Abstract Data Types & Implementations

Lecture 16

CS211 – Fall 2005

Program Design Strategies

- Goal: Make it easier to design/create programs
- Basic Data Structures
  - I recognize this; I can use this well-known data structure
  - Examples: Stack, Queue, Priority Queue, Hashtable, Binary Search Tree
- Algorithm Design Methods
  - I can design an algorithm to solve this
  - Examples: Divide & Conquer, Greedy, Dynamic Programming
- Problem Reductions
  - I can change this problem into another with a known solution
  - Or, I can show that a reasonable algorithm is most-likely impossible
  - Examples: reduction to network flow, NP-complete problems

Abstract Data Types (ADTs)

- A method for achieving abstraction for data structures and algorithms
- ADT = model + operations
- Describes what each operation does, but not how it does it
- An ADT is independent of its implementation
- In Java, an *interface* corresponds well to an ADT
  - The interface describes the operations, but says *nothing at all* about how they are implemented
- Example: Stack interface/ADT

```
public interface Stack {
    public void push (Object x);
    public Object pop ( );
    public Object peek ( );
    public boolean isEmpty ( );
    public void makeEmpty ( );
}
```

Queues & Priority Queues

- ADT Queue

  Operations:

  ```
  void enQueue (Object x);
  Object deQueue ( );
  Object peek ( )
  boolean isEmpty ( );
  void makeEmpty ( );
  ```

Where used:
- Simple job scheduler (e.g., print queue)
- Wide use within other algorithms

- ADT PriorityQueue

  Operations:

  ```
  void insert (Object x);
  Object getMax ( );
  Object peekAtMax ( );
  boolean isEmpty ( );
  void makeEmpty ( );
  ```

Where used:
- Job scheduler for OS
- Event-driven simulation
- Can be used for sorting
- Wide use within other algorithms

Sets & Dictionaries

- ADT Set

  Operations:

  ```
  void insert (Object element);
  boolean contains (Object element);
  void remove (Object element);
  boolean isEmpty ( );
  void makeEmpty ( );
  ```

  - Note: no duplicates allowed
    - A "set" with duplicates is usually called a *bag*
- Where used:
  - Wide use within other algorithms

- ADT Dictionary

  Operations:

  ```
  void insert (Object key, Object value);
  void update (Object key, Object value);
  Object find (Object key);
  void remove (Object key);
  boolean isEmpty ( );
  void makeEmpty ( );
  ```

  - Think of

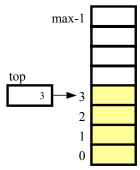
    key = word; value = definition
- Where used:
  - Symbol tables
  - Wide use within other algorithms

Data Structure Building Blocks

- These are *implementation* "building blocks" that are often used to build more-complicated data structures
  - Arrays
  - Linked Lists
    - Singly linked
    - Doubly linked
  - Binary Trees
  - Graphs
    - Adjacency matrix
    - Adjacency list

Array Implementation of Stack

```
class StackArray implements Stack {
   Object [ ] s;                    // Holds the stack
   int top;                         // Index of stack top
   public StackArray(int max)  // Constructor
         {s = new Object [max]; top = -1;}
   public void push (Object item) {s [++top] = item;}
   public Object pop ( ) {return s [top – –];}
   public Object peek ( ) {return s [top];}
   public boolean isEmpty( ) {return top == -1;}
   public void makeEmpty( ) {top = -1;}
}
// Better for garbage collection if makeEmpty( ) also cleared the
   array
```



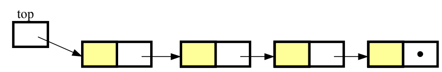
O(1) worst-case time for each operation

Linked List Implementation of Stack

```
class StackLinked implements Stack {
   class Node {Object data; Node next;  // An inner class
        Node (Object d, Node n)          // Constructor for Node
             {data = d; next = n;}
   }
   Node top;                             // Top Node of stack
   public StackLinked ( ) {top = null;}  // Constructor
   public void push (Object item) {top = new Node(item,top);}
   public Object pop ( ) {
        Object temp = top.data; top = top.next; return temp;}
   public boolean isEmpty ( ) {return top == null;}
   public void makeEmpty ( ) {top = null;}
}
```

