Priority Queues (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. Heaps are also crucial in several efficient graph algorithms. In this chapter, we will discuss two types of heaps—binary heaps and binomial heaps (or queues).

4.1 INTRODUCTION — PRIORITY QUEUES

A *priority queue* is a data structure in which each element is assigned a priority. The priority of the element will be used to determine the order in which the elements will be processed. The general rules of processing the elements of a priority queue are

- An element with higher priority is processed before an element with a lower priority.
- Two elements with the same priority are processed on a first-come-first-served (FCFS) basis.

A priority queue can be thought of as a modified queue in which when an element has to be removed from the queue, the one with the highest-priority is retrieved first. The priority of the element can be set based on various factors. Priority queues are widely used in operating systems to execute the highest priority process first. The priority of the process may be set based on the CPU time it requires to get executed completely. For example, if there are three processes, where the first process needs 5 ns to complete, the second process needs 4 ns, and the third process needs 7 ns, then the second process will have the highest priority and will thus be the first to be executed. However, CPU time is not the only factor that determines the priority, rather it is just one among several factors. Another factor is the importance of one process over another. In case we have to run two processes at the same time, where one process is concerned with online order booking and the second with printing of stock details, then obviously the online booking is more important and must be executed first.

Implementation of a Priority Queue

There are two ways to implement a priority queue. We can either use a sorted list to store the elements so that when an element has to be taken out, the queue will not have to be searched for the element with the highest priority or we can use an unsorted list so that insertions are always done at the end of the list. Every time when an element has to be removed from the list, the element with the highest priority will be searched and removed. While a sorted list takes O(n) time to insert an element in the list, it takes only O(1) time to delete an element. On the contrary, an unsorted list will take O(1) time to insert an element and O(n) time to delete an element from the list.

Practically, both these techniques are inefficient and usually a blend of these two approaches is adopted that takes roughly O(log n) time or less.

Linked Representation of a Priority Queue

In the computer memory, a priority queue can be represented using arrays or linked lists. When a priority queue is implemented using a linked list, then every node of the list will have three parts: (a) the information or data part, (b) the priority number of the element, and (c) the address of the next element. If we are using a sorted linked list, then the element with the higher priority will precede the element with the lower priority.

Consider the priority queue shown in Fig. 4.1.

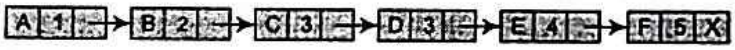


Figure 4.1 Priority queue

Lower priority number means higher priority. For example, if there are two elements A and B, where A has a priority number 1 and B has a priority number 5, then A will be processed before B as it has higher priority than B.

The priority queue in Fig. 4.1 is a sorted priority queue having six elements. From the queue, we cannot make out whether A was inserted before E or whether E joined the queue before A because the list is not sorted based on FCFS. Here, the element with a higher priority comes before the element with a lower priority. However, we can definitely say that C was inserted in the queue before D because when two elements have the same priority the elements are arranged and processed on FCFS principle.

Insertion When a new element has to be inserted in a priority queue, we have to traverse the entire list until we find a node that has a priority lower than that of the new element. The new node is inserted before the node with the lower priority. However, if there exists an element that has the same priority as the new element, the new element is inserted after that element. For example, consider the priority queue shown in Fig. 4.2.

Figure 4.2 Priority queue

If we have to insert a new element with data = F and priority number = 4, then the element will be inserted before D that has priority number 5, which is lower priority than that of the new element. So, the priority queue now becomes as shown in Fig. 4.3.

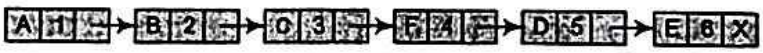


Figure 4.3 Priority queue after insertion of a new node

However, if we have a new element with `data = F` and `priority number = 2`, then the element will be inserted after `B`, as both these elements have the same priority but the insertions are done on FCFS basis as shown in Fig. 4.4.

Figure 4.4 Priority queue after insertion of a new node

Deletion Deletion is a very simple process in this case. The first node of the list will be deleted and the data of that node will be processed first.

Array Representation of a Priority Queue

When arrays are used to implement a priority queue, then a separate queue for each priority number is maintained. Each of these queues will be implemented using circular arrays or circular queues. Every individual queue will have its own FRONT and REAR pointers.

We use a two-dimensional array for this purpose where each queue will be allocated the same amount of space. Look at the two-dimensional representation of a priority queue given below. Given the FRONT and REAR values of each queue, the two-dimensional matrix can be formed as shown in Fig. 4.5.

FRONT[K] and REAR[K] contain the front and rear values of row K, where K is the priority number. Note that here we are assuming that the row and column indices start from 1, not 0. Obviously, while programming, we will not take such assumptions.

FRONT	REAR
3	3
1	2
4	5
4	4

	1	2	3	4	5
1			A		
2	B	C	D		
3				E	F
4	I			G	H

Figure 4.5 Priority queue matrix

Insertion To insert a new element with priority K in the priority queue, add the element at the rear end of row K, where K is the row number as well as the priority number of that element. For example, if we have to insert an element R with priority number 3, then the priority queue will be given as shown in Fig. 4.6.

FRONT	REAR
3	3
1	2
4	5
4	4

	1	2	3	4	5
1			A		
2	B	C	D		
3			R	E	F
4	I			G	H

Deletion To delete an elem

```
struct node *start=NULL;
struct node *insert(struct node *);
struct node *del(struct node *);
void display(struct node *);
int main()
{
int option;

do
{
cout<<"\n ********** MAIN MENU **********\n";
cout<<"\n 1. INSERT";
cout<<"\n 2. DELETE";
cout<<"\n 3. DISPLAY";
cout<<"\n 4. EXIT";
cout<<"\n ENter your option : ";
cin>>option;
switch(option)
{
case 1:
start=insert(start);
break;
case 2:
start = del(start);
break;
case 3:
display(start);
break;
}
}while(option!=4);
}
struct node *insert(struct node *start)
{
int val, pri;
struct node *ptr, *p;
ptr = new node;
cout<<"\n Enter the value and its priority : ";
cin>>val>>pri;
ptr->data = val;
ptr->priority= pri;
if(start == NULL || pri <start->priority)
{
ptr->next = start;
start = ptr;
}
else
{
p = start;
while(p->next != NULL && p->next->priority <= pri)

p = p->next;
ptr->next = p->next;
p->next = ptr;
}
return start;
}
struct node *del(struct node *start)
{
struct node *ptr;
if(start == NULL)
{
cout<<"\n UNDERFLOW";
```

```
exit;
}
else
{
ptr = start;
cout<<"\n Deleted item is : "<<ptr->data;
start = start->next;
delete ptr;
}
return start;
}
void display(struct node *start)
{
struct node *ptr;
ptr = start;
if(start == NULL)
cout<<"\nQUEUE IS EMPTY";
else
{
cout<<"\n PRIORITY QUEUE IS :" ;
while(ptr != NULL)
{
cout<<"\t"<<ptr->data<<"["<<ptr->priority<<"]";
ptr = ptr->next;
}
}
}
```

Output

```
*****MAIN MENU*****
1. INSERT
2. DELETE
3. DISPLAY
4. EXIT
Enter your option : 1
Enter the value and its priority : 5 2
Enter the value and its priority : 10 1
Enter your option : 3
PRIORITY QUEUE IS :
10[priority = 1] 5[priority = 2]
Enter your option : 4
```

4.2 BINARY HEAPS—MODEL AND SIMPLE IMPLEMENTATION

We have already seen in the previous section how priority queues can be implemented using linked lists. 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. Conceptually, we can think of a priority queue as a bag of priorities shown in Fig. 4.7. In this bag you can insert any priority but you can take out one with the highest value.

