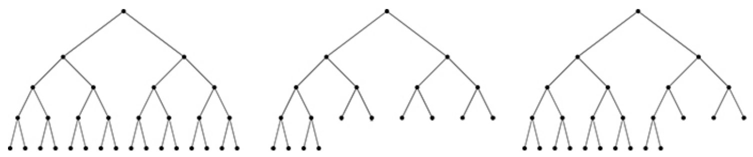
Heaps

In this lecture we study the tree-based structure known as a **heap**. This data structure is an exceptional tool that enables us to quickly access the maximum (or minimum) element in a set of data. We saw in our previous lecture how to select the maximum and minimum in *linear* time. Heaps perform much better because although they take $O(n)$ to first "process" the data, but after that they allow us to find the maximum in **constant** time. Furthermore, **removing** the maximum (and maintaining the heap structure) can be done in $O(\log n)$ time. The key to this fast speed is in the interesting structure maintained by heaps. They are designed in such a way that allows for easy access to the max. Furthermore, when the max is deleted, the heap structure is quickly updated so that the *next* call to the maximum can again be done in constant time.

Heaps are used in order to carry out some of the most fundamental algorithms in computer science, in particular **Dijkstra**'s shortest-path algorithm which we shall see later in this course. We shall also see in this lecture how we can use heaps to sort, simply by repetitively removing the maximum from the heap.

1 What is a heap?

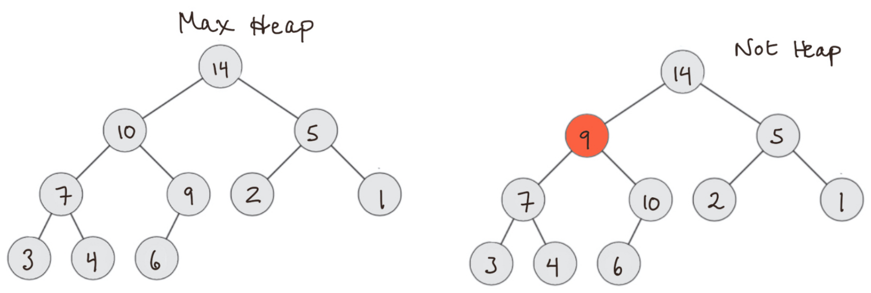
A **binary heap** is a data structure can be visualized as a **complete binary tree**. Such a binary tree is defined as having all levels that are full, except possibly the last level. Furthermore, the last level must be filled from left to right. Below we show three examples of complete binary trees:



In a heap structure, the values are stored in the **nodes** of the tree. A heap can be defined as either a **max-heap** or a **min-heap**. The max-heap has the following property:

Max-heap Property:
Every node in the tree has a value that is *greater than or equal to* all the values in its subtree.

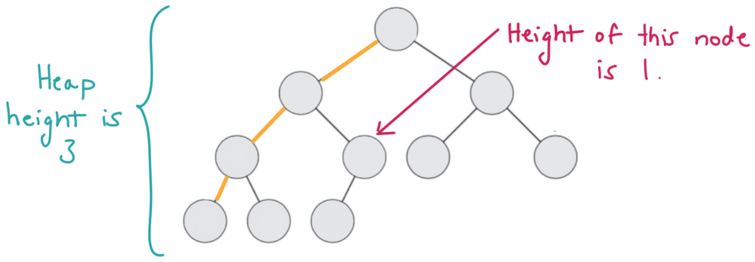
This property must be true for any node in the tree. For example, the figure on the left is a max-heap. Every node in that tree is greater than or equal to all nodes in its subtree. The figure on the right is not a max-heap - node 9 violates the heap property, since it contains a 10 in its subtree.



In lecture, we shall assume we are working with max-heaps. Min-heaps are defined symmetrically.

Since a heap is visualized as a tree, we can define its **height** just as we do for trees. We will not prove the formula for the height of a complete binary tree here, but the result below is essential in our analysis of operations on heaps.

Height of heap:
The height of a heap is the number of edges in the longest simple path from the root to a leaf. For a heap on n nodes, the height is $\lfloor \log_2 n \rfloor$ or $\Theta(\log n)$. The height of a particular *node* is the longest path from it to a leaf.



