HEAPS, PRIORITY QUEUES AND HEAP SORT

After studying this chapter, you should be able to

- Describe a priority queue at the logical level and to implement a priority queue as a list
- Show how a binary tree can be represented in an array, with implicit positional links between the elements
- Define the terms full binary tree and complete binary tree
- Describe the shape and order properties of a heap, and implement a heap using a tree represented by implicit links in an array
- Implement a priority queue as a heap
- Compare the implementations of a priority queue using a heap, linked list, and binary search tree
- Sort a set of values using a heap

Topics

- Array representation of Binary tree
- Complete Binary Tree
- Heap
- Insert and Delete
- Heap Sort
- Heapify
- Priority queue

ADT Priority Queue

- **Priority Queue:** An ADT in which only the highest priority element can be accessed

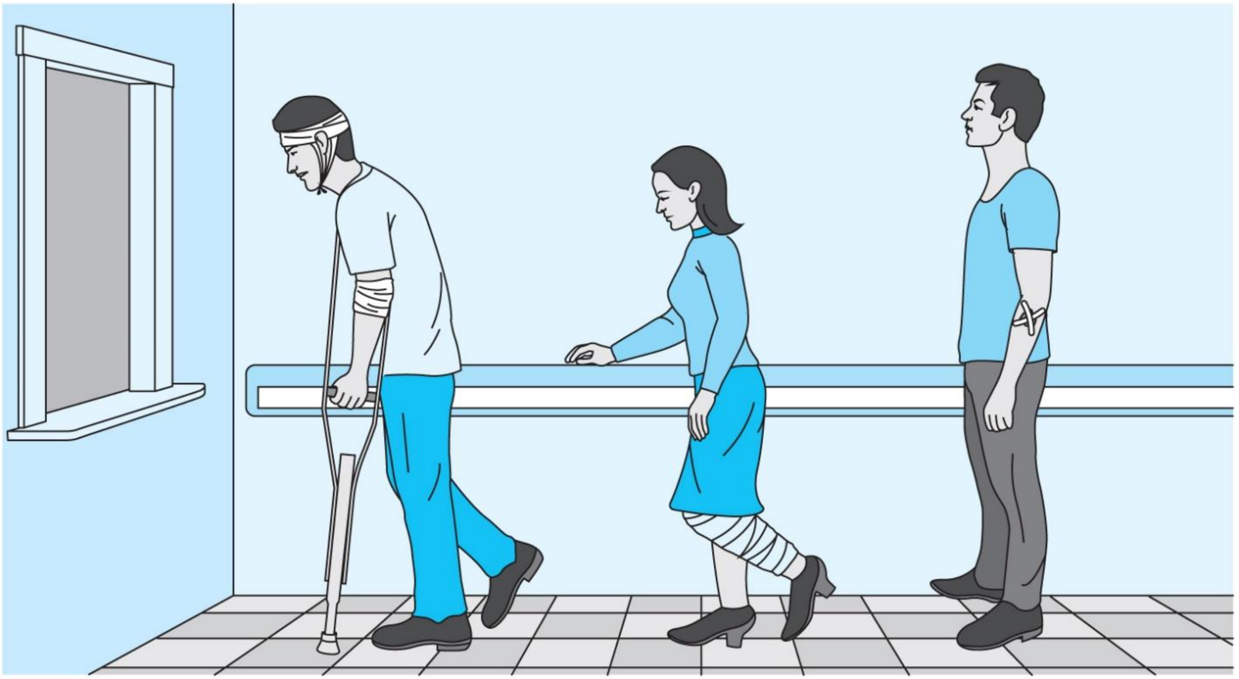

Figure 9.1 Real-life priority queue

Priority Queue: Logical Level

- The operations on a priority queue are almost identical to those of a regular queue
- Dequeue returns the highest-priority item instead of the oldest item
- Transformers: Enqueue, Dequeue, MakeEmpty
- Observers: IsFull, IsEmpty

Priority Queue: Application Level

- Emergency rooms triage patients so that the most urgent cases are handled first
- Graduating seniors are given first choice when registering classes
- Can also be used in sorting

Priority Queue: Implementation

- Unsorted List: Enqueue would be easy with an unsorted list; simply insert it at the end of the list. Dequeuing would required searching for the highest-priority item
- Array-based Sorted List: Enqueuing would be O(*N*). We have to find the place to enqueue the element.
- Linked Sorted List: Enqueuing is O(*N*) because finding the insertion point is a linear search
- Binary Search Tree: O(log*N*) Enqueue and Dequeue – winner!

