Algorithms

Gabriel Istrate, West University of Timișoara.

Course 6: Sorting algorithms: Quicksort, Heapsort, Lower bounds.

Slide credit: David Luebke (Virginia)

Quicksort

- Sorts in place
- Sorts O(n lg n) in the average case
- Sorts $O(n^2)$ in the worst case
  - But in practice, it's quick
  - And the worst case doesn't happen often (but more on this later…)

Quicksort

- Another divide-and-conquer algorithm
  - The array A[p..r] is *partitioned* into two non-empty subarrays A[p..q] and A[q+1..r]
    - Invariant: All elements in A[p..q] are less than all elements in A[q+1..r]
  - The subarrays are recursively sorted by calls to quicksort
  - Unlike merge sort, no combining step: two subarrays form an already-sorted array

Quicksort Code

```
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r);
        Quicksort(A, p, q);
        Quicksort(A, q+1, r);
    }
}
```

Partition

- Clearly, all the action takes place in the **`partition()`** function
  - Rearranges the subarray in place
  - End result:
    - Two subarrays
    - All values in first subarray ≤ all values in second
  - Returns the index of the "pivot" element separating the two subarrays
- *How do you suppose we implement this?*

Partition In Words

- Partition(A, p, r):
  - Select an element to act as the “pivot” (*which?*)
  - Grow two regions, A[p..i] and A[j..r]
    - All elements in A[p..i] <= pivot
    - All elements in A[j..r] >= pivot
  - Increment i until A[i] >= pivot
  - Decrement j until A[j] <= pivot
  - Swap A[i] and A[j]
  - Repeat until i >= j
  - Return j

Partition Code

```
Partition(A, p, r)
    x = A[p];
    i = p - 1;
    j = r + 1;
    while (TRUE)
        repeat
            j--;
        until A[j] <= x;
        repeat
            i++;
        until A[i] >= x;
        if (i < j)
            Swap(A, i, j);
        else
            return j;
```

`Illustrate on`

`A = {5, 3, 2, 6, 4, 1, 3, 7};`

What is the running time of ***`partition()`****?*

Partition Code

```
Partition(A, p, r)
    x = A[p];
    i = p - 1;
    j = r + 1;
    while (TRUE)
        repeat
            j--;
        until A[j] <= x;
        repeat
            i++;
        until A[i] >= x;
        if (i < j)
            Swap(A, i, j);
        else
            return j;
```

`partition()` *runs in O(n) time*

Analyzing Quicksort

- *What will be the worst case for the algorithm?*
  - Partition is always unbalanced
- *What will be the best case for the algorithm?*
  - Partition is perfectly balanced
- *Which is more likely?*
  - The latter, by far, except...
- *Will any particular input elicit the worst case?*
  - Yes: Already-sorted input

Analyzing Quicksort

- In the worst case:

  $T(1) = \Theta(1)$

  $T(n) = T(n - 1) + \Theta(n)$

- Works out to

  $T(n) = \Theta(n^2)$

Analyzing Quicksort

- In the best case:

  $T(n) = 2T(n/2) + \Theta(n)$

- What does this work out to?

  $T(n) = \Theta(n \lg n)$

Improving Quicksort

- The real liability of quicksort is that it runs in $O(n^2)$ on already-sorted input
- Solutions:
  - Randomize the input array, OR
  - *Pick a random pivot element, OR*
  - *Choose the pivot wisely (median)*
- *How will these solve the problem?*
  - By insuring that no particular input can be chosen to make quicksort run in $O(n^2)$ time

Analyzing Quicksort: Average Case

- Assuming random input, average-case running time is much closer to O(n lg n) than O(n^2)
- First, a more intuitive explanation/example:
  - Suppose that partition() always produces a 9-to-1 split. This looks quite unbalanced!
  - The recurrence is thus:

    T(n) = T(9n/10) + T(n/10) + n

    Use n instead of O(n) for convenience (how?)
  - *How deep will the recursion go?* (draw it)

Analyzing Quicksort: Average Case

- Intuitively, a real-life run of quicksort will produce a mix of "bad" and "good" splits
  - Randomly distributed among the recursion tree
  - Pretend for intuition that they alternate between best-case (n/2 : n/2) and worst-case (n-1 : 1)
  - *What happens if we bad-split root node, then good-split the resulting size (n-1) node?*

Analyzing Quicksort: Average Case

- Intuitively, a real-life run of quicksort will produce a mix of "bad" and "good" splits
  - Randomly distributed among the recursion tree
  - Pretend for intuition that they alternate between best-case (n/2 : n/2) and worst-case (n-1 : 1)
  - *What happens if we bad-split root node, then good-split the resulting size (n-1) node?*
    - We end up with three subarrays, size 1, (n-1)/2, (n-1)/2
    - Combined cost of splits = n + n -1 = 2n -1 = O(n)
    - No worse than if we had good-split the root node!

Analyzing Quicksort: Average Case

- Intuitively, the O(n) cost of a bad split (or 2 or 3 bad splits) can be absorbed into the O(n) cost of each good split
- Thus running time of alternating bad and good splits is still O(n lg n), with slightly higher constants
- How can we be more rigorous?

Analyzing Quicksort: Average Case

- For simplicity, assume:
  - All inputs distinct (no repeats)
  - Slightly different **`partition()`** procedure
    - partition around a random element, which is not included in subarrays
    - all splits (0:n-1, 1:n-2, 2:n-3, … , n-1:0) equally likely
- *What is the probability of a particular split happening?*
- Answer: 1/n

Analyzing Quicksort: Average Case

- So partition generates splits
  (0:n-1, 1:n-2, 2:n-3, … , n-2:1, n-1:0)
  each with probability 1/n
- If T(n) is the expected running time,

$$T(n) = \frac{1}{n}\sum_{k=0}^{n-1}\left[T(k) + T(n-1-k)\right] + \Theta(n)$$

- *What is each term under the summation for?*
- *What is the Θ(n) term for?*

Analyzing Quicksort: Average Case

- So…

$$T(n) = \frac{1}{n}\sum_{k=0}^{n-1}\left[T(k) + T(n-1-k)\right] + \Theta(n)$$

$$= \frac{2}{n}\sum_{k=0}^{n-1} T(k) + \Theta(n) \longleftarrow$$

Write it on the board

  - We'll take care of that in a second

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
  - Assume that the inductive hypothesis holds
  - Substitute it in for some value < n
  - Prove that it follows for n

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - *What's the answer?*
  - Assume that the inductive hypothesis holds
  - Substitute it in for some value < n
  - Prove that it follows for n

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - $T(n) = O(n \lg n)$
  - Assume that the inductive hypothesis holds
  - Substitute it in for some value < n
  - Prove that it follows for n

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - T(n) = O(n lg n)
  - Assume that the inductive hypothesis holds
    - *What's the inductive hypothesis?*
  - Substitute it in for some value < n
  - Prove that it follows for n

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - $T(n) = O(n \lg n)$
  - Assume that the inductive hypothesis holds
    - $T(n) \leq an \lg n + b$ for some constants a and b
  - Substitute it in for some value < n
  - Prove that it follows for n

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - $T(n) = O(n \lg n)$
  - Assume that the inductive hypothesis holds
    - $T(n) \le an \lg n + b$ for some constants a and b
  - Substitute it in for some value $< n$
    - *What value?*
  - Prove that it follows for n

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - $\mathrm{T}(n) = \mathrm{O}(n \lg n)$
  - Assume that the inductive hypothesis holds
    - $\mathrm{T}(n) \le an \lg n + b$ for some constants a and b
  - Substitute it in for some value < n
    - The value k in the recurrence
  - Prove that it follows for n

Analyzing Quicksort: Average Case

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - $T(n) = O(n \lg n)$
  - Assume that the inductive hypothesis holds
    - $T(n) \leq an \lg n + b$ for some constants a and b
  - Substitute it in for some value < n
    - The value k in the recurrence
  - Prove that it follows for n
    - Grind through it…

Analyzing Quicksort: Average Case

$$T(n) = \frac{2}{n}\sum_{k=0}^{n-1} T(k) + \Theta(n)$$

The recurrence to be solved

$$\le \frac{2}{n}\sum_{k=0}^{n-1} (ak\lg k + b) + \Theta(n)$$

Plug in inductive hypothesis

$$\le \frac{2}{n}\left[b + \sum_{k=1}^{n-1} (ak\lg k + b)\right] + \Theta(n)$$

Expand out the k=0 case

$$= \frac{2}{n}\sum_{k=1}^{n-1} (ak\lg k + b) + \frac{2b}{n} + \Theta(n)$$

2b/n is just a constant, so fold it into Θ(n)

$$= \frac{2}{n}\sum_{k=1}^{n-1} (ak\lg k + b) + \Theta(n)$$

Analyzing Quicksort: Average Case

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1}(ak\lg k + b) + \Theta(n)$$

