Resizable arrays
Lower bounds on sorting

Lecture 21
CS 211 Spring 2006

Administrivia

- A5 due tomorrow
- A6 out very soon
  - Implement the game Risk with graphical UI
- Prelim 2 in one week
  - 7:30pm Tuesday, Upson B17
  - Open book

The need for resizing

- Hash tables with collision resolution by chaining: O(1)?
  - Expected look-up time: $1+\lambda/2$ (if there), λ (if not)
  - But… $\lambda = n/m$ is O(n)!
- Solution 1: always preallocate a big enough hash table
  - Can't always predict
  - Bigger array → wasted space, slower accesses
- Solution 2: grow the hash table when load factor λ exceeds a threshold

How to resize an array

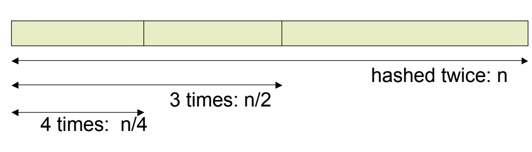
- To resize a hash table:
  - Allocate a new array for the slots
  - Rehash all the elements from old array for the new array length, insert into new array
- To grow an array:
  - Multiply array length by 2 (or some constant factor>1)
  - Do when load factor exceeds some threshold λ_{max} (e.g. λ_{max} = 1, 2, 3)
  - Effect: $\lambda_{max} \rightarrow \lambda_{max}/2$
- To shrink an array:
  - Divide array length by 2 (or some constant factor)
  - Do when load factor goes below $\lambda_{max}/4$
  - Effect: $\lambda_{max}/4 \rightarrow \lambda_{max}/2$

Amortization

- Expected run time can now be O(n)!
  - Rehashing and copying take linear time in array size
- But…resizing doesn't happen very often
- Idea: **amortize** the run time over a sequence of many operations
  - **Amortized complexity**: worst case for total run time *divided* by number of operations

Amortized array resize

- Consider n insertions into an array with resizing at λ=1
- Worst case: just resized at $n = 2^k$
- n elements: 2n + n/2 + n/4 + n/8 + … + 1 = 3n-1
- Total time: O(n)   Amortized per operation: O(1)



Why geometric growth?

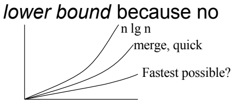
- Suppose we instead added space for k elements when array was full
- n insertions → n/k resizes
- Total work = k + 2k + 3k + …+ n = (1 + 2 + 3 + … + n/k) k = k * (n/k(n/k+1)/2) = n(n/k + 1)/2
- This is $O(n^2)$, so amortized time O(n) per element added!

Sorting algorithm summary

- The ones we have discussed
  - Insertion Sort
  - Selection Sort
  - Merge Sort
  - Quick Sort
- Other sorting algorithms
  - Heap Sort (uses priority queue)
  - Shell Sort (in text)
  - Bubble Sort (nice name, slow)
  - Radix Sort
  - Bin Sort
  - Counting Sort
- Why so many?
  - Stable sorts: *Ins, Sel, Mer*
  - Worst-case O(n log n): *Mer, Hea*
  - Expected-case O(n log n): *Mer, Hea, Qui*
  - Best for nearly-sorted sets: *Ins*
  - No extra space needed: *Ins, Sel, Hea*
  - Fastest in practice: *Qui*
  - Fastest on uniform integer keys (O(*n*)!): *Radix*
  - Least data movement: *Sel*

Problem complexity

- Asymptotically fastest sorting algorithms are O(*n* lg *n*)
  - *kn* lg *n* is an *upper bound* on run time (for some *k*)
  - Can we do better?
- Some problems have an intrinsic complexity -- no algorithm can do better
  - Complexity of a problem is a *lower bound* because no algorithm can run faster



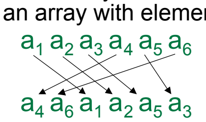
- What is the intrinsic complexity of sorting?

Lower bounds on sorting

- Goal: Determine the minimum time *required* to sort *n* items (no matter what order they come in)
  - Want *worst-case* time for the *best possible* algorithm
- Assumption: sorting algorithm works by comparing pairs of elements
  - in general: that's all you can do

The sorting job

- Sorting takes a array of *n* elements and produces an array with elements in sorted order

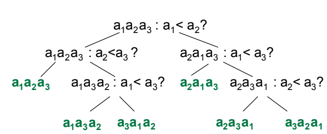


- Sorting finds a *permutation* of the original array
  - There are 1 × 2 × 3 × … × *n* = *n*! permutations
  - Desired permutation is just the *inverse* permutation of the original ordering
- Any sorting algorithm must decide which of n! permutations will undo initial permutation

Comparison trees

- Any sorting algorithm performs some sequence of comparisons, depending on input:
  - Insertion sort on 2 3 1 4: 2<3? 3<1? 2<1? 3<4?
  - Insertion sort on 1 3 2 4: 1<3? 2<3? 1<2? 3<4?
- Comparisons form a *comparison tree*
  - At least *n*! leaves : every permutation must be a leaf
  - Worst-case # comparisons = height of tree

Comparison tree for insertion sort of three items



Time vs. Height

- Worst-case time for a sorting method must be ≥ the height of its comparison tree
  - The algorithm is doing more than just comparisons, but can use comparisons alone for lower bound
- Minimum possible height for a binary tree with *x* leaves is lg *x*
- With n! leaves?
  Height ≥ lg(n!) =
  lg(1 × 2 × … × n) =
  lg(1×n × 2×(n-1) × 3 × … × n/2)
  = lg(1 × n) + lg (2 × (n-1)) + …
  ≥ (n/2) * lg n
- **Any** comparison-based sorting algorithm **must** have a worst-case time of Ω(n lg n)
  - Lower bound; so we use big-Omega (Ω) instead of big-O
  - f(n) is Ω(n lg n) if there exists k such that f(n) ≥ kn for large n

Using the Lower Bound on Sorting

Claim: I have a priority queue
- add time: O(1)
- removeMax time: O(1)

- True or false?

False (for general sets) because it could be used to sort in time O(*n*) using heapsort.

Heapsort: insert all elements into priority queue, extract in priority order.

Sorting in Linear Time

Several sorting methods take only linear time
- Counting Sort
  - Sorts integers from a small range: [0..k] where k = O(n)
- Radix Sort
  - The method used by card-sorters
  - Sorting time O(dn) where d is the number of "digits"
- Others…

- How do they get around the Ω(n lg n) lower bound?
  - Don't use comparisons: use keys as numbers to index into arrays

Lower vs. upper bounds

- Many problems have provable lower bounds
  - When algorithm is O(f(n)) and problem is Ω (f(n))… great!
- But for many important problems, big space between lower and upper bounds!
  - Factoring
  - Many games (e.g., chess)
  - Traveling salesman and other optimization problems
  - Boolean satisfiability
  - Take 381/481 for more…