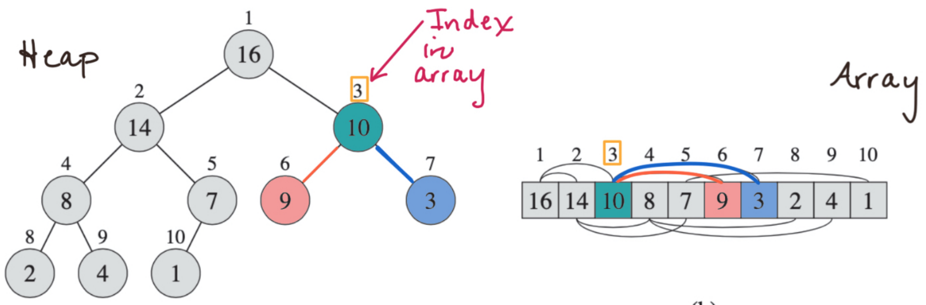
We can verify the height formula using the example above. Here we have 10 nodes, and $\lfloor \log_2 10 \rfloor = 3$, which is the height of the tree. The height of a heap is extremely important to our analysis in this lecture, since most operations run in time proportional to the height of the heap.

1.1 Heap implementation

Heaps can be implemented using arrays, where each node in the heap corresponds to an element in the array, A. The values of the nodes are placed at specific indices in A:

- The root is in position $A[1]$
- The parent of node i is $A[\lfloor i/2 \rfloor]$
- The left child of node i is $A[2i]$
- The right child of node i is $A[2i+1]$

The following example is taken from CLRS. The tree on the left shows both the internal values of the nodes, and the corresponding indices in the array.



(b)

The heap implementation requires an attribute *A.heapsize* which specifies how many elements of the array are actually part of the heap. In other words, it is possible that the heap is smaller than the array that is used to store the elements. In the above example, $A.heapsize = 10$. This attribute is updated as elements are deleted from the heap, or inserted into the heap. This will be relevant in our last section on *Heapsort*. We now look at how to build a heap from scratch, given a set of n distinct numbers.

2 How to build a heap

The algorithms to build the heap are not as efficient as the delete or insert operations. However, by taking the time to build the heap structure, we can later take advantage of the fast insert and delete operations that are possible because of its structure. We look at two methods in this section:

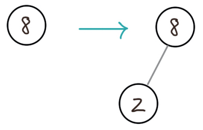
- Iterative method
- Bottom-up method

2.1 Iterative method: runs in time $O(n \log n)$

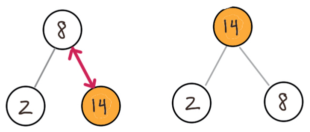
Suppose we are given a set of n elements and we wish to create a heap out of these elements. The iterative approach builds the heap by adding one element at a time. After each element is added as a *leaf*, the heap must be adjusted so the the heap property is maintained. We illustrated the process with the following elements :

$$\{8, 2, 14, 3, 9, 16, 7, 5\}$$

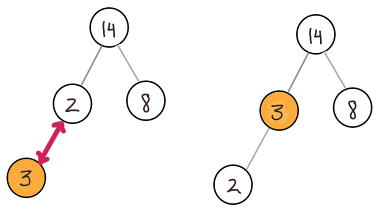
We begin by inserting the 8, which creates a heap of size one. The 2 is inserted next as a leaf. Note that we do not need to adjust any values: the heap property is maintained.



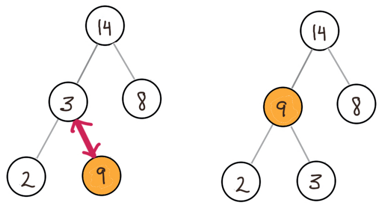
Next when the 14 is inserted as a leaf, we have to *swap* the 14 with its parent in order to maintain the heap property.



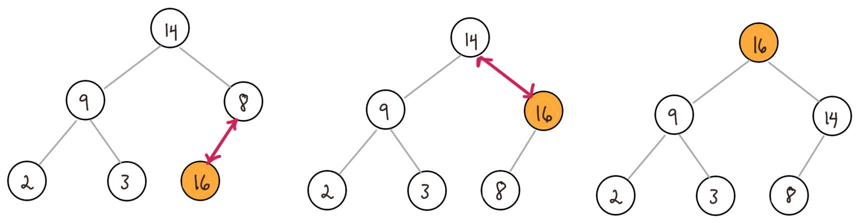
When the 3 is inserted, the same procedure is applied: we insert it as a leaf and then continuously swap with the parent until the heap property is maintained.