The recurrence to be solved

$$= \frac{2}{n}\sum_{k=1}^{n-1} ak\lg k + \frac{2}{n}\sum_{k=1}^{n-1} b + \Theta(n)$$

Distribute the summation

$$= \frac{2a}{n}\sum_{k=1}^{n-1} k\lg k + \frac{2b}{n}(n-1) + \Theta(n)$$

Evaluate the summation: b+b+…+b = b (n-1)

$$\le \frac{2a}{n}\boxed{\sum_{k=1}^{n-1} k\lg k} + 2b + \Theta(n)$$

Since n-1<n, 2b(n-1)/n < 2b

This summation gets its own set of slides later

Analyzing Quicksort: Average Case

$$T(n) \le \frac{2a}{n}\sum_{k=1}^{n-1} k\lg k + 2b + \Theta(n)$$

The recurrence to be solved

$$\le \frac{2a}{n}\left(\frac{1}{2}n^2\lg n - \frac{1}{8}n^2\right) + 2b + \Theta(n)$$

We'll prove this later

$$= an\lg n - \frac{a}{4}n + 2b + \Theta(n)$$

Distribute the (2a/n) term

$$= an\lg n + b + \left(\Theta(n) + b - \frac{a}{4}n\right)$$

Remember, our goal is to get $T(n) \le an\lg n + b$

$$\le an\lg n + b$$

Pick a large enough that an/4 dominates $\Theta(n)+b$

Analyzing Quicksort: Average Case

- So $T(n) \le an \lg n + b$ for certain a and b
  - Thus the induction holds
  - Thus T(n) = O(n lg n)
  - Thus quicksort runs in O(n lg n) time on average (phew!)
- Oh yeah, the summation…

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k = \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k$$

Split the summation for a tighter bound

$$\leq \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg n$$

The lg *k in the second term is bounded by* lg *n*

$$= \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

Move the lg *n outside the summation*

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k \le \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

The summation bound so far

$$\le \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg(n/2) + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

The lg k in the first term is bounded by lg n/2

$$= \sum_{k=1}^{\lceil n/2 \rceil - 1} k(\lg n - 1) + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

lg *n/2* = lg *n* - 1

$$= (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

Move (lg n - 1) outside the summation

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k \le (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

The summation bound so far

$$= \lg n \sum_{k=1}^{\lceil n/2 \rceil - 1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

Distribute the (lg n - 1)

$$= \lg n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

The summations overlap in range; combine them

$$= \lg n \left(\frac{(n-1)(n)}{2} \right) - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

The Gaussian series

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k \le \left(\frac{(n-1)(n)}{2}\right)\lg n - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

The summation bound so far

$$\le \frac{1}{2}[n(n-1)]\lg n - \sum_{k=1}^{n/2-1} k$$

Rearrange first term, place upper bound on second

$$\le \frac{1}{2}[n(n-1)]\lg n - \frac{1}{2}\left(\frac{n}{2}\right)\left(\frac{n}{2}-1\right)$$

X Gaussian series

$$\le \frac{1}{2}\left(n^2 \lg n - n \lg n\right) - \frac{1}{8}n^2 + \frac{n}{4}$$

Multiply it all out

Tightly Bounding The Key Summation

$$\sum_{k=1}^{n-1} k \lg k \le \frac{1}{2}\left(n^2 \lg n - n \lg n\right) - \frac{1}{8}n^2 + \frac{n}{4}$$

$$\le \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2 \text{ when } n \ge 2$$

Done!!!

Sorting Revisited

So far we've talked about two algorithms to sort an array of numbers

What is the advantage of merge sort?

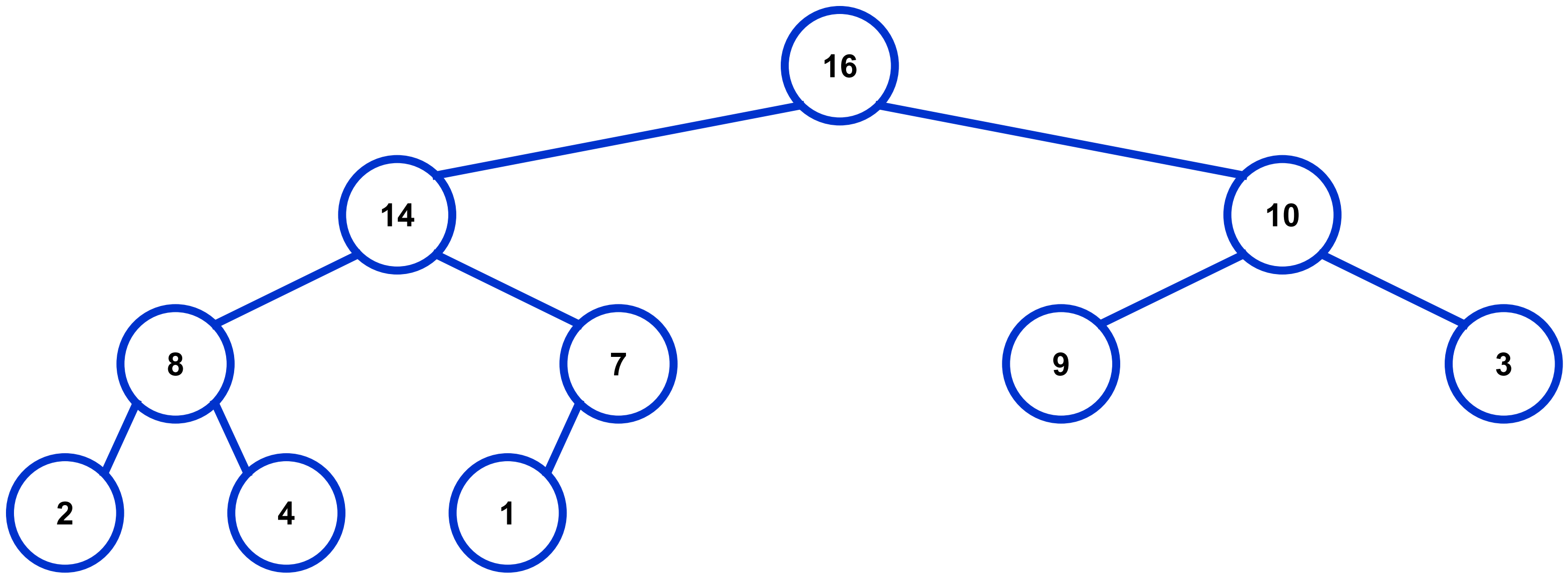
What is the advantage of insertion sort?

Next on the agenda: *Heapsort*

Combines advantages of both previous algorithms

Heaps

A *heap* can be seen as a complete binary tree:

