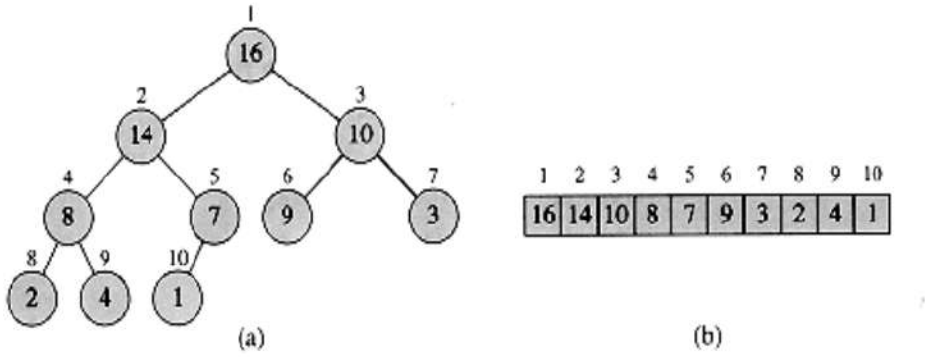
Heaps, Heap Sort and Prioirty Queues

The (binary) heap data structure is an array that can be viewed as a complete binary tree, as shown in figure below.



Each node of the tree corresponds to an element of the array that stores the record with key (simply call value) in the node. Being a complete binary tree, it is completely filled on all levels except possibly the lowest, which is filled from the left up to a point. The root of the tree is A[1], and given the index i of a node, the indices of its parent PARENT(i), left child LEFT(i), and right child RIGHT(i) can be computed simply as follows.

- PARENT(i) : { return $\lfloor i/2 \rfloor$}
- LEFT(i) : { return $2i$}
- RIGHT(i) : { return 2i+1}

Additionally, heaps also satisfy the **heap property**: for every node i other than the root, $A[PARENT(i)] \leq A[i]$ for a min-heap and $A[PARENT(i)] \geq A[i]$ for a max-heap. The basic heap operations may be listed as follows.

1 Maintaining the heap property

HEAPIFY is an important subroutine for manipulating heaps. Its inputs are an array A and an index i into the array. When HEAPIFY is called, it is assumed that the binary trees rooted at LEFT(i) and RIGHT(i) are heaps, but that A[i] may be smaller than its children, thus violating the (max) heap property. The function of HEAPIFY is to let the value at A[i] "float down" in the heap so that the subtree rooted at index i becomes a heap.

```
HEAPIFY(A, i)
{
    l = LEFT(i);
    r = RIGHT(i);
    if ((l <= heap-size) && (A[l] > A[i]))
        largest = l;
    else largest = i;
    if ((r <= heap-size) && (A[r] > A[largest]))
        largest = r;
    if (largest != i)
        exchange(A[i], A[largest]);
    HEAPIFY(A,largest);
}
```

