Data Structures

Giri Narasimhan
Office: ECS 254A
Phone: x-3748
giri@cs.fiu.edu

Motivation

- Many applications where
  - Items have associated priorities
    - Job scheduling
      - Long print jobs vs short ones; OS jobs vs user jobs
      - Doctor's office
- Abstract Data Structure: PriorityQueue
  - Insert(x, priority) // insert item with priority value
  - DeleteMin // delete item with highest prioriy
- Simple Implementations:
  - …

Possible Implementations

	insert(x, p)	deleteMin
LinkedList	O(1)	O(N)
SortedList	O(N)	O(1)
ArrayList	O(1)	O(N)
SortedArrays	O(N)	O(1)
Stacks	O(1)	N/A
Queues	O(1)	N/A
Binary Search Tree	O(h)	O(h)
AVL Trees	O(log N)	O(log N)
Binary Heaps	O(log N) **	O(log N)

What is a Binary Heap?

- Heap is
  - a **complete binary tree**
  - Priority of node is at least as large as priority of children

Heap Property

- Useful observations
  - Highest priority is at the root of the tree
  - The number of nodes in a **complete binary tree** of height h is between 2^h and $2^{h+1} - 1$
  - The height of a **complete binary tree** with n nodes is floor(log n)
  - A complete binary tree can be stored in an array.
    - How?

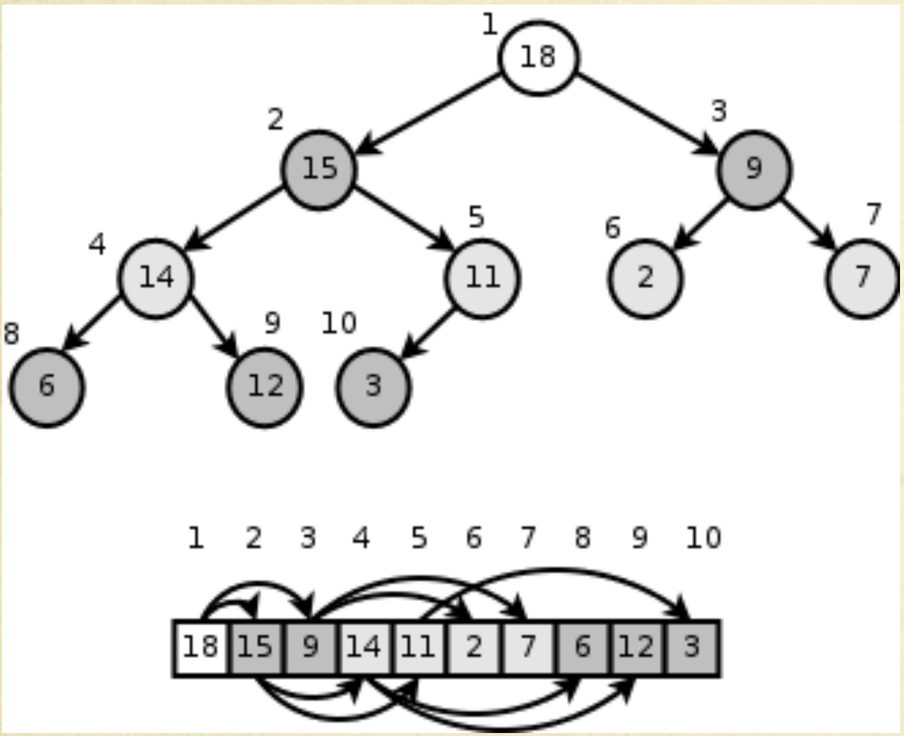
Possible Array Implementation

Index	1	2	3	4	5	6	7	8	9	10
Node	18	15	14	6	12	11	3	9	2	7
Left Child	2	3	4	N/A	N/A	7	N/A	9	N/A	N/A
Right Child	8	6	5	N/A	N/A	N/A	N/A	10	N/A	N/A
Parent	N/A	1	2	3	3	2	6	1	8	8

Index	1	2	3	4	5	6	7	8	9	10
Node	18	15	9	14	11	2	7	6	12	3
Left Child	2	4	6	8	10	N/A	N/A	N/A	N/A	N/A
Right Child	3	5	7	9	N/A	N/A	N/A	N/A	N/A	N/A
Parent	N/A	1	1	2	2	3	3	4	4	5

- No gaps in array
  - Because binary heaps are complete binary trees

Binary Heap: An example



- Root is always in position 1
- For any array position i
  - Left child in position 2i
  - Right child in position 2i+1
  - Parent in floor(i/2)
- All tree links are therefore implicit

http://users.cecs.anu.edu.au/~Alistair.Rendell/Teaching/apac_comp3600/module2/images/Heaps_HeapStructure.png

Array Implementations

- Why is it better?
  - Speed
    - Array operations tend to be faster (indexing is faster than referencing)
    - no need to read and write node references
    - cache performance is better
  - Memory
    - Trees have a storage overhead (pointers to chidren)

