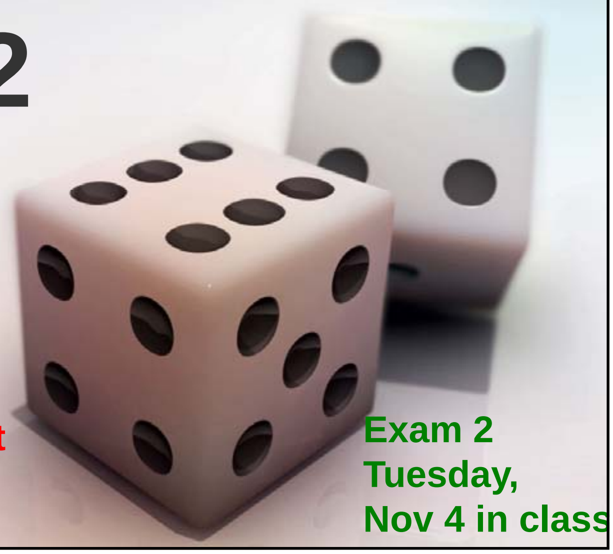
MA/CSSE 473 Day 22



Binary Heaps

Heapsort

Answers to student questions

Exam 2 Tuesday, Nov 4 in class

Binary (max) Heap Quick Review

Representation change example

See also Weiss, Chapter 21 (Weiss does min heaps)

- An almost-complete Binary Tree
  - All levels, except possibly the last, are full
  - On the last level all nodes are as far left as possible
  - No parent is smaller than either of its children
  - A great way to represent a Priority Queue
- Representing a binary heap as an array:

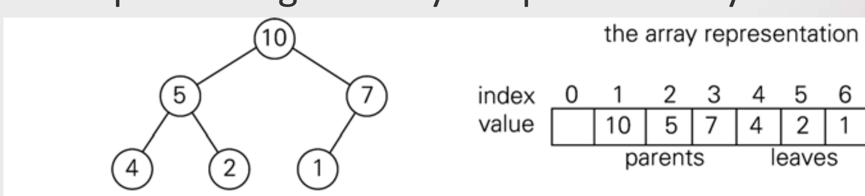


FIGURE 6.10 Heap and its array representation

Insertion and RemoveMax

- Insertion:
  - Insert at the next position (end of the array) to maintain an almost-complete tree, then "percolate up" within the tree to restore heap property.
- RemoveMax:
  - Move last element of the heap to replace the root, then "percolate down" to restore heap property.
- Both operations are $\Theta(\log n)$.
- Many more details (done for min-heaps):
  - http://www.rose-hulman.edu/class/csse/csse230/201230/Slides/18-Heaps.pdf

Heap utilitiy functions

```
def percolateDown(a,i, n):
    """Within the n elements of A to be "re-heapified", the two subtrees of A[i]
       are already maxheaps. Repeatedly exchange the element currently in A[i] with
       the largest of its children until the tree whose root is a[i] is a max heap. """
    current = i # root position for subtree we are heapifying
    lastNodeWithChild = n//2  # if a node number is higher than this, it is a leaf.
    while current <= lastNodeWithChild:
        max = current
        if a[max] < a[2*current]:  # if it is larger than its left child.
            max = 2*current
        if 2*current < n and a[max] < a[2*current+1]:  # But if there is a right child,
            max = 2*current + 1                        # right child may be larger than either
        if max == current:
            break  # larger than its children, so we are done.
        swap(a, current, max) # otherwise, exchange, move down tree, and check again.
        current = max

def percolateUp(a,n):
    'Assume that elements 1 through n-1 are a heap; add element n and "re-heapify"'
    # compare to parent and swap until not larger than parent.
    current = n
    while current > 1:  # or until this value is in the root.
        if a[current//2] >= a[current]:
            break
        swap(a, current, current//2)
        current //= 2
```

Code is on-line, linked from the schedule page

HeapSort

- Arrange array into a heap. (details next slide)
- for i = n downto 2:
  a[1]↔a[i], then "reheapify" a[1]..a[i-1]
- Animation: http://www.cs.auckland.ac.nz/software/AlgAnim/heapsort.html
- Faster heap building algorithm: **buildheap** http://students.ceid.upatras.gr/~perisian/data_structure/HeapSort/heap_applet.html