Figure 4.7 Priority queue visualization

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.

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*.

Figure 4.8 shows a binary min heap and a binary max heap.

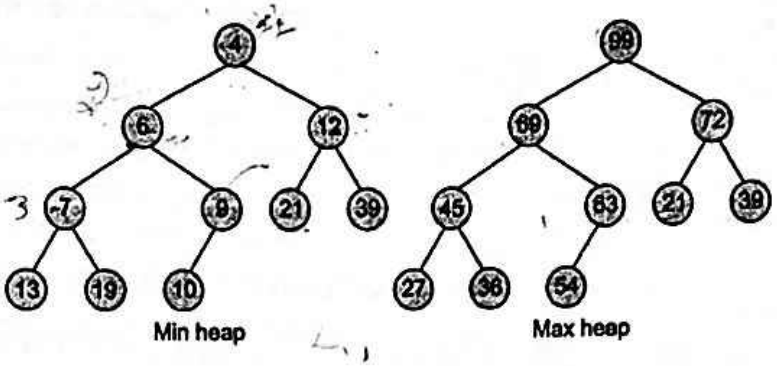


Figure 4.8 Binary heaps

4.2.1 Binary Heap – Structure Property

For an efficient implementation of binary heap, we need to guarantee logarithmic performance by keeping the tree balanced. A balanced binary tree has roughly the same number of nodes in the left and right sub-trees of the root. While building a binary heap, we balance the tree by creating a *complete binary tree.*

Note In a complete binary tree, each level (except the bottom level) has all of its nodes. The nodes in the bottom level are filled from left to right.

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.

4.2.2 Binary Heap – Order Property

The technique for inserting and deleting items from a heap always maintains the heap order property. The *heap order property* states that for every node x with parent p, the key in p is smaller than or equal to the key in x. Figure 4.9 illustrates a complete binary tree that satisfies the heap order property.

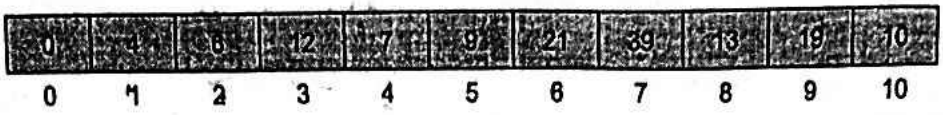


Figure 4.9 Binary tree that illustrates heap order property

4.3 BASIC HEAP OPERATIONS

4.3.1 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 4.1 Consider the max heap given in Fig. 4.10 and insert 99 in it.

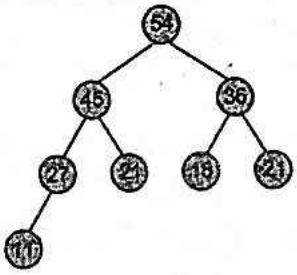


Figure 4.10 Binary heap

Solution

The first step is to insert the element in the heap such 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. 4.11.

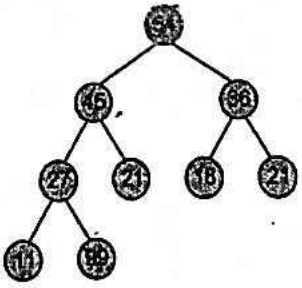


Figure 4.11 Binary heap after insertion of 99

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. 4.12.

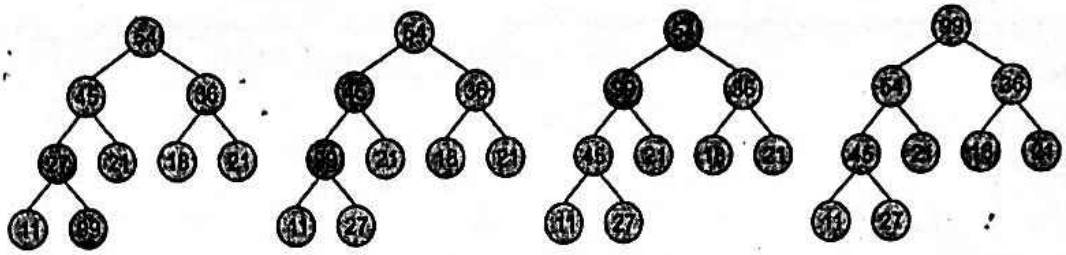


Figure 4.12 Heapify the binary heap

Example 4.2 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

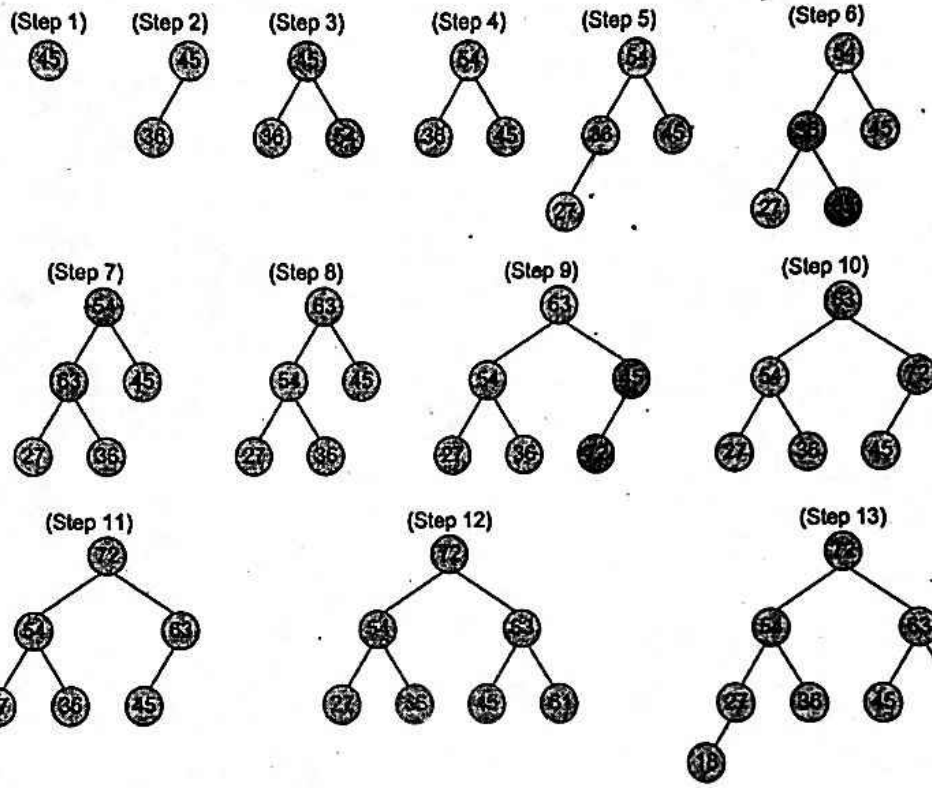


Figure 4.13

The memory representation of H can be given as shown in Fig. 4.14.

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 4.14 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. 4.15. 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.

