Heap

Heaps are, after the search trees, the second most studied type of data structure. As abstract structure they are also called priority queues, and they keep track of a set of objects, each object having a key value (the priority), and support the operations to `insert` an object, find the object of minimum key (`find min`), and delete the object of minimum key (`delete min`). So unlike the search trees, there are neither arbitrary find operations nor arbitrary delete operations possible. Of course, we can replace everywhere the minimum by maximum; where this distinction is important, one type is called the min-heap and the other the max-heap. If we need both types of operations, the structure is called a double-ended heap, which is a bit more complicated.

The heap structure was originally invented by Williams (1964) for the very special application of sorting, although he did already present it as a separate data structure with possibly further applications. But it was recognized only much later that heaps have many other, and indeed more important, applications. Still, the connection to sorting is important because the lower bound of $\Omega(n \log n)$ on comparison-based sorting of n objects implies a lower bound on the complexity of the heap operations. We can sort by first inserting all objects in the heap and then performing `find min` and `delete min` operations to recover the objects, sorted in increasing order. So we can sort by performing n operations each of `insert`, `find min`, and `delete min`; thus, at least one of these operations must have (in a comparison-based model) a complexity $\Omega(\log n)$. This connection works in both directions; there is an equivalence between the speed of sorting and heap operations in many models – even in which the comparison-based lower bound for sorting does not hold.

The various methods to realize the heap structure differ mainly by the additional operations they support. The most important of these are the merging of several heaps (taking the union of the underlying sets of objects), which is sometimes also called melding, and the change of the key of an object (usually decreasing the key), which requires a finger to the object in the structure.

The most important applications of heaps are all kinds of event queues, as they occur in many diverse applications: sweeps in computational geometry, discrete event systems (Evans 1986), schedulers, and many classical algorithms such as Dijkstra's shortest path algorithm.

In this chapter, we are going to explore an important data structures named heap. We will start with the basic concept of heap and types of heap. Later we will dive deeper with lots of examples and their explanations. Finally, we will be ending this chapter by learning the applications of heap.

2.1 Heap Types

A heap is an almost complete binary tree with all levels is full, except possibly the last one, which is filled from left to right.

Definition 8.1: We define "height" of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf. The height of a heap is the height of its root. A heap of `n` nodes has a height of $\lfloor \log n \rfloor$. (Hint: if a heap has height h, then the minimum and maximum possible number of elements is $2h \leq n \leq 2h + 1 - 1$.

Definition 8.2: The "max-heap property" (of a heap A) is the property we were talking about, where for every node `i` other than the root, `A[parent[i]]≥A[i]`. The "min-heap property" is defined analogously.

Max heap can be viewed as a binary tree, where each node has two (or fewer) children, and the key of each node (i.e. the number inside the node) is greater than the keys of its child nodes. (From now on I'm going to say node `i` is larger than node `j` when I really mean that the key of `i` is larger than the key of `j`.) There is also min heaps, where each node is smaller than its child nodes, but here we will talk about max heaps, with the understanding that the algorithms for min heaps are analogous.

For example, the root of a max heap is the largest element in the heap. However, note that it's possible for some nodes on level 3 to be smaller than nodes on level 4 (if they're in different branches of the tree). In figure 8.1, it doesn't matter that 4 in level 1 is smaller than 5 in level 2. Figure 8.2 shows the example of max heap and figure 8.3 shows the example of max heap and min heap.

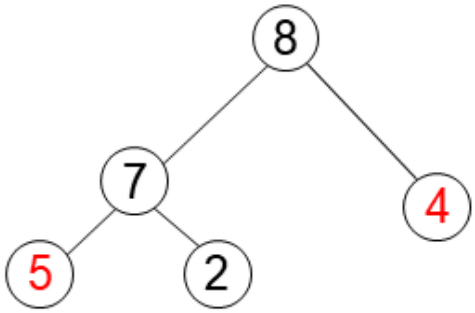


Figure 8.1: Heap (top to bottom and left to right)