BST Drawbacks

- BST efficiency is based on the tree's height
- The nature of Enqueue and Dequeue would lead to tall, "narrow," and inefficient trees
- Priority Queues only care about the highest-priority node; by relaxing the binary search property, we turn the tree into a ***heap*** that still provides O(log*N*) operations

Array-Based Binary Search Tree

- BSTs can be stored in arrays; the relationships between nodes becomes an implicit property of the algorithms
- The node at index X has its left child at
  - (X*2) + 1 and its right child at (X*2) + 2
- The parent of a node is at (X-1)/2

Array-Based BST

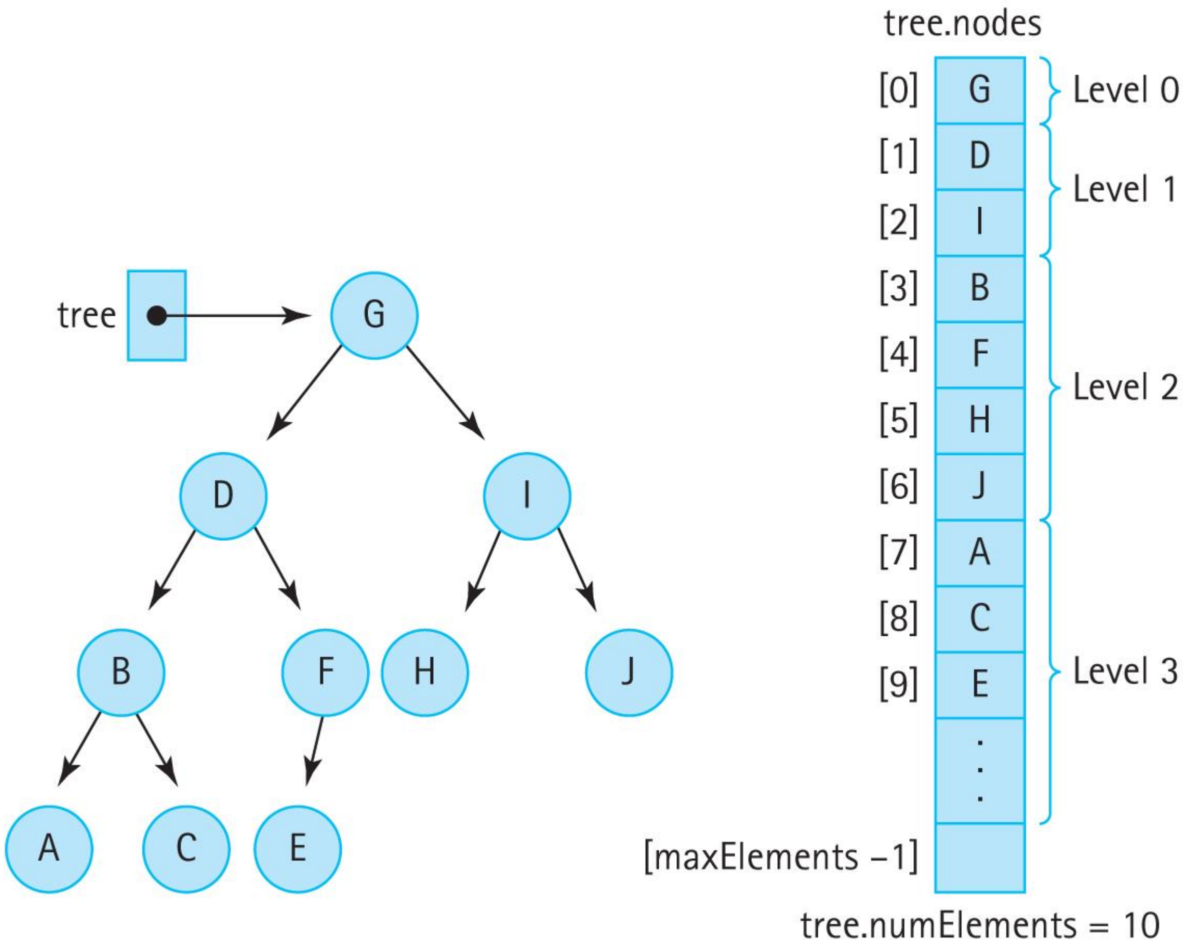

Figure 9.2 A binary tree and its array representation

For any node `tree.nodes[index]`, its left child is in `tree.nodes[index*2+1]`

Example: finding a left child of node F at index 4
`tree.nodes[4]`
`tree.nodes[4 * 2 + 1] = tree.nodes[9]`
Therefore, the left child of F (index 4) is E (index 9)

For any node `tree.nodes[index]`, its right child is in `tree.nodes[index*2+2]`

Example: finding a right child of node B at index 3
`tree.nodes[3]`
`tree.nodes[3 * 2 + 2] = tree.nodes[8]`
Therefore, the right child of B (index 3) is C (index 8)

Full and Complete Trees

- **Full Binary Tree:** All leaves are located on the same level and all non-leaf nodes have two children
- **Complete Binary Tree:** The tree is full, or full to the last level, with the leaves on the last level located as far left as possible
- Array-based binary trees must be full or complete, because the elements occupy contiguous array slots. If a tree is not full or complete, we must account for the gaps created by missing nodes. In order to maintain the proper parent-child relationship, it must be filled a dummy value in those gaps.

Full and Complete Trees (cont.)

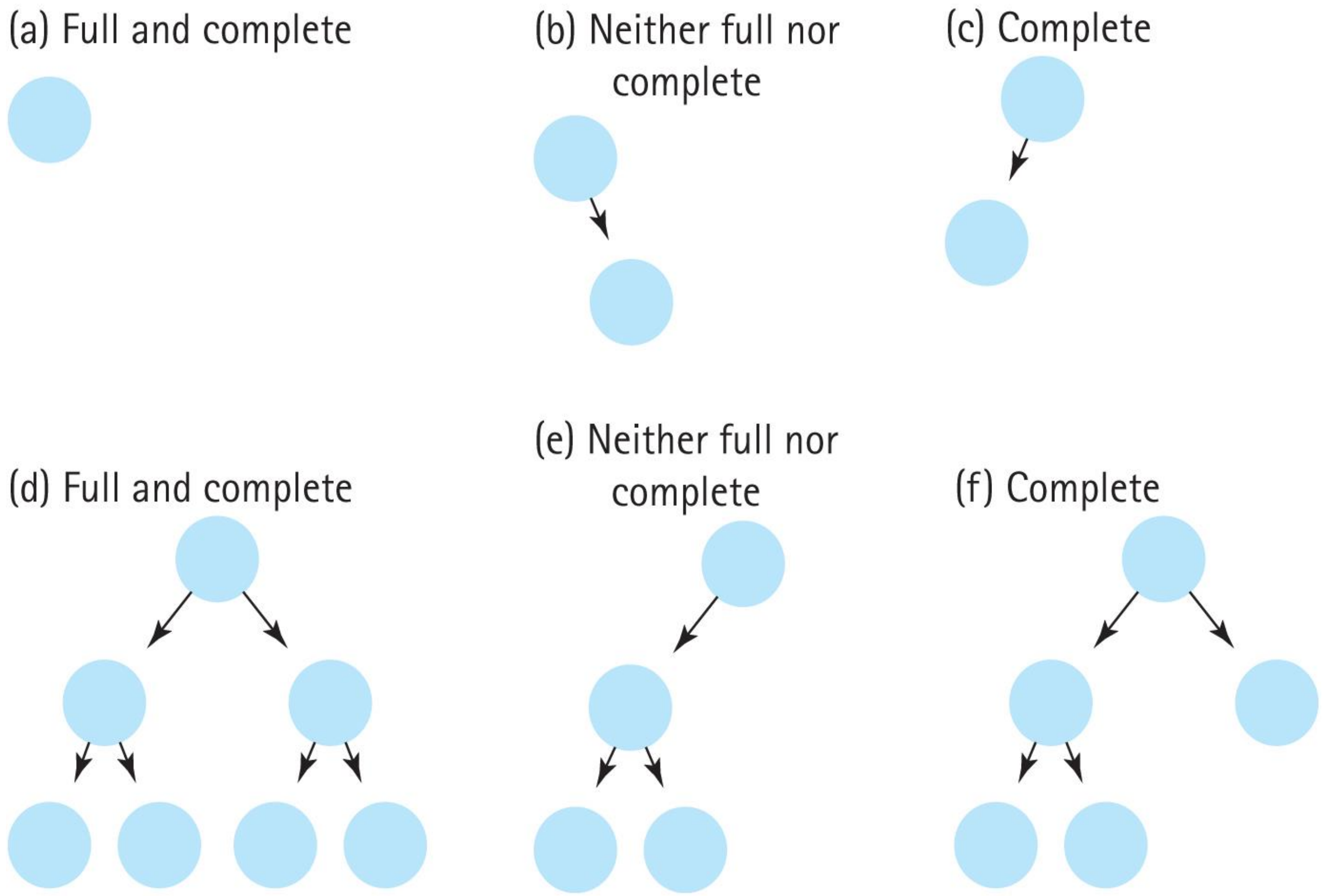

Figure 9.3 Examples of binary trees (a) Full and complete (b) Neither full nor complete (c) Complete (d) Full and complete (e) Neither full nor complete (f) Complete

