TEMP
Step 9:           Link_Fib-Heap(H, TEMP, PTR)
Step 10:          SET A[deg] = NULL
Step 11:   SET deg = deg + 1
Step 12:   SET A[deg] = PTR
Step 13: SET min[H] = NULL
Step 14: Repeat for i = 0 to D(n(H])
Step 15:     IF A[i] != NULL, then
Step 16:          Add A[i] to the root list of H
Step 17:          IF min[H] = NULL OR Val[A[i]] <
                  Val[min[H]], then
Step 18:     SET min[H] = A[i]
Step 19: END
```

Figure 12.32 Algorithm to consolidate a Fibonacci heap

```
Link_Fib-Heap (H, x, y)

Step 1: Remove node y from the root list of H
Step 2: Make x the parent of y
Step 3: Increment the degree of x
Step 4: SET mark[y] = FALSE
Step 5: END
```

Figure 12.33 Algorithm to link two Fibonacci heaps

nodes must be linked with each other. Of course, the node with the smaller key becomes the parent of the other as a result of the link operation and so if need arises, we exchange the pointers to `PTR` and `TEMP`.

Next, we link `TEMP` to `PTR` using the `Link_Fib-Heap` procedure. The `Link_Fib-Heap` procedure (Fig. 12.33) increments the degree of `x` but leaves the degree of `y` unchanged. Since node `y` is no longer a root, the pointer to it in array `A` is removed in Step 10. Note that the value of degree of `x` is incremented in the `Link_Fib-Heap` procedure, so Step 13 restores the value of `d = degree[x]`. The `while` loop is repeated until `A[d] = NULL`, that is until no other root with the same degree as `x` exists in the root list of `H`.

Example 12.9 Remove the minimum node from the Fibonacci heap given below.

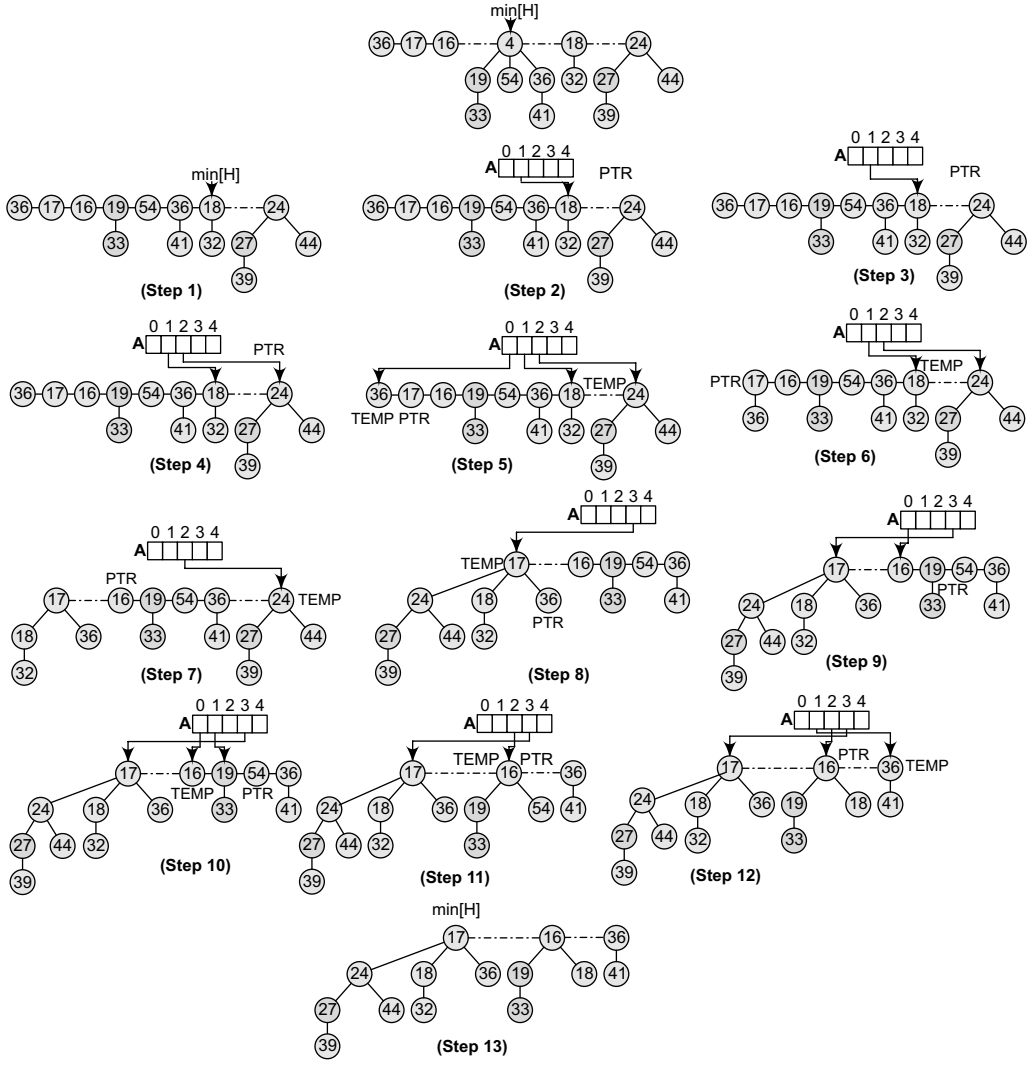


Figure 12.34 Fibonacci heap

```
Decrease-Val_Fib-Heap (H, PTR, v)

Step 1: IF v > Val[PTR]
            PRINT "ERROR"
        [END OF IF]
Step 2: SET Val[PTR] = v
Step 3: SET PAR = Parent[PTR]
Step 4: IF PAR != NULL and Val[PTR] < Val[PAR]
            Cut (H, PTR, PAR)
            Cascading-Cut(H, PAR)
        [END OF IF]
Step 5: IF Val[PTR] < Val[min[H]]
            SET min[H] = PTR
        [END OF IF]
Step 6: END
```

Figure 12.35 Algorithm to decrease the value of a node

Decreasing the Value of a Node

The algorithm to decrease the value of a node in O(1) amortized time is given in Fig. 12.35.

In the Decrease-Val_Fib-Heap (Fig. 12.35), we first ensure that the new value is not greater than the current value of the node and then assign the new value to PTR. If either the PTR points to a root node or if Val[PTR] >= Val[PAR], where PAR is PTR's parent, then no structural changes need to be done. This condition is checked in Step 4.

However, if the IF condition in Step 4 evaluates to a false value, then the heap order has been violated and a series of changes may occur. First, we call the Cut procedure to disconnect (or cut) any link between PTR and its PAR, thereby making PTR a root.

If PTR is a node that has undergone the following history, then the importance of the mark field can be understood as follows:

- Case 1: PTR was a root node.
- Case 2: Then PTR was linked to another node.
- Case 3: The two children of PTR were removed by the Cut procedure.

Note that when PTR will lose its second child, it will be cut from its parent to form a new root. mark[PTR] is set to TRUE when cases 1 and 2 occur and PTR has lost one of its child by the Cut operation. The Cut procedure, therefore, clears mark[PTR] in Step 4 of the Cut procedure.

However, if PTR is the second child cut from its parent PAR (since the time that PAR was linked to another node), then a Cascading-Cut operation is performed on PAR. If PAR is a root, then the IF condition in Step 2 of Cascading-Cut causes the procedure to just return. If PAR is unmarked, then it is marked as it indicates that its first child has just been cut, and the procedure returns. Otherwise, if PAR is marked, then it means that PAR has now lost its second child. Therefore, PTR is cut and Cascading-Cut is recursively called on PAR's parent. The Cascading-Cut procedure is called recursively up the tree until either a root or an unmarked node is found.

```
Cut(H, PTR, PAR)

Step 1: Remove PTR from the child list of PAR
Step 2: SET Degree[PAR] = Degree[PAR] - 1
Step 3: Add PTR to the root list of H
Step 4: SET Parent[PTR] = NULL
Step 5: SET Mark[PTR] = FALSE
Step 6: END
```

Figure 12.36 Algorithm to perform cut procedure

```
Cascading-Cut (H, PTR)

Step 1: SET PAR = Parent[PTR]
Step 2: IF PAR != NULL
                IF mark[PTR] = FALSE
                        SET mark[PTR] = TRUE
                ELSE
                        Cut (H, PTR, PAR)
                        Cascading-Cut(H, PAR)
        [END OF IF]
Step 3: END
```

Figure 12.37 Algorithm to perform cascade

Once we are done with the Cut (Fig. 12.36) and the Cascading-Cut (Fig. 12.37) operations, Step 5 of the Decrease-Val_Fib-Heap finishes up by updating min[H].

Note that the amortized cost of Decrease-Val_Fib-Heap is O(1). The actual cost of Decrease-Val_Fib-Heap is O(1) time plus the time required to perform the cascading cuts. If Cascading-Cut procedure is recursively called c times, then each call of

`Cascading-Cut` takes `O(1)` time exclusive of recursive calls. Therefore, the actual cost of `Decrease-Val_Fib-Heap` including all recursive calls is `O(c)`.

Example 12.10 Decrease the value of node 39 to 9 in the Fibonacci heap given below.

Solution

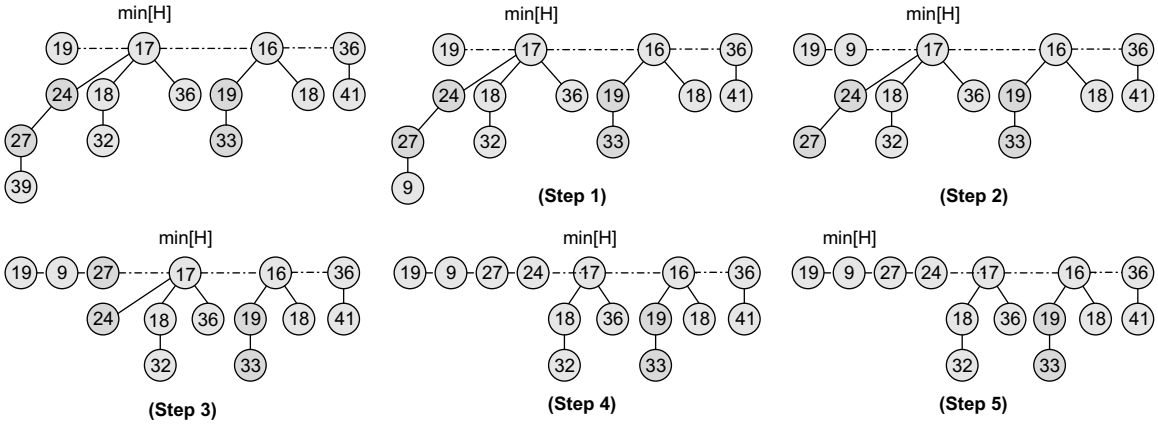


Figure 12.38 Fibonacci heap

Deleting a Node

A node from a Fibonacci heap can be very easily deleted in `O(D(n))` amortized time. The procedure to delete a node is given in Fig. 12.39.

```
Del_Fib-Heap (H, x)

