**6.046 Recitation 3: Heaps** Bill Thies, Fall 2004

**Acknowledgement**

Parts of these notes are adapted from a recitation in Spring 1998.

**My contact information:**

Bill Thies

thies@mit.edu

Office hours changed again:

Starting NEXT week (10/2): Sat 1-3pm, 36-153 (this room)

See website for other TA's (should be Tues, Wed 7-9pm, Thur)

**Today:**

- Probability
- Heaps
- PS 1 back

**Expected Interval Between Events**

p = probability that trial succeeds

Expected # trials in order to succeed = 1/p

<< example: # times to flip a coin before you get a heads? 2 >>

- result used multiple times in lecture

random variable X = number of trials needed to succeed

$E[X] = \sum_i i * Pr(X=i)$ [Def. of Expectation]

<< What is probability that X=i? >>

prob. success on 1st try: $p$

2nd try: $p(1-p)$

3rd try: $p(1-p)^2$

ith try: $p(1-p)^{(i-1)}$

$$E[X] = \sum_{i=0}^{\infty} i * p * (1-p)^{(i-1)}$$
$$= p * \sum_{i=0}^{\infty} i * (1-p)^{(i-1)}$$

note $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ $|x|<1$ [infinite geometric series]

$\sum_{i=0}^{\infty} i * x^{i-1} = \frac{1}{(1-x)^2}$ [d/dx]

Substitute x = 1-p:

$= p / (1-(1-p))^2$

$= p / p^2$

$= 1/p$

See also: CLRS p. 1112

**Linearity of Expectation**

Roll a die. What is expected value of sum of top and bottom faces?

Random vars:
T = value on top
B = value on bottom
X = T + B

<< these are for a single throw >>

E[X] = E[T + B] = E[T] + E[B] [Linearity of Expectation] << even though variables are dependent >>

$= \sum_i i * Pr(T=i) + \sum_i i * Pr(B=i)$

$= \sum_{i=1}^{6} i * (1/6) + \sum_i i * (1/6)$

$= 1/6 * 2 * \sum_{i=1}^{6} i$

$= 1/6 * 2 * (6 * 7 / 2)$

$= 7$

**Dynamic Sets**

Dynamic Sets

| | |
|---|---|
| Insert(S, x) | $S \leftarrow S \cup \{x\}$ |
| Delete(S, x) | $S \leftarrow S - \{x\}$ |
| Contains(S, x) | whether or not $x \in S$ |
| Max(S) | largest $x \in S$ |
| etc. | |

Priority Queue: one kind of dynamic set
- Operations << draw table headings below >>

<< Example: job scheduling. Value is priority; comes attached with job. >>

| | Insert | Extract-Max | [Worst case] |
|---|---|---|---|
| Array | $\Theta(1)$ | $\Theta(n)$ | |
| Sorted array | $\Theta(n)$ | $\Theta(1)$ | << linked list has same behavior >> |
| Binary heap | $\Theta(\lg n)$ | $\Theta(\lg n)$ | |

Sorting with heaps:

```
for i ← 1 to n
  do Insert(S, A[i])
for i ← n downto 1
  do A[i] ← Extract-Max(S)
```

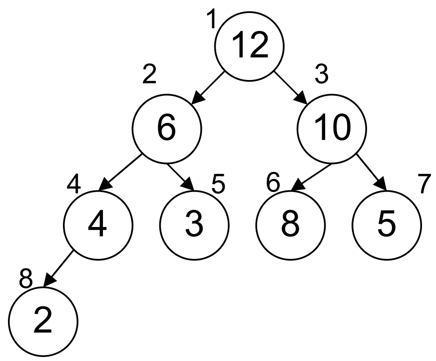
Running time? $\Theta(n \lg n)$

<< Will see that you can be clever, make it run in place >>

**Binary max-heap: nearly complete binary tree**

- heap property: $A[parent(x)] \geq A[x]$

<< also have min-heaps, k-ary heaps >>




A = 12, 6, 10, 4, 3, 8, 5, 2

Can store in array (add indices to graph, draw array)

  Parent(i) = $\lfloor$ i/2 $\rfloor$

  Left(i) = 2*i

  Right(i) = 2*i + 1

Extract-max (A)

  max $\leftarrow$ A[1]

