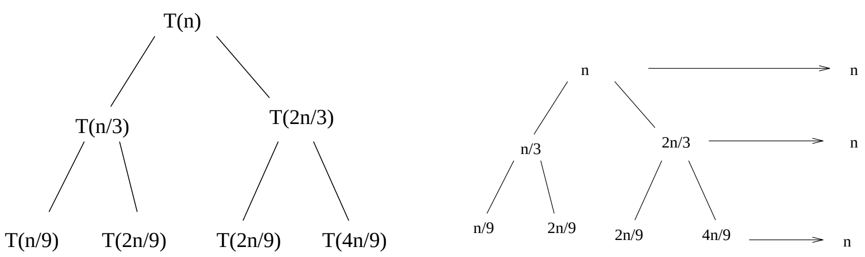
4.2-2 Argue the solution to

$$T(n) = T(n/3) + T(2n/3) + n$$

is $\Omega(n \lg n)$ by appealing to the recursion tree.

Draw the recursion tree.



How many levels does the tree have? This is equal to the longest path from the root to a leaf.

The shortest path to a leaf occurs when we take the heavy branch each time. The height k is given by $n(1/3)^k \leq 1$, meaning $n \leq 3^k$ or $k \geq \lg_3 n$.

The longest path to a leaf occurs when we take the light branch each time. The height k is given by $n(2/3)^k \leq 1$, meaning $n \leq (3/2)^k$ or $k \geq \lg_{3/2} n$.

The problem asks to show that $T(n) = \Omega(n \lg n)$, meaning we are looking for a lower bound

On any *full* level, the additive terms sums to n. There are $\log_3 n$ full levels. Thus $T(n) \geq n \log_3 n = \Omega(n \lg n)$

4.2-4 Use iteration to solve $T(n) = T(n-a) + T(a) + n$, where $a \geq 1$ is a constant.

Note iteration is backsubstitution.

$$
\begin{aligned}
T(n) &= T(n-a) + T(a) + n \\
&= (T(n-2a) + T(a) + n - a) + T(a) + n \\
&= (T(n-3a) + T(a) + n - 2a) + 2T(a) + 2n - 3a \\
&\vdots \\
&\approx \sum_{i=0}^{n/a} T(a) + \sum_{i=0}^{n/a} n - ia \\
&\approx (n/a)T(a) + \sum_{i=0}^{n/a} n - a \sum_{i=0}^{n/a} i \\
&\approx (n/a)T(a) + n \sum_{i=0}^{n/a} 1 - a \sum_{i=0}^{n/a} i \\
&\approx (n/a)T(a) + n(n/a) - a(n/a)^2/2
\end{aligned}
$$

Why don't CS profs ever stop talking about sorting?!

1. Computers spend more time sorting than anything else, historically 25% on mainframes.

2. Sorting is the best studied problem in computer science, with a variety of different algorithms known.

3. Most of the interesting ideas we will encounter in the course can be taught in the context of sorting, such as divide-and-conquer, randomized algorithms, and lower bounds.

You should have seen most of the algorithms - we will concentrate on the analysis.

Applications of Sorting

One reason why sorting is so important is that once a set of items is sorted, many other problems become easy.

Searching

Binary search lets you test whether an item is in a dictionary in $O(\lg n)$ time.

Speeding up searching is perhaps the most important application of sorting.

Closest pair

Given n numbers, find the pair which are closest to each other.

Once the numbers are sorted, the closest pair will be next to each other in sorted order, so an $O(n)$ linear scan completes the job.

Element uniqueness

Given a set of n items, are they all unique or are there any duplicates?

Sort them and do a linear scan to check all adjacent pairs.

This is a special case of closest pair above.

Frequency distribution – Mode

Given a set of n items, which element occurs the largest number of times?

Sort them and do a linear scan to measure the length of all adjacent runs.

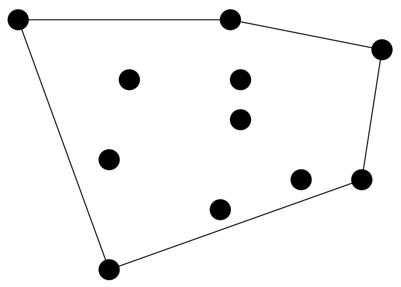
Median and Selection

What is the kth largest item in the set?