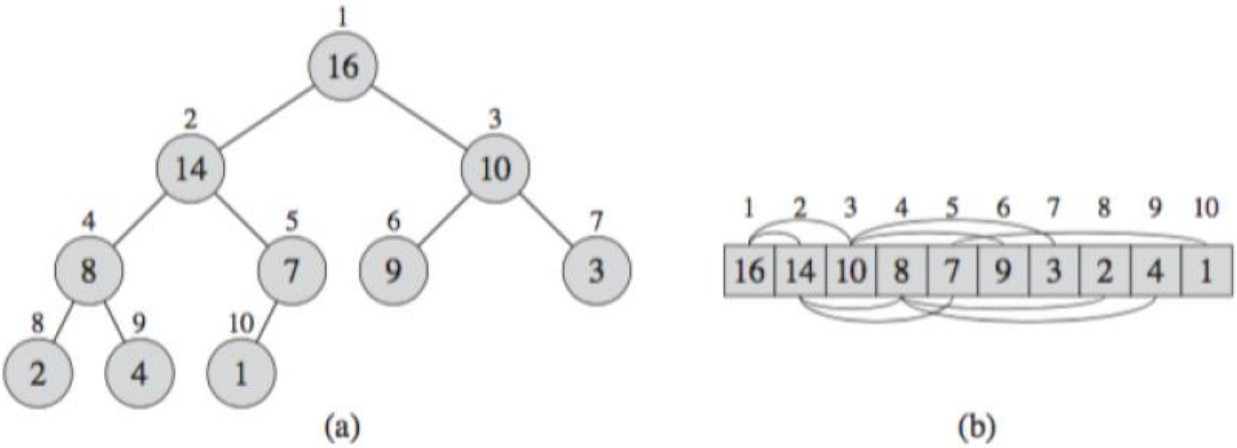


Figure 8.2 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

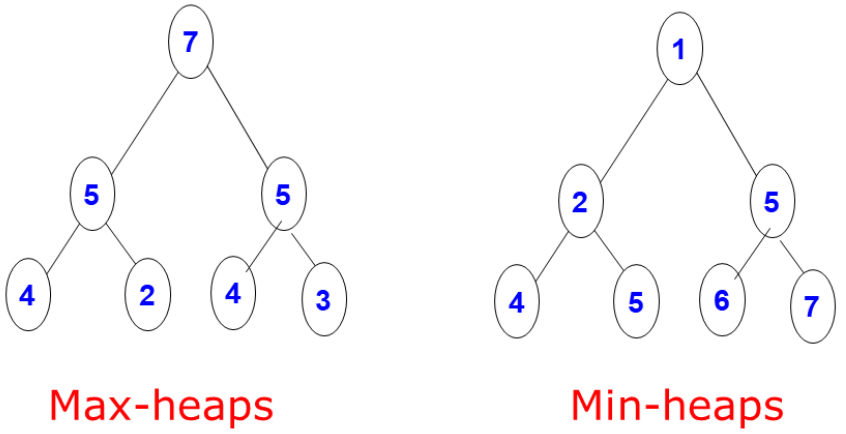


Figure 8.3: The example of max heap and min heap

2.2 Heap as an Array

Heap can be implemented as an array. This array is essentially populated by "reading of" the numbers in the tree, from left to right and from top to bottom. The root is stored at index 0, and if a node is at index i, then

- Its left child has index 2i+1
- Its left child has index 2i+2
- Its parent has index $\lfloor i - 1/2 \rfloor$

Furthermore, for the heap array A, we also store two properties: length n, which is the number of elements in the array, and heap size, which is the number of array elements that are actually part of the heap. Even though the array A is filled with numbers, only the elements in A[0…heapsize] are actually part of the heap. Therefore, Heapsize[A] ≤ length[A]. The elements in $A[\lfloor n/2 \rfloor]$ to A[n-1] are leaves and Parents are A[0] to $A[\lfloor n/2 \rfloor - 1]$. The root have the maximum element of the heap.

2.3 Operations on Heap

The common operation involved using heaps are:

- Maintaining the heap property :
  It is a process to rearrange the elements of the heap in order to maintain the heap property. It is done when a certain node causes an imbalance in the heap due to some operation on that node. For example, Max-Heapify .