Using Dummy Values

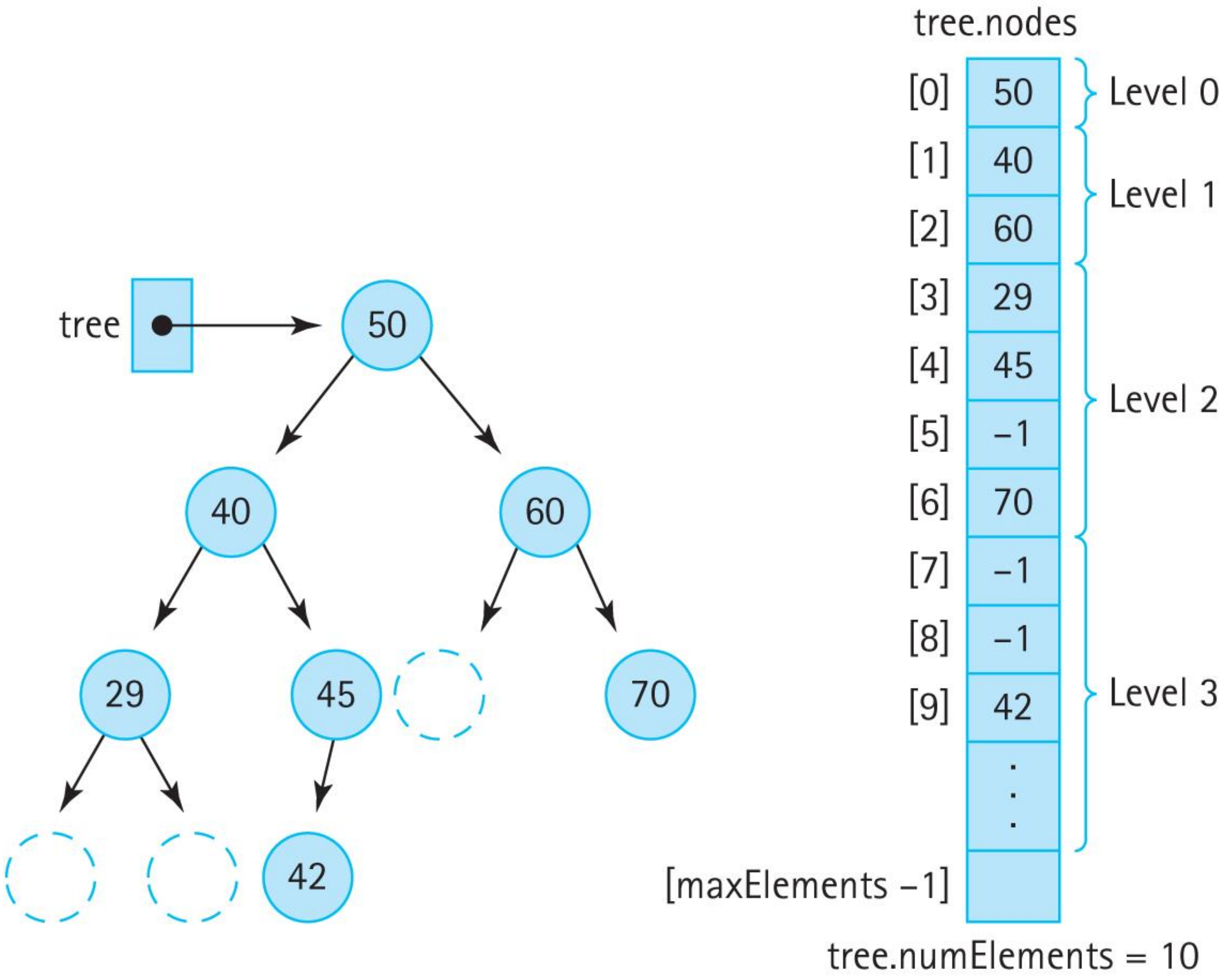

Figure 9.4 A binary search tree stored in an array with dummy values

Heaps: Logical Level

- **Heap:** A binary tree that fulfills two properties:
  - **Shape Property:** The heap is a complete tree
  - **Order Property:** The value of each node in the tree is greater than or equal to the values of its children (max-heap) or the value of each node in the tree is smaller than or equal to the values of its children (min-heap)
- Max-heap: The root node has the highest value
- Min-heap: The parents’ values are less than their children’s values
- Not to be confused with the memory heap

Heaps

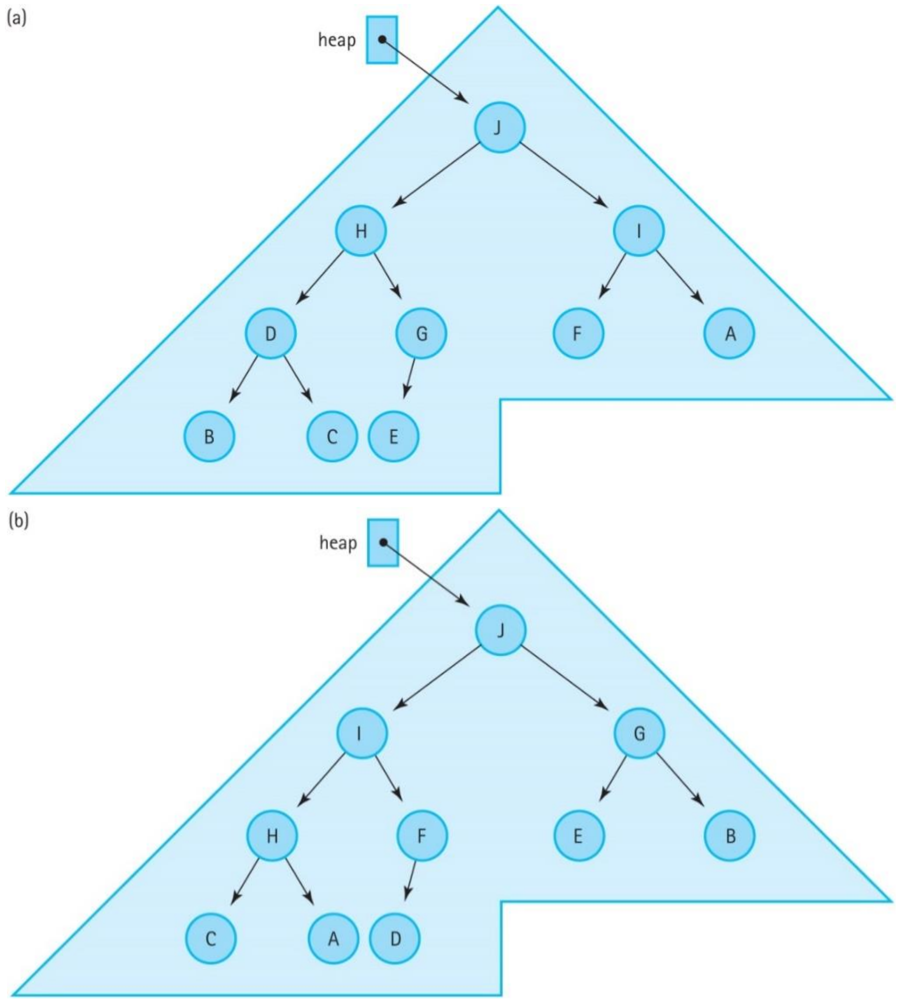

Figure 9.5 Two heaps containing the letters "A" through "J"

The placement of the values differs in two trees, but the **shape property** remains the same: a complete binary tree of ten elements.
A group of values can be stored in a binary tree in many ways and still satisfy the **order** property of heaps.