Once the keys are placed in sorted order in an array, the kth largest can be found in constant time by simply looking in the kth position of the array.

Convex hulls

Given n points in two dimensions, find the smallest area polygon which contains them all.



The convex hull is like a rubber band stretched over the points.

Convex hulls are the most important building block for more sophisticated geometric algorithms.

Once you have the points sorted by x-coordinate, they can be inserted from left to right into the hull, since the rightmost point is always on the boundary.

Without sorting the points, we would have to check whether the point is inside or outside the current hull.

Adding a new rightmost point might cause others to be deleted.

Huffman codes

If you are trying to minimize the amount of space a text file is taking up, it is silly to assign each letter the same length (ie. one byte) code.

Example: e is more common than q, a is more common than z.

If we were storing English text, we would want a and e to have shorter codes than q and z.

To design the best possible code, the first and most important step is to sort the characters in order of frequency of use.

Character	Frequency	Code
f	5	1100
e	9	1101
c	12	100
b	13	101
d	16	111
a	45	0

Selection Sort

A simple $O(n^2)$ sorting algorithm is selection sort.

Sweep through all the elements to find the smallest item, then the smallest remaining item, etc. until the array is sorted.

```
Selection-sort(A)
      for i = 1 to n
            for j = i + 1 to n
                  if (A[j] < A[i]) then swap(A[i],A[j])
```

It is clear this algorithm must be correct from an inductive argument, since the ith element is in its correct position.

It is clear that this algorithm takes $O(n^2)$ time.

It is clear that the analysis of this algorithm cannot be improved because there will be $n/2$ iterations which will require at least $n/2$ comparisons each, so at least $n^2/4$ comparisons will be made. More careful analysis doubles this.

Thus selection sort runs in $\Theta(n^2)$ time.

Binary Heaps

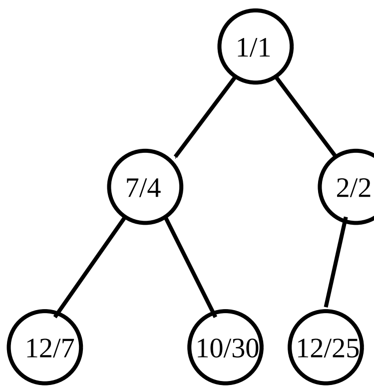
A *binary heap* is defined to be a binary tree with a key in each node such that:

1. All leaves are on, at most, two adjacent levels.

2. All leaves on the lowest level occur to the left, and all levels except the lowest one are completely filled.

3. The key in root is $\geq$ all its children, and the left and right subtrees are again binary heaps.

Conditions 1 and 2 specify shape of the tree, and condition 3 the labeling of the tree.



The ancestor relation in a heap defines a *partial order* on its elements, which means it is reflexive, anti-symmetric, and transitive.

1. *Reflexive:* x is an ancestor of itself.

2. *Anti-symmetric:* if x is an ancestor of y and y is an ancestor of x, then $x = y$.

3. *Transitive:* if x is an ancestor of y and y is an ancestor of z, x is an ancestor of z.

Partial orders can be used to model heirarchies with incomplete information or equal-valued elements. One of my favorite games with my parents is fleshing out the partial order of "big" old-time movie stars.

The partial order defined by the heap structure is weaker than that of the total order, which explains

1. Why it is easier to build.

2. Why it is less useful than sorting (but still very important).

Constructing Heaps

Heaps can be constructed incrementally, by inserting new elements into the left-most open spot in the array.

If the new element is greater than its parent, swap their positions and recur.

Since at each step, we replace the root of a subtree by a larger one, we preserve the heap order.

Since all but the last level is always filled, the height h of an n element heap is bounded because:

$$\sum_{i=1}^{h} 2^i = 2^{h+1} - 1 \geq n$$

so $h = \lfloor \lg n \rfloor$.

Doing n such insertions takes $\Theta(n \log n)$, since the last $n/2$ insertions require $O(\log n)$ time each.

Heapify

The bottom up insertion algorithm gives a good way to build a heap, but Robert Floyd found a better way, using a *merge* procedure called *heapify*.

Given two heaps and a fresh element, they can be merged into one by making the new one the root and trickling down.

```
Build-heap(A)
        n = |A|
        For i = ⌊n/2⌋ to 1 do
                Heapify(A,i)


Heapify(A,i)
        left = 2i
        right = 2i + 1
        if (left ≤ n) and (A[left] > A[i]) then
                max = left
                else max = i
        if (right ≤ n) and (A(right] > A[max]) then
                max = right
        if (max ≠ i) then
                swap(A[i],A[max])
                Heapify(A,max)
```

Rough Analysis of Heapify

Heapify on a subtree containing n nodes takes

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

The 2/3 comes from merging heaps whose levels differ by one. The last row could be exactly half filled. Besides, the asymptotic answer won't change so long the fraction is less than one.

Solve the recurrence using the Master Theorem.

Let $a = 1$, $b = 3/2$ and $f(n) = 1$.

Note that $\Theta(n^{\log_{3/2} 1}) = \Theta(1)$, since $\log_{3/2} 1 = 0$.

Thus Case 2 of the Master theorem applies.

The Master Theorem: Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n)$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows:

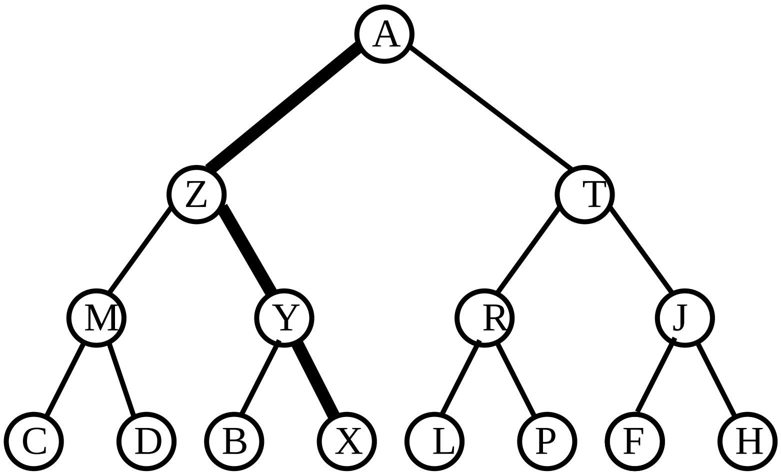
1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.

3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$, and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Exact Analysis of Heapify

In fact, Heapify performs better than $O(n \log n)$, because most of the heaps we merge are extremely small.



In a full binary tree on n nodes, there are $n/2$ nodes which are leaves (i.e. height 0), $n/4$ nodes which are height 1, $n/8$ nodes which are height 2, . . .

In general, there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h, so the cost of building a heap is:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil O(h) = O(n \sum_{h=0}^{\lfloor \lg n \rfloor} h/2^h)$$

Since this sum is not quite a geometric series, we can't apply the usual identity to get the sum. But it should be clear that the series converges.

Proof of Convergence

Series convergence is the "free lunch" of algorithm analysis.

The identify for the sum of a geometric series is

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

If we take the derivative of both sides, . . .

$$\sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2}$$

Multiplying both sides of the equation by x gives the identity we need:

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

Substituting $x = 1/2$ gives a sum of 2, so Build-heap uses at most $2n$ comparisons and thus linear time.

The Lessons of Heapsort, I

"Are we doing a careful analysis? Might our algorithm be faster than it seems?"

Typically in our analysis, we will say that since we are doing at most x operations of at most y time each, the total time is $O(xy)$.

However, if we overestimate too much, our bound may not be as tight as it should be!

Heapsort

Heapify can be used to construct a heap, using the observation that an isolated element forms a heap of size 1.

```
Heapsort(A)
      Build-heap(A)
      for i = n to 1 do
            swap(A[1],A[i])
            n = n - 1
            Heapify(A,1)
```

If we construct our heap from bottom to top using Heapify, we do not have to do anything with the last $n/2$ elements.

With the implicit tree defined by array positions, (i.e. the ith position is the parent of the $2i$th and $(2i+1)$st positions) the leaves start out as heaps.