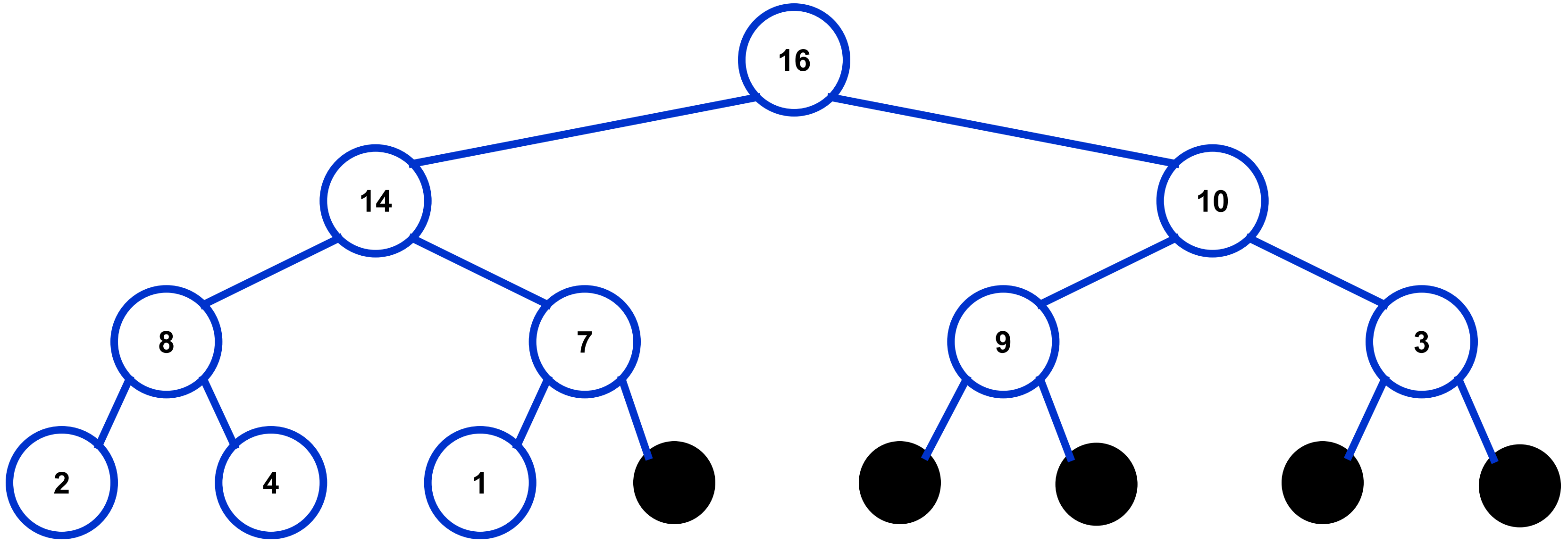


What makes a binary tree complete?

Is the example above complete?

Heaps

A *heap* can be seen as a complete binary tree:



The book calls them "nearly complete" binary trees; can think of unfilled slots as null pointers

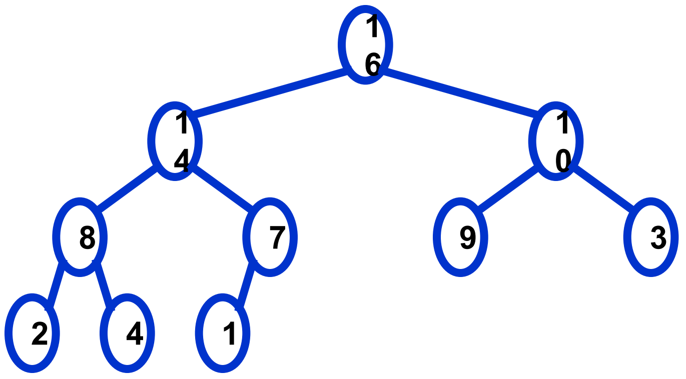
Heaps

In practice, heaps are usually implemented as arrays:

A =

16	*14*	*10*	*8*	*7*	*9*	*3*	*2*	*4*	*1*

=



Heaps

To represent a complete binary tree as an array:

The root node is A[1]

Node *i* is A[*i*]

The parent of node *i* is A[*i*/2] (note: integer divide)

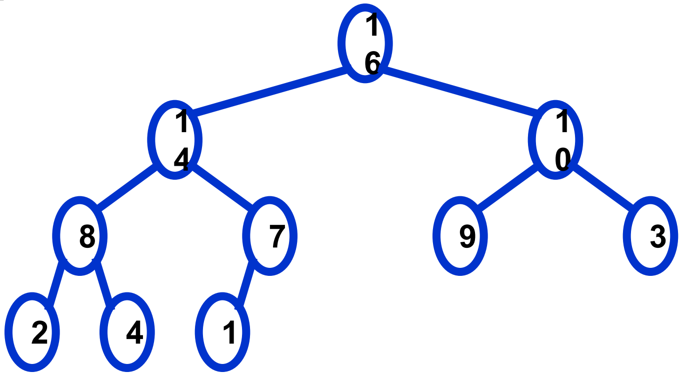
The left child of node *i* is A[2*i*]

The right child of node *i* is A[2*i* + 1]

A =

16	*14*	*10*	*8*	*7*	*9*	*3*	*2*	*4*	*1*

=



Referencing Heap Elements

So…

```
Parent(i) { return ⌊i/2⌋; }
Left(i) { return 2*i; }
right(i) { return 2*i + 1; }
```

An aside: *How would you implement this most efficiently?*

Another aside: *Really?*

The Heap Property

Heaps also satisfy the *heap property*:

$A[\mathit{Parent}(i)] \geq A[i]$ for all nodes $i > 1$

In other words, the value of a node is at most the value of its parent

Where is the largest element in a heap stored?

Definitions:

The *height* of a node in the tree = the number of edges on the longest downward path to a leaf

The height of a tree = the height of its root

Heap Height

What is the height of an n-element heap? Why?

This is nice: basic heap operations take at most time proportional to the height of the heap

Heap Operations: Heapify()

`Heapify()`: maintain the heap property

Given: a node *i* in the heap with children *l* and *r*

Given: two subtrees rooted at *l* and *r*, assumed to be heaps

Problem: The subtree rooted at *i* may violate the heap property (*How?*)

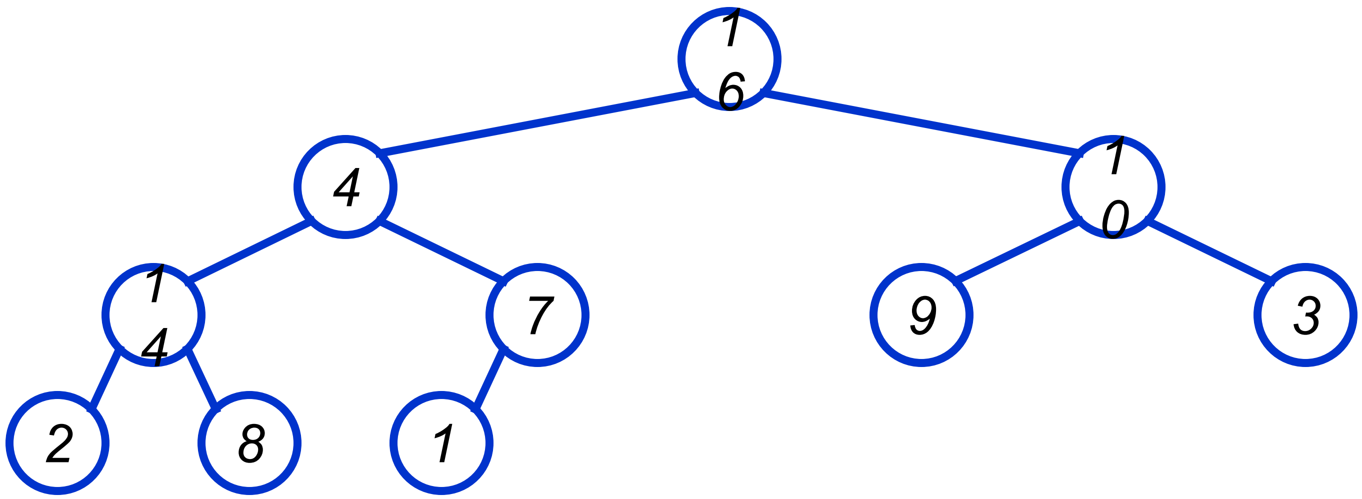
Action: let the value of the parent node "float down" so subtree at *i* satisfies the heap property

- *What do you suppose will be the basic operation between i, l, and r?*

Heap Operations: Heapify()

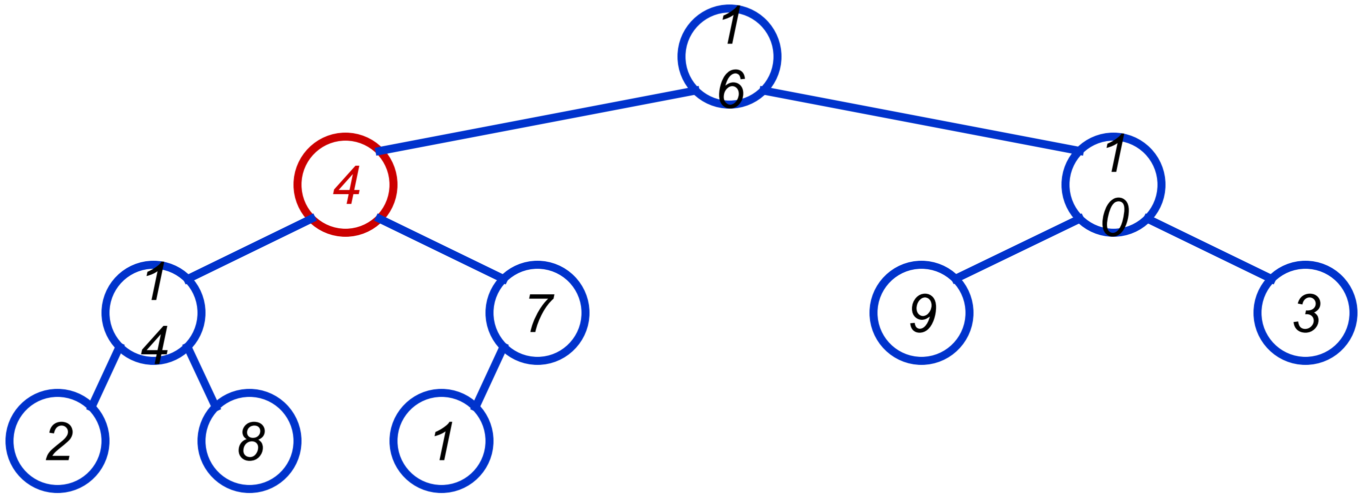
```
Heapify(A, i)
{
   l = Left(i); r = Right(i);
   if (l <= heap_size(A) && A[l] > A[i])
    largest = l;
   else
    largest = i;
   if (r <= heap_size(A) && A[r] > A[largest])
    largest = r;
   if (largest != i)
    Swap(A, i, largest);
    Heapify(A, largest);
}
```