Binary Heap interface

```
// **********PUBLIC OPERATIONS*************
// void insert( x )       --> Insert x
// Comparable deleteMin( )--> Return and remove smallest item
// Comparable findMin( )  --> Return smallest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// **********************************
```

Insert Operation

- Let's try the animation first
  - http://www.cs.usfca.edu/~galles/JavascriptVisual/Heap.html
- Basic Idea:
  - Insert item at last item on last level
    - Same as last location in array
  - Percolate item up the tree until Heap Property is satisfied

Insert Implementation

```
public void insert( AnyType x ) {
  if( currentSize == array.length - 1 )
     enlargeArray( array.length * 2 + 1 );

  // Percolate up
  int hole = ++currentSize;
  for(array[0] = x; x.compareTo(array[hole/2]) < 0; hole /= 2 )
        array[hole] = array[hole / 2];
  array[ hole ] = x;
  }
```

Time Complexity = O(log n)

deleteMin Operation

- Basic Idea: First Attempt
  - Delete root
  - Percolate next highest priority value up the tree
- Does not work
  - Result may not be a complete tree
- Let's try the animation now
  - http://www.cs.usfca.edu/~galles/JavascriptVisual/Heap.html
- Basic Idea: SecondAttempt
  - Swap root with last item in array
  - Percolate value down the tree

deleteMin Implementation

```
public AnyType deleteMin( )
  {
      if( isEmpty( ) )
          throw new UnderflowException( );

      AnyType minItem = findMin( );        // returns array[1]
      array[ 1 ] = array[ currentSize-- ];
      percolateDown( 1 );

      return minItem;
  }
```

percolateDown

```
private void percolateDown( int hole )
  {
      int child;

      AnyType tmp = array[ hole ];
      for( ; hole * 2 <= currentSize; hole = child )
      {
          child = hole * 2;
          if( child != currentSize &&
                  array[ child + 1 ].compareTo( array[ child ] ) < 0 )
              child++;     // "child" is now the higher priority of the 2 children
          if( array[ child ].compareTo( tmp ) < 0 )
              array[ hole ] = array[ child ];  // now compare with "child" & swap
          else
              break;
      }
      array[ hole ] = tmp;
  }
```

Time Complexity = O(log n)

Possible Implementations

	insert(x, p)	deleteMin
LinkedList	O(1)	O(N)
SortedList	O(N)	O(1)
ArrayList	O(1)	O(N)
SortedArrays	O(N)	O(1)
Stacks	O(1)	N/A
Queues	O(1)	N/A
Binary Search Tree	O(h)	O(h)
AVL Trees	O(log N)	O(log N)
Binary Heaps	O(log N) **	O(log N)

Rethinking Priority Queues

- We have 2 operations
  - Insert(x)
  - deleteMin()
- Amazingly, this can be used to **sort** a list. How?
  - For (each item in unsorted list) { Insert(x); }
  - While (not IsEmpty()) { deleteMin(); }
- Both steps above take O(n log n) time. Why?
- We also want to rethink the first step.
- If all items are inserted at start before any deletes, can inserts be done faster?

Revisit Insert

- If all items are inserted at start before any deletes, can inserts be done faster?
- Yes!
  - buildHeap

```
private void buildHeap( )
  {   // build heap efficiently from unsorted list
      for( int i = currentSize / 2; i > 0; i-- )
          percolateDown( i );
  }
```

Analysis of buildHeap

- Useful Fact:
  - percolateDown(i) has time complexity O(d) where d is height of node represented by heap location i
- Theorem: For complete binary tree with height h and with $n = 2^{h+1} - 1$ nodes, the sum of heights of the nodes is $2^{h+1} - 1 - (h+1) = O(n)$
- BuildHeap does job of n inserts, but more efficiently
- Since buildHeap can be performed in O(n) time, each insert operation effectively takes O(1) time on the average.

Applications of Priority Queues

- Sorting
  - buildHeap and then perform n deleteMins
    - $O(n) + n \times O(\log n) = O(n \log n)$
- Selection – find kth smallest item in set
  1. buildHeap and then perform only k deleteMins
     - $O(n) + k \times O(\log n) = O(n + k \log n)$
     - If $k = O(n / \log n)$, then time complexity is $O(n)$
     - If k is much larger (say $k = n/2$), then this takes $O(n \log n)$
  2. buildHeap on first k items and then, if needed, insert each remaining item after a deleteMin operation
     - $O(k) + (n-k) \times O(\log k) = O(n \log k)$

Minor Problem

- Heap has largest item at the root
- Thus items deleted would be in reverse order
- One option is to create a heap where the smallest item is at root instead of the largest and to assume that values in the heap increase as you traverse from root to leaf
- A better solution is already achieved by deleteMin()
  - How?
- Remember how deleteMin swaps with last position in array before proceeding to percolateDown() that item?
  - N calls to deleteMin() would place the items in incr order!

Other Heap Operations

- decreaseKey(p, Delta) // make item higher priority
- increaseKey(p, Delta) // make item lower priority
- delete(p) // delete arbitrary item

Sorting with AVL Trees

- N insert() operations, followed by
- N findMin() and N delete()
- Time complexity is O(N log N) again