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.

```
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 4.15 Algorithm to insert an element in a max heap

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.

4.3.2 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).

Example 4.3 Consider the max heap H shown in Fig. 4.16 and delete the root node's value.

Solution

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.

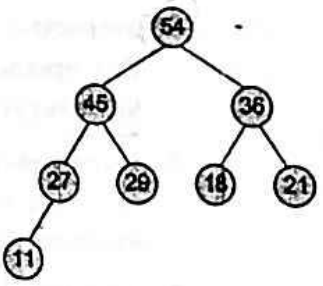


Figure 4.16 Binary heap

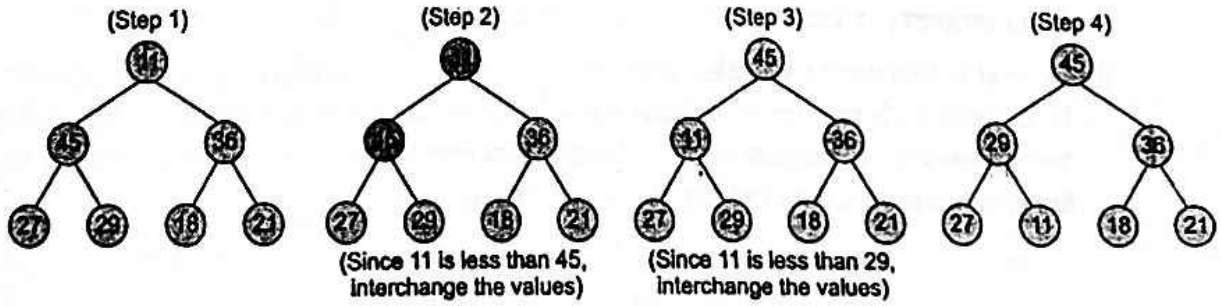


Figure 4.17 Binary heap

After discussing the concept behind deleting the root element from the heap, let us look at the algorithm given in Fig. 4.18. 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 4.18 Algorithm to delete the root element from a max heap

4.4 OTHER HEAP OPERATIONS

The following are some more operations that can be performed on the heaps.

1. `getMax()`: It returns the root element of max heap. The time complexity of this operation is O(1).
2. `extractMax()`: It removes the maximum element from max heap. The time complexity of this operation is O(log n) as this operation needs to maintain the heap property (by calling `heapify()`) after removing root.
3. `increaseKey()`: It increases the value of key. The time complexity of this operation is O(log n). If the increased key is less than that of the parent node, then the heap property is not violated. Otherwise, we need to traverse up the heap to fix the violated heap property.
4. `insert()`: Inserting a new key takes O(log n)time. The new value is added at the end of the tree. If the new key is less than the key of its parent, then the heap is fine. Otherwise, the heap property is violated and we need to traverse up to fix it.
5. `delete()`: Deleting a key also takes O(log n) time. To delete a key, we first replace the key to be deleted with maximum infinite value by using the `increaseKey()` function. After using the `increasekey()`, the largest value (which is maximum infinite value) becomes the root. Finally, the root is removed by calling the `extractMax()` function.

Programming Example

2. Write a program to illustrate binary heap operations on a max heap.

```
#include<iostream>
#include<climits>
using namespace std;

// Prototype of a utility function to swap two integers
void swap(int *x, int *y);

// A class for Min Heap
class MaxHeap
{
    int *harr; // pointer to array of elements in heap
    int capacity; // maximum possible size of min heap
    int heap_size; // Current number of elements in min heap
public:
    MaxHeap(int capacity);

    // to heapify a subtree with root at given index
    void MaxHeapify(int );

    int parent(int i) { return (i-1)/2; }

    // to get index of left child of node at index i
    int left(int i) { return (2*i + 1); }

    // to get index of right child of node at index i
    int right(int i) { return (2*i + 2); }

    // to extract the root which is the maximum element
    int extractMax();

    // Decreases key value of key at index i to new_val
    void increaseKey(int i, int new_val);

    // Returns the maximum key (key at root)
    int getMax() { return harr[0]; }

    // Deletes a key stored at index i
    void deleteKey(int i);

    // Inserts a new key 'k'
    void insertKey(int k);

    void display();
};
MaxHeap::MaxHeap(int n)
{
    heap_size = 0;
    capacity = n;
    harr = new int[n];
}

void MaxHeap::insertKey(int k)
{
if (heap_size == capacity)
{
```

```
        cout << "\n OVERFLOW";
        return;
    }

    // Insert the new key at the end
    heap_size++;
    int i = heap_size - 1;
    harr[i] = k;

    // Fix the max heap property
    while (i != 0 && harr[parent(i)] < harr[i])
    {
       swap(&harr[i], &harr[parent(i)]);
       i = parent(i);
    }
}

// Increase value of key at index 'i' to new_val.  It is assumed that
// new_val is greater than harr[i].
void MaxHeap::increaseKey(int i, int new_val)
{
    harr[i] = new_val;
    while (i != 0 && harr[parent(i)] < harr[i])
    {
       swap(&harr[i], &harr[parent(i)]);
       i = parent(i);
    }
}

int MaxHeap::extractMax()
{
    if (heap_size == 1)
    {
        heap_size--;
        return harr[0];
    }

    // Remove the maximum value from heap
    int root = harr[0];
    harr[0] = harr[heap_size-1];
    heap_size--;
    MaxHeapify(0);

    return root;
}


// This function deletes key at index i. It first reduced value to minus
// infinite, then calls extractMin()
void MaxHeap::deleteKey(int i)
{
    increaseKey(i, INT_MAX);
    extractMax();
}

void MaxHeap::display()
{
    int i;
    for(i=0;i<heap_size;i++)
```

```
    cout<<"\t"<<harr[i];
}
// A recursive method to heapify a subtree with root at given index
// This method assumes that the subtrees are already heapified
void MaxHeap::MaxHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int largest = i;
    if (l < heap_size && harr[l] > harr[i])
        largest = l;
    if (r < heap_size && harr[r] > harr[largest])
        largest = r;
    if (largest != i)
    {
        swap(&harr[i], &harr[largest]);
        MaxHeapify(largest);
    }
}

// A utility function to swap two elements
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

// Driver program to test above functions
int main()
{
    MaxHeap h(10);
    h.insertKey(27);
    h.insertKey(36);
    h.insertKey(18);
    h.insertKey(9);
    h.insertKey(54);
    h.insertKey(45);
    cout<<"\n After insertion heap is : ";
    h.display();

    h.deleteKey(1);
    cout<<"\n After deletion heap is : ";
    h.display();

    cout << "\n Extracting max value ...."<<h.extractMax() << " ";
    cout <<"\n Maximum value = "<< h.getMax() << " ";
    cout<<"\n Increasing key....";
    h.increaseKey(2, 100);
    cout<<"\n After increasing key, the heap is : ";
    h.display();
    return 0;
}
```

Output