Heapify() Example



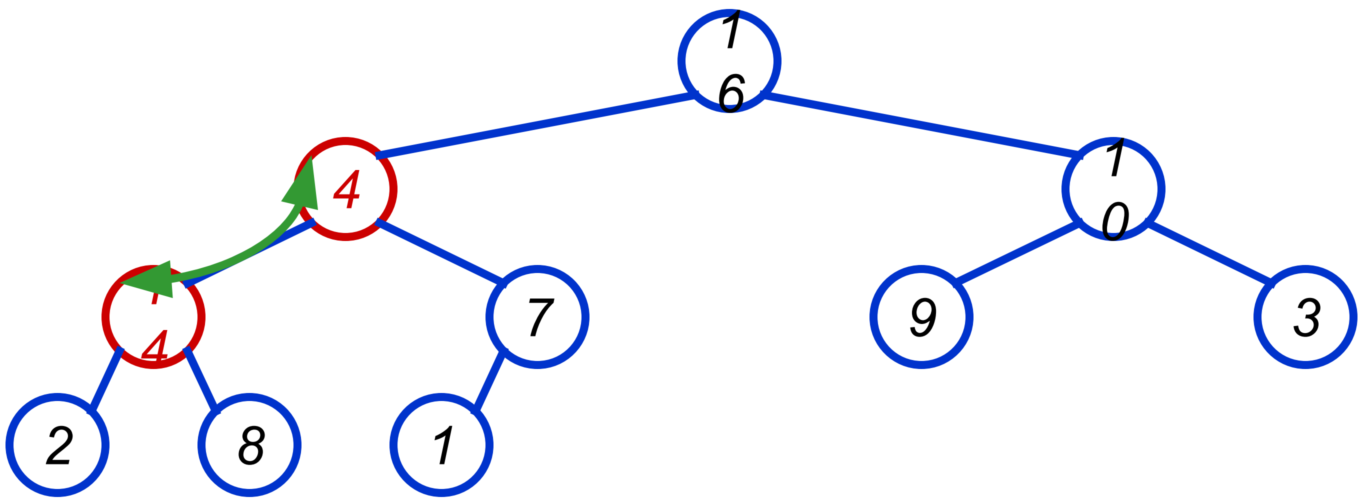
A =	16	4	10	14	7	9	3	2	8	1

Heapify() Example



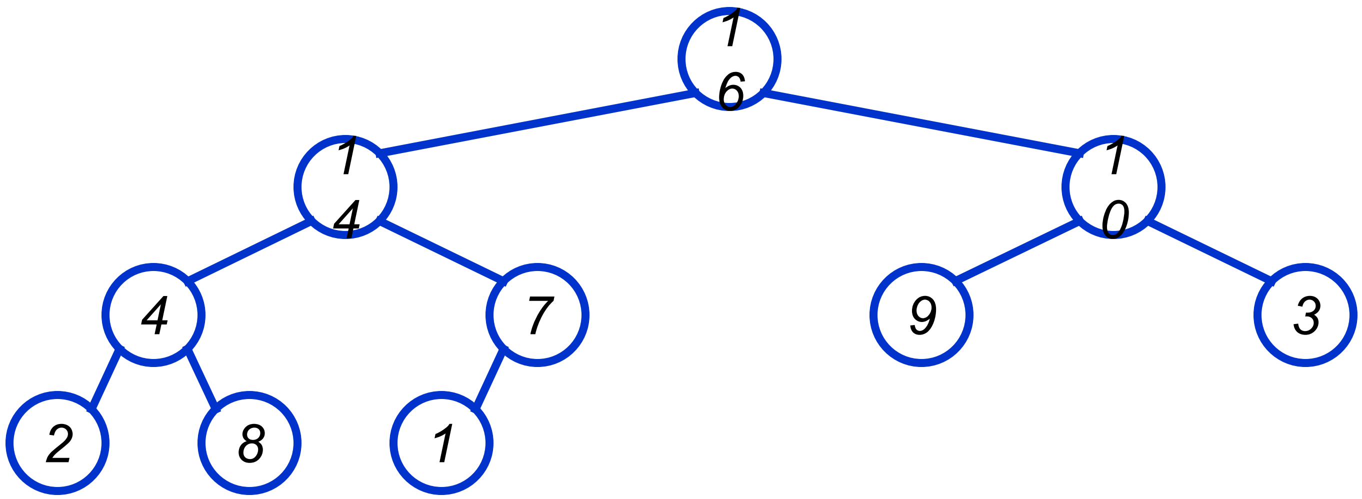
A =	16	4	10	14	7	9	3	2	8	1

Heapify() Example



A =	16	4	10	14	7	9	3	2	8	1

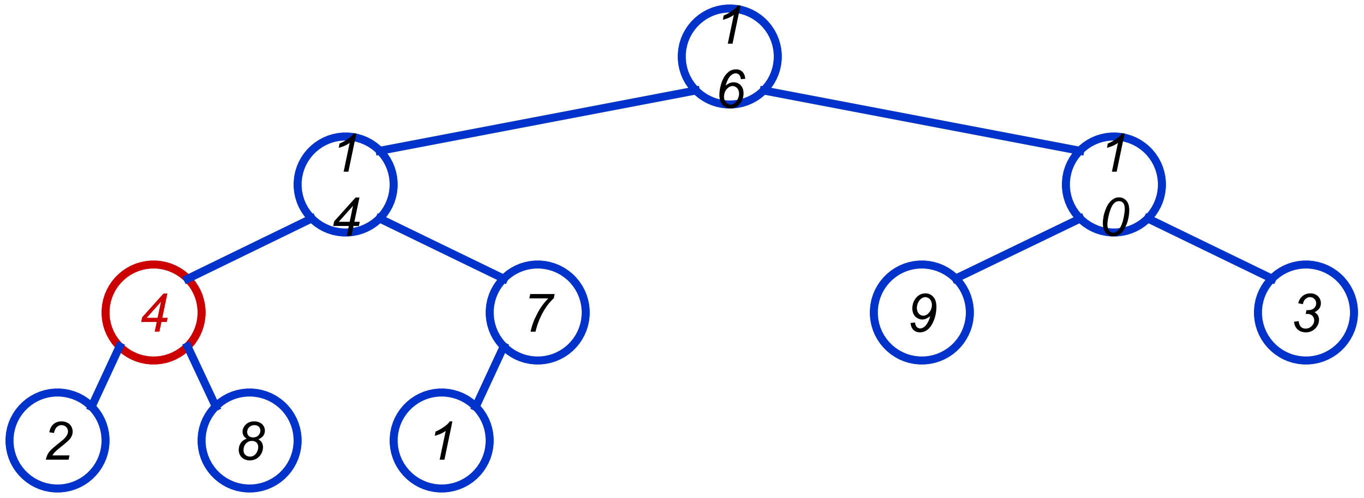
Heapify() Example



A =

16	14	10	4	7	9	3	2	8	1

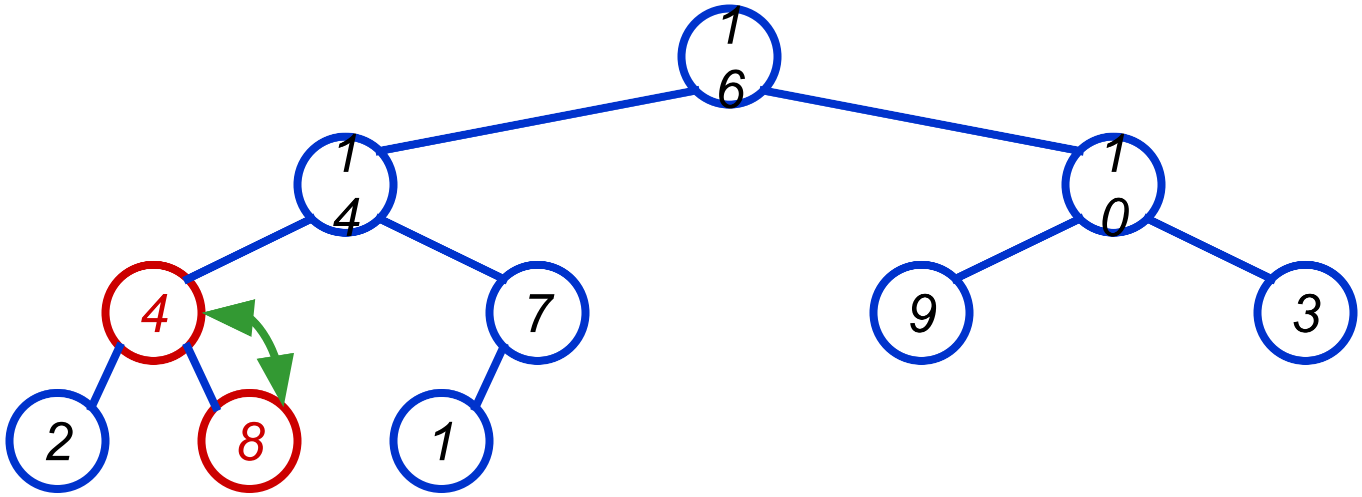
Heapify() Example

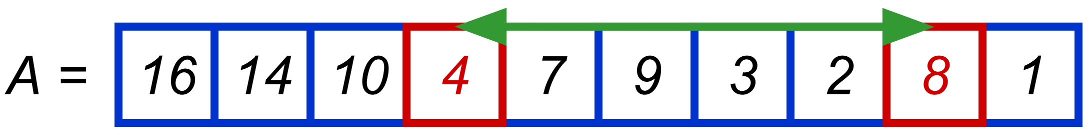


A =

