Algorithms in Systems Engineering
IE170

Lecture 7

Dr. Ted Ralphs

References for Today's Lecture

- Required reading
  - CLRS Chapter 6
- References
  - D.E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching* (Third Edition), 1998.
  - R. Sedgewick, *Algorithms in C++* (Third Edition), 1998.

The Sorting Problem

- We will now undertake a more formal study of algorithms for the *sorting problem*.
- This problem is fundamental to the study of algorithms.
- Most often, the items to be sorted are individual *records*, usually consisting of a *key* and related *satellite data*.
- Recall our previous definition (slightly generalized here).
  Input: A sequence of n records $a_1, a_2, \ldots, a_n$.
  Output: A reordering $a'_1, a'_2, \ldots, a'_n$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.
- Note that the records can be anything for which a "$\leq$" operator can be defined (usually by comparing the specified key).
- We may be interested in sorting the same list in more that one way.
- What are some contexts in which sorting is important?

Sorting Algorithms

- It is safe to say that there are more algorithms for sorting than any other single problem.
- There are so many fundamentally different ways of solving this problem that entire books have been devoted to the topic.
- It is known that the running time of any comparison-based sorting algorithm is in $\Omega(n \lg n)$ (why?).
- Any algorithm whose worst-case running time matches this lower bound is said to be *asymptotically optimal* or just *optimal*.
- Many of the known algorithms, including merge sort are optimal.
- However, this does not necessarily translate into good performance in practice.

Properties of Sorting Algorithms

- In addition to worst-case running time, there are a few important properties of sorting algorithm that we may need to consider.
  - A *stable* sorting algorithm is one that leaves duplicate keys in the same relative order that they were in the original list.
  - This is an important property if you want to be able to sort on multