CS 361
Data Structures & Algs
Lecture 9

Prof. Tom Hayes
University of New Mexico
09-21-2010

Today

Orderings

Searching

Sorting

Priority Queues & Heaps

Order Relation

We say a binary relation R is an “order relation” if it is

(1) transitive a R b and b R c implies a R c

and (2) antisymmetric a R b implies not(b R a).

example: <   >   “is a prefix of”
“is younger than”   “is a descendent of”
“is a superclass of”

Total Ordering

An order relation R is total if, for every a, b, either a=b, a R b or b R a. “trichotomy”

Example: < on real numbers.

Otherwise: partial ordering. example: “is an ancestor of”. “came to class before”. (Why are these orderings only partial?)

Sorting

Prerequisites: a total ordering, R.

Input: an unordered collection of things.

Output: a list containing the same things, now in order

(A[0] R A[1] R A[2] R ... R A[n-1])

How to Sort

Basic idea: Divide and Conquer!

(a) QuickSort. Choose a "pivot" P = A[i]. Split the rest of A into 2 sides: less than P, and more than P. Place these sides in the correct order: (< P) P (> P). Finally, recursively sort the two sides.

(b) MergeSort. Split the list into two equal parts. Recursively sort each part. Finally, "splice" them together in O(n) time.

Analysis of MergeSort

Let $T(n)$ denote the (worst case) running time on an array of length n.

$T(n) \leq 2T(n/2) + C\,n$ (recurrence relation)

Analysis of MergeSort

Let $T(n)$ denote the (worst case) running time on an array of length n.

$T(n) \le 2T(n/2) + C\, n$ (recurrence relation)

$\le 2(2T(n/4) + C(n/2)) + C\, n)$

$= 4\, T(n/4) + 2\, C\, n$

$\le 4(2T(n/8) + C(n/4)) + 2\, C\, n$

$= 8\, T(n/8) + 3\, C\, n$

... $= n\, T(1) + \log(n)\, C\, n = O(n \log n)$.

“Analysis” of QuickSort

How big do we expect the side “< P” to be, typically?

“Analysis” of QuickSort

How big do we expect the side “< P” to be, typically?

If it were always close to (n/2), then we would expect to get the same recurrence as for MergeSort:

$T(n) \leq 2\ T(n/2) + C\ n.$

So, the solution would be the same,
$O(n \log n)$ “nearly linear time”

Analysis of QuickSort

Let $T(n)$ = worst-case running time of QuickSort.

$T(n) \leq C\,n + T(\text{left side}) + T(\text{right side})$.

left side + right side = n-1. So, $T(\text{left side}) + T(\text{right side}) \leq T(n-1)$. This case can happen! (Pivot could be max or min elt.)

$T(n) \leq C\,n + T(n-1)$. Unroll this recurrence.

$T(n) = O(n^2)$. (Actually, also $\Omega(n^2)$.)

QuickSort Rmks

Although worst-case performance is quadratic, QuickSort tends to perform very well in practice.

(a) With randomized pivot, average running time is provably O(n log n).

(b) Practical advantages over MergeSort: sorting in place, improved locality of reference (good for memory caching).

Searching

Say we want to store a set of data, such as UNM student names. How quickly can we check whether an input equals the name of a student?

Searching

Say we want to store a set of data, such as UNM student names. How quickly can we check whether an input equals the name of a student?

Goal: O(log N), where N is the total number of students.

Method: Binary search!

Binary Search

Prerequisite: Totally Ordered Data.

Store Data in a Sorted Array.

Let $T(N)$ = worst-case time to test a name.

Recurrence: $T(N) \leq C + T(N/2)$.

unroll!

Binary Search

Prerequisite: Totally Ordered Data.

Store Data in a Sorted Array.

Let $T(N)$ = worst-case time to test a name.

Recurrence: $T(N) \leq C + T(N/2)$.

$\leq 2\ C + T(N/4)$

$\leq 3\ C + T(N/8) \leq \ldots \leq (\log N)\ C + T(1)$

$= O(\log N)$.

Summary

With a total ordering, can sort in time O(N log N). In a sorted list, can search in time O(log N).