16	14	10	4	7	9	3	2	8	1

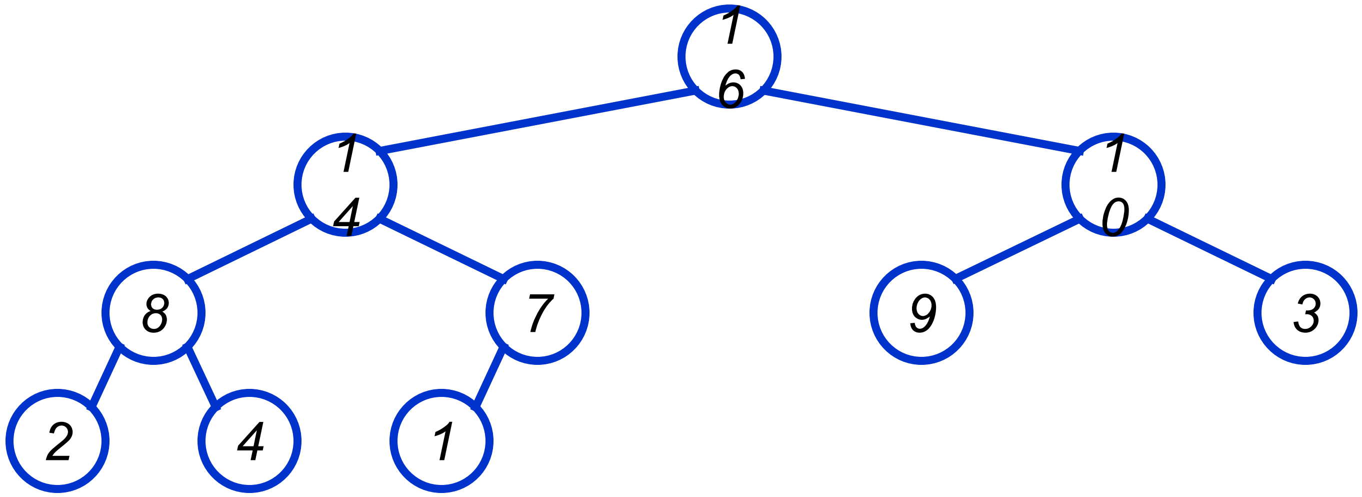
Heapify() Example





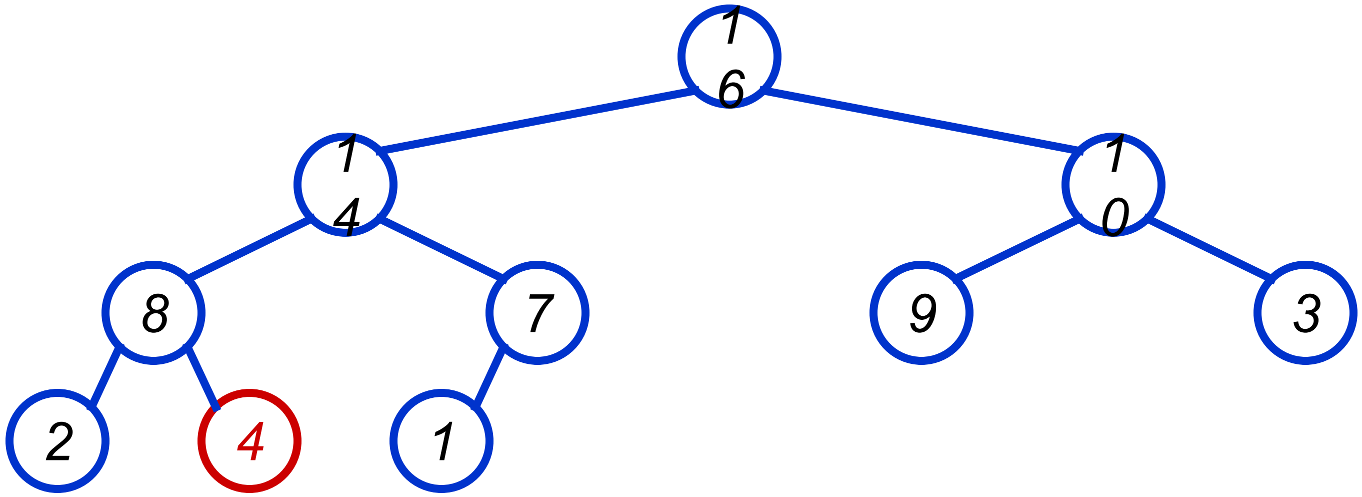
16	14	10	4	7	9	3	2	8	1

Heapify() Example



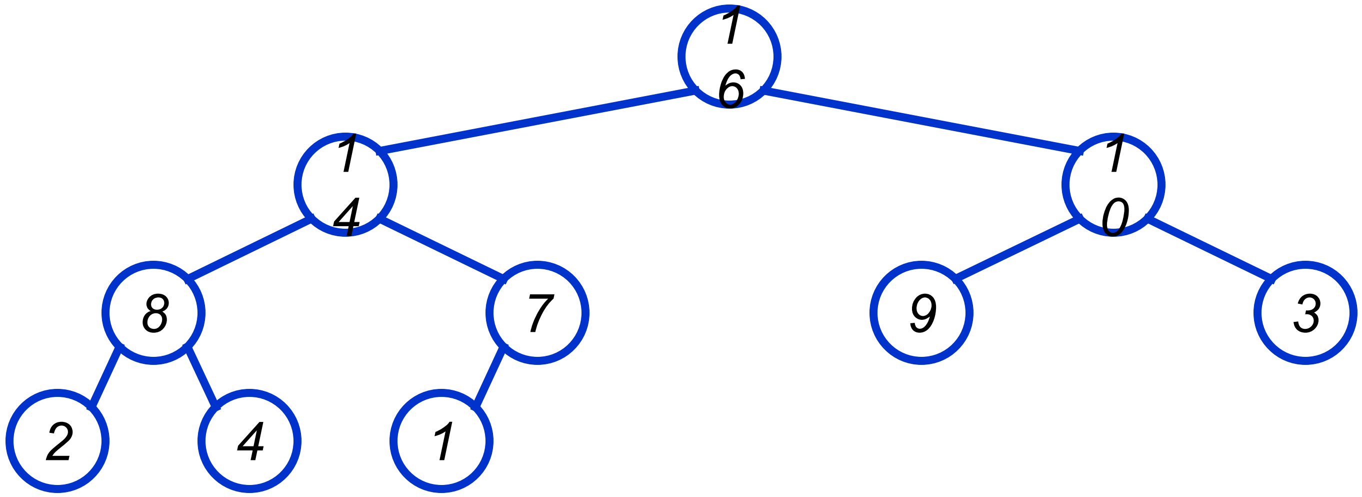
A =	16	14	10	8	7	9	3	2	4	1

Heapify() Example



A =	16	14	10	8	7	9	3	2	4	1

Heapify() Example



A =	16	14	10	8	7	9	3	2	4	1

Analyzing Heapify(): Informal

Aside from the recursive call, what is the running time of **`Heapify()`**?