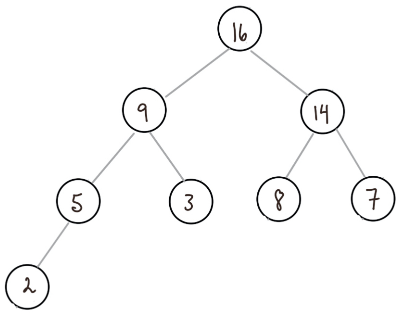
Next the 9 is inserted:



Then the 16 is inserted. Note that the swaps *continue* up the path towards the root until the heap property is maintained.



This process continues until all elements are inserted. The final heap is shown below:



This process of performing swaps *up the path towards the root* is referred to as *bubbling-up* the tree. Therefore the iterative heap-building method requires the use of a procedure called **Bubble-up(A,i)**, summarized below:

Bubble-up(A,i)

Input: Array A and an index i. The array contains a valid heap in the range 1 to $i-1$. The element at index i might be larger than it's parent, and therefore may violate the heap property.

Procedure: Bubble-up performs a series of swaps starting with the element at index i. The element is swapped with its parent, whenever it is larger than its parent. This continues all the way up to the root, if necessary.

Result: The heap property is maintained for all elements in the range 1 to i.

Runtime: Each step of Bubble-up runs in constant time (it simply performs a comparison and a swap). However, we may have to bubble each element *all the way up to the root*. Therefore the runtime of Bubble-up depends on the height of the heap. We have seen that the height of the heap is $O(\log n)$. Thus a bubble-up procedure runs in time $O(\log n)$ in the worst case.

The iterative heap building method simply calls the above Bubble-up procedure one element at a time. The pseudo-code below builds a heap using the input array A.

```
BuildHeap($A[1,\ldots,n]$)
    A.heapsize = 2
    for $i = 2$ to $n$
        Bubble-up(A,i)
        A.heapsize++
```

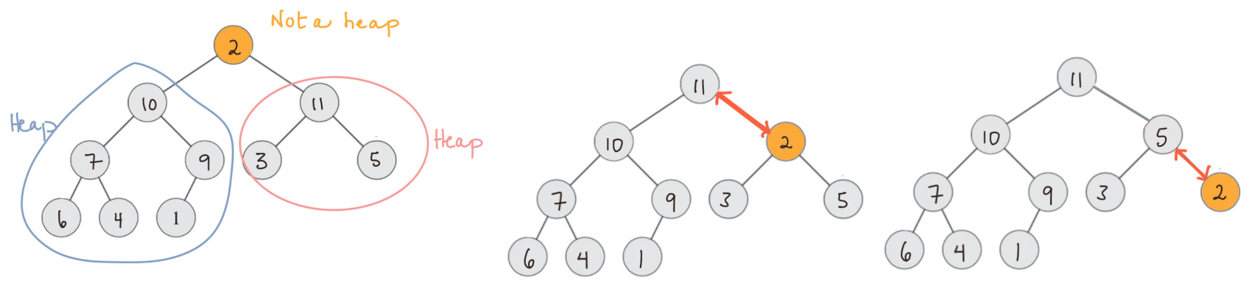
What is the runtime of this heap-building algorithm? Recall that a bubble-up procedure runs in time $O(\log n)$ in the worst case. So each element can be inserted into the heap in time $O(\log n)$. Since there are n elements to be inserted overall, the total runtime of the iterative heap-building algorithm is $O(n \log n)$ in the worst case. This seems rather inefficient, since we could actually have sorted all the elements in this time! In the next section we look at a much faster way of building a heap.

2.2 Bottom-up method: runs in time $O(n)$

The second method works by restructuring the elements in a tree from the bottom up, until the heap property is maintained. This algorithm relies on a procedure called **bubble-down** which is similar to bubble-up from the previous section. We start by looking the bubble-down procedure, which takes as input an index, i

Bubble-down:

Suppose the node at index i is such that each of its children is in fact a proper max-heap. Assume that node i may or may not satisfy the heap property, as shown on the left below (node i is yellow). We can correct this easily, by swapping the node at i with its **largest** child. We repeat this *down the tree* until the heap property is maintained.



The summary of the Bubble-down procedure is given below:

Bubble-down(A,i)

Input: Array A and an index i. The left and right children of the node at index i are valid heaps. However, the element at index i may be larger than its children, which means the subarray $A[i, \ldots, n]$ is not a valid heap.