Step 1: DECREASE-VAL_FIB-HEAP(H, x, -∞)
Step 2: EXTRACT-MIN_FIB-HEAP(H)
Step 3: END
```

Figure 12.39 Algorithm to delete a node from a Fibonacci heap

`Del_Fib-Heap` assigns a minimum value to `x`. The node `x` is then removed from the Fibonacci heap by making a call to the `Extract-Min_Fib-Heap` procedure. The amortized time of the delete procedure is the sum of the `O(1)` amortized time of `Decrease-Val_Fib-Heap` and the `O(D(n))` amortized time of `Extract-Min_Fib-Heap`.

12.4 COMPARISON OF BINARY, BINOMIAL, AND FIBONACCI HEAPS

Table 12.1 makes a comparison of the operations that are commonly performed on heaps.

Table 12.1 Comparison of binary, binomial, and Fibonacci heaps

Operation	Description	Time complexity in Big O Notation		
		Binary	Binomial	Fibonacci
Create Heap	Creates an empty heap	O(n)	O(n)	O(n)
Find Min	Finds the node with minimum value	O(1)	O(log n)	O(n)
Delete Min	Deletes the node with minimum value	O(log n)	O(log n)	O(log n)
Insert	Inserts a new node in the heap	O(log n)	O(log n)	O(1)
Decrease Value	Decreases the value of a node	O(log n)	O(log n)	O(1)
Union	Unites two heaps into one	O(n)	O(log n)	O(1)

12.5 APPLICATIONS OF HEAPS

Heaps are preferred for applications that include:

- ***Heap sort*** It is one of the best sorting methods that has no quadratic worst-case scenarios. Heap sort algorithm is discussed in Chapter 14.

- ***Selection algorithms*** These algorithms are used to find the minimum and maximum values in linear or sub-linear time.
- ***Graph algorithms*** Heaps can be used as internal traversal data structures. This guarantees that runtime is reduced by an order of polynomial. Heaps are therefore used for implementing Prim's minimal spanning tree algorithm and Dijkstra's shortest path problem.

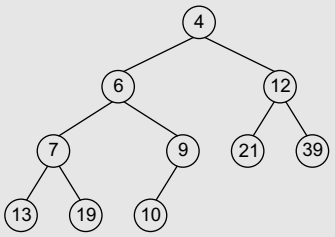
Points to Remember

- A binary heap is defined as a complete binary tree in which every node satisfies the heap property. There are two types of binary heaps: max heap and min heap.
- In a min heap, elements at every node will either be less than or equal to the element at its left and right child. Similarly, in a max heap, elements at every node will either be greater than or equal to element at its left and right child.
- 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`, `0`.
- A binomial tree B_i of height `i` has 2^i nodes.
- 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.
  - There can be one or zero binomial trees for each order including zero order.
- A Fibonacci heap is a collection of trees. Fibonacci heaps differ from binomial heaps, as they have a more relaxed structure, allowing improved asymptotic time bounds.

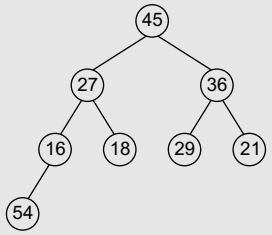
Exercises

Review Questions

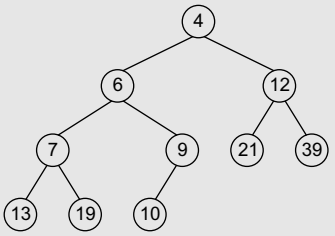
1. Define a binary heap.
2. Differentiate between a min-heap and a max-heap.
3. Compare binary trees with binary heaps.
4. Explain the steps involved in inserting a new value in a binary heap with the help of a suitable example.
5. Explain the steps involved in deleting a value from a binary heap with the help of a suitable example.
6. Discuss the applications of binary heaps.
7. Form a binary max-heap and a min-heap from the following sequence of data:
   50, 40, 35, 25, 20, 27, 33.
8. Heaps are excellent data structures to implement priority queues. Justify this statement.
9. Define a binomial heap. Draw its structure.
10. Differentiate among binary, binomial, and Fibonacci heaps.
11. Explain the operations performed on a Fibonacci heap.
12. Why are Fibonacci heaps preferred over binary and binomial heaps?
13. Analyse the complexity of the algorithm to unite two binomial heaps.
14. The running time of the algorithm to find the minimum key in a binomial heap is `O(log n)`. Comment.