```
After insertion heap is :  54    36    45    9    27    18
After deletion heap is :  54    27    45    9    18
Extracting max value ....54
Maximum value = 45
Increasing key....
After increasing key, the heap is :  100    27    45    9
```

Note To implement a min heap, the operations would be same as that given for max heap with minimal changes as given below.

- Rename `increasekey()` as `decreasekey()`.
- Rename `getmax()` as `getmin()`.
- In `insertkey()`, replace < with > as given here; `while (i != 0 && harr[parent(i)] > harr[i])`
- In `increaseKey()`, replace < with > as given here; `while (i != 0 && harr[parent(i)] > harr[i])`
- Rename `extractMax` as `extractMin`
- Rename `maxHeapify` as `minHeapify`
- In `deleteKey`, replace `increaseKey(i, INT_MAX);` with `decreaseKey(i, INT_MIN);`
- In `minHeapify()`, instead of largest use the variable smallest. Replace > with < in all if conditions. Call `minHeapify`.

4.5 APPLICATIONS OF PRIORITY QUEUES

We have seen that a priority queue is a special queue in which values come out in order by priority and not in FIFO order. A priority queue is typically implemented using heaps and has a wide range of applications. Some of these applications are given below.

- Dijkstra's Shortest Path Algorithm
- In Prim's algorithm to store keys of nodes and extract minimum key node at every step.
- In Huffman codes for data compression.
- In operating systems for process management and sharing limited computer resources with multiple processes. The priority queue is used to arrange all processes according to their priority so that the highest priority process is serviced before others.
- Efficiently sorting numbers (heap sort).
- Scheduling events. For example, to simulate traffic light as in when a light changes to red, the next change is scheduled to green after n seconds.

Priority queues are also applied to solve two common problems in computing—The Selection Problem and Event Simulation which are discussed in the section given below.

4.5.1 The Selection Problem

Priority queues are widely used to manage the events in a discrete event simulation. The events are added to the queue along with their simulation time. The simulation time determines their priority. While execution, the event at the top of the queue is pulled out and executed. This process repeats as long as events last in the priority queue.

Therefore, technically we can define this problem as,

Problem: Given a sequence of N elements and an integer k, k ∈ N, find the kth largest element.

Solution 1: Sort the elements in an array and return the element in the kth position.
Complexity: While simple sorting algorithm takes $O(N^2)$ time, advanced sorting algorithm takes $O(N \log N)$ time to execute.

Solution 2: Read k elements in an array and sort them. Let S_k be the smallest element. For each next element E do the following:

If $E > S_k$ remove S_k and insert E in the appropriate position in the array

Return S_k

In simple words, we read read k elements into an array and sort them. The smallest of these elements is present in the *k*th position. The remaining elements are processed one by one. As an

element arrives, it is compared with *k*th element in the array. If it is larger, then the *k*th element is removed, and the new element is placed in the correct place among the remaining k - 1 elements. When the algorithm ends, the element in the *k*th position is the answer.

Complexity: O(N * k). This is because,

- $c(k^2)$ time is used for the initial sorting of k elements
- $O((N-k)*k)$ time for inserting each next element, as only (N - k) elements are left (unsorted).

Therefore, on combining both steps the complexity can be given as,

$O(k^2) + O((N-k)*k)$

$= O(k^2 + N*k - k^2)$

$= O(N * k)$

In the worst case if k = N/2, then the runtime complexity is $O(N^2)$.

Note If k = N/2, then both Solutions 1 and 2 takes $O(N^2)$ time to execute.

Solution 3: We can use a max heap. Therefore, to find the *k*th smallest element, we read the n elements into an array and apply the `build_heap` algorithm to this array. Finally, k `Delete_Min` operations are performed so that the last element extracted from the heap gives the desired answer. In this case, you need to follow the steps given below.

Step 1: Read N elements in an array.

Step 2: Build a heap of these N elements. This would require O(N) time.

Step 3: Perform k `DeleteMax` operations which will take another O(k log N) time.

Therefore, total time required to implement this solution is O(N) + O(k log N)

When k = N/2, the complexity would be O(N log N).

Solution 4: We can use a min heap having the smallest element at the top. The *k*th largest element among k elements is the smallest element in the heap and thus at the topmost position. This can be done in O(k) time. For other elements, compare each element with the top element which will take O(1) time. If the new element is larger than the value of the root node, use `DeleteMin` to remove the top element and insert the new element in the heap. This will take O(log(k)).

After all the elements are processed, the final complexity can be given as,

`O(k) + O((N-k)*log(k)) = O(Nlog(k))`

4.5.2 Event Simulation

Problem: The event simulation problem is a very important queuing problem. To understand this problem, just think of a bank where customers arrive and wait in a queue until one of k tellers is available. Customer arrival as well as the service time (time needed to serve a customer once a teller is available) depends on a probability distribution function. In this problem, we need to consider how long on average, a customer has to wait or how long the line might be.

If given some probability distributions and values of k, the answers to these questions can easily be found. When the value of k becomes big, computation gets tougher. We therefore need a computer to simulate the operation of the bank so that the bank officers can determine how many tellers are needed for providing a smooth service to its customers.

Solution: A simulation consists of processing events - customer arriving and customer departing in our bank example. Further, probability functions are used to generate an input stream consisting of ordered pairs of arrival time and service time for each customer. The ordered pair is sorted by arrival time.

One way to simulate processing of events is to start a simulation clock at zero ticks. The clock is then advanced one tick at a time, to check if there is an event. In case a customer arrival event is generated, the event is processed and statistics are compiled. The simulation gets over when all the customers have been serviced and all the tellers are free.

A severe issue with this strategy is that the running time does not depend on the number of customers or events. It depends on the number of ticks, which is not even a part of the input.

In order to overcome this issue, we can advance the clock to the next event time at each stage. At any point, the next event that can occur is either the arrival of next customer, or departure of one of the customers at a teller. Since we already have pairs of arrival and departure times, we just need to find the event that will happen in the nearest future. The next event thus found is then processed.

For example, if the event is a departure, statistics for the departing customer is gathered and a check is made to find if any other customer is waiting for a teller. If so, we use the statistics to compute the time when that customer will leave, and add the arrival and departure to the set of events waiting to happen.

If the event is an arrival, we check for an available teller. If no teller is available, we add the arrival in the queue. In case a teller is available, he/she is allotted to the customer. Use the statistics to compute the time when that customer will leave, and add the arrival and departure to the set of events waiting to happen.

In our simulated environment, waiting line for customers can be implemented as a queue. Since we need to find the event *nearest* in the future, it is best to organize the set of departures waiting to happen in a priority queue. The next event is thus the next arrival or next departure (whichever is sooner).

Complexity: If there are `c` customers, there are 2c events. With `k` tellers, the running time of the simulation would be `O(C log(k + 1))` because computing and processing each event takes `O(log H)` time, where `H = k + 1` is the size of the heap.

4.6 BINOMIAL HEAPS (OR QUEUES)

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. 4.19 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.

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.

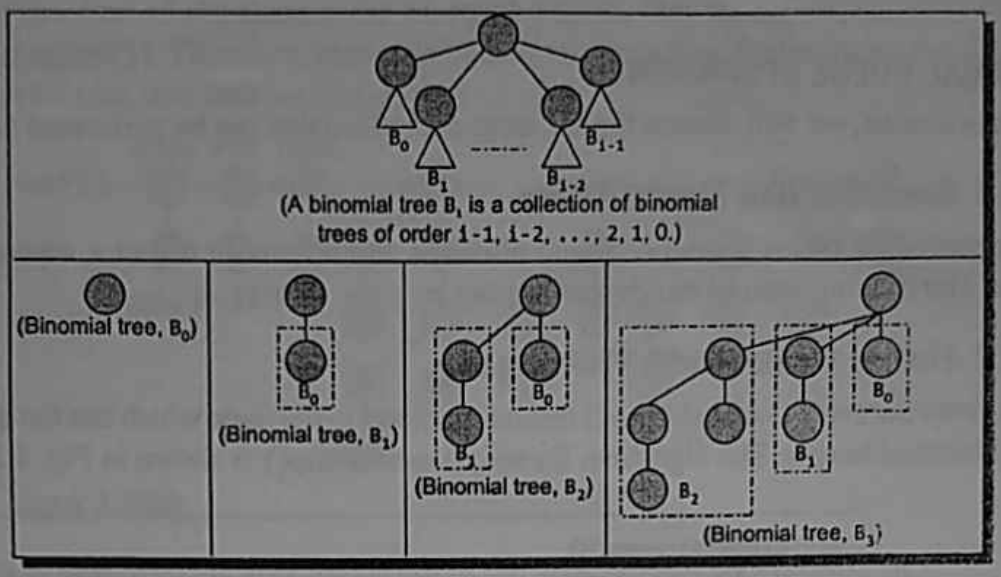


Figure 4.19 Binomial trees

4.7 BINOMIAL HEAP (OR QUEUE) STRUCTURE AND IMPLEMENTATION

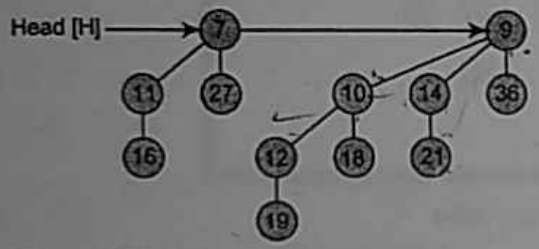


Figure 4.20 Binomial heap

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] = NIL`.

In addition to this, every node N has a degree field which stores the number of children of N.

Implementation of Binomial Heap

Look at the binomial heap shown in Fig. 4.20. Figure 4.21 shows its corresponding linked representation. You can even use an array to hold references of root nodes of each binomial tree.

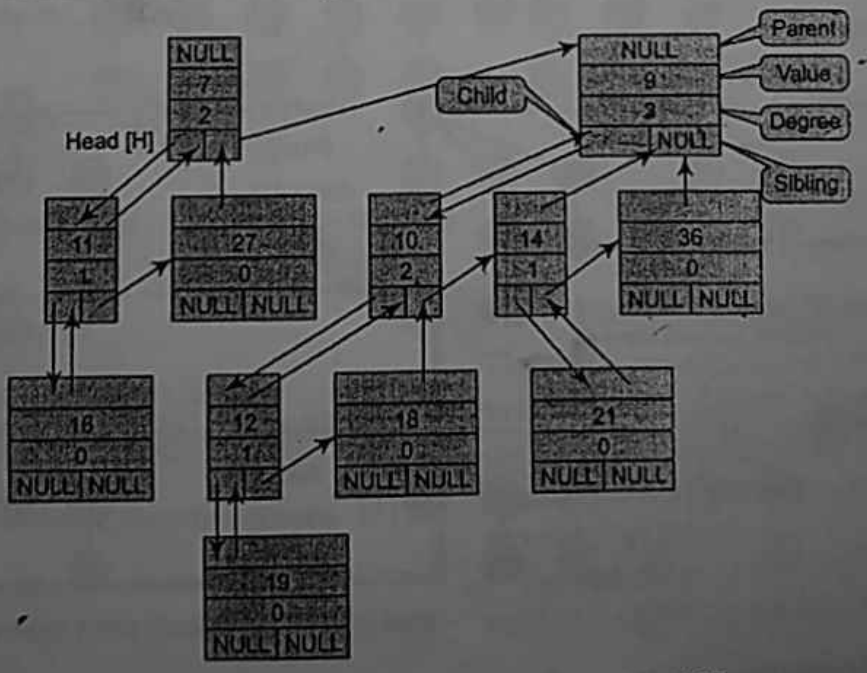


Figure 4.21 Linked representation of the binomial tree shown in Fig. 4.20

4.8 BINOMIAL QUEUE OPERATIONS

In this section, we will discuss the different operations that can be performed on binomial heaps.

4.8.1 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)`.

4.8.2 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. 4.22.