Basic Heap Operations

- **ReheapDown (heapify):** Given a heap that fulfills the order property except for the root, move the root node down until it is in a position that fulfills the order property.
- **ReheapUp:** Given a heap that fulfills the heap property except for the very last node, move the last node up the heap until the order property is restored

- *Demonstration of ReheapDown and ReheapUP are posted along with this chapter.*

Heaps: Application Level

- Heaps are primarily used to implement other structures and ADTs, such as priority queues

Heaps: Implementation Level

- Because heaps are always complete, an array-based implementation is used.
- Heaps are rarely used alone; it makes sense to write the HeapType as a struct so that all members are public

HeapType

```
template<class ItemType>
// Assumes ItemType is either a built-in simple type
// or a class with overloaded relational operators.
struct HeapType
{
  void ReheapDown(int root, int bottom);
  void ReheapUp(int root, int bottom);
  ItemType* elements;
  int numElements;
};
```

Implementing ReheapDown

- One of the root's two children is the maximum value in the heap, due to the order property
- By swapping the root and this child, the root is again the root of a heap that needs to be reshaped by ReheapDown
- This continues recursively until the root is a leaf node or is in the proper place

ReheapDown Algorithm

- If the root is a leaf node, do nothing
- Find the maximum of the root’s children
- If the root is less than the max child, swap the two nodes and recurse on the max child’s index, which now contains the root’s value

Determining Leaf Nodes

- How do we check that a node is a leaf node?
- Every non-leaf node must at least have a left child due to the shape property
- The index of a node’s left child is index*2 + 1
- If the index of a node’s left child is greater than the index of the last node in the tree, the node is a leaf

Implementing ReheapUp

- If the node is the root of the heap, do nothing
- If the node is greater than its parent, swap the node and its parent
- Repeat the previous step recursively until the node is the root of the heap or it is less than or equal to its parent
- Index of parent = (index of node - 1) / 2

Heaps and Priority Queues

- Dequeue returns the highest priority item, which is the root of the heap
- This leaves a hole at the top of the heap; like with UnsortedType, this hole can be filled with the bottom element of the heap
- But now the order property may be violated by the root
- Call ReheapDown to fix it

Heaps and Priority Queues (cont.)

- Enqueue puts an element in the appropriate place in the queue by priority. How?
- Start by putting it as the bottom element, thus preserving the shape property.
- Now the bottom element of the heap may be violating the order property. Fix it using ReheapUp.

Heaps vs. Other Implementations

- ReheapUp and ReheapDown are O(log*N*), so Enqeueu and Dequeue are O(log*N*)

Table 9.1 Comparison of Priority Queue Implementations

	Enqueue	Dequeue
Heap	$O(\log_2 N)$	$O(\log_2 N)$
Linked list	$O(N)$	$O(1)$
Binary search tree		
Balanced	$O(\log_2 N)$	$O(\log_2 N)$
Skewed	$O(N)$	$O(N)$

Table 9.1 Comparison of Priority Queue Implementations

Heap Sort

- A simple sort algorithm is to search for the highest value, insert it at the last position, repeat for the next highest value, and so on
- Searching for the next highest value in the list makes this **selection sort** inefficient
- By using a heap, there is no need to search for the next largest element
- Remove the root, ReheapDown, repeat

Building a Heap

- The unsorted list needs to be turned into a heap
- It already satisfies the shape property: there are no holes in the array
- ReheapUp and ReheapDown can reshape a heap, but only if their preconditions are met
- Does the array meet the preconditions?