How many times can **`Heapify()`** *recursively call itself?*

What is the worst-case running time of **`Heapify()`** *on a heap of size n?*

Analyzing Heapify(): Formal

Fixing up relationships between *i*, *l*, and *r* takes $\Theta(1)$ time

If the heap at i has n elements, how many elements can the subtrees at l or r have?

Draw it

Answer: $2n/3$ (worst case: bottom row 1/2 full)

So time taken by **`Heapify()`** is given by

$$T(n) \leq T(2n/3) + \Theta(1)$$

Analyzing Heapify(): Formal

So we have

$$T(n) \leq T(2n/3) + \Theta(1)$$

By case 2 of the Master Theorem,

$$T(n) = O(\lg n)$$

Thus, **`Heapify()`** takes linear time

Heap Operations: BuildHeap()

We can build a heap in a bottom-up manner by running **`Heapify()`** on successive subarrays

Fact: for array of length *n*, all elements in range A[$\lfloor n/2 \rfloor$ + 1 .. n] are heaps (*Why?*)

So:

- Walk backwards through the array from n/2 to 1, calling **`Heapify()`** on each node.
- Order of processing guarantees that the children of node *i* are heaps when *i* is processed

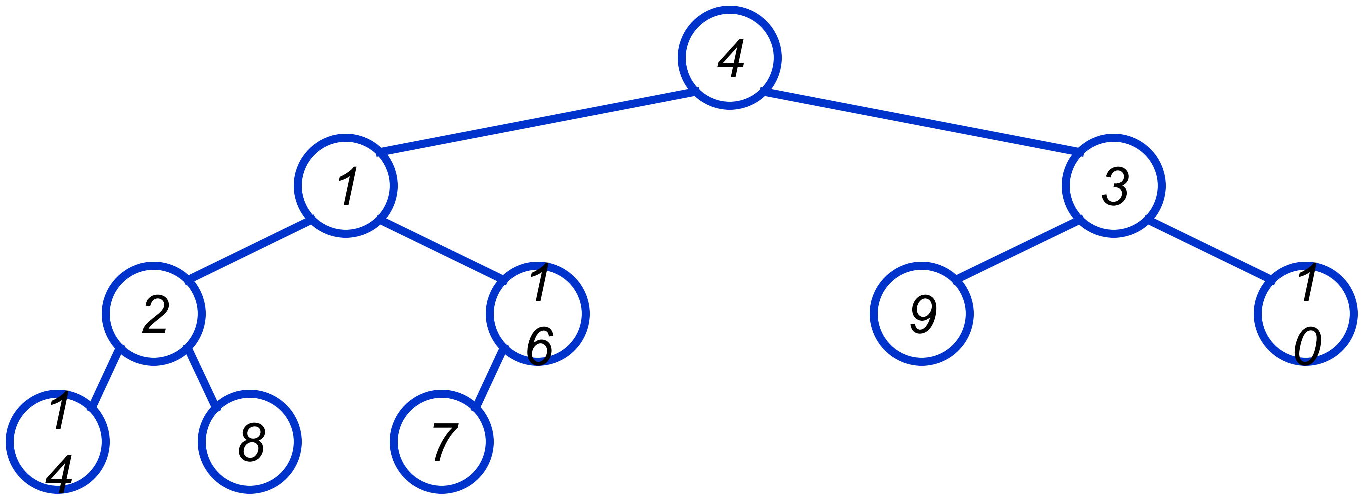
BuildHeap()

```
// given an unsorted array A, make A a heap
BuildHeap(A)
{
  heap_size(A) = length(A);
  for (i = ⌊length[A]/2⌋ downto 1)
   Heapify(A, i);
}
```

BuildHeap() Example

Work through example

A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}



Analyzing BuildHeap()

Each call to **`Heapify()`** takes O(lg n) time

There are O(n) such calls (specifically, $\lfloor n/2 \rfloor$)

Thus the running time is O(n lg n)

Is this a correct asymptotic upper bound?

Is this an asymptotically tight bound?

A tighter bound is O(n)

How can this be? Is there a flaw in the above reasoning?

Analyzing BuildHeap(): Tight

To **`Heapify()`** a subtree takes O(h) time where h is the height of the subtree

h = O(lg m), m = # nodes in subtree

The height of most subtrees is small

Fact: an n-element heap has at most $\lceil n/2^{h+1} \rceil$ nodes of height h

CLR 7.3 uses this fact to prove that **`BuildHeap()`** takes O(n) time

Heapsort

Given **`BuildHeap()`**, an in-place sorting algorithm is easily constructed:

Maximum element is at A[1]

Discard by swapping with element at A[n]

- Decrement heap_size[A]
- A[n] now contains correct value

Restore heap property at A[1] by calling **`Heapify()`**

Repeat, always swapping A[1] for A[heap_size(A)]

Heapsort

```
Heapsort(A)
{
   BuildHeap(A);
   for (i = length(A) downto 2)
   {
      Swap(A[1], A[i]);
      heap_size(A) -= 1;
      Heapify(A, 1);
   }
}
```

Analyzing Heapsort

The call to **`BuildHeap()`** takes O(n) time

Each of the n - 1 calls to **`Heapify()`** takes O(lg n) time

Thus the total time taken by **`HeapSort()`**
= O(n) + (n - 1) O(lg n)
= O(n) + O(n lg n)
= O(n lg n)

Priority Queues

Heapsort is a nice algorithm, but in practice Quicksort (coming up) usually wins

But the heap data structure is incredibly useful for implementing *priority queues*

A data structure for maintaining a set *S* of elements, each with an associated value or *key*

Supports the operations **`Insert()`**, **`Maximum()`**, and **`ExtractMax()`**

What might a priority queue be useful for?

Priority Queue Operations

Insert(S, x) inserts the element x into set S

Maximum(S) returns the element of S with the maximum key

ExtractMax(S) removes and returns the element of S with the maximum key

How could we implement these operations using a heap?

Sorting So Far

- Insertion sort:
  - Easy to code
  - Fast on small inputs (less than ~50 elements)
  - Fast on nearly-sorted inputs
  - $O(n^2)$ worst case
  - $O(n^2)$ average (equally-likely inputs) case
  - $O(n^2)$ reverse-sorted case

Sorting So Far

- Merge sort:
  - Divide-and-conquer:
    - Split array in half
    - Recursively sort subarrays
    - Linear-time merge step
  - O(n lg n) worst case
  - Doesn’t sort in place

Sorting So Far

- Heap sort:
  - Uses the very useful heap data structure
    - Complete binary tree
    - Heap property: parent key > children's keys
  - O(n lg n) worst case
  - Sorts in place
  - Fair amount of shuffling memory around

Sorting So Far

- Quick sort:
  - Divide-and-conquer:
    - Partition array into two subarrays, recursively sort
    - All of first subarray < all of second subarray
    - No merge step needed!
  - O(n lg n) average case
  - Fast in practice
  - O(n^2) worst case
    - Naïve implementation: worst case on sorted input
    - Address this with randomized quicksort

How Fast Can We Sort?

