Priority Queues (Heaps)

We will discuss

- Efficient implementation of the priority queue ADT
- Applications of priority queues
- Implementations of priority queues

In Chapter 3, we studied LIFO and FIFO disciplines. However, a more sophisticated discipline is sometimes desirable. Examples:

- Printers: which job should run first?
- Operating System Scheduler: which process should run first?

Priority Queue ADT

Operations:

- INSERT(item)
- DELETE_MIN(item): finds, returns, and removes the minimum element in the priority queue.

How do you decide what the minimal element is?

How would you implement a priority queue?

	INSERT	DELETE_MIN
LINKED LIST		
SORTED LINKED LIST		
BINARY SEARCH TREE		

BINARY HEAP

We will implement binary heaps (or just "heaps") as trees implemented on top of arrays.

Can you implement a tree as an array (without pointers!)?

A complete binary tree is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right (See Fig. 6.2).

A complete binary tree *can* be implemented in an array.

Example

Consider node D in the binary tree. It's position in the array is position 4.
$LEFT\_CHILD(4) = 8$,
$RIGHT\_CHILD(4) = 9$, and
$PARENT(4) = 2$.

In general, we have
$LEFT\_CHILD(i) = 2i$,
$RIGHT\_CHILD(i) = 2i + 1$, and
$PARENT(i) = \lfloor i/2 \rfloor$

These equations allow us to traverse the tree by moving through the array.

What is required to implement a tree as an array?

We will assume an array implementation of heaps in everything that follows.

Heap Order Property

For any node in the tree, Parent is smaller than *both* its Children.

So, what node contains the minimum of the tree?

When a tree obeys the heap order property, how much time is required to compute the DELETE_MIN?

The trick is to compute insert so that it preserves the heap order property.

The trick is to insert a new item at the bottom of the tree and then to *percolate up* until the correct location is found. (See Figs. 6.6 and 6.7.)

DELETE_MIN Operation

Finding the minimum of a heap is easy. The hard part is removing it.

Removal is handled using the *percolate down* strategy with a technique that's analogous to insert.

When an element is removed, a hole is created at the root. We now have to percolate the hole down to a leaf.

The key is to choose the smaller of the holes' children to move up. (See Figs. 6.9 through 6.11.)

Observations About Heaps

- Worst case running time for the percolate up and down operations is $O(\log n)$.

- Finding the minimal element in a heap is a constant time operation, but finding the *maximum* element would take linear time.

- A heap contains very little ordering information. So finding a particular key would require linear time as well.

- When implementing a heap, be careful not to assume that every node has two children.

Linear Time BUILD_HEAP

The BUILD_HEAP operation takes n keys as input and places them into an empty heap.

What's the worse case upper bound for n successive insert operations?

The following algorithm enables us to do BUILD_HEAP in $O(n)$ time:

```
for(i = n/2; i > 0; i-- )
    percolate_down( i );
```

How can this algorithm *possibly* work if we are only moving nodes $n/2$ through 1?

Let's look at an example: (Figures 6.15–18).

Analysis of BUILD_HEAP

Since we are moving nodes down, the number of operations per node is bounded by the node's height.

The number of nodes moved is bounded by the number of nodes in the tree.

Thus, if we can show that the sum of the heights of nodes in the tree is bounded, we have a bound on the running time of the BUILD_HEAP algorithm!

Theorem 6.1
For the perfect binary tree of height h containing $2^{h+1} - 1$ nodes, the sum of the heights of the nodes is $2^{h+1} - 1 - (h + 1)$.

Proof:

The sum of the heights of all nodes is

$$\begin{aligned} S &= \Sigma_{i=0}^{h} 2^i(h-i) \\ &= h + 2(h-1) + 4(h-2) + 8(h-3) + \ldots + 2^{h-1}(1) \end{aligned}$$

$$2S = 2h + 4(h-1) + 8(h-2) + 16(h-3) + \ldots + 2^{h}(1)$$

$$\begin{aligned} S &= -h + 2 + 4 + 8 + \ldots + 2^{h-1} + 2^{h} \\ &= (2^{h+1} - 1) - (h+1) \end{aligned}$$

The Selection Problem

Input: a list of n elements and an integer k.

Output: find the k-th largest element.

Algorithm 1A: sort the elements and return the k-th element.

Algorithm 1B:

Input in $I[1]\ldots I[n]$
    Sort $I[1]\ldots I[k]$ into $A[1]\ldots A[k]$
    For $J := 1$ to $n-k$ DO
        IF $I[n-j] > A[k]$
        THEN INSERT $I[n-j]$ into $A[1]\ldots A[k]$

<u>Worst case:</u> $k^2 + (n-k)k = O(nk)$

What is the maximal size for k?

Heap Algorithms

Algorithm 6A: finding the k-th *smallest* elements

1. Insert the n elements into a heap.
2. Perform k delete_min operations.

What is the complexity of the algorithm?

Note: for $k = n$, this is actually a sorting algorithm called *heapsort* (see Ch. 7).

Weakness of the heap implementation is that it does not support merging of two heaps.

How would you merge two heaps in the simplest way possible?

In Section 6.6, the book describes much more complex mechanisms.