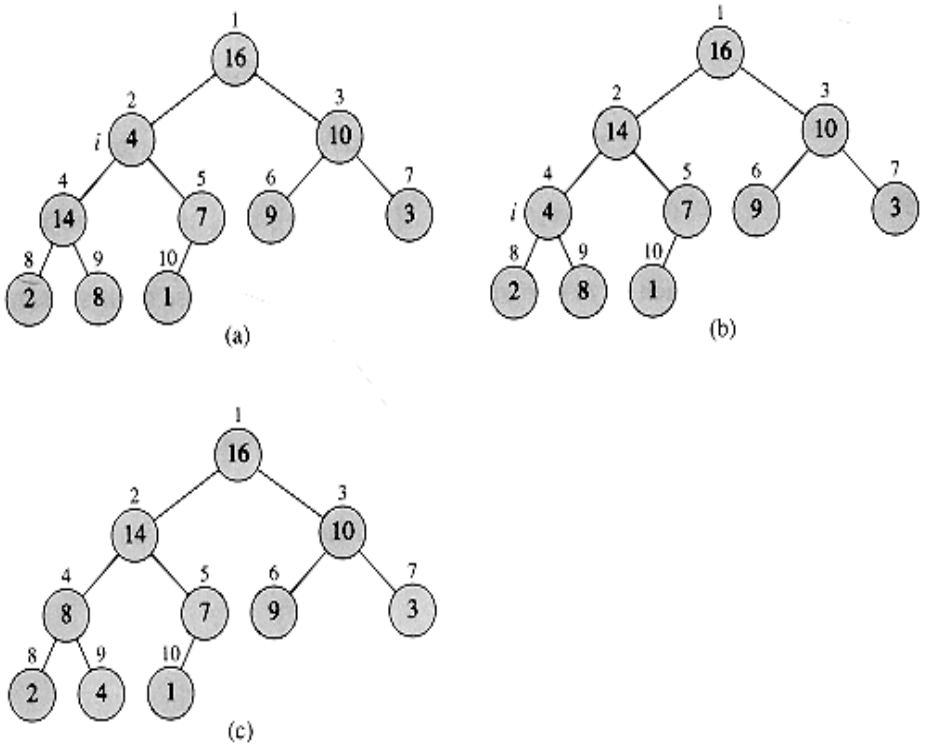


Figure 1: The action of HEAPIFY(A, 2), where heap-size[A] = 10. (a) The initial configuration of the heap, with A[2] at node i = 2 violating the heap property since it is not larger than both children. The heap property is restored for node 2 in (b) by exchanging A[2] with A[4], which destroys the heap property for node 4. The recursive call HEAPIFY(A, 4) now sets i = 4. After swapping A[4] with A[9], as shown in (c), node 4 is fixed up, and the recursive call HEAPIFY(A, 9) yields no further change to the data structure.

2 Building a heap

We can use the procedure HEAPIFY in a bottom-up manner to convert an array A[1 . . n], where n = length[A], into a heap. Since the elements in the subarray A[(n/2 + l) . . n] are all leaves of the tree, each is a 1-element heap to begin with. The procedure BUILD-HEAP goes through the remaining nodes of the tree and runs HEAPIFY on each one. The order in which the nodes are processed guarantees that the subtrees rooted at children of a node i are heaps before HEAPIFY is run at that node.

```
BUILD-HEAP(A)
{
    heap-size = length(A);
    for(i = heap-size/2; i >= 1; i--)
        HEAPIFY(A, i);
}
```

3 The heapsort algorithm

The heapsort algorithm starts by using BUILD-HEAP to build a heap on the input array A[1 . . n], where n = length[A]. Since the maximum element of the array is stored at the root A[1], it can be put into its correct final position by exchanging it with A[n]. If we now "discard" node n from the heap (by decrementing heap-size(A)), we observe that A[1 . . (n - 1)] can easily be made into a heap. The children of the root remain heaps, but the new root element may violate the heap

