What is a Heap Data Structure?

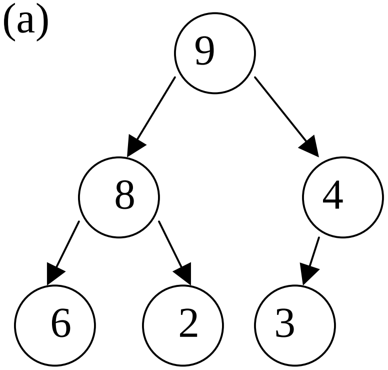
- A Heap data structure is a binary tree with the following properties :

1. It is a ***complete binary tree***; that is, each level of the tree is completely filled, except possibly the bottom level. At this level, it is filled from left to right.

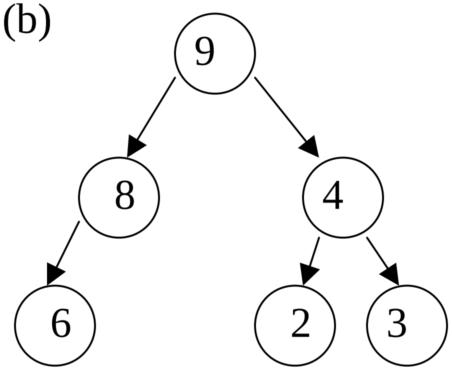
2. It satisfies the heap-order property: The data item stored in each node is greater than or equal to the data items stored in its children.

- Examples:

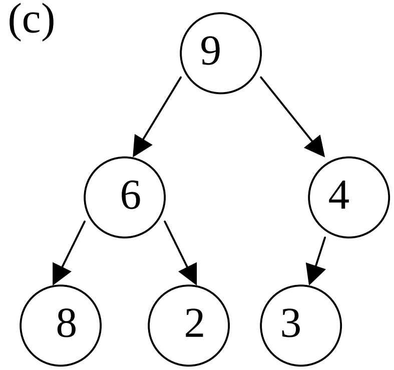
(a)



(b)



(c)

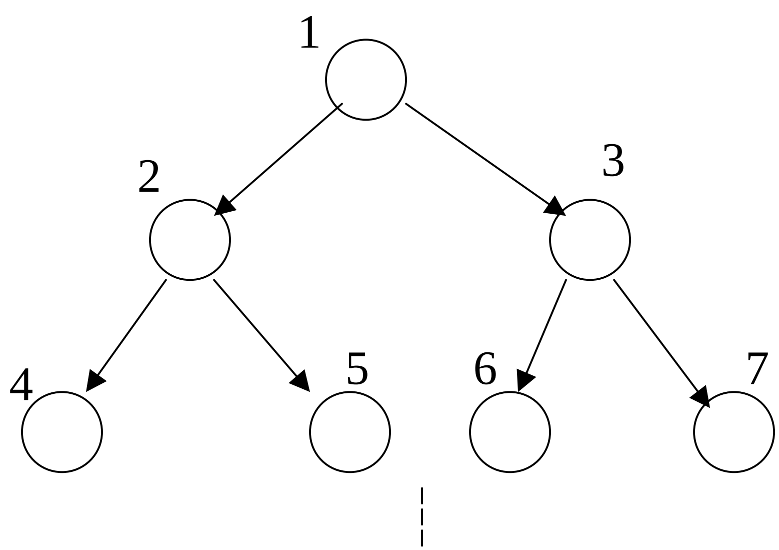


- In the above examples **<u>only (a) is a heap</u>**. (b) is not a heap as it is not complete and (c) is complete but does not satisfy the second property defined for heaps.

Heap as an Array

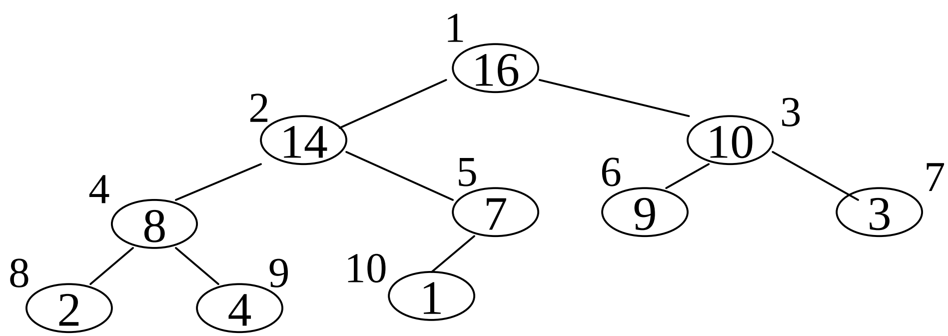
- We could implement heaps using a linked list like structure, like we did with binary trees but in this instance it is actually easier to implement heaps using arrays.