  A[1] $\leftarrow$ A[size(A)]

  size(A) $\leftarrow$ size(A)- 1

  run Heapify(A)  // restore heap property

  return max

EXTRACT-MAX EXAMPLE  (Remove node 12, move 4 to top, run Heapify)

Time = Heapify time + $\Theta(1)$

**Maintaining Heap Property**

```
Heapify(A, i)  // Subtrees of i are heaps.  Makes i's subtree a heap.
  largest ← i
  if Left(i) ≤ size(A) and A[Left(i)] > A[largest]
    largest ← Left(i)
  if Right(i) ≤ size(A) and A[Right(i)] > A[largest]
    largest ← Right(i)
  if largest ≠ i
    then    exchange A[i] ↔ A[largest]
            Heapify(A, largest)
```

<< note: tighter code via looping >>

HEAPIFY EXAMPLE  (See CLRS, p. 131)

Correctness:  induction on height of node i

Running time:  proportional to height, O(lg n)  (for n-node heap)

**Inserting Elements into Heap**

```
Insert(A, key)
  size(A) ← size(A) + 1
  i ← size(A)
  A[i] ← key
  while i > 1 and A[Parent(i)] < key
    do exchange A[i] ↔ A[Parent(i)]
          i ← Parent(i)
```

<< like insertion of insertion sort >>

INSERT EXAMPLE  (See CLRS, p. 141)

Running time:  O(lg n) for n-elem heap

**Sorting Using Heaps**

Heapsort
- O(n lg n) worst-case << known >>
- sorts in place << to show >>

<< none of the sorts we've seen so far have both of these properties >>

```
HeapSort(A)
  Build-Heap(A)
  for i ← size(A) downto 2
    do swap A[1] ↔ A[i]
        size(A) ← size(A) – 1   << note: effects only size of heap, not real array >>
        Heapify (A)
```

HEAPSORT EXAMPLE (See CLRS, p. 137)

Running time = Build-Heap time + O(n lg n)

**Building a Heap from an Array**

We didn't cover this in detail in recitation.

Build-Heap(A)
  for i $\leftarrow$ size(A) downto 1
    do Heapify(A, i)

BUILD-HEAP EXAMPLE (See CLRS, p. 134)

Correctness: induction on i
- invariant: all trees rooted at m > i are heaps

Running time (naïve analysis):
n calls to Heapify = n * O(lg n) = O(n lg n)

*** **This is the naïve analysis; running time is actually O(n) [see below]**

<< Good enough for O(n lg n) bound on heapsort, but sometimes we build heaps for other reasons. >>

Tighter analysis:

Time of Heapify = O(height(i))
Assume $n = 2^k - 1$ (complete binary tree)

$$T(n) = O\left( \frac{n+1}{2} + \frac{n+1}{4} * 2 + \frac{n+1}{8} * 3 \; ... \; 1 * k \right)$$

$$T(n) = O((n+1) \textstyle\sum_{i=1}^{k} i/2^i$$

$= O((n+1) \sum_{i=1}^{k} i * (1/2)^i$ [refactor]

$= O((n+1) \sum_{i=0}^{\infty} i * (1/2)^i$ [upper bound]

$= O((n+1) * 1/2 \sum_{i=0}^{\infty} i * (1/2)^{i-1})$ [refactor to what we saw before]

$= O((n+1) * 1/2 * 1 / (1-1/2)^2)$ [solve sum]

$= O((n+1) * 1/2 / 1/4) = O(2(n+1) = O(n)$

Thus, **Build-Heap(A) = O(n)**, where n=size(A)

**Sorting Review**

| **Sort** | **Average** | **Worst Case** | **In place?** |
|---|---|---|---|
| Heapsort | $\Theta(n \lg n)$ | $\Theta(n \lg n)$ | yes |
| Quicksort | $\Theta(n \lg n)$ | $\Theta(n^2)$ | yes |
| MergeSort | $\Theta(n \lg n)$ | $\Theta(n \lg n)$ | no |
| InsertionSort | $\Theta(n^2)$ | $\Theta(n^2)$ | yes |