Procedure: Bubble-down performs a series of swaps starting with the element at index i. The element is swapped with its larger child, whenever there is a child that is larger than the parent. This continues all the way down the heap, if necessary

Result: The heap property is maintained for all elements in the array from index i to n.

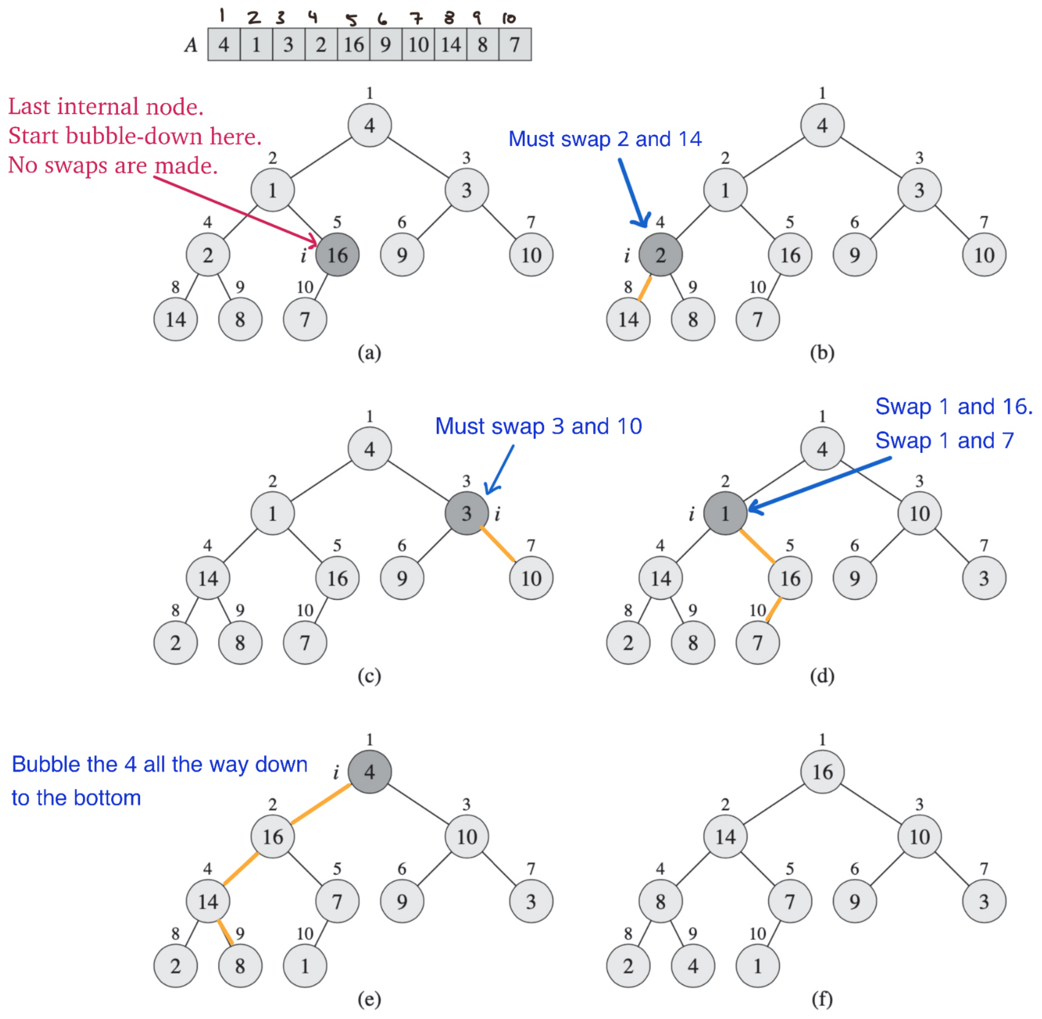
Runtime: Each step of Bubble-down runs in constant time (it simply performs a comparison and a swap). However, we may have to bubble each element *all the way down to a leaf.* Therefore the runtime of Bubble-down depends on the height of the element at index i. If node i is at height h, then the number of swaps performed is $O(h)$. We have seen that the height of the heap is $O(\log n)$. In the worst-case then Bubble-down procedure runs in time $O(\log n)$.