- Create a max-heap from an unordered array:
  The Build-Max-Heap function that follows, converts an array *A* which stores a complete binary tree with *n* nodes to a max-heap by repeatedly using Max-Heapify in a bottom up manner. It is based on the observation that the array elements indexed by $A[\lfloor n/2 \rfloor]$ to A[n-1] are all leaves for the tree (assuming that indices start at 0) , thus each is a one-element heap. Build-Max-Heap runs Max-Heapify on each of the remaining tree nodes.

- Sort an Heap Array
  Heapsort can be thought of as an improved selection sort like selection sort, heapsort divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region. Unlike selection sort, heapsort does not waste time with a linear-time scan of the unsorted region; rather, heap sort maintains the unsorted region in a heap data structure to more quickly find the largest element in each step [1].

2.3.1 Maintaining the heap property

Heaps use this one weird trick to maintain the max-heap property. Unlike most of the algorithms we've seen so far, Max-Heapify assumes a very special type of input. Given a heap A and a node i inside that heap, Max-Heapify attempts to "heapify" the subtree rooted at i, i.e., it changes the order of nodes in A so that the subtree at i satisfies the max-heap property. However, it assumes that the left subtree of i and the right subtree of i are both max-heaps in and of themselves, so the only possible violation is that i might be smaller than its immediate children.

In order to fix this violation, the algorithm finds which node is largest: i, i's left child, or i's right child. If i is largest, the max-heap property is already satisfied. Otherwise, it swaps i with the largest node, which causes the top of the heap to be fine. But this might mess up the subtree that used to be under the largest node, because i might be smaller than some of the

nodes in that subtree. So now we call `Max-Heapify` on that subtree (because the only possible violation of the max-heap property occurs at the top of the subtree). `Max-Heapify` runs in time `O(h)`, where `h` is the height of the node.