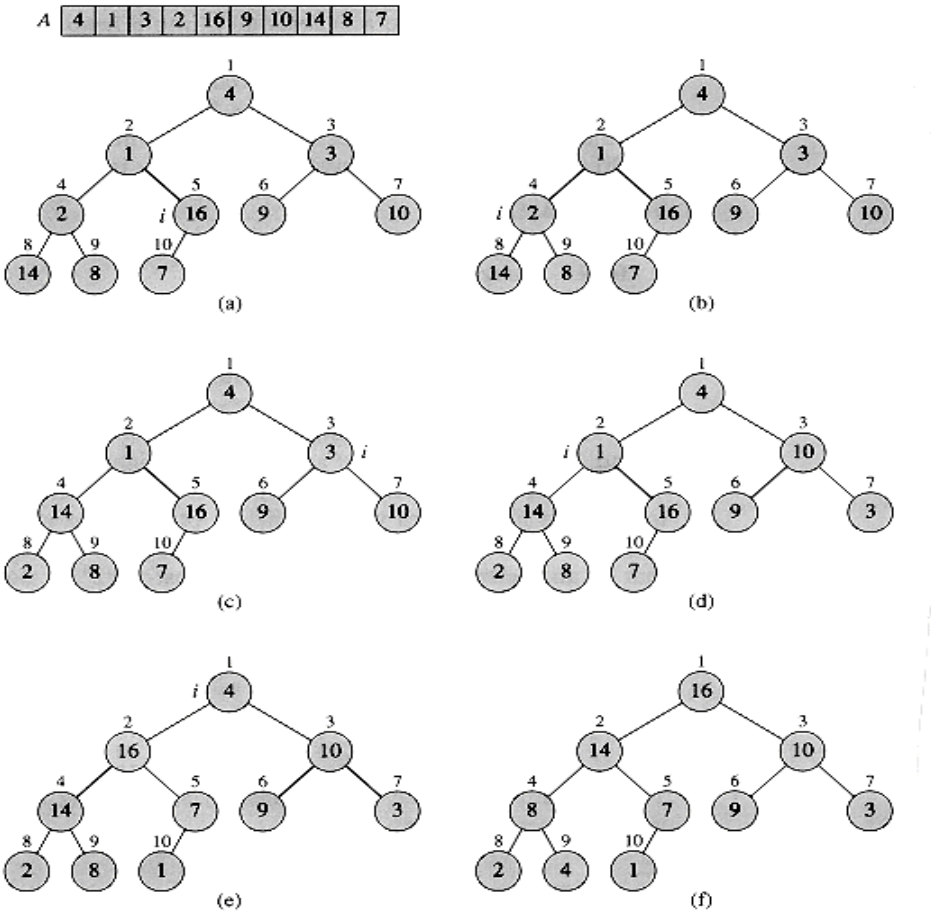


Figure 2: The operation of BUILD-HEAP, showing the data structure before the call to HEAPIFY in line 3 of BUILD-HEAP. (a) A 10-element input array A and the binary tree it represents. The figure shows that the loop index i points to node 5 before the call HEAPIFY(A, i). (b) The data structure that results. The loop index i for the next iteration points to node 4. (c)-(e) Subsequent iterations of the for loop in BUILD-HEAP. Observe that whenever HEAPIFY is called on a node, the two subtrees of that node are both heaps. (f) The heap after BUILD-HEAP finishes.

property. All that is needed to restore the heap property, however, is one call to HEAPIFY(A, 1), which leaves a heap in A[1 . . (n - 1)]. The heapsort algorithm then repeats this process for the heap of size n - 1 down to a heap of size 2.

```
HEAPSORT(A)
{
     BUILD-HEAP(A);
     for(i = length(A); i>=2; i--)
     {
          exchange(A[1], A[i]);
          heapsize--;
          HEAPIFY(A,1);
     }
}
```

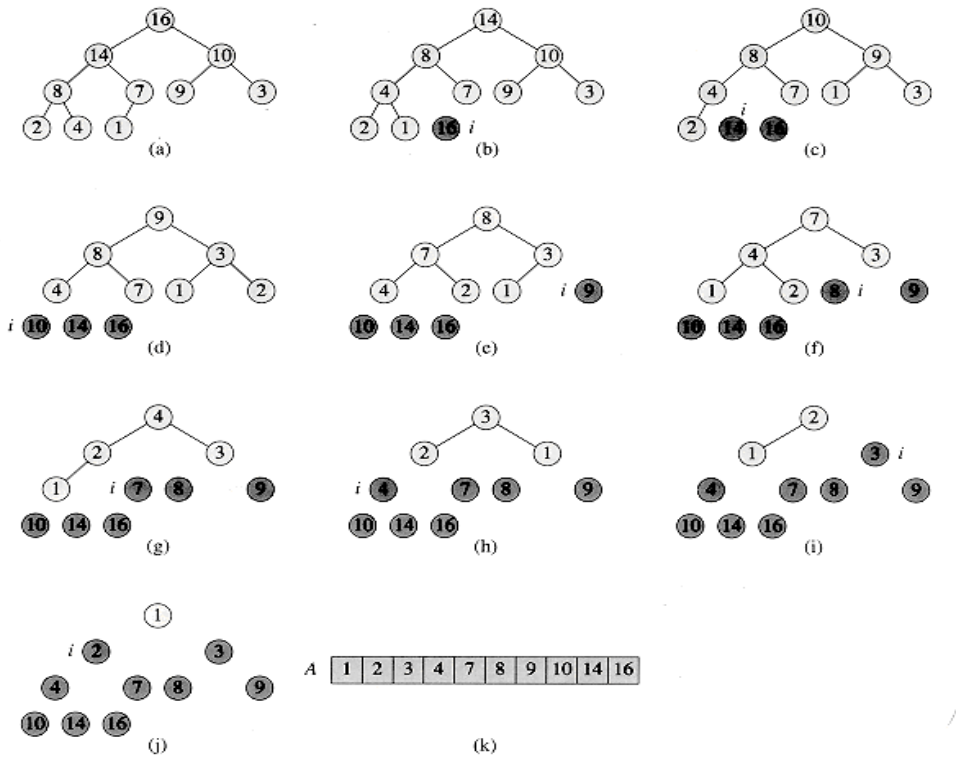


Figure 3: The operation of HEAPSORT. (a) The heap data structure just after it has been built by BUILD-HEAP. (b)-(j) The heap just after each call of HEAPIFY. The value of i at that time is shown. Only lightly shaded nodes remain in the heap. (k) The resulting sorted array A.