- We simply number the nodes in the heap from top to bottom, numbering the nodes on each level from left to right and store the ith node in the ith location of the array. (of course remembering that array indexing in Java starts at zero).



Heap as an Array (2)

- An array A that represents a heap is an array with two attributes
  - ***length***, the number of elements in the array
  - ***heap-size***, the number of heap elements stored in the array
- Viewed as a binary tree and as an array :



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

• The root of the tree is stored at A[0], its left-child at A[1], its right child at A[2] etc.

Accessing the Heap Values

- Given the index *i* of a node, the indices of its parent *Parent(i)*, left-child *LeftChild(i)* and right child *RightChild(i)* can be computed simply :

```
Parent(i)
   return i/2
```

```
LeftChild(i)
   return 2i
```

```
RightChild(i)
   return 2i+1
```

- Heaps must also satisfy the **heap property** for every node, *i*, other than the root.

  $$A[\text{Parent}(i)] \geq A[i]$$

- Therefore, the largest element in a heap is stored at the root, and the subtrees rooted at a node contain smaller values than does the node itself.

Heap Operations

- The **height** of a node in a tree is the number of edges on the longest simple downward path from the node to a leaf. (i.e. maximum depth from that node – Remember?)

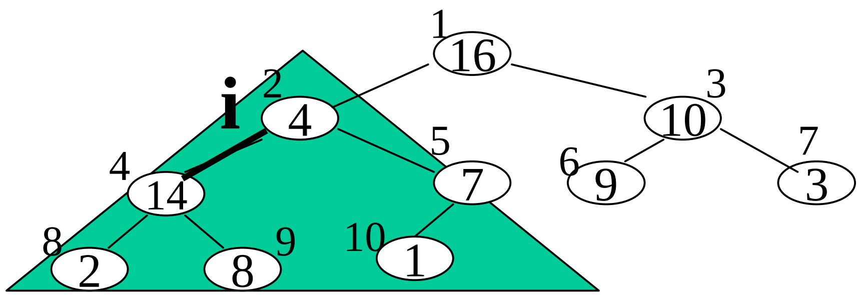
- The height of an n-element heap based on a binary tree is lg n

- The basic operations on heaps run in time at most proportional to the height of the tree and thus take O(lg n) time.

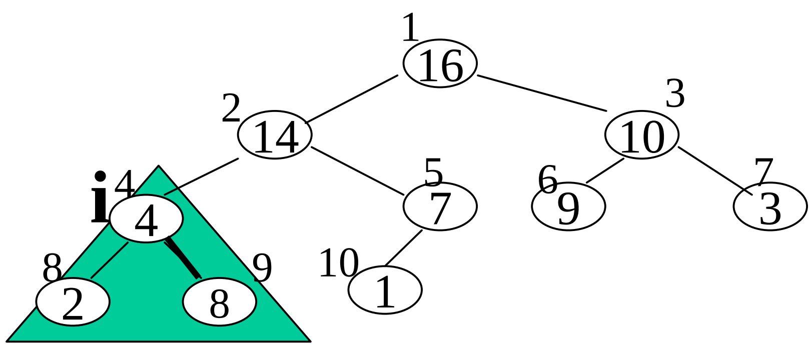
Maintaining the Heap Property

- One of the more basic heap operations is converting a complete binary tree to a heap.
- Such an operation is called "Heapify".
- 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 LeftChild(i) and RightChild(i) are heaps, but that A[i] may be smaller than its children, thus violating the 2nd 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.
- The action required from Heapify is as follows:

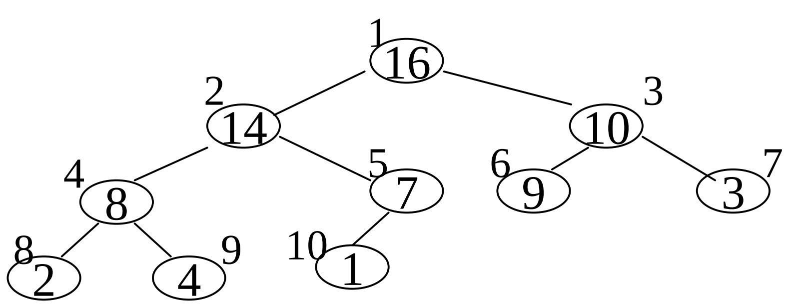
Example of Heapify



- Subtree to be heapified is marked by the shaded triangle.
- Node **i** if smaller than its children is swapped with the greater of its two children – in this case the left child with value 14 is swapped for value 4 in node **I**.



- If the next subtree down (again marked with the shaded triangle region) is not a heap, we "percolate" down to the next subtree level and find the larger of the two children again to swap with current root node.



- We keep "percolating" down until either the subtree conforms to the heap properties or a leaf has been reached.

The Heapify Algorithm

```
Begin
Heapify(A[], i)          // Heapify takes in our heap and index of current root node of subtree to be heapified
{
    left = LeftChild(i);                         // index of left child
    right = RightChild(i);                       // index of right child

    if left ≤ A.heap-size AND A[left] > A[i]      {    // if still within bounds AND left child
            largest = left;                             // greather than parent – remember largest as left child
    }
    else   {
         largest = i;                                   // else parent still has largest value for now
    }

    if right ≤ A.heap-size AND A[right] > A[largest] {   // if still within bounds  AND right child
         largest = right                                // greather than parent – remember largest as right child
    }
    if largest  NOT EQUAL i   {                  // if parent does not hold largest value
         swap A[i] and A[largest]                // swap parent with child that does have largest value
         Heapify(A, largest)                     // Percolate down and Heapify next subtree
    }
}
End
```

The Heapify Algorithm

- At each step, the index of the largest of the elements A[*i*], A[Left(*i*)], and A[Right(*i*)] is stored in the variable largest.
- If A[*i*] is largest, then the subtree rooted at node *i* is a heap and the procedure ends.
- Otherwise, one of the two children has the largest element, and A[*i*] is swapped with A[largest], which causes node *i* and its children to satisfy the heap property.
- The node largest, however, now has the original value A[*i*], and thus the subtree rooted at largest may violate the heap property.
- Therefore, Heapify must be called recursively on that subtree.

Analysis of Heapify

- The running time of Heapify on a subtree of size n rooted at a given node i is the O(1) time to fix up the relationships among the elements A[i], A[Left(i)] and A[Right(i)], plus the time to run Heapify on a subtree rooted at one of the children of node i.

- The subtrees can have size of at most $2n/3$, and the running time of Heapify can therefore be described by the recurrence,

$$T(n) \leq T(2n/3) + O(1)$$

- Using either the iteration method or the master's theorem we can find that the solution to this works out as:

$$T(n) = O(\log n)$$

Building a Heap

- For the general case of converting a complete binary tree to a heap, we begin at the last node that is not a leaf, apply the "percolate down" routine to convert the subtree rooted at this current root node to a heap.

- We then move onto the preceding node and "percolate down" that subtree.

- We continue on in this manner, working up the tree until we reach the root of the given tree.

- We use the Heapify procedure here in a bottom up manner. This means also means that to convert an array A[1..n], where n = A.length, into a heap we can apply the same procedure as described above.

- Now, since the subarray A[($\lfloor n/2 \rfloor$ + 1)..n] are all leaves of the tree, each is a 1-element heap.

- The procedure "Build-Heap" goes through the remaining nodes and runs Heapify on each.

- The order of processing guarantees that the subtrees rooted at children of node i are heaps before Heapify is run at that node.

Build Heap Algorithm

```
Begin:
Build-Heap (A[])      {                          // takes in an array to be heapified
    A.heap-size = A.length;                      // heap size is number of elements in the array
    for i = ⌊A.length/2⌋ downto 1 // starting at first subtree up to root full tree
    {
        Heapify(A, i);                 // Heapify the current subtree
    }
}
End
```