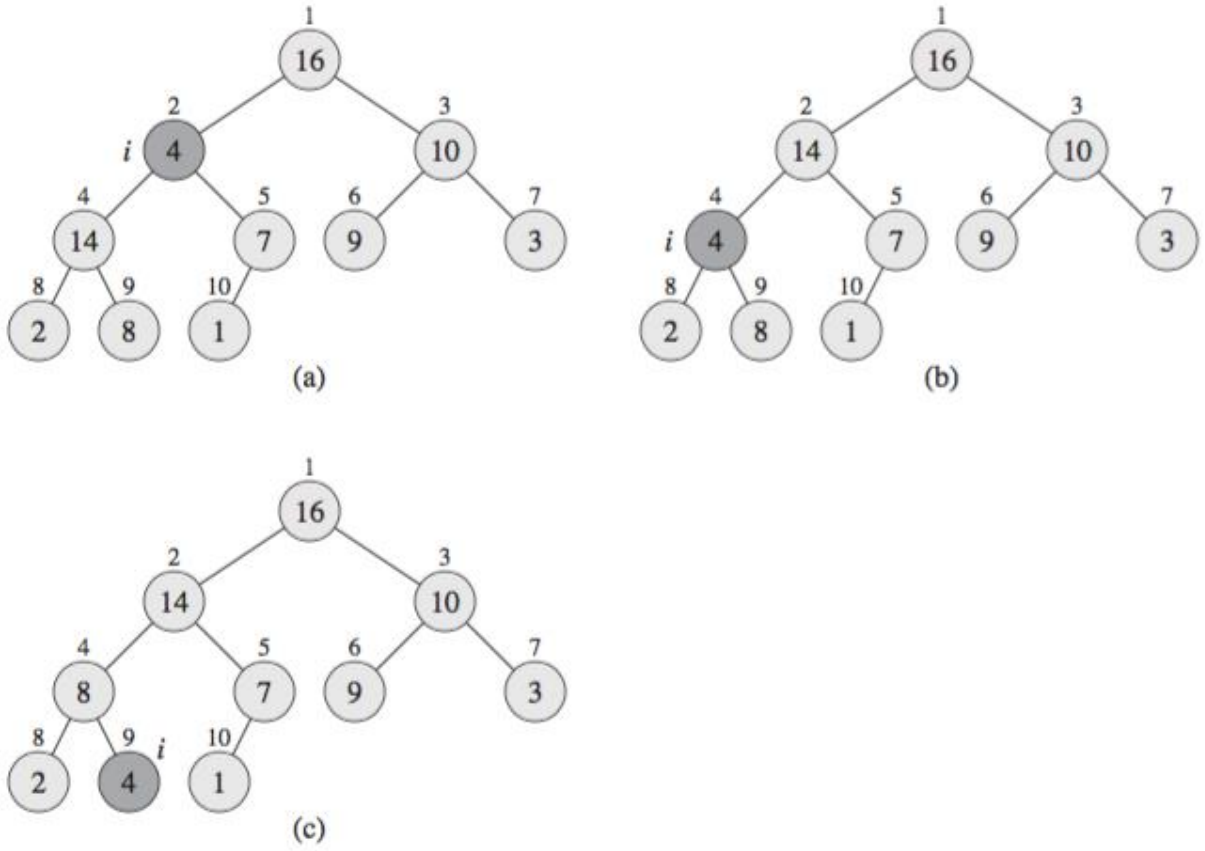


Figure 8.4 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.

Recall that we are implementing heaps as arrays, so the actual code of `Max-Heapify` takes as input an array `A` and an index `i` into that array.

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7       largest = r
 8  if largest ≠ i
 9       exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

2.3.2 Building a heap

The `Build-Max-Heap` procedure runs `Max-Heapify` on all the nodes in the heap, starting at the nodes right above the leaves and moving towards the root. We start at the bottom because in order to run `Max-Heapify` on a node, we need the subtrees of that node to already be heaps.

Recall that the leaves of the heap are the nodes indexed by `A[⌊n/2⌋]` ..... `A[n-1]` so when we use our array implementation we just start at the index `A[⌊n/2⌋-1]` and move towards `0`. For example in figure 8.5, we will start from index `4` and move towards `0`.

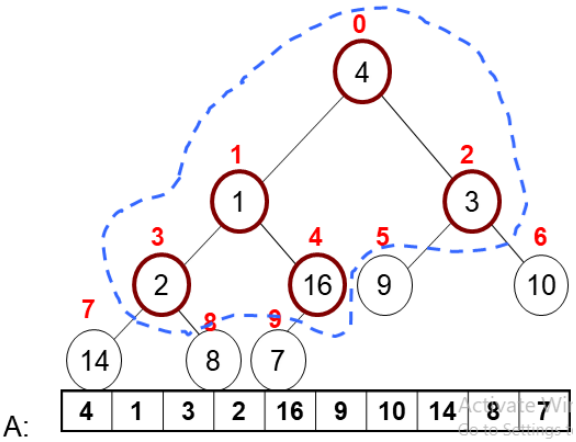


Figure 8.5: The example of Build-Max-Heap range

```
BUILD-MAX-HEAP(A)
   1. n = length of A[]
   2.  for i = ⌊n/2⌋-1 down to 0
   3.        do MAX-HEAPIFY(A, i)
```

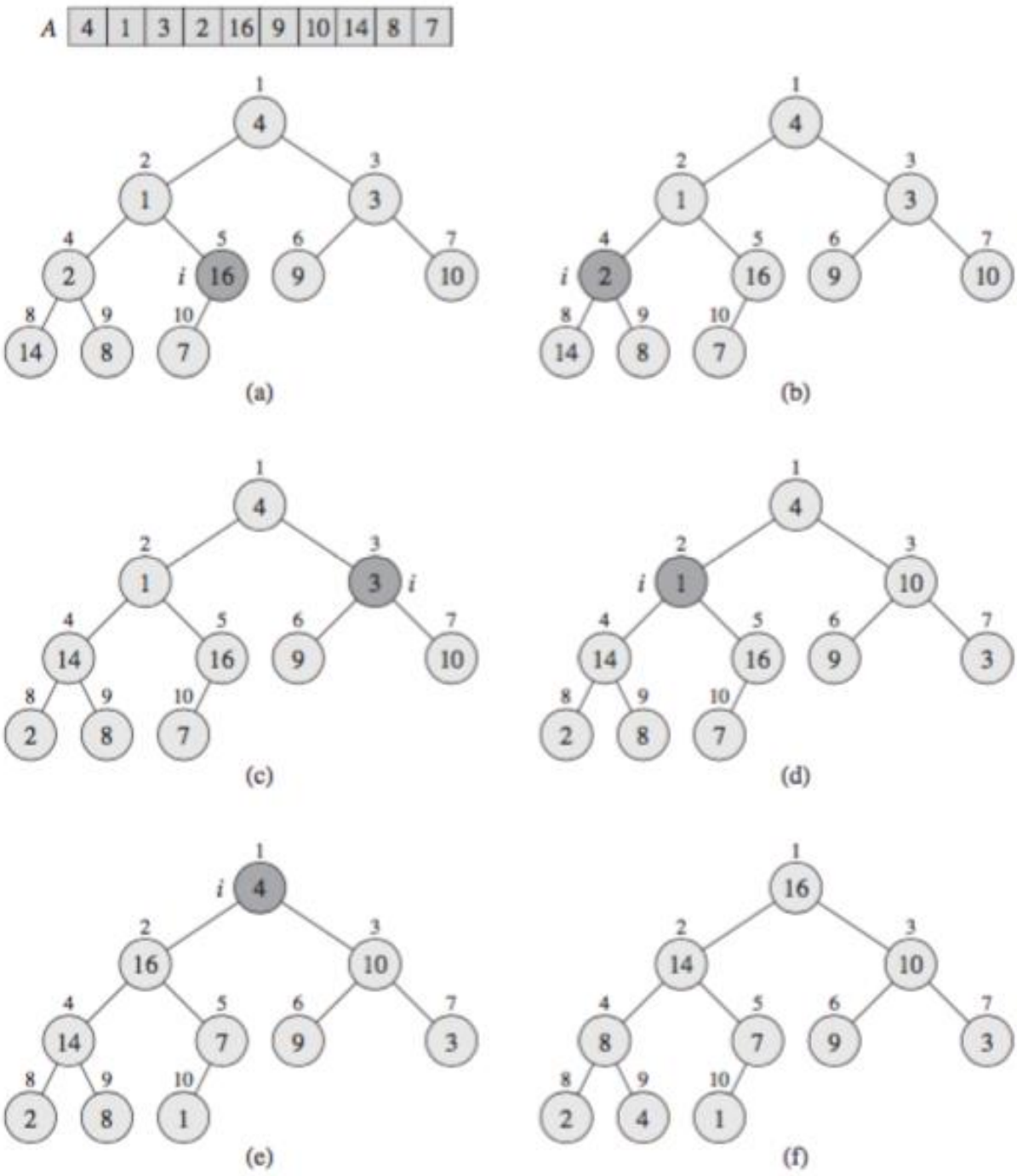


Figure 8.6 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. **(a)** A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). **(b)** The data structure that results. The loop index i for the next iteration refers to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. **(f)** The max-heap after BUILD-MAX-HEAP finishes.

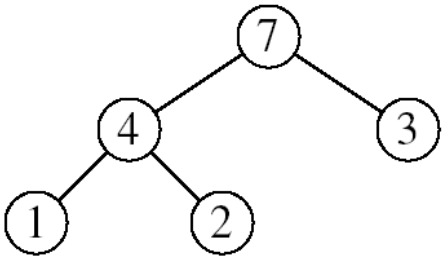
2.3.3 Heap Sort

`Heapsort` is a way of sorting arrays. First we use `Build-Max-Heap` to turn the array into a max-heap. Now we can extract the maximum (i.e. the root) from the heap, swapping it with the last element in the array and then shrinking the size of the heap so we never operate on the max element again. At this point, the heap property is violated because the root may be smaller than other elements, so we call `Max-Heapify` on the root, which restores the max heap property. Now we can keep repeating this until the whole array is sorted.

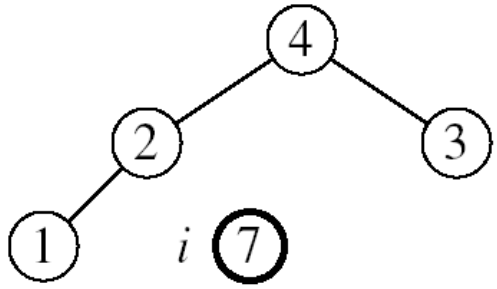
Note that each call to `Max-Heapify` takes `O(log n)` because the height of the heap is `O(log n)`, so heapsort takes `O(n log n)` overall.

```
HeapSort(A)
   1. BUILD-MAX-HEAP(A)
   2. for i ← n-1 down to 1
   3.     do exchange A[0] ↔ A[i]
   4.         n=n-1
   5.         MAX-HEAPIFY(A,0)
```