Example Array

Here’s an unsorted array and the tree it forms:

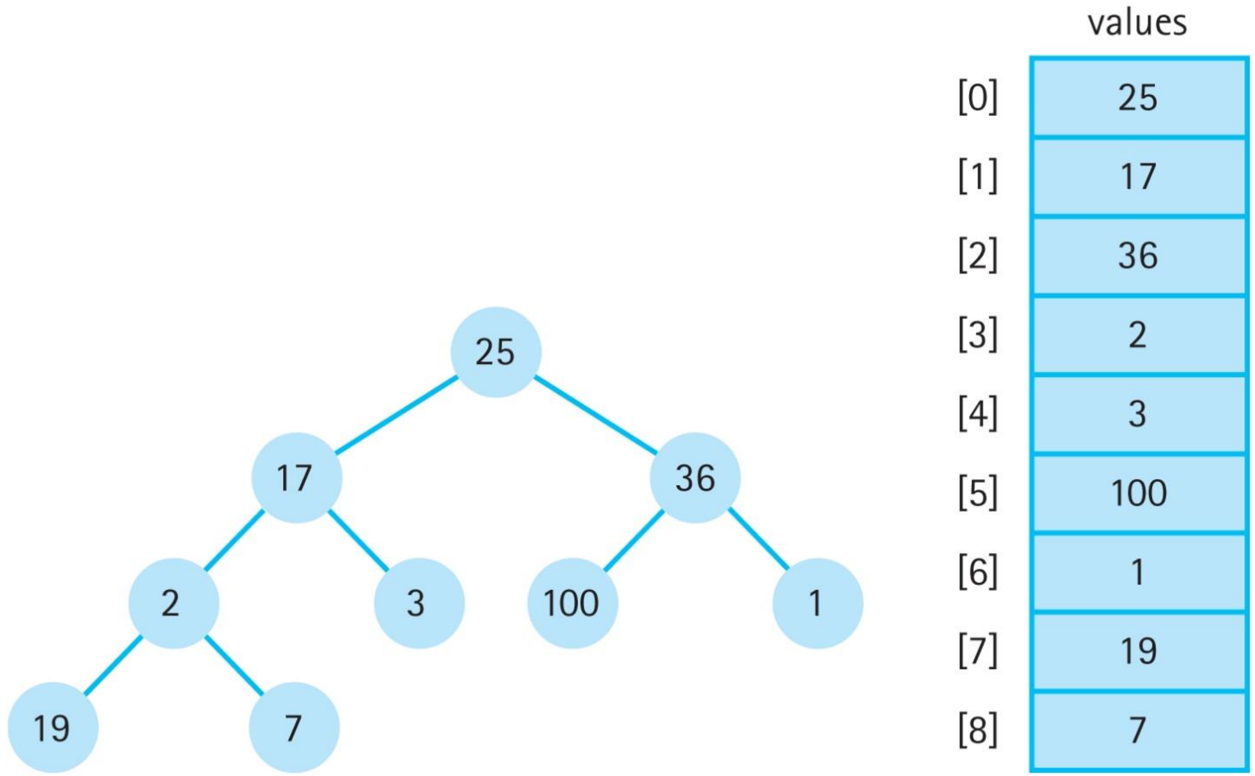

Figure 9.11 An unsorted array and its tree

Building a Heap

- All the leaf nodes are heaps already
- The heap rooted at 2 is almost a heap except for the root; this is perfect for ReheapDown
- Calling ReheapDown on each non-leaf node from the bottom up turns the array into a heap
- For convenience, ReheapDown is written as a global function that takes the heap as an additional parameter

Building a Heap (cont.)