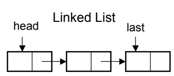
O(1) worst-case time for each operation

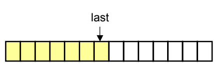
Note that the array implementation can overflow, but the linked list version can't



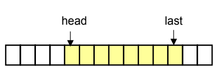
Queue Implementations

- Possible implementations





Array with head always at A[0]
(deQueue( ) becomes expensive)
(can overflow)



Array with wraparound
(can overflow)

- Recall: operations are enQueue, deQueue, peek,…
  - For linked-list
    - All operations are O(1)
  - For array with head at A[0]
    - deQueue takes time O(n)
    - Other ops are O(1)
    - Can overflow
  - For array with wraparound
    - All operations are O(1)
    - Can overflow

Choosing an Implementation

Issues:

- What operations do I need to perform on the data?
  - Insertion, deletion, searching, reset to initial state?
- How efficient do the operations need to be?
- Are there any additional constraints on the operations or on the data structure?
  - Can there be duplicates?
  - When extracting elements, does order matter?
- Is there a known upper bound on the amount of data? Or can it grow unboundedly large?

Priority Queue Implementations

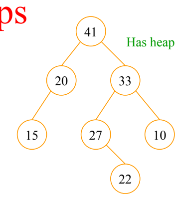
	Unordered List	Ordered List	Unordered Array	Ordered Array	BST*	Balanced BST
insert(item)	O(1)	O(n)	O(1)	O(n)	O(log n) expected	O(log n) worst-case
removeMax( )	O(n)	O(1)	O(n)	O(1)	O(log n) expected	O(log n) worst-case

* BST becomes unbalanced as PQ is used

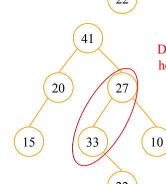
Can we do better than balanced trees?
Well no, not in terms of big-O bounds, but…

Heaps

- A heap is a tree that
  - Has a particular shape (we'll come back to this) and
  - Has the *heap property*
- *Heap property*
  - Each node's value (its *priority*) is ≤ the value of its parent
  - This version is for a max-heap (max value at the root)
    - There is a similar heap property for a min-heap (min at the root)



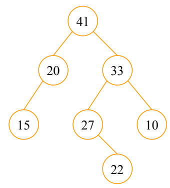
Has heap property



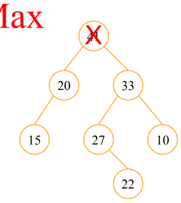
Does not have heap property

Heap Property Examples

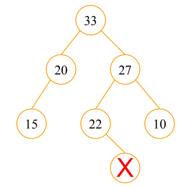
- Ages of people in a family tree
  - Child is younger than parent
  - But an aunt can be younger than her niece
- Salaries of people in an organization
  - A boss makes more money than a subordinate
  - But a 2nd level manager in one region may make more than a 1st level manager in another region
- Crime family ordered by "ruthlessness" (measured by number of murders each member is responsible for)
  - Max, the top crime boss, must be the most ruthless



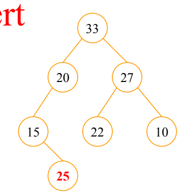
GetMax



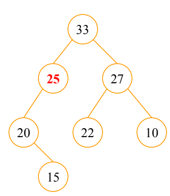
- What would happen if someone were to "get" Max (the top boss)?
  - This leaves a hole at the root
  - We must maintain the heap property so…
    - The most ruthless subordinate moves up to fill the hole
  - This leaves another hole that we fill in the same way
  - We finally create an empty leaf which we delete



Insert

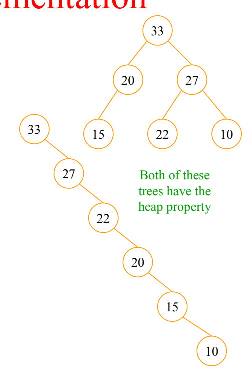


- What happens when "Fat Tony" arrives from Detroit?
  - He starts as a leaf
  - We must maintain the heap property, so…
    - If he is more ruthless than his boss, they swap positions



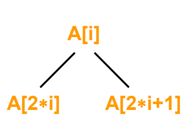
Heap Implementation

- This works great, but…
  - Operations insert and getMax can be slow if the tree is "skinny"
    - Both take linear time on a skinny tree and O(log n) time on a fat tree
- How can we ensure that our heap-tree is fat?