HeapSort Code

```
# The next two functions tdo the same thing; take an unordered
# array and turn it into a max-heap.  In HW 10, you will show
# that the secondis much more efficient than the first.
# So this first one is not actually called in this code.
def heapifyByInsert(a, n):
    """ Repeatedly insert elements into the heap.
        Worst case number of element exchanges:
            sum of depths of nodes."""
    for i in range(2, n+1):
        percolateUp(a, i)

def buildHeap(a, n):
    """ Each time through the loop, each of node i's two
        subtreees is already a heap.
        Find the correct position to move the root down to
        in order to "reheapify."
        Worst case number of element exchanges:
            sum of heights of nodes."""
    for i in range (n//2, 0, -1):
        percolateDown(a, i, n)

def heapSort(a, n):
    buildHeap(a, n)
    for i in range(n, 1, -1):
        swap(a, 1, i)
        percolateDown(a, 1, i-1)
```

Recap: HeapSort: Build Initial Heap

- Two approaches:
  - for i = 2 to n
    percolateUp(i)
  - for j = n/2 downto 1
    percolateDown(j)
- Which is faster, and why?
- What does this say about overall big-theta running time for HeapSort?

Polynomial Evaluation

Problem Reductiion

TRANSFORM AND CONQUER

Recap: Horner's Rule

- We discussed it in class previously
- It involves a representation change.
- Instead of $a_n x^n + a_{n-1} x^{n-1} + \ldots. + a_1 x + a_0$, which requires a lot of multiplications, we write
- $(\ldots (a_n x + a_{n-1})x + \ldots + a_1)x + a_0$
- code on next slide

Recap: Horner's Rule Code

- This is clearly Θ(n).

```
def polyEvalHorner(p, x):
    """ p is a list representing the coefficients.
        p[i] is the coefficient of x^i.
        x is where we are to evaluate p. """
    sum = 0
    for i in range(len(p)-1, -1, -1):
        sum = sum * x + p[i]

    return sum

# evaluate 4x^3 + 3x^2 + 2x + 1 at x=2
print polyEvalHorner([1, 2, 3, 4], 2)
```

Problem Reduction

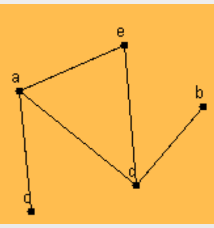
- Express an instance of a problem in terms of an instance of another problem that we already know how to solve.
- There needs to be a one-to-one mapping between problems in the original domain and problems in the new domain.
- **Example:** In quickhull, we reduced the problem of determining whether a point is to the left of a line to the problem of computing a simple 3x3 determinant.
- **Example:** Moldy chocolate problem in HW 9. The big question: What problem to reduce it to? (You'll answer that one in the homework)

Least Common Multiple

- Let m and n be integers. Find their LCM.
- Factoring is hard.
- But we can reduce the LCM problem to the GCD problem, and then use Euclid's algorithm.
- Note that $\text{lcm}(m,n) \cdot \text{gcd}(m,n) = m \cdot n$
- This makes it easy to find lcm(m,n)

Paths and Adjacency Matrices

- We can count paths from A to B in a graph by looking at powers of the graph's adjacency matrix.



	a	b	c	d	e
a	0	0	1	1	1
b	0	0	0	1	0
c	1	0	0	0	0
d	1	1	0	0	1
e	1	0	0	1	0

	a	b	c	d	e
a	3	1	0	1	1
b	1	1	0	0	1
c	0	0	1	1	1
d	1	0	1	3	1
e	1	1	1	1	2

For this example, I used the applet from http://oneweb.utc.edu/~Christopher-Mawata/petersen2/lesson7.htm, which is no longer accessible

Linear programming

- We want to maximize/minimize a linear function $\sum_{i=1}^{n} c_i x_i$, subject to **constraints**, which are linear equations or inequalities involving the n variables $x_1,\ldots,x_n$.
- The constraints define a region, so we seek to maximize the function within that region.
- If the function has a maximum or minimum in the region it happens at one of the vertices of the convex hull of the region.
- The simplex method is a well-known algorithm for solving linear programming problems. We will not deal with it in this course.
- The Operations Research courses cover linear programming in some detail.

Integer Programming

- A linear programming problem is called an **integer programming** problem if the values of the variables must all be integers.
- The knapsack problem can be reduced to an integer programming problem:
- maximize $\sum_{i=1}^{n} x_i v_i$ subject to the constraints $\sum_{i=1}^{n} x_i w_i < W$ and $x_i \in \{0, 1\}$ for $i=1, \ldots, n$

Sometimes using a little more space saves a lot of time

SPACE-TIME TRADEOFFS

Space vs time tradeoffs

- Often we can find a faster algorithm if we are willing to use additional space.
- Examples: