A New Problem…

- Application: Find the smallest (or highest priority) item quickly
  - Operating system needs to schedule jobs according to priority
  - Doctors in ER take patients according to severity of injuries
  - Event simulation (bank customers arriving and departing, ordered according to when the event happened)

Priority Queue ADT

- Priority Queue can efficiently do:
  - FindMin (and DeleteMin)
  - Insert
- What if we use…
  - Lists: If sorted, what is the run time for Insert and FindMin? Unsorted?
  - Binary Search Trees: What is the run time for Insert and FindMin?

Less flexibility → More speed

- Lists
  - If sorted: FindMin is O(1) but Insert is O(N)
  - If not sorted: Insert is O(1) but FindMin is O(N)
- Balanced Binary Search Trees (BSTs)
  - Insert is O(log N) and FindMin is O(log N)
- BSTs look good but…
  - BSTs are efficient for all Finds, not just FindMin
  - We only need FindMin

Better than a speeding BST

- We can do better than Balanced Binary Search Trees?
  - Very limited requirements: Insert, FindMin, DeleteMin
  - FindMin is O(1)
  - Insert is O(log N)
  - DeleteMin is O(log N)
Binary Heaps

• A binary heap is a binary tree that is:
  › Complete: the tree is completely filled except possibly the bottom level, which is filled from left to right
  › Satisfies the heap order property
    • every node is less than or equal to its children
    • or every node is greater than or equal to its children
  • The root node is always the smallest node
    › or the largest, depending on the heap order

Heap order property

• A heap provides limited ordering information
  • Each path is sorted, but the subtrees are not sorted relative to each other
    › A binary heap is NOT a binary search tree

Binary Heap vs Binary Search Tree

<table>
<thead>
<tr>
<th>Binary Heap</th>
<th>Binary Search Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent is less than both left and right children</td>
<td>Parent is greater than left child, less than right child</td>
</tr>
</tbody>
</table>

Structure property

• A binary heap is a complete tree
  › All nodes are in use except for possibly the right end of the bottom row

Examples

Array Implementation of Heaps (Implicit Pointers)

• Root node = A[1]
• Keep track of current size N (number of nodes)
### FindMin and DeleteMin

- **FindMin**: Easy!
  - Return root value $A[1]$
  - Run time = ?

- **DeleteMin**:
  - Delete (and return) value at root node

### Maintain the Structure Property

- We now have a “Hole” at the root
  - Need to fill the hole with another value
- When we get done, the tree will have one less node and must still be complete

### DeleteMin: Percolate Down

- Copy smaller child up and go down one level
- Done if both children are $\geq$ item or reached a leaf node
- What is the run time?

### DeleteMin

- Delete (and return) value at root node

### Maintain the Heap Property

- The last value has lost its node
  - We need to find a new place for it
- We can do a simple insertion sort operation to find the correct place for it in the tree

### Percolate Down

```pseudocode
PercDown(i:integer, x:integer): {
// N is the number of entries in queue
j: integer;
Case{
2i > N: A[i] := x; //at bottom//
2i = N: if A[2i] < x then
else A[i] := x;
else j := 2i+1;
if A[j] < x then
A[i] := A[j]; PercDown(j,x);
else A[i] := x;
}
}```
**DeleteMin: Run Time Analysis**

- Run time is $O(\text{depth of heap})$
- A heap is a complete binary tree
- Depth of a complete binary tree of $N$ nodes?
  - $\text{height} = \lceil \log_2(N) \rceil - 1$
- Run time of DeleteMin is $O(\log N)$

**Insert**

- Add a value to the tree
- Structure and heap order properties must still be correct when we are done

**Maintain the Structure Property**

- The only valid place for a new node in a complete tree is at the end of the array
- We need to decide on the correct value for the new node, and adjust the heap accordingly

**Maintain the Heap Property**

- The new value goes where?
- We can do a simple insertion sort operation to find the correct place for it in the tree

**Insert: Percolate Up**

- Start at last node and keep comparing with parent $A[i/2]$
- If parent larger, copy parent down and go up one level
- Done if parent $\leq$ item or reached top node $A[1]$
- Run time?

**Insert: Done**

- Run time?
PercUp

• Class participation
• Define PercUp which percolates new entry to correct spot.
• Note: the parent of i is i/2

```c
PercUp(i : integer, x : integer): {
    ????
}
```

Sentinel Values

• Every iteration of Insert needs to test:
  › if it has reached the top node A[1]
  › if parent ≤ item
• Can avoid first test if A[0] contains a very large negative value
  › sentinel < item, for all items
• Second test alone always stops at top

Binary Heap Analysis

• Space needed for heap of N nodes: O(MaxN)
  › An array of size MaxN, plus a variable to store the size N, plus an array slot to hold the sentinel
• Time
  › FindMin: O(1)
  › DeleteMin and Insert: O(log N)
  › BuildHeap from N inputs: O(N)

Build Heap

```c
BuildHeap {
    for i = N/2 to 1 by -1 PercDown(i, A[i])
}
```

Build Heap
Analysis of Build Heap

• Assume \( N = 2^k - 1 \)
  › Level 1: \( k - 1 \) steps for 1 item
  › Level 2: \( k - 2 \) steps of 2 items
  › Level 3: \( k - 3 \) steps for 4 items
  › Level \( i \): \( k - i \) steps for \( 2^{i-1} \) items

Total Steps = \( \sum_{i=1}^{k-1} (k - i)2^{i-1} = 2^{k-1} - k - 1 \)
= \( O(N) \)

Other Heap Operations

• Find\( (X, H) \): Find the element \( X \) in heap \( H \) of \( N \) elements
  › What is the running time? \( O(N) \)
• FindMax\( (H) \): Find the maximum element in \( H \)
  › What is the running time? \( O(N) \)
• We sacrificed performance of these operations in order to get \( O(1) \) performance for FindMin

Other Heap Operations

• DecreaseKey\( (P, \Delta, H) \): Decrease the key value of node at position \( P \) by a positive amount \( \Delta \). Eg, to increase priority
  › First, subtract \( \Delta \) from current value at \( P \)
  › Heap order property may be violated
  › so percolate up to fix
  › Running Time: \( O(\log N) \)

Other Heap Operations

• IncreaseKey\( (P, \Delta, H) \): Increase the key value of node at position \( P \) by a positive amount \( \Delta \). Eg, to decrease priority
  › First, add \( \Delta \) to current value at \( P \)
  › Heap order property may be violated
  › so percolate down to fix
  › Running Time: \( O(\log N) \)

Other Heap Operations

• Delete\( (P, H) \): E.g. Delete a job waiting in queue that has been preemptively terminated by user
  › Use DecreaseKey\( (P, \infty, H) \) followed by DeleteMin
  › Running Time: \( O(\log N) \)

Other Heap Operations

• Merge\( (H_1, H_2) \): Merge two heaps \( H_1 \) and \( H_2 \) of size \( O(N) \). \( H_1 \) and \( H_2 \) are stored in two arrays.
  › Can do \( O(N) \) Insert operations: \( O(N \log N) \) time
  › Better: Copy \( H_2 \) at the end of \( H_1 \) and use BuildHeap. Running Time: \( O(N) \)
PercUp Solution

PercUp(i : integer, x : integer): {
    if i = 1 then A[1] := x
    else if A[i/2] < x then
        A[i] := x;
    else
        A[i] := A[i/2];
        Percup(i/2,x);
    }