Exchanging the maximum element with the last element and calling heapify repeatedly gives an $O(n \lg n)$ sorting algorithm, named *Heapsort*.

Heapsort Animations

The Lessons of Heapsort, II

Always ask yourself, "Can we use a different data structure?"

Selection sort scans throught the entire array, repeatedly finding the smallest remaining element.

For $i = 1$ to n
A: Find the smallest of the first $n - i + 1$ items.
B: Pull it out of the array and put it first.

Using arrays or unsorted linked lists as the data structure, operation A takes $O(n)$ time and operation B takes $O(1)$.

Using heaps, both of these operations can be done within $O(\lg n)$ time, balancing the work and achieving a better tradeoff.

Priority Queues

A *priority queue* is a data structure on sets of keys supporting the following operations:

- *Insert(S, x)* - insert x into set S
- *Maximum(S)* - return the largest key in S
- *ExtractMax(S)* - return and remove the largest key in S

These operations can be easily supported using a heap.

- *Insert* - use the trickle up insertion in $O(\log n)$.
- *Maximum* - read the first element in the array in $O(1)$.
- *Extract-Max* - delete first element, replace it with the last, decrement the element counter, then heapify in $O(\log n)$.

Applications of Priority Queues

Heaps as stacks or queues

- In a stack, *push* inserts a new item and *pop* removes the most recently pushed item.
- In a queue, *enqueue* inserts a new item and *dequeue* removes the least recently enqueued item.

Both stacks and queues can be simulated by using a heap, when we add a new *time* field to each item and order the heap according it this time field.

- To simulate the stack, increment the time with each insertion and put the maximum on top of the heap.
- To simulate the queue, decrement the time with each insertion and put the maximum on top of the heap (or increment times and keep the minimum on top)

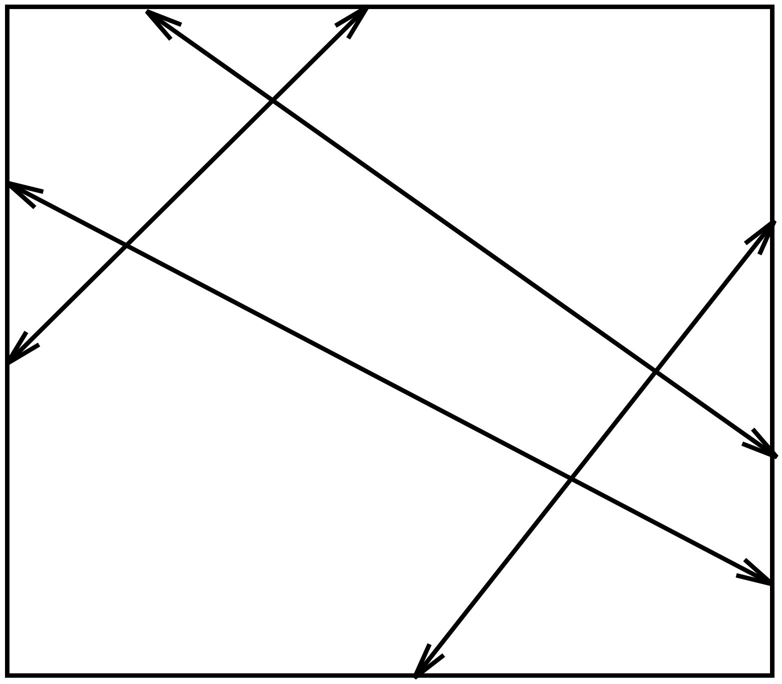
This simulation is not as efficient as a normal stack/queue implementation, but it is a cute demonstration of the flexibility of a priority queue.

Discrete Event Simulations

In simulations of airports, parking lots, and jai-alai – priority queues can be used to maintain who goes next.

The stack and queue orders are just special cases of orderings. In real life, certain people cut in line.

Sweepline Algorithms in Computational Geometry



In the priority queue, we will store the points we have not yet encountered, ordered by x coordinate. and push the line forward one stop at a time.

Greedy Algorithms

In greedy algorithms, we always pick the next thing which locally maximizes our score. By placing all the things in a priority queue and pulling them off in order, we can improve performance over linear search or sorting, particularly if the weights change.

Example: Sequential strips in triangulations.

Danny Heep