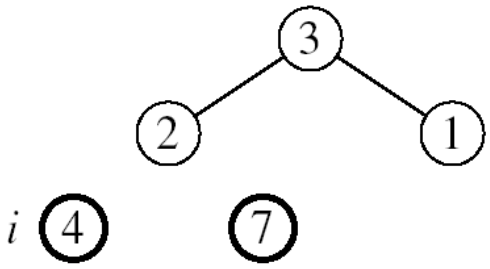
For example, we have a array `A` with values `A=[7,4,3,1,2],` figure 8.7 shows the step by step simulation of heap sort.



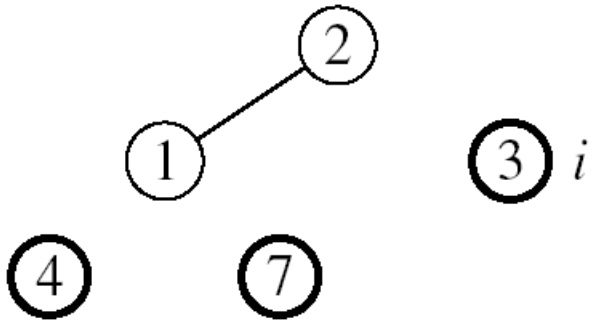
(6) n=4 , Max-Heapify(A,0)



(1) n=3 , Max-Heapify(A,0)



(5) n=2 , Max-Heapify(A,0)



(2) n=1 , Max-Heapify(A,0)

2.4 Complexity of Heapsort

There are two steps to sort an array, or list, containing `N` values, first insert each value into a heap (initially empty) and then remove each value form the heap in ascending order (this is done by `N` successive calls to get smallest).

Hence the the complexity of the Heapsort algorithm comprises of two operations like (`N` insert operations) + (`N` delete operations). Each insert and delete operation is `O(logN)` at the very worst - the heap does not always have all N values in it. So, the complexity is certainly no greater than `O(NlogN)`.

2.5 Uses of Heap

There are two main uses of heaps.

The first is as a way of implementing a special kind of queue, called a priority queue. Recall that in an ordinary queue, elements are added at one end of the queue and removed from the other end, so that the elements are removed in the same order they are added (FIFO). In a priority queue, each element has a priority; when an element is removed it must be the element on the queue with the highest priority.

A very efficient way to implement a priority queue is with a heap ordered by priority - each node is higher priority than everything below it. The highest priority element, then, is at the top of the heap.

The second application is sorting. Heapsort is especially useful for sorting arrays. Because heaps - unlike almost all other types of trees - are usually implemented in arrays, not as linked data structures!

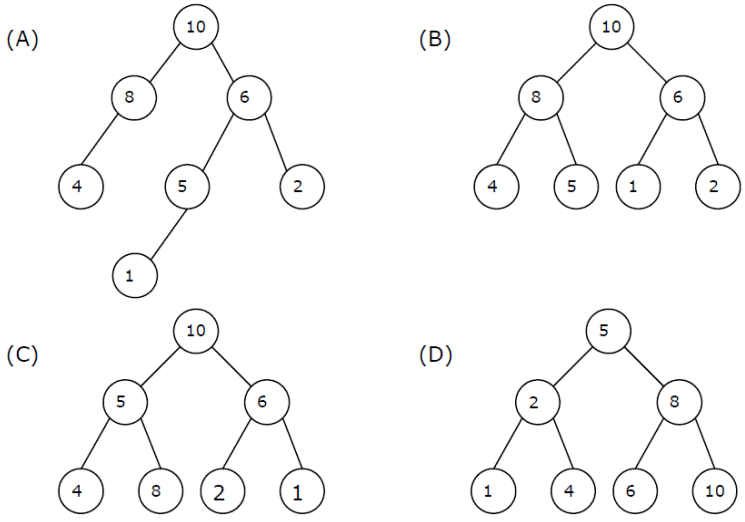
2.6 Some Important Properties of a Heap

- Heaps are based upon trees. These trees maintain the heap property.
    - The Heap invariant. The value of Every Child is greater than the value of the parent. We are describing Min-heaps here (Use less than for Max-heaps).
- The trees must be mostly balanced for the costs listed below to hold.
- Access to elements of a heap usually has the following costs.
    - The cost to find the smallest (largest) element takes constant time.
    - The cost to delete the smallest (largest) element takes time proportional to the log of the number of elements in the set.
    - The cost to add a new element takes time proportional to the log of the number of elements in the set.
- Heaps can be implemented using arrays (using the tree embedding described above) or by using balanced binary trees