- We will provide a lower bound, then beat it
  - *How do you suppose we'll beat it?*
- First, an observation: all of the sorting algorithms so far are *comparison sorts*
  - The only operation used to gain ordering information about a sequence is the pairwise comparison of two elements
  - Theorem: all comparison sorts are Ω(n lg n)
    - A comparison sort must do O(n) comparisons (*why?*)
    - What about the gap between O(n) and O(n lg n)

Decision Trees

- *Decision trees* provide an abstraction of comparison sorts
  - A decision tree represents the comparisons made by a comparison sort. Every thing else ignored
  - (Draw examples on board)
- *What do the leaves represent?*
- *How many leaves must there be?*

Decision Trees

- Decision trees can model comparison sorts. For a given algorithm:
  - One tree for each *n*
  - Tree paths are all possible execution traces
  - *What's the longest path in a decision tree for insertion sort? For merge sort?*
- *What is the asymptotic height of any decision tree for sorting n elements?*
- Answer: $\Omega(n \lg n)$ (now let's prove it…)

Lower Bound For Comparison Sorting

- Thm: Any decision tree that sorts n elements has height $\Omega(n \lg n)$
- *What's the minimum # of leaves?*
- *What's the maximum # of leaves of a binary tree of height h?*
- Clearly the minimum # of leaves is less than or equal to the maximum # of leaves

Lower Bound For Comparison Sorting

- So we have…
  $n! \le 2^h$
- Taking logarithms:
  $\lg(n!) \le h$
- Stirling's approximation tells us:
  $$n! > \left(\frac{n}{e}\right)^n$$
- Thus: $h \ge \lg\left(\frac{n}{e}\right)^n$

Lower Bound For Comparison Sorting

- So we have $$h \geq \lg\left(\frac{n}{e}\right)^n$$

$$= n \lg n - n \lg e$$

$$= \Omega(n \lg n)$$

- Thus the minimum height of a decision tree is $\Omega(n \lg n)$

Lower Bound For Comparison Sorts

- Thus the time to comparison sort n elements is $\Omega(n \lg n)$
- Corollary: Heapsort and Mergesort are asymptotically optimal comparison sorts
- But the name of this lecture is “Sorting in linear time”!
  - *How can we do better than* $\Omega(n \lg n)$*?*

Sorting In Linear Time

- Counting sort
  - No comparisons between elements!
  - ***But***...depends on assumption about the numbers being sorted
    - We assume numbers are in the range *1.. k*
  - The algorithm:
    - Input: A[1..*n*], where A[j] ∈ {1, 2, 3, …, *k*}
    - Output: B[1..*n*], sorted (notice: not sorting in place)
    - Also: Array C[1..*k*] for auxiliary storage

Counting Sort

```
CountingSort(A, B, k)
    for i=1 to k
        C[i]= 0;
    for j=1 to n
        C[A[j]] += 1;
    for i=2 to k
        C[i] = C[i] + C[i-1];
    for j=n downto 1
        B[C[A[j]]] = A[j];
        C[A[j]] -= 1;
```

Work through example: A={4 1 3 4 3}, k = 4

Counting Sort

```
CountingSort(A, B, k)
    for i=1 to k
        C[i]= 0;
    for j=1 to n
        C[A[j]] += 1;
    for i=2 to k
        C[i] = C[i] + C[i-1];
    for j=n downto 1
        B[C[A[j]]] = A[j];
        C[A[j]] -= 1;
```

Takes time $O(k)$

Takes time $O(n)$

What will be the running time?

Counting Sort

- Total time: O(n + k)
  - Usually, k = O(n)
  - Thus counting sort runs in O(n) time
- But sorting is Ω(n lg n)!
  - No contradiction--this is not a comparison sort (in fact, there are *no* comparisons at all!)
  - Notice that this algorithm is *stable*

Counting Sort

- Cool! *Why don't we always use counting sort?*
- Because it depends on range *k* of elements
- *Could we use counting sort to sort 32 bit integers? Why or why not?*
- Answer: no, *k* too large ($2^{32} = 4{,}294{,}967{,}296$)

Counting Sort

- *How did IBM get rich originally?*
- Answer: punched card readers for census tabulation in early 1900's.
  - In particular, a *card sorter* that could sort cards into different bins
    - Each column can be punched in 12 places
    - Decimal digits use 10 places
  - Problem: only one column can be sorted on at a time

Radix Sort

- Intuitively, you might sort on the most significant digit, then the second msd, etc.
- Problem: lots of intermediate piles of cards (read: scratch arrays) to keep track of
- Key idea: sort the *least* significant digit first

```
RadixSort(A, d)
    for i=1 to d
        StableSort(A) on digit i
```

  - Example: Fig 9.3

Radix Sort

- *Can we prove it will work?*
- Sketch of an inductive argument (induction on the number of passes):
  - Assume lower-order digits {j: j<i}are sorted
  - Show that sorting next digit i leaves array correctly sorted
    - If two digits at position i are different, ordering numbers by that digit is correct (lower-order digits irrelevant)
    - If they are the same, numbers are already sorted on the lower-order digits. Since we use a stable sort, the numbers stay in the right order

Radix Sort

- *What sort will we use to sort on digits?*
- Counting sort is obvious choice:
  - Sort n numbers on digits that range from $1..k$
  - Time: $O(n + k)$
- Each pass over n numbers with d digits takes time $O(n+k)$, so total time $O(dn+dk)$
  - When d is constant and $k=O(n)$, takes $O(n)$ time
- *How many bits in a computer word?*

Radix Sort

- Problem: sort 1 million 64-bit numbers
  - Treat as four-digit radix 2^{16} numbers
  - Can sort in just four passes with radix sort!
- Compares well with typical O(n lg n) comparison sort
  - Requires approx lg n = 20 operations per number being sorted
- *So why would we ever use anything but radix sort?*

Radix Sort

- In general, radix sort based on counting sort is
  - Fast
  - Asymptotically fast (i.e., O(n))
  - Simple to code
  - A good choice
- To think about: *Can radix sort be used on floating-point numbers?*

Review: Radix Sort

- *How did IBM get rich originally?*
- Answer: punched card readers for census tabulation in early 1900's.
  - In particular, a *card sorter* that could sort cards into different bins
    - Each column can be punched in 12 places
    - Decimal digits use 10 places
  - Problem: only one column can be sorted on at a time

Radix Sort

- *What sort will we use to sort on digits?*
- Counting sort is obvious choice:
  - Sort n numbers on digits that range from $1..k$
  - Time: $O(n + k)$
- Each pass over n numbers with d digits takes time $O(n+k)$, so total time $O(dn+dk)$
  - When d is constant and $k=O(n)$, takes $O(n)$ time
- *How many bits in a computer word?*

Radix Sort

- Problem: sort 1 million 64-bit numbers
  - Treat as four-digit radix 2^{16} numbers
  - Can sort in just four passes with radix sort!
- Compares well with typical $O(n \lg n)$ comparison sort
  - Requires approx $\lg n = 20$ operations per number being sorted
- *So why would we ever use anything but radix sort?*

Radix Sort

- In general, radix sort based on counting sort is
  - Fast
  - Asymptotically fast (i.e., O(n))
  - Simple to code
  - A good choice
- To think about: *Can radix sort be used on floating-point numbers?*

Summary: Radix Sort

- Radix sort:
  - Assumption: input has d digits ranging from 0 to k
  - Basic idea:
    - Sort elements by digit starting with *least* significant
    - Use a stable sort (like counting sort) for each stage
  - Each pass over n numbers with d digits takes time $O(n+k)$, so total time $O(dn+dk)$
    - When d is constant and $k=O(n)$, takes $O(n)$ time
  - Fast! Stable! Simple!
  - Doesn't sort in place

Bucket Sort

- Bucket sort
  - Assumption: input is *n* reals from [0, 1)
  - Basic idea:
    - Create *n* linked lists (*buckets*) to divide interval [0,1) into subintervals of size 1/*n*
    - Add each input element to appropriate bucket and sort buckets with insertion sort
  - Uniform input distribution $\Rightarrow$ O(1) bucket size
    - Therefore the expected total time is O(n)
  - These ideas will return when you will see *hash tables (not in this course)*

Order Statistics

- The *i*th *order statistic* in a set of *n* elements is the *i*th smallest element
- The *minimum* is thus the 1st order statistic
- The *maximum* is (duh) the *n*th order statistic
- The *median* is the *n*/2 order statistic
  - If *n* is even, there are 2 medians
- *How can we calculate order statistics?*
- *What is the running time?*

Order Statistics

- *How many comparisons are needed to find the minimum element in a set? The maximum?*
- *Can we find the minimum and maximum with less than twice the cost?*
- Yes:
  - Walk through elements by pairs
    - Compare each element in pair to the other
    - Compare the largest to maximum, smallest to minimum
  - Total cost: 3 comparisons per 2 elements = O(3n/2)

