Lower bounds on sorting & Standard collection ADTs

Lecture 19
CS211 Spring 06

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)
  - k n 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 x 2 x 3 x ... x 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 4: 2<3? 3<1? 2<1? 3<4?
  - Insertion sort on 1 3 2 4: 1<3? 3<4? 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

n lg n

merge, quick

Fastest possible?
Time vs. Height

- Worst-case time for a sorting method must be \( \geq \) 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 of tree: \( \lg(n! + 2^{(n-1)} + 3 \times \ldots + n/2) \)
    - Lower bound: so we use big-Omega (\( \Omega \)) instead of big-O
- Any comparison-based sorting algorithm must have a worst-case time of \( \Omega(n \lg n) \)

Using the Lower Bound on Sorting

Claim: I have a priority queue
- Add time: \( O(1) \)
- Remove Max time: \( O(1) \)
- True or false?

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

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 \( k \in O(n) \)
  - The method used by card sorters
  - Sorting time \( O(dn) \) where \( d \) is the number of "digits"
- Radix Sort
- Others...

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

Collection ADTs

- What are the useful abstractions for organizing data in collections?
  - So far: sets, priority queues
- How can they be implemented efficiently?
  - So far: lists, arrays, trees
  - This lecture: more useful abstractions (but not how to implement them).

Set abstractions

```java
class Set<T> { 
  boolean contains(T elem);
}
```

- Mutable sets: elements can be added and removed from set (the usual approach)
  - `void add(T elem)`
- Immutable sets: sets don’t change. Insertion or union produce new sets
  - `Set add(T elem)`
  - `Set union(Set<T> s)`
    - Implementable with data structures that share data (e.g., lists, trees), useful when related sets must coexist.

Set abstractions, cont’d

- Unordered sets:
  - No ordering on elements
  - Iterator produces elements in no particular order
  - No way to get from one element to next or previous
  - The basic abstraction & the most efficient approach if you don’t need ordering
- Ordered sets:
  - Elements are (abstractly) kept in sorted order, can be iterated in order.
  - May be able to search within a range
  - May be able to find next or previous element in order
  - Useful if elements have natural ordering (e.g., dates)
  - Usually implemented as trees
- Bags (multisets): can contain same element more than once
Map abstractions

```java
class Map<K, V> {
    V get(K key);
}
```

- Maintains an association between keys and values
- Every key occurs only once
- Can look up value associated with a key
- Also known as associative arrays, dictionaries (esp. with string keys)
- Java interface: java.util.Map

Map varieties

- Mutable maps
  - void put(K key, V value)
- Immutable maps
  - Map put(K key, V value) // non-destructive
  - Unusual, implementable as tree
- Ordered maps: mappings are ordered by keys
  - Can view a map a set of (key, value) pairs where two pairs are considered “equal” or “less than” if their keys are.
  - Implemented as a tree with key and value at each node.
- Can use as an index.
  - Example: Collection of employee records might be a set of objects.
    - Might also have several maps as indices: from employee name to record object, from employee number to record object, ...

Queues

- Queues contain elements but do not support random lookup

- FIFO queues
  - Push at one end, pop at the other
  - Helpful for delaying work (buffering)
- LIFO queues (stacks)
  - Push and pop from top
  - Good for saving and restoring state
- Priority queues
  - Elements have priority, are popped in priority order.

Summary

- Several useful abstractions for organizing, finding, updating information
- Choose the right abstraction for your program
  - May use several abstractions together
- Next: how to implement unordered sets and maps efficiently