Heap Implementation (the Big Trick)

- Can avoid using pointers!



- Use a complete binary tree stored in an array
  - Definition: *Complete* means that each level of the tree is filled except possibly the last, which is filled from left to right
- For A[i]
  - left child = 2 * i
  - right child = 2 * i + 1
  - parent = $\lfloor i / 2 \rfloor$

Insert and GetMax Pseudocode

```
insert (Item):
   Place item in a leaf (= next empty position in array);
   while (item > parent) {Swap item with parent;}        // BubbleUp

getMax ():
   max = root.value;
   Swap root and last item (call it v) in heap;  // Ensures same shape for heap
   Decrease heap size by 1 (i.e., access less of the array);
   while (v < one of its children)           // BubbleDown
         {Swap v with its largest child;}
   return max;
```

To Build a Heap

- How long to construct a heap, given the items?
- Worst-case time for insert() is O(log n)
- Total time to build heap using insert() is
  - O(log 1) + O(log 2) + ... + O(log n) or O(n log n)

Can we do better?

- We had two heap-fixing methods
  - bubbleUp: move up the tree as long as we're > our parent
  - bubbleDown: move down the tree as long as we're < one of our children
- If we build the heap from the bottom-up using bubbleDown then we can build it in time O(n) (Wow!)

Efficient Heap Building

- Build from the bottom-up
- If there are n items in the heap then...
  - There are about n/2 mini-heaps of height 1
  - There are about n/4 mini-heaps of height 2
  - There are about n/8 mini-heaps of height 3 and so on
- The time to fix up a mini-heap is O(its height)
- Total time spent fixing heaps is thus bounded by
  - n/2 + 2n/4 + 3n/8 + ...
- This can be rewritten as
  - $n(1/2 + 2/4 + ... + i/2^i + ...)$ $= n(2)$
- Thus total heap-building time (using the bottom-up method) is O(n)

HeapSort

- Given a Comparable[ ] array of length n,
  - Put all n elements into a heap: O(n) or O(n log n)
  - Repeatedly get the min: O(n log n)

```
public static void heapSort(Comparable[] a) {
   PriorityQueue<Comparable> pq = new PQ<Comparable>();
   for (Comparable x : a) { pq.put(x); }
   for (int i = 0; i < a.length; i++) { a[i] = pq.get(); }
}
```

PQ Application: Simulation

- Example: Given a probabilistic model of bank-customer arrival times and transaction times, how many tellers are needed
  - Assume we have a way to generate random inter-arrival times
  - Assume we have a way to generate transaction times
  - Can simulate the bank to get some idea of how long customers must wait

<u>Time-Driven Simulation</u>

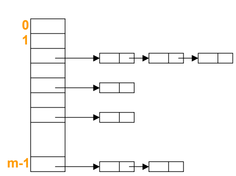
- Check at each *tick* to see if any event occurs

<u>Event-Driven Simulation</u>

- Advance clock to next event, skipping intervening *ticks*
- This uses a PQ!

Another PQ Implementation

- If there are only a few possible priorities then can use an array of queues
  - Each array position represents a priority (0..m-1 where m is the array size)
  - Each queue holds all items that have that priority
- One text [Skiena] calls this a *bounded height priority queue*
- Time for insert: O(1)
- Time for getMax:
  - O(m) in the worst-case
  - Generally, faster
- Example: airline check-in



Other PQ Operations

<u>delete</u>
a particular item

<u>update</u>
an item (change its priority)

<u>join</u>
two priority queues

- For delete and update, we need to be able to find the item
  - One way to do this: Use a Dictionary to keep track of the item's position in the heap
- Efficient joining of 2 Priority Queues requires another data structure
  - Skew Heaps or Pairing Heaps (Chapter 23 in text)