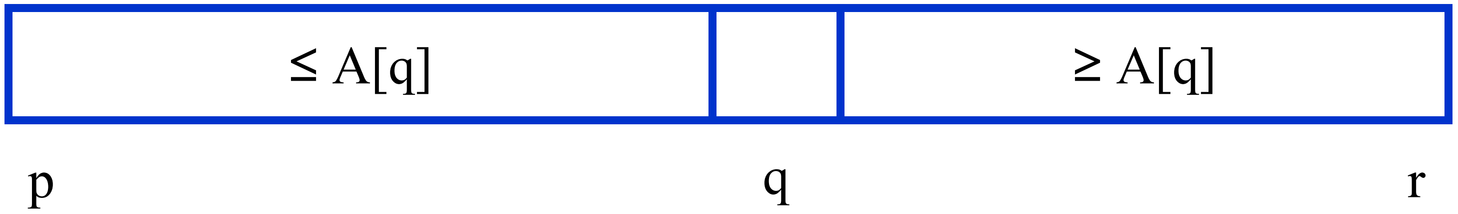
Finding Order Statistics: The Selection Problem

- A more interesting problem is *selection*: finding the *i*th smallest element of a set
- We will show:
  - A practical randomized algorithm with O(n) expected running time
  - A cool algorithm of theoretical interest only with O(n) worst-case running time

Randomized Selection

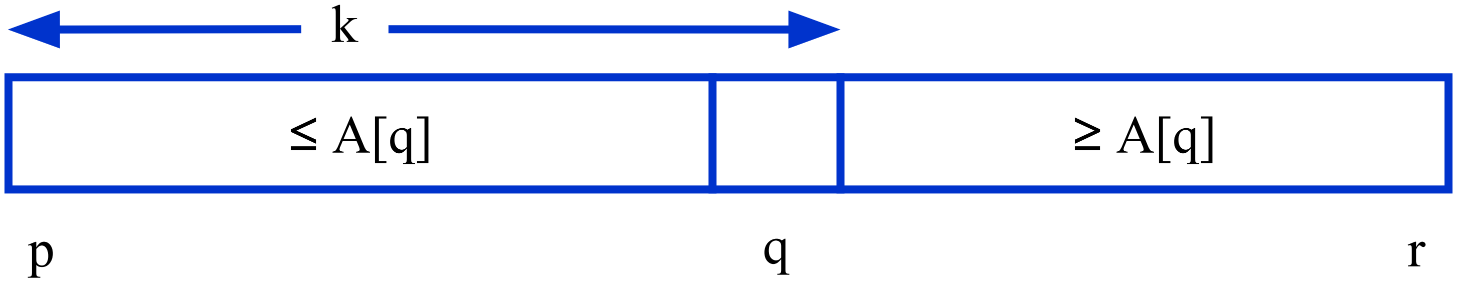
- Key idea: use partition() from quicksort
  - But, only need to examine one subarray
  - This savings shows up in running time: O(n)
- We will again use a slightly different partition than the book:

  q = RandomizedPartition(A, p, r)



Randomized Selection

```
RandomizedSelect(A, p, r, i)
    if (p == r) then return A[p];
    q = RandomizedPartition(A, p, r)
    k = q - p + 1;
    if (i == k) then return A[q];    // not in book
    if (i < k) then
        return RandomizedSelect(A, p, q-1, i);
    else
        return RandomizedSelect(A, q+1, r, i-k);
```



Randomized Selection

- Analyzing **`RandomizedSelect()`**
  - Worst case: partition always 0:n-1

    T(n) = T(n-1) + O(n) *= ???*

    = O(n^2) (arithmetic series)

    - No better than sorting!
  - "Best" case: suppose a 9:1 partition

    T(n) = T(9n/10) + O(n) *= ???*

    = O(n) (Master Theorem, case 3)

    - Better than sorting!
    - *What if this had been a 99:1 split?*

Randomized Selection

- Average case
  - For upper bound, assume *i*th element always falls in larger side of partition:

$$T(n) \le \frac{1}{n}\sum_{k=0}^{n-1} T(\max(k, n-k-1)) + \Theta(n)$$

$$\le \frac{2}{n}\sum_{k=n/2}^{n-1} T(k) + \Theta(n)$$

What happened here?

  - Let's show that T(*n*) = O(*n*) by substitution

Randomized Selection

- Assume T(n) ≤ cn for sufficiently large c:

$$T(n) \le \frac{2}{n}\sum_{k=n/2}^{n-1} T(k) + \Theta(n)$$

The recurrence we started with

What happened here?

$$\le \frac{2}{n}\sum_{k=n/2}^{n-1} ck + \Theta(n)$$

Substitute T(n) ≤ cn for T(k)

What happened here?

$$= \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k\right) + \Theta(n)$$

"Split" the recurrence

Expand arithmetic series

$$= \frac{2c}{n}\left(\frac{1}{2}(n-1)n - \frac{1}{2}\left(\frac{n}{2}-1\right)\frac{n}{2}\right) + \Theta(n)$$

What happened here?

What happened here?

$$= c(n-1) - \frac{c}{2}\left(\frac{n}{2}-1\right) + \Theta(n)$$

Multiply it out

Randomized Selection

- Assume T(*n*) ≤ *cn* for sufficiently large *c*:

$$T(n) \le c(n-1) - \frac{c}{2}\left(\frac{n}{2}-1\right) + \Theta(n)$$

The recurrence so far

What happened here?

$$= cn - c - \frac{cn}{4} + \frac{c}{2} + \Theta(n)$$

Multiply it out

What happened here?

$$= cn - \frac{cn}{4} - \frac{c}{2} + \Theta(n)$$

Subtract c/2

What happened here?

$$= cn - \left(\frac{cn}{4} + \frac{c}{2} - \Theta(n)\right)$$

Rearrange the arithmetic

$$\le cn \quad \text{(if c is big enough)}$$

What happened here?

What we set out to prove

Worst-Case Linear-Time Selection

- Randomized algorithm works well in practice
- What follows is a worst-case linear time algorithm, really of theoretical interest only
- Basic idea:
  - Generate a good partitioning element
  - Call this element *x*

Worst-Case Linear-Time Selection

- The algorithm in words:
  1. Divide *n* elements into groups of 5
  2. Find median of each group (*How? How long?*)
  3. Use Select() recursively to find median *x* of the $\lfloor n/5 \rfloor$ medians
  4. Partition the *n* elements around *x*. Let *k* = rank(*x*)
  5. **if** (i == k) **then** return x

     if (i < k) **then** use Select() recursively to find *i*th smallest element in first partition

     else (i > k) use Select() recursively to find (*i*-*k*)th smallest element in last partition

Worst-Case Linear-Time Selection

- (Sketch situation on the board)
- *How many of the 5-element medians are ≤ x?*
  - At least 1/2 of the medians = $\lfloor \lfloor n/5 \rfloor / 2 \rfloor = \lfloor n/10 \rfloor$
- *How many elements are ≤ x?*
  - At least $3 \lfloor n/10 \rfloor$ elements
- For large n, $3 \lfloor n/10 \rfloor \geq n/4$ *(How large?)*
- So at least $n/4$ elements ≤ x
- Similarly: at least $n/4$ elements ≥ x

Worst-Case Linear-Time Selection

- After partitioning around x, step 5 will call Select() on <= $3n/4$ elements
- The recurrence is therefore:

$$T(n) \le T(\lfloor n/5 \rfloor) + T(3n/4) + \Theta(n)$$

$$\le T(n/5) + T(3n/4) + \Theta(n)$$

$$\le cn/5 + 3cn/4 + \Theta(n)$$

Substitute T(n) = cn

$$= 19cn/20 + \Theta(n)$$

Combine fractions

$$= cn - (cn/20 - \Theta(n))$$

Express in desired form

$$\le cn \quad \text{if } c \text{ is big enough}$$

What we set out to prove

Worst-Case Linear-Time Selection

- Intuitively:
  - Work at each level is a constant fraction (19/20) smaller
    - Geometric progression!
  - Thus the O(n) work at the root dominates

Linear-Time Median Selection

- Given a "black box" O(n) median algorithm, what can we do?
  - *i*th order statistic:
    - Find median *x*
    - Partition input around *x*
    - if ($i \leq (n+1)/2$) recursively find *i*th element of first half
    - else find ($i - (n+1)/2$)th element in second half
    - $T(n) = T(n/2) + O(n) = O(n)$
  - *Can you think of an application to sorting?*

Linear-Time Median Selection

- Worst-case O(n lg n) quicksort
  - Find median *x* and partition around it
  - Recursively quicksort two halves
  - $T(n) = 2T(n/2) + O(n) = O(n \lg n)$

So ....

We're done with sorting (phew!)

From now on work on more general problems

Concentrate on alg. design techniques

Next on the agenda: Greedy Algorithms.