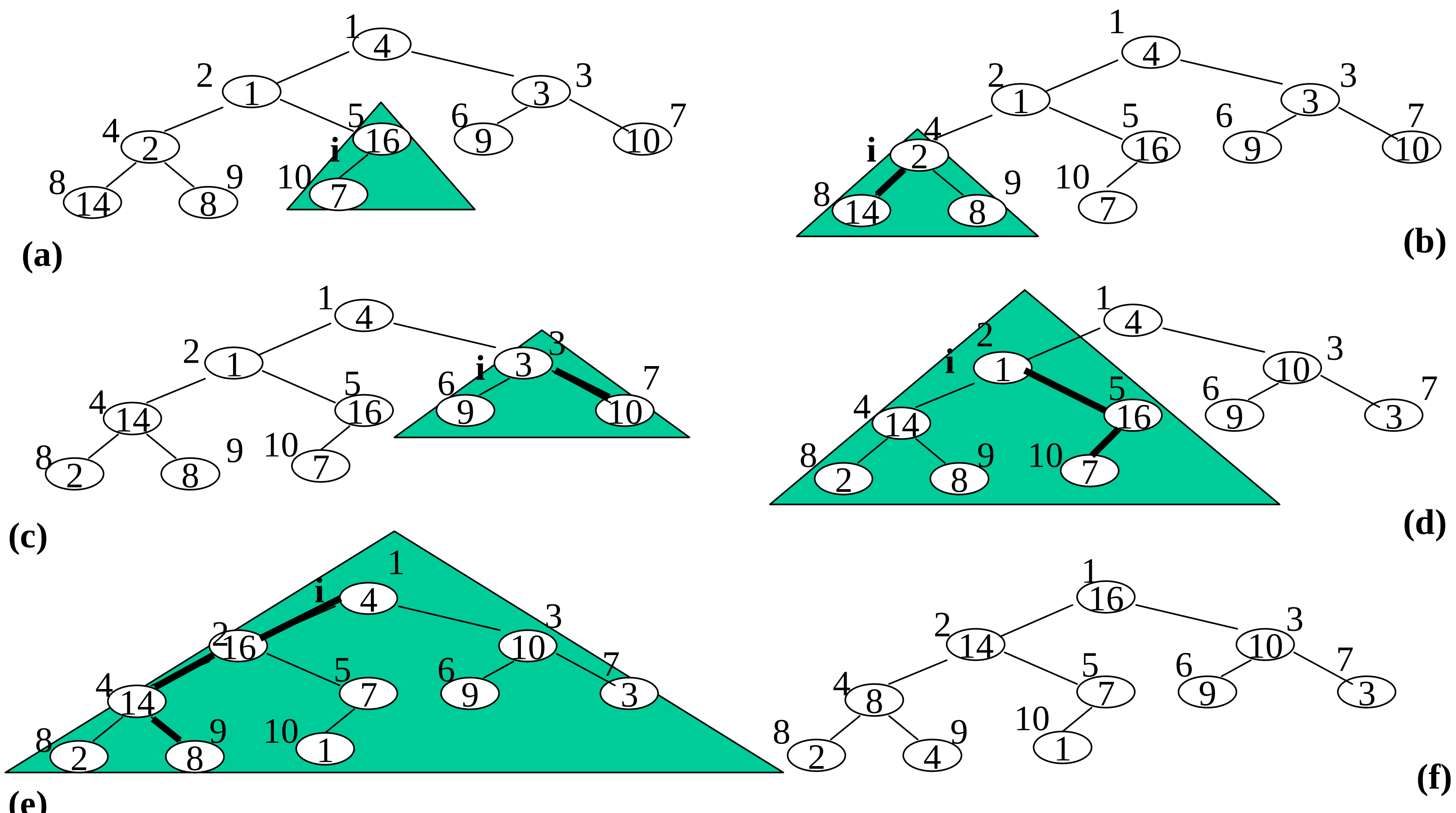
- Each call to Heapify costs O(lg n) time, and there are O(n) such calls.
- Thus, the running time is at most O(n lg n)
- A more complex analysis, however, gives us a tighter upper bound of O(n).
- Hence, we can build a heap from an unordered array in linear time.

- Lets have a look at a visual example of this operation (have a look at the next slide).
- The current subtrees that are being worked on are marked inside a shaded triangle.
- Any swapping that goes on is marked with a thicker edge.

Example of Build Heap

A

4	1	3	2	16	9	10	14	8	7



Heap Practical Example :Priority Queue

- Many applications require that we process records with keys/priorities in order, but not necessarily in full sorted order. (e.g. CPU process scheduling).

- Items in a priority queue are not processed strictly in order of entry into the queue

- Items can be placed on the queue at any time, but processing always takes the item with the largest key/ priority

- The priority queue is a generalised queue Abstract Data Structure (ADT)

- In fact, the priority queue is a proper generalisation of the stact and the queue ADTs, as we can implement either with priority queues, using appropriate priorities

Priority Queue - Definition

- A priority queue is a data structure of items with keys that support two basic operations : insert a new item, and delete the item with the largest key.

6	5	5	3	2	1			
6	6	5	5	3	2	1		Add item with priority 6
6	6	5	5	4	3	2	1	Add item with priority 4
6	5	5	4	3	2	1		Remove item with highest priority

- Applications of priority queues include:
  - Simulation Systems, where events need to be processed chronologically.
  - Job scheduling in computer systems.

- The Heap Data Structure is an efficient structure for implementing a simple priority queue.

- In practice, priority queues are more complex than the simple definition above.

Priority Queue

- One of the reasons that priority queue implementations are so useful is their flexibility

- For that reason any implementation will need to support some of the following operations
  - *Construct* a priority queue from N given items
  - *Insert* a new item
  - *Delete the maximum* item
  - *Change the priority* of an arbitrary specified item
  - *Delete* an arbitrary specified item

- We take “maximum” to mean “any record with the largest key value”.

Other Heap Operations

- Other important heap operations include :
  - Heap Insert   (inserting an element into a heap)
  - Heap Extract Max/Min (extracting an Max/Min element from a heap)
  - Heap Delete (remove an element from a heap)
  - And of course…. HeapSort - O(N lg N) sorting algorithm
    + very basic idea :
      * swap and remove root value with final position element value
      * decrement size of heap and Heapify from position one (i.e. new root) again to restore new heap
      * Keep doing this while there is more nodes. This eventually gives us a sorted list.

- Unfortunately, due to time constraints, we cannot examine these operations in any more detail. ☹