	inplace	stable	worst	average	best	remarks
quick	X		N^2	NlogN	NlogN	fastest in practice, NlogN probabilistic guarantee
merge		X	NlogN	NlogN	NlogN	N log N guarantee, stable
heap	X		NlogN	NlogN	NlogN	N log N guarantee,in-place

4 Properties of Heapsort

So, if you have to sort in O(N log N) worst-case without using extra memory, you may consider using heapsort. Mergesort will consume linear amount of extra space. Quicksort takes quadratic time in the worst case. However, as a downside,

- inner loop of heapsort is longer than quicksorts leading to extra costs.
- heapsort makes poor use of cache memory.

5 Priority Queues

Heapsort is an excellent algorithm, but a good implementation of quicksort, usually beats it in practice. Nevertheless, the heap data structure itself has enormous utility. In this section, we present one of the most popular applications of a heap: its use as an efficient priority queue. A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. A priority queue supports the following operations.

- INSERT(S, x) inserts the element x into the set S.
- MAXIMUM/MINIMUM(S) returns the element of S with the largest/smallest key.
- EXTRACT-MAX/MIN(S) removes and returns the element of S with the largest/smallest key.

One application of priority queues is to schedule jobs on a shared computer. The priority queue keeps track of the jobs to be performed and their relative priorities. When a job is finished or interrupted, the highest-priority job is selected from those pending using EXTRACT-MAX. A new job can be added to the queue at any time using INSERT.

A priority queue can also be used in an event-driven simulator. The items in the queue are events to be simulated, each with an associated time of occurrence that serves as its key. The events must be simulated in order of their time of occurrence, because the simulation of an event can cause other events to be simulated in the future. For this application, it is natural to reverse the linear order of the priority queue and support the operations MINIMUM and EXTRACT-MIN instead of MAXIMUM and EXTRACT-MAX. The simulation program uses EXTRACT-MIN at each step to choose the next event to simulate. As new events are produced, they are inserted into the priority queue using INSERT.

Not surprisingly, we can use a heap to implement a priority queue. The operation HEAP-MAXIMUM returns the maximum heap element in constant time by simply returning the value A[1] in the heap. The HEAP-EXTRACT-MAX procedure is similar to the for loop body of the HEAPSORT procedure.

```
HEAP-EXTRACT-MAX(A)
{
     if heap-size < 1
          error "heap underflow";
     max = A[1];
     A[1] = A[heap-size];
     heap-size=heap-size - 1;
     HEAPIFY(A, 1);
     return max;
}
```

The running time of HEAP-EXTRACT-MAX is O(lg n), since it performs only a constant amount of work on top of the O(lg n) time for HEAPIFY. The HEAP-INSERT procedure inserts a node into heap A. To do so, it first expands the heap by adding a new leaf to the tree. Then, it traverses a path from this leaf toward the root to find a proper place for the new element.

```
HEAP-INSERT(A,key)
{    In array representation, you should for overflow here;
     heap-size=heap-size + 1;
     i = heap-size;
     while i > 1 and A[PARENT(i)] < key
     {
          A[i] = A[PARENT(i)];
          i = PARENT(i);
     }
     A[i] = key;
}
```

Observe the key difference between heap and BST in this context. A min-heap is a rooted tree such that the key stored at every node is less or equal to the keys of all its descendants. Similarly a max-heap is one in which the key at a node is greater or equal to all its descendants. A search-tree is a rooted tree such that the key sorted at every node is greater than (or equal to) all the keys in its left subtree and less than all the keys in its right subtree. Heaps maintain only a partial ordering, whereas search trees maintain a total ordering. Finding a min/max in BST incurs logarithmic time, while in heap it is constant time. Hence, for a priority queue implementation, we go for a heap.