Priority Queues & Binary Heaps

CSE 373
Data Structures
Spring 2007

Readings

- Chapter 6
  - › Section 6.1-6.4

FindMin Problem

- Quickly find the smallest (or highest priority) item in a set
- Applications:
  - › Operating system needs to schedule jobs according to priority instead of FIFO
  - › Event simulation (bank customers arriving and departing, ordered according to when the event happened)
  - › Find student with highest grade, employee with highest salary etc.
  - › Find "most important" customer waiting in line

Priority Queue ADT

- Priority Queue can efficiently do:
  - › FindMin()
    - Returns minimum value but does not delete it
  - › DeleteMin( )
    - Returns minimum value and deletes it
  - › Insert (k)
    - Or Insert (k,x) where k is the key and x the value. In all algorithms the important part is the key, a "comparable" item. We'll skip the value.
  - › size() and isEmpty()

List implementation of a Priority Queue

- What if we use unsorted lists:
  - › FindMin and DeleteMin are O(n)
    - In fact you have to go through the whole list
  - › Insert(k) is O(1)
- What if we used sorted lists
  - › FindMin and DeleteMin are O(1)
    - Be careful if we want both Min and Max (circular array or doubly linked list)
  - › Insert(k) is O(n)

BST implementation of a Priority Queue

- Worst case (degenerate tree)
  - › FindMin, DeleteMin and Insert (k) are all O(n)
- Best case (completely balanced BST)
  - › FindMin, DeleteMin and Insert (k) are all O(log n)
- Balanced BSTs
  - › FindMin, DeleteMin and Insert (k) are all O(log n)

Better than a speeding BST

- Can we do better than Balanced Binary Search Trees?
- Very limited requirements: Insert, FindMin, DeleteMin. The goals are:
  - › FindMin is O(1)
  - › Insert is O(log N)
  - › DeleteMin is O(log N)

Binary Heaps

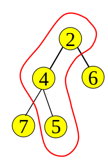
- A binary heap is a binary tree (***NOT*** a BST) that satisfies
  - › Structure property:
    - Complete - the tree is completely filled except possibly the bottom level, which is filled from left to right
  - › Heap order property
    - every node is less than or equal to its children
    - or every 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
  - › All nodes are in use except for possibly the right end of the bottom row

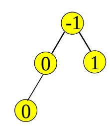
Structure property

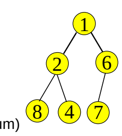
- A binary heap is a complete tree

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



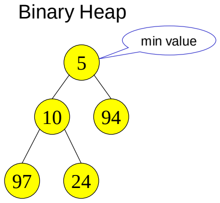




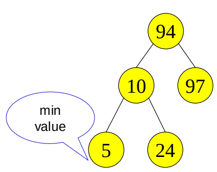
These are all valid binary heaps (minimum)

Binary Heap vs Binary Search Tree

Binary Heap



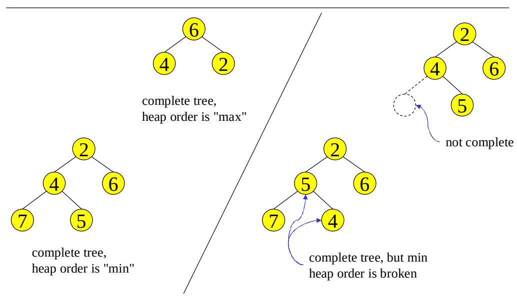
Binary Search Tree



Parent is less than both left and right children

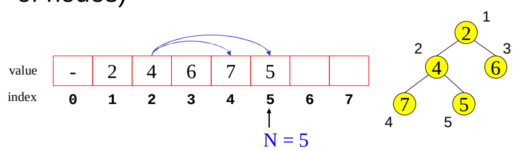
Parent is greater than left child, less than right child