Build Heap:

We can use the bubble-down procedure to build our heap from the bottom up, starting with the last *internal node* (a node which is not a leaf). How can we identify this node in the array? If the last node of the tree has index n, then its parent node has index $\lfloor n/2 \rfloor$. Thus the index of the last internal node is exactly $\lfloor n/2 \rfloor$.

```
BottomBuildHeap($A[1, \ldots, n]$)
    A.heapsize = n
    for $j = \lfloor n/2 \rfloor$ downto 1
        Bubble-down(A,i)
```

The following example is taken from CLRS, with some extra notation:



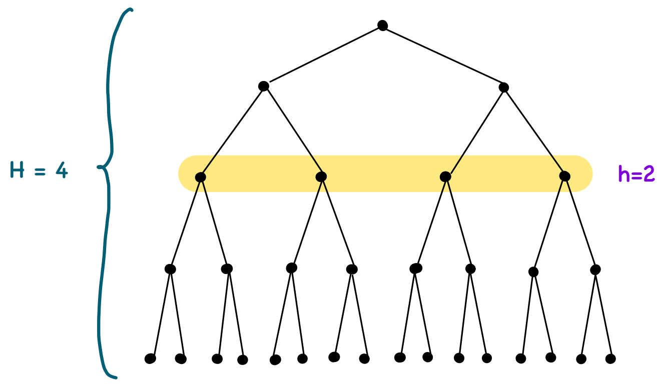
What is the runtime of the above algorithm? Recall that the number of steps carried about by the bubble-down procedure at most the height of the *node*. Since the maximum height of any node is $O(\log n)$ and there are n nodes in the loop of our build-heap procedure, we could conclude that the algorithm carries out $O(n \log n)$ steps. This is indeed a correct bound, but in fact it is not the tightest one. If we are careful with our analysis, we can actually show that the runtime is $O(n)$. This is based on the fact that the *majority* of the nodes are at the lower half of the tree. So many of those nodes will not have to bubble very far down the tree, and we can take advantage of that fact in our analysis.

We will piece together the following facts in order to determine the runtime. Let H be the height of the tree, and n the number of nodes.

- The number of nodes which have height h in a complete binary tree is at most 2^{H-h}, where H is the height of the tree.

$$2^{H-h} = \frac{2^H}{2^h} \leq \frac{n}{2^h}$$

  In the example below, $n = 31$ and $H = 4$. The nodes at height 2 are those highlighted in yellow. How many of them are there? According to the above formula, at *most* $\frac{n}{2^h} = \frac{31}{4} = 7.75$. In fact, there are only four of them, but the formula is an upper bound.



- For each node at height h, the bubble-down operation takes time $O(h)$, and so by the definition of big-oh, we assume this is at most ch. Since there are at most $n/2^h$ nodes at height h, the *total* time for the bubble-down procedure for *all nodes* at height h is at most :

$$ch \cdot \frac{n}{2^h}$$

- The total cost of building the heap is the total cost of *all* the bubble-down operations, over all nodes at every possible height, from those at $h = 1$ to H. Then the total run time $T(n)$ of build-heap is bounded above by:

$$\begin{aligned} T(n) &\leq \sum_{h=1}^{H} ch \cdot \frac{n}{2^h} \\ &\leq \sum_{h=1}^{\infty} ch \cdot \frac{n}{2^h} \\ &= cn \sum_{h=1}^{\infty} h \left(\frac{1}{2}\right)^h \\ &= cn(2) \end{aligned}$$

The last line above replaces the infinite sum with its value, 2 (see class worksheet). Since the final term is simply a constant multiplied by n, then $T(n)$ is $O(n)$. Thus this second method of building a heap is much faster than our first iterative method. Now that we can efficiently build heaps, we look next at how they can be used to sort our numbers.

3 HeapSort

The heap structure is extremely powerful in that it allows as to *efficiently update the heap* when the *maximum is removed*. We look first at how this is done, before moving onto the heapsort algorithm.

3.1 Deleting the max of a Heap

If array A satisfies the max-heap property, the max is at position $A[1]$. We will delete the max in a rather interesting way: we simply *swap* it with the very *last* element in our heap. Then we fix the heap by using the **bubble-down** procedure seen above. The procedure for deleting the max is given below. Note that since there is only one call to the bubble-down procedure, the algorithm to delete the maximum from a heap runs in time $O(\log n)$.

```
DeleteMax($A$)
    k = A.heapsize
    Swap $A[1]$ and $A[k]$
    Bubble-down(A,1)
    A.heapsize −−
```

After one execution of Delete-Max, the A.heapsize has decreased, and the deleted maximum element is now in the *last* position of the array. This illustrates what we mentioned at the beginning of this lecture: there may be elements in the array that have been removed and are technically no longer part of the heap. However, we can use this fact to our advantage when we carry out the Heapsort algorithm.