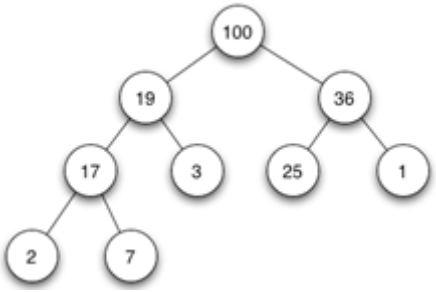
- Trees with the leftist property have the following invariant.
    - The leftist invariant. The rank of every left-child is equal to or greater than the rank of the corresponding right-child. The rank of a tree is the length of the right-most path.
- Heaps form the basis for an efficient sort called heap sort that has cost proportional to `n*log(n)` where n is the number of elements to be sorted.
- Heaps are the data structure most often used to implement priority queues.

2.7 Exercises

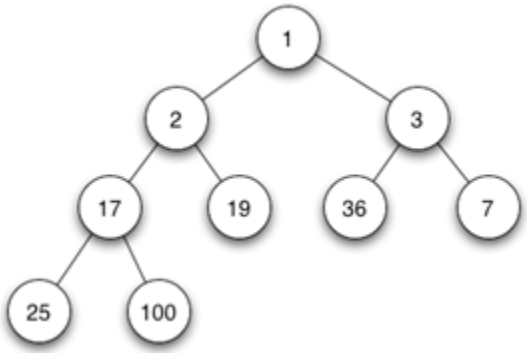
1. What is the time complexity of Build Heap operation. Build Heap is used to build a max(or min) binary heap from a given array. Build Heap is used in Heap Sort as a first step for sorting.
2. Suppose we are sorting an array of eight integers using heapsort, and we have just finished some heapify (either maxheapify or minheapify) operations. The array now looks like this: 16 14 15 10 12 27 28 How many heapify operations have been performed on root of heap?
3. Consider a binary max-heap implemented using an array. Which one of the following array represents a binary max-heap?
   (A) 25,12,16,13,10,8,14
   (B) 25,12,16,13,10,8,14
   (C) 25,14,16,13,10,8,12
   (D) 25,14,12,13,10,8,16
4. A max-heap is a heap where the value of each parent is greater than or equal to the values of its children. Which of the following is a max-heap?



5. If we implement heap as maximum heap , adding a new node of value 15 to the left most node of right subtree . What value will be at leaf nodes of the right subtree of the heap?



6. If we implement heap as min-heap, deleting root node (value 1)from the heap. What would be the value of root node after second iteration if leaf node (value 100) is chosen to replace the root at start?



2.8 References

[1]. *Skiena, Steven (2008). "Searching and Sorting". The Algorithm Design Manual. Springer. p. 109. doi:10.1007/978-1-84800-070-4_4. ISBN 978-1-84800-069-8. [H]eapsort is nothing but an implementation of selection sort using the right data structure.*

[2] https://en.wikipedia.org/wiki/Heapsort

[3] https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture4.pdf

[4] **"Schaum's Outline of Data Structures with C++"**. By John R. Hubbard (Can be found in university Library)

[5] "Data Structures and Program Design", Robert L. Kruse, 3rd Edition, 1996.

[6] "Data structures, algorithms and performance", D. Wood, Addison-Wesley, 1993

[7] "Advanced Data Structures", Peter Brass, Cambridge University Press, 2008

[8] "Data Structures and Algorithm Analysis", Edition 3.2 (C++ Version), Clifford A. Shaffer, Virginia Tech, Blacksburg, VA 24061 January 2, 2012

[9] "C++ Data Structures", Nell Dale and David Teague, Jones and Bartlett Publishers, 2001.

[10] "Data Structures and Algorithms with Object-Oriented Design Patterns in C++", Bruno R. Preiss,