```
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 4.22 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 4.4 Consider the binomial heap shown in Fig. 4.23(a) and see how the procedure works in this case.

Solution

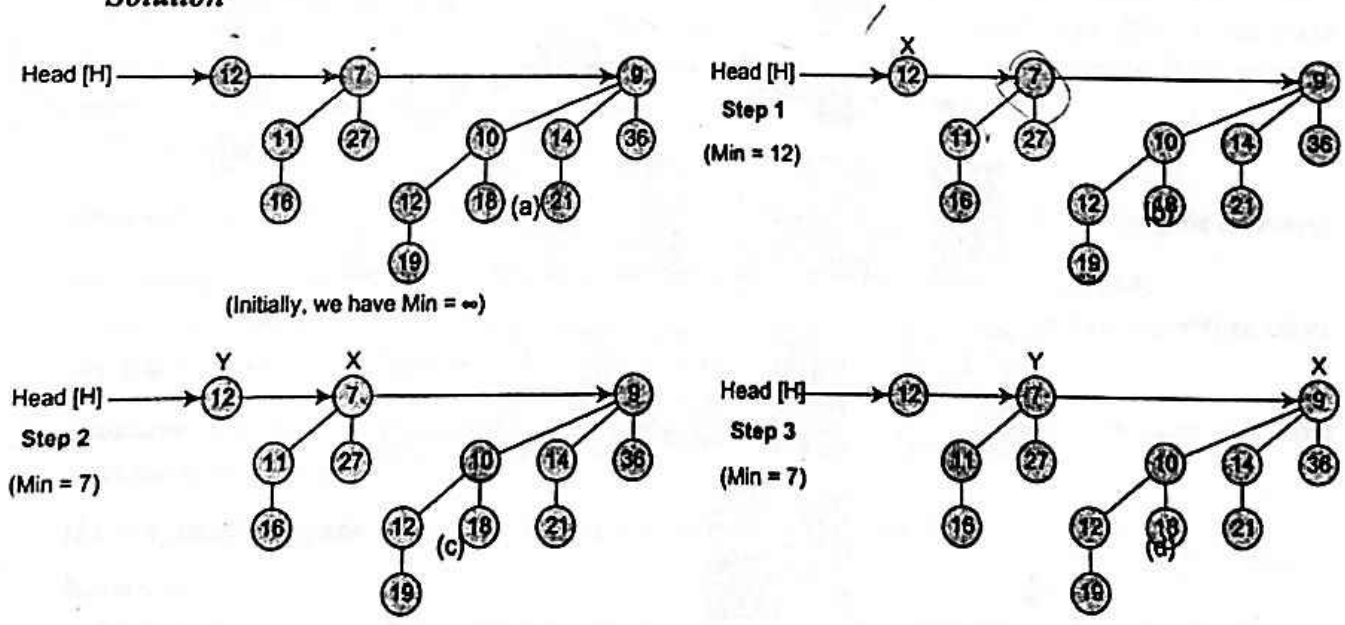


Figure 4.23 Binomial heap

4.8.3 Linking and 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. 4.24.

```
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 4.24 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. 4.25.

```
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 4.25 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 4.5 Unite the two binomial heaps shown in Fig. 4.26(a).

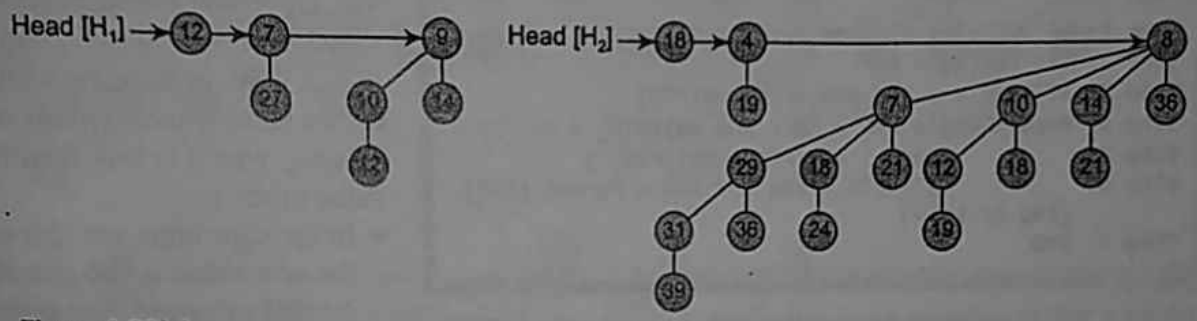


Figure 4.26(a)

Solution

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

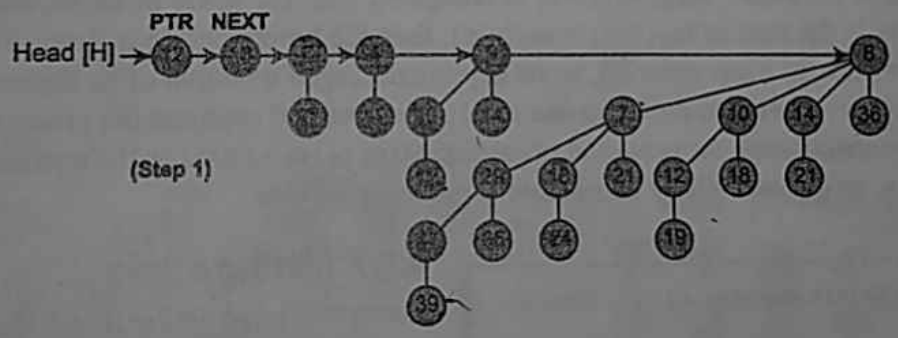


Figure 4.26(b)

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

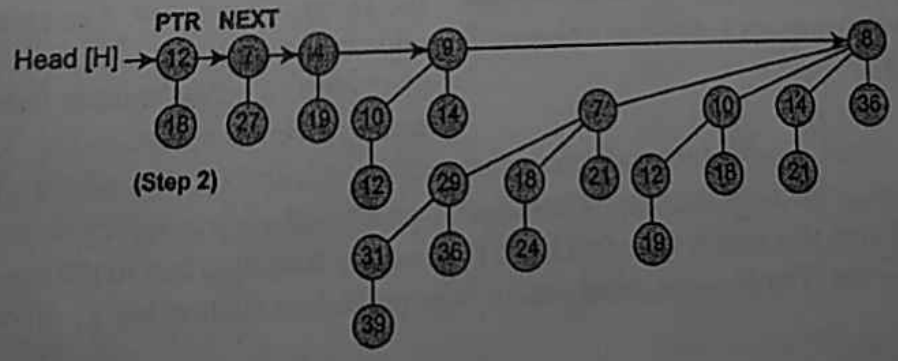


Figure 4.26(c)

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]`.

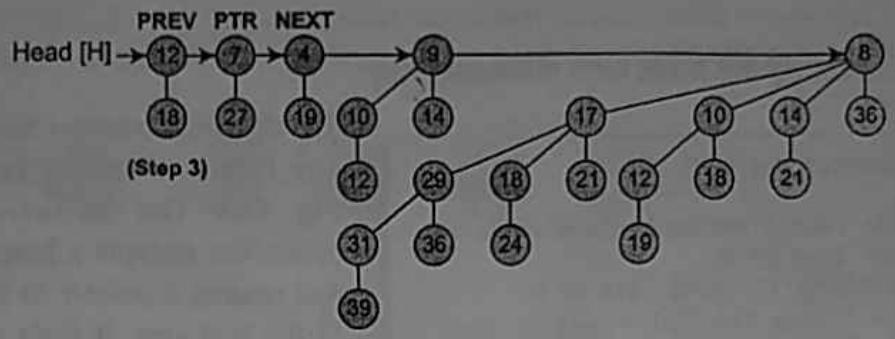


Figure 4.26(d)

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

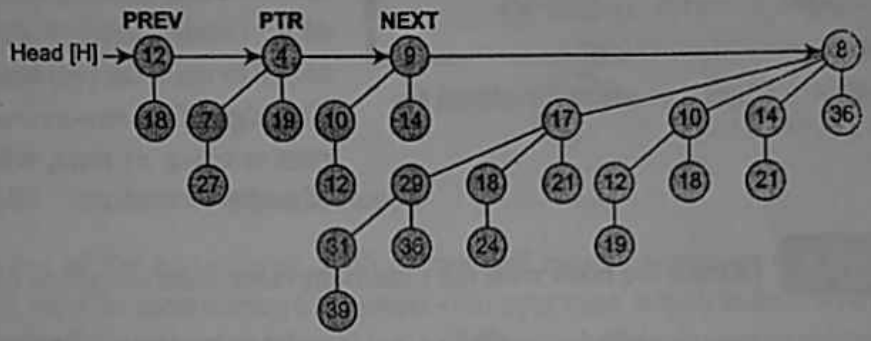


Figure 4.26(e)

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

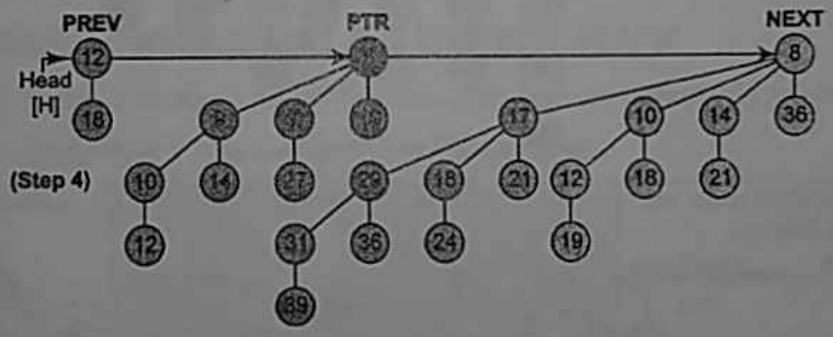


Figure 4.26(f) Resultant binomial heap

```
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 4.27 Algorithm to insert a new element in a binomial heap

4.8.4 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. 4.27 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.

4.8.5 Extracting the Node with Minimum Key

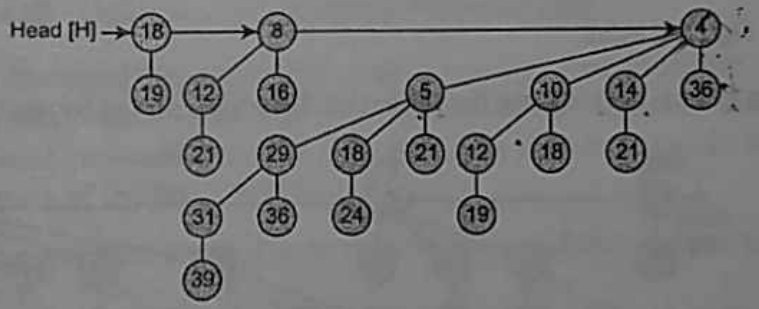
```
Min-Extract_Binomial Heap (H)