Examples



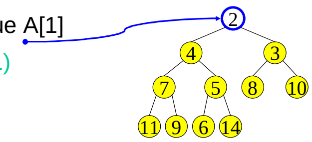
Array Implementation of Heaps

- Root node = A[1]
- Children of A[i] = A[2i], A[2i + 1]
- Keep track of current size N (number of nodes)



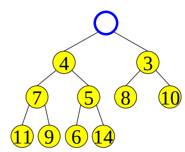
FindMin and DeleteMin

- FindMin: Easy!
  - › Return root value A[1]
  - › Run time = O(1)
- DeleteMin:
  - › Delete (and return) value at root node



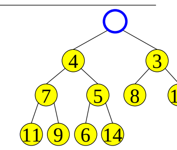
DeleteMin

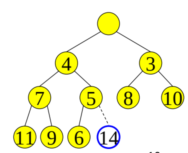
- Delete (and return) value at root node



Maintain the Structure Property

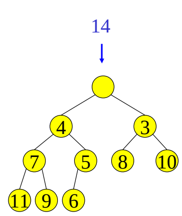
- We now have a “Hole” at the root
  - › Need to fill the hole with another value
- When we get done, the tree will have one less node and must still be complete



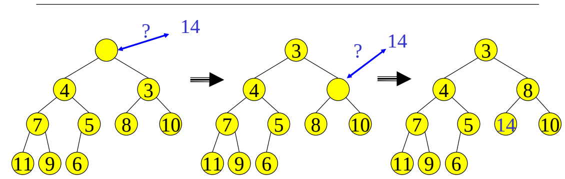


Maintain the Heap Property

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



DeleteMin: Percolate Down



- Keep comparing with children A[2i] and A[2i + 1]
- Copy smaller child up and go down one level
- Done if both children are ≥ item or reached a leaf node
- What is the run time? O(log n)

Percolate Down

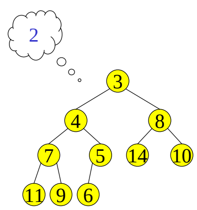
```
PercDown(i:integer, x :integer): {
// N is the number of entries in heap//
j : integer;
Case{
  2i > N : A[i] := x; //at bottom//
  2i = N : if A[2i] < x then
              A[i] := A[2i]; A[2i] := x;
           else A[i] := x;
  2i < N : if A[2i] < A[2i+1] then j := 2i;
           else j := 2i+1;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


DeleteMin: Run Time Analysis

- Run time is O(depth of heap)
- A heap is a complete binary tree
- Depth of a complete binary tree of N nodes?
  - › depth = $\lfloor \log_2(N) \rfloor$
- Run time of DeleteMin is O(log N)

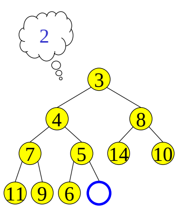
Insert

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



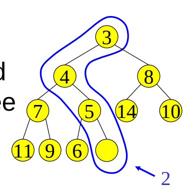
Maintain the Structure Property

- 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

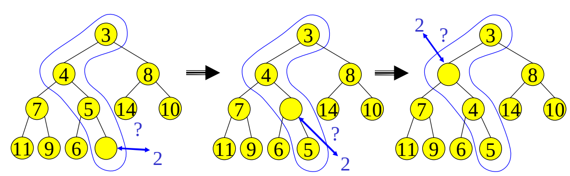


Maintain the Heap Property

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

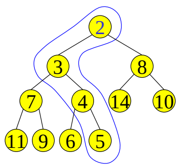


Insert: Percolate Up



- 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 ≤ item or reached top node A[1]
- Run time?

Insert: Done



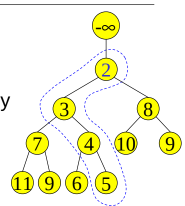
- Run time?

PercUp

```
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);
}
```


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



value	-∞	2	3	8	7	4	10	9	11	9	6	5		
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13

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) (forthcoming)