Variant: Binary Search Tree

BST is a data structure providing the following operations:

add

remove

is_element

Goal: These 3 operations run fast!

Variant: Binary Search Tree

Data structure:

```
class Node<T> {
    T data;
    Node leftChild, rightChild;
}
```

Recall: these are “references” or “pointers”

Invariant: leftChild.data < data < rightChild.data

Variant: Binary Search Tree

Time for these operations:

add

remove

is_element

All proportional to depth(tree).

Ideally, log(N). “complete” binary tree.

Problem: may become unbalanced.

Priority Queues

Stores a collection of data

- Each data has a numeric “key value”
- operations supported: add, delete, extract_min.
- guarantees: O(log n) time per operation
- n = current size of collection.

Where does the name come from?

The Name

Queue: operations add, delete, get_next

first-in, first-out (FIFO) data flow.

Often implemented as linked list.

Priority Queue: whenever-in, highest-priority first-out (WHiPFO) data flow.

(A priority queue is not a kind of queue, in the standard sense)

In Java

java.util.PriorityQueue

(look at API)

Implementing a PQ

Idea 1: Maintain a sorted list

Then, to extract_min, just need to grab the first element, and point to the second one.

What will it cost to insert an element?

Implementing a PQ

Idea 1: Maintain a sorted list

Takes $\Omega(n)$ time to find/add/delete. No good.

Idea 2: Maintain a sorted array.

Implementing a PQ

Idea 1: Maintain a sorted list

Takes $\Omega(n)$ time to find/add/delete. No good.

Idea 2: Maintain a sorted array.

Can use binary search to find, but add/delete still take $\Omega(n)$ time. No good!

Implementing a PQ

Idea 1: Maintain a sorted list

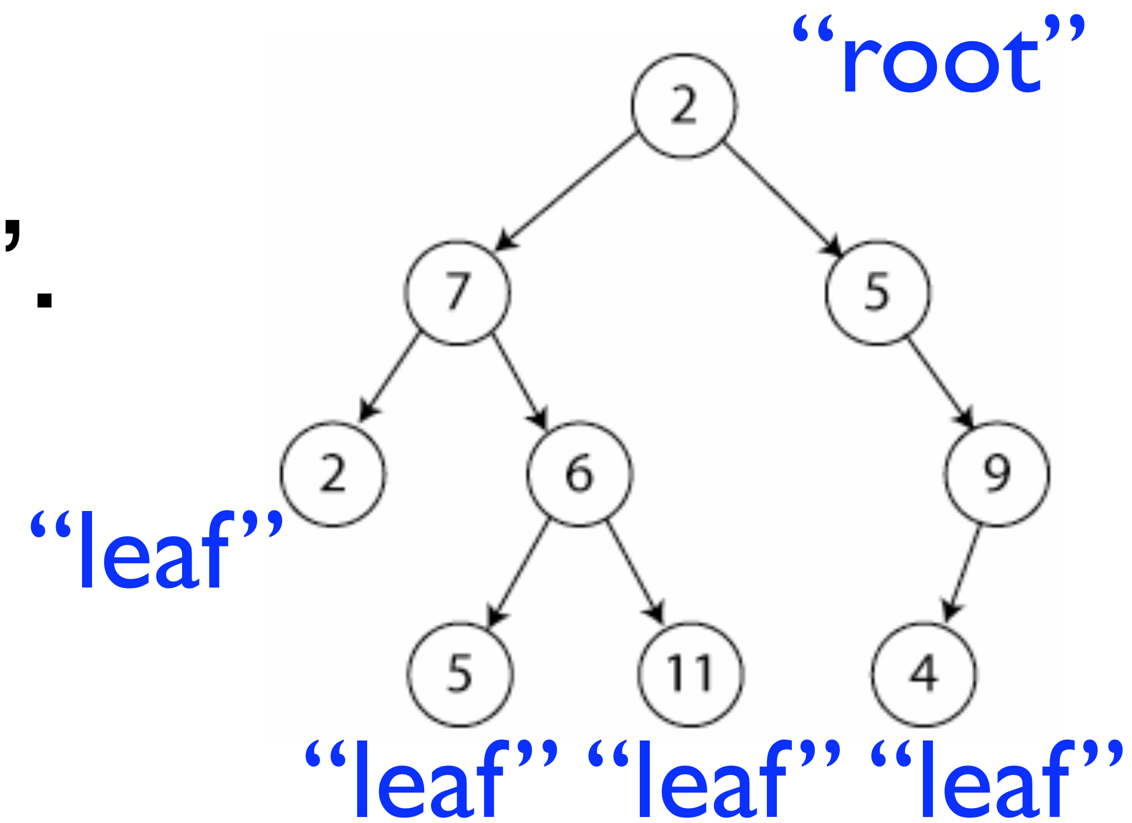
Takes $\Omega(n)$ time to find/add/delete. No good.