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')
```

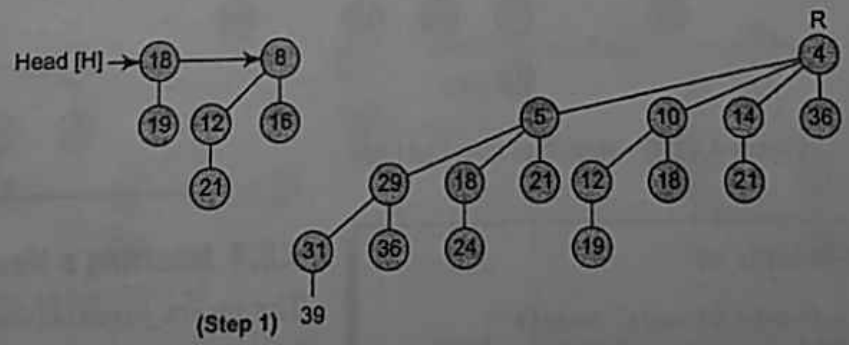
Figure 4.28 Algorithm to extract the node with minimum key from a binomial heap

The algorithm to extract the node with minimum key from a binomial heap H is shown in Fig. 4.28. 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.

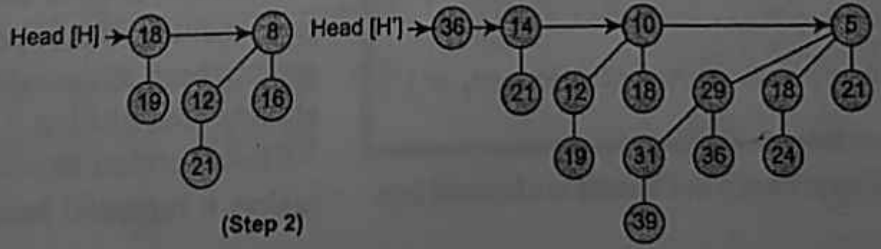
Example 4.6 Extract the node with the minimum value from the given binomial heap.



Solution



(Step 1)



(Step 2)

(Contd)

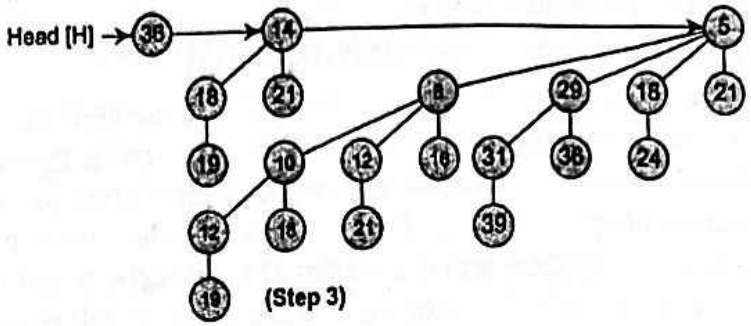


Figure 4.29 Binomial heap

```
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 4.30 Algorithm to decrease the value of a node x in a binomial heap H

4.8.6 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. 4.30. 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-Heap_Delete-Node(H, x)

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

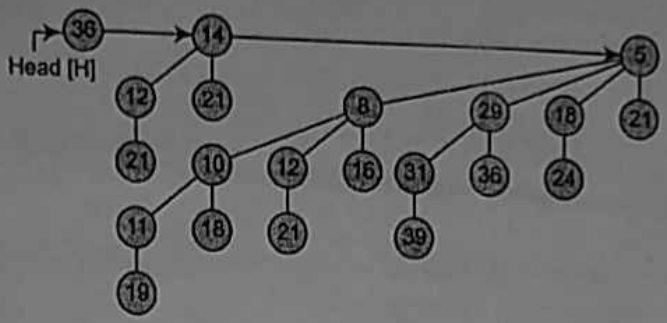
Figure 4.31 Algorithm to delete a node from a bionomial heap

4.8.7 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 $-\infty$. Assuming that there is no node in the heap that has a value less than $-\infty$, the algorithm to delete a node from a binomial heap can be given as shown in Fig. 4.31.

The `Binomial-Heap_Delete-Node` procedure sets the value of x to $-\infty$, 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 4.7 Delete the node with the value 11 from the binomial heap H.