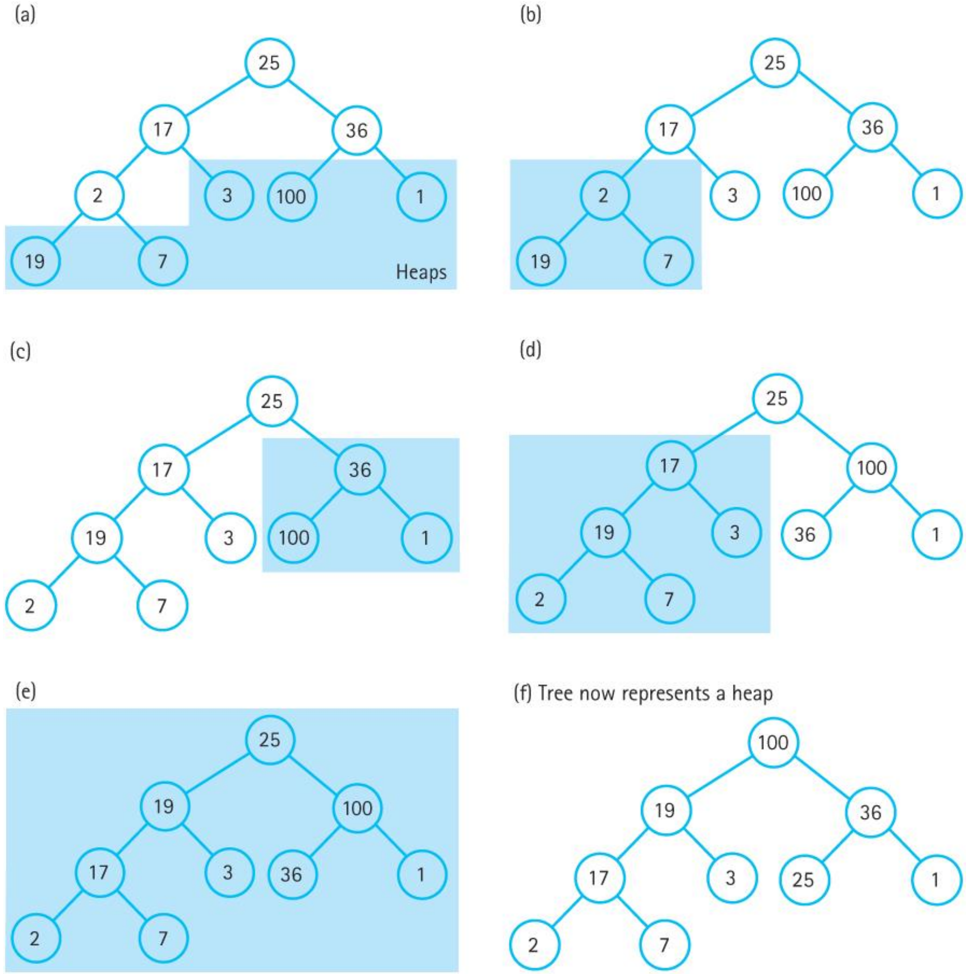

Figure 9.12 The heap-building process (f) Tree now represents a heap

Sorting with the Heap

- Recall that Priority Queue’s Dequeue removes the root and replaces it with the bottom value, reducing the size of the heap by 1
- Heap Sort swaps the root and the bottom value, then calls ReheapDown on the heap
- The root, now at the end of the array, is in the correct position in the sorted list
- The sorted list and the heap coexist in the array

Sorting with the Heap (cont.)

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
values	100	19	36	17	3	25	1	2	7
Swap	7	19	36	17	3	25	1	2	100
ReheapDown	36	19	25	17	3	7	1	2	100
Swap	2	19	25	17	3	7	1	36	100
ReheapDown	25	19	7	17	3	2	1	36	100
Swap	1	19	7	17	3	2	25	36	100
ReheapDown	19	17	7	1	3	2	25	36	100
Swap	2	17	7	1	3	19	25	36	100
ReheapDown	17	3	7	1	2	19	25	36	100
Swap	2	3	7	1	17	19	25	36	100
ReheapDown	7	3	2	1	17	19	25	36	100
Swap	1	3	2	7	17	19	25	36	100
ReheapDown	3	1	2	7	17	19	25	36	100
Swap	2	1	3	7	17	19	25	36	100
ReheapDown	2	1	3	7	17	19	25	36	100
Swap	1	2	3	7	17	19	25	36	100
ReheapDown	1	2	3	7	17	19	25	36	100
Exit from sorting loop	1	2	3	7	17	19	25	36	100

Figure 9.14 Effect of HeapSort on the array

Heap Sort

- The heap was only a temporary structure, used internally by the sorting algorithm
- ReheapDown is $O(\log_2 N)$ and was called each time an element was removed, so Heap Sort is an $O(N \log_2 N)$ sort
- Using an external heap would have cost twice as much memory

Heap Sort Code

```
template<class ItemType>
void HeapSort(ItemType values[], int numValues)
// Assumption: Function ReheapDown is available.
// Post: The elements in the array values are sorted by key.
{
  int index;
  // Convert the array of values into a heap.
  for (index = numValues/2 - 1; index >= 0; index--)
    ReheapDown(values, index, numValues-1);
  // Sort the array.
  for (index = numValues-1; index >= 1; index--)
  {
    Swap(values[0], values[index]);
    ReheapDown(values, 0, index-1);
  }
}
```

The End!

The demonstration of ReheapDown, ReheapUp and Implementing A Heap posted along with this lecture.