Idea 2: Maintain a sorted array.

Can use binary search to find, but add/delete still take $\Omega(n)$ time. No good!

Idea 3: Special kind of tree called a “heap”

Binary Trees



Data is stored in "nodes".

Each node has 4 fields:

- data
- parent (either a ref to a node, or "null")
- left_child (reference to a node or null)
- right_child (reference to a node or null)

Heap Order Property

We say a tree storing “key” values satisfies the (min-) heap order property if it is always the case that the parent of a node stores a value ≤ than the node does.

Heap Order Property

We say a tree storing “key” values satisfies the (min-) heap order property if it is always the case that the parent of a node stores a value ≤ than the node does.

Such a tree is called a “heap.”

Heap Order Property

We say a tree storing "key" values satisfies the (min-) heap order property if it is always the case that the parent of a node stores a value ≤ than the node does.

Such a tree is called a "heap."

A heap is called "balanced" if every layer except perhaps the bottom one, has the maximum possible number of nodes.

Heap Order Property

We say a tree storing "key" values satisfies the (min-) heap order property if it is always the case that the parent of a node stores a value ≤ than the node does.

Such a tree is called a "heap."

A heap is called "balanced" if every layer except perhaps the bottom one, has the maximum possible number of nodes.

Q: What is this number?

Heap Order Property

We say a tree storing “key” values satisfies the (min-) heap order property if it is always the case that the parent of a node stores a value ≤ than the node does.

Such a tree is called a “heap.”

A heap is called “balanced” if every layer except perhaps the bottom one, has the maximum possible number of nodes.

Q: What is this number? 2^L for the L’th level away from the root. (“depth L”)

Heap Order Property

What can we do with a heap?

Can we search it quickly?

Heap Order Property

What can we do with a heap?

Can we search it quickly?

No. (whiteboard)

Heap Order Property

What can we do with a heap?

Can we search it quickly?

No. (whiteboard)

Can we extract_min quickly?

Heap Order Property

What can we do with a heap?

Can we search it quickly?

No. (whiteboard)

Can we extract_min quickly?

Well, we can find_min quickly. But if we extract it, we will have to replace the root.

Heap Order Property

What can we do with a heap?

Can we search it quickly?

No. (whiteboard)

Can we extract_min quickly?

Well, we can find_min quickly. But if we extract it, we will have to replace the root.

What about adding an element?

Heap Order Property

What can we do with a heap?

Can we search it quickly?

No. (whiteboard)

Can we extract_min quickly?

Well, we can find_min quickly. But if we extract it, we will have to replace the root.

What about adding an element? Stick it in a leaf.

Heap Order Property

What can we do with a heap?

Can we search it quickly?

No. (whiteboard)

Can we extract_min quickly?

Well, we can find_min quickly. But if we extract it, we will have to replace the root.

What about adding an element? Stick it in a leaf. But it might violate the order property!

Reading: Heapify-Up, Heapify-Down

Goal: Fix a near-heap that has just one value out of place.

If it's smaller than its parent, swap it with its parent. Recurse! (Heapify-up)

If it's bigger than a child, swap it with the smaller child. Recurse! (Heapify-down)

Applet at http://people.ksp.sk/~kuko/bak/index.html

Next P.A.

A 2-sided Priority Queue.

Can extract_min and extract_max, both in time O(log N).

Design: How can this be achieved?

Don’t Forget: Quiz on Thursday. Finish HW!