Solution

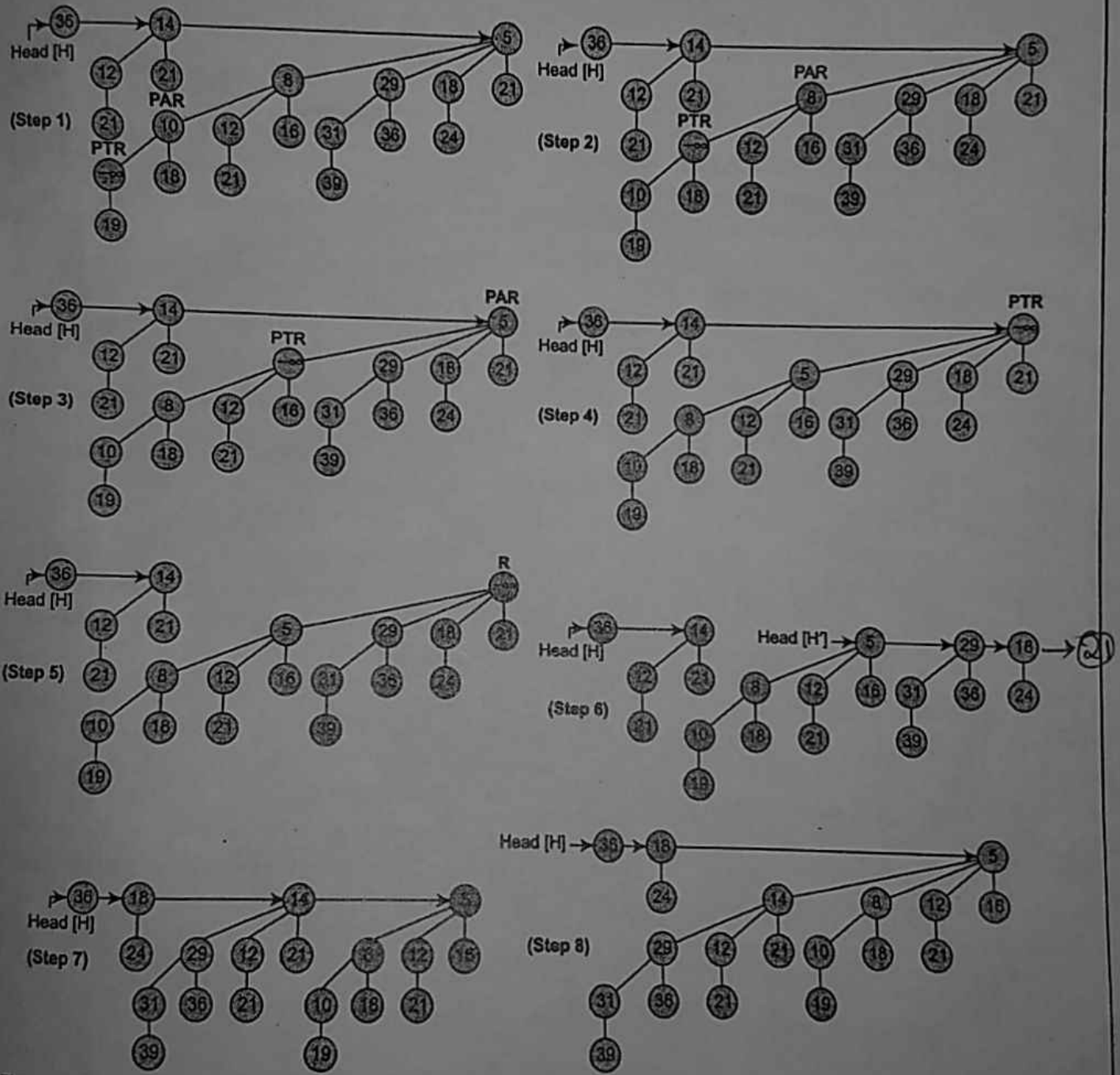


Figure 4.32 Binomial heap

4.8.8 Lazy Binomial Queue

The word 'lazy' in a lazy binomial heap has been specifically used because this variant of binomial heap leaves the work for later. In a lazy binomial queue, merging is done lazily. In order to merge two binomial queues, the two queues are simply concatenated as two lists of binomial trees. The resulting forest may have several trees of the same order. (Refer to Fig. 4.33).

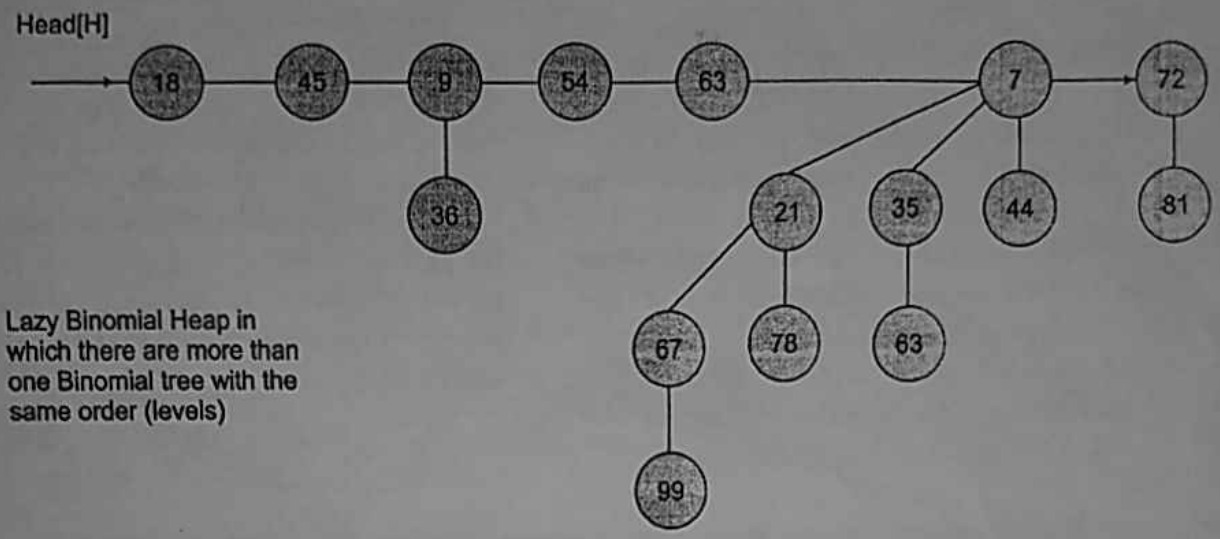


Figure 4.33 Concatenation of binomial queue

Because of the lazy merge, both merge and insert operations take O(1) time in worst case. The real work is done during the extract-min operation which is followed by linking trees of the same order. This operation takes O(log n) time which is same as that required on standard queue. However, the Deletemin operation first converts the lazy binomial queue into a standard binomial queue and then performs the Deletemin operation on the standard queue.

4.9 COMPARISON BETWEEN BINARY AND BINOMIAL HEAPS

Table 4.1 makes a comparison of the operations that are commonly performed on heaps.

Table 4.1 Comparison of binary, binomial, and Fibonacci heaps

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

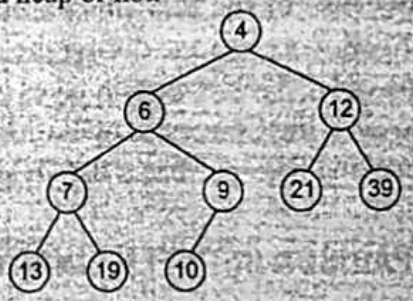
Points to Remember

- 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 priority queue is a data structure in which each element is assigned a priority. The priority of the element will be used to determine the order in which the elements will be processed.
- When a priority queue is implemented using a linked list, then every node of the list will have three parts: (a) the information or data part, (b) the priority number of the element, and (c) the address of the next element.

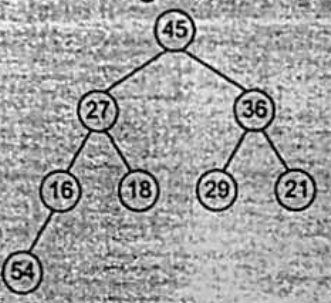
Exercises

Review Questions

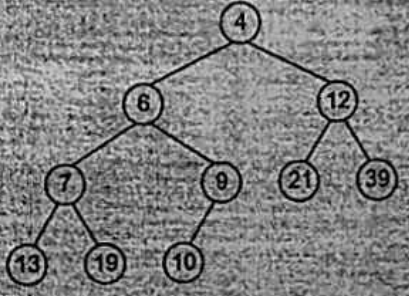
1. Define a binary heap.
2. Differentiate between a min-heap and a max-heap.
3. Compare binary trees with binary heaps.
4. Explain the steps involved in inserting a new value in a binary heap with the help of a suitable example.
5. Explain the steps involved in deleting a value from a binary heap with 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 and binomial heaps.
11. Analyse the complexity of the algorithm to unite two binomial heaps.
12. The running time of the algorithm to find the minimum key in a binomial heap is O(log n). Comment.
13. Discuss the process of inserting a new node in a binomial heap. Explain with the help of an example.
14. The algorithm Min-Extract_Binomial-Heap() runs in O(log n) time where n is the number of nodes in H. Justify this statement.
15. Explain how an existing node is deleted from a binomial heap with the help of a relevant example.
16. Consider the figure given below and state whether it is a heap or not.



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



18. Show the array implementation of the following heap.



19. 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.

20. 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

21. 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?

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

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

24. Analyse the complexity of heapify algorithm.

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) $O(1)$

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. The running time of `Min_Binomial-Heap()` procedure is $O(\log n)$.
9. In a priority queue, two elements with the same priority are processed on a FCFS basis.

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. _____ are appropriate data structures to process batch computer programs submitted to the computer centre.