15. Discuss the process of inserting a new node in a binomial heap. Explain with the help of an example.
16. The algorithm `Min-Extract_Binomial-Heap()` runs in `O(log n)` time where n is the number of nodes in `H`. Justify this statement.
17. Explain how an existing node is deleted from a binomial heap with the help of a relevant example.
18. Explain the process of inserting a new node in a Fibonacci heap.
19. Write down the algorithm to unite two Fibonacci heaps.
20. What is the procedure to extract the node with the minimum value from a Fibonacci heap? Give the algorithm and analyse its complexity.
21. Consider the figure given below and state whether it is a heap or not.



22. Reheap the following structure to make it a heap.



23. Show the array implementation of the following heap.



24. Given the following array structure, draw the heap.

45	27	36	18	16	21	23	10

Also, find out

(a) the parent of nodes 10, 21, and 23, and

(b) index of left and right child of node 23.

25. Which of the following sequences represents a binary heap?

(a) 40, 33, 35, 22, 12, 16, 5, 7

(b) 44, 37, 20, 22, 16, 32, 12

(c) 15, 15, 15, 15, 15, 15

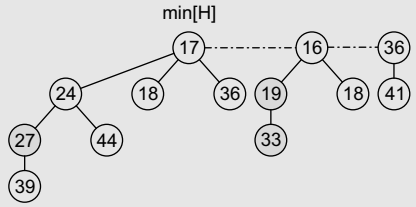
26. A heap sequence is given as: 52, 32, 42, 22, 12, 27, 37, 12, 7. Which element will be deleted when the deletion algorithm is called thrice?

27. Show the resulting heap when values 35, 24, and 10 are added to the heap of the above question.

28. Draw a heap that is also a binary search tree.

29. Analyse the complexity of heapify algorithm.

30. Consider the Fibonacci heap given below and then decrease the value of node 33 to 9. Insert a new node with value 5 and finally delete node 19 from it.



Multiple-choice Questions

1. The height of a binary heap with `n` nodes is equal to

(a) `O(n)` (b) `O(log n)`

(c) `O(n log n)` (d) $O(n^2)$

2. An element at position `i` in an array has its left child stored at position

(a) `2i` (b) `2i + 1`

(c) `i/2` (d) `i/2 + 1`

3. In the worst case, how much time does it take to build a binary heap of `n` elements?

(a) `O(n)` (b) `O(log n)`

(c) `O(n log n)` (d) $O(n^2)$

4. The height of a binomial tree B_i is

(a) `2i` (b) `2i + 1`

(c) `i/2` (d) `i`

5. How many nodes does a binomial tree of order 0 have?

(a) 0 (b) 1

(c) 2 (d) 3

6. The running time of `Link_Binomial-Tree()` procedure is

(a) `O(n)` (b) `O(log n)`

(c) `O(n log n)` (d) `O(1)`

7. In a Fibonacci heap, how much time does it take to find the minimum node?

(a) `O(n)` (b) `O(log n)`

(c) `O(n log n)` (d) `O(1)`

True or False

1. A binary heap is a complete binary tree.
2. In a min heap, the root node has the highest key value in the heap.
3. An element at position `i` has its parent stored at position `i/2`.
4. All levels of a binary heap except the last level are completely filled.
5. In a min-heap, elements at every node will be greater than its left and right child.
6. A binomial tree B_i has `2i` nodes.
7. Binomial heaps are ordered.
8. Fibonacci heaps are rooted and ordered.
9. The running time of `Min_Binomial-Heap()` procedure is `O(log n)`.
10. If there are `m` roots in the root lists of H_1 and H_2, then `Merge_Binomial-Heap()` runs in `O(m log m)` time.
11. Fibonacci heaps are preferred over binomial heaps.

Fill in the Blanks

1. An element at position `i` in the array has its right child stored at position _____.
2. Heaps are used to implement _____.
3. Heaps are also known as _____.
4. In _____, elements at every node will either be less than or equal to the element at its left and right child.
5. An element is always deleted from the _____.
6. The height of a binomial tree B_i is _____.
7. A binomial heap is defined as _____.
8. A binomial tree B_i has _____ nodes.
9. A binomial heap is created in _____ time.
10. A Fibonacci heap is a _____.
11. In a Fibonacci heap, `mark[x]` indicates _____.