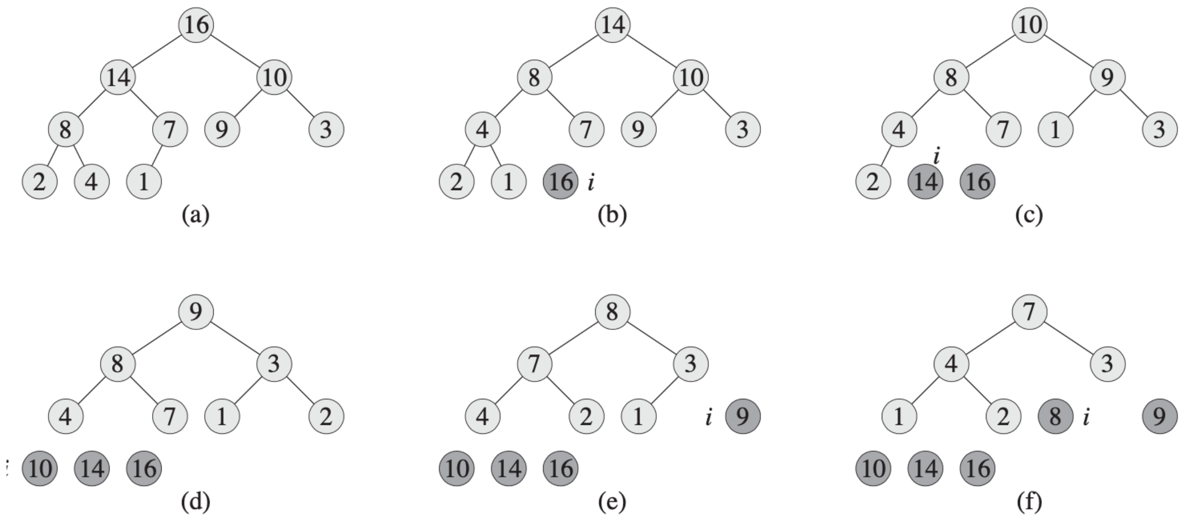
3.2 HeapSort Procedure

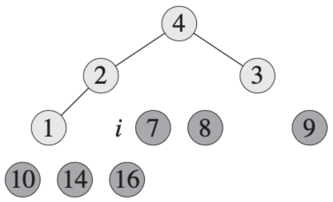
In order to sort the elements in A, we simply build a heap, and then repeatedly remove the maximum element. What is wonderful about the Delete-Max operation, is that is already places the removed maximum in the last position of the heap. This means that when all elements have been removed from the heap, the array will in fact contain the elements in sorted order.

```
HeapSort($A$)
    BottomBuildHeap(A)
    for $i = n$ downto 2
        Delete-Max(A)
```

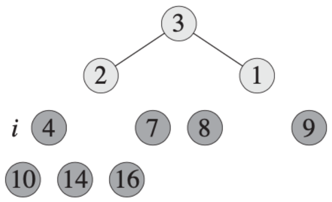
What is the complexity of this algorithm? Each call to Delete-max runs in time $O(\log n)$, since the procedure bubbles down from the root. There are $O(n)$ calls to Delete-Max in total, meaning that the overall Heapsort algorithm runs in time $O(n \log n)$.

Below is an example from CLRS showing the running of Heapsort after the initial heap has been built.

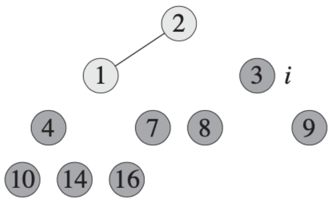



4
2
3
1
i
7
8
9
10
14
16

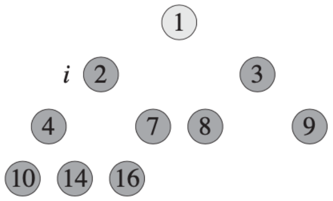
(g)


3
2
1
i
4
7
8
9
10
14
16

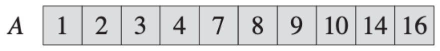
(h)


2
1
3
i
4
7
8
9
10
14
16

(i)


1
i
2
3
4
7
8
9
10
14
16

(j)


A
1
2
3
4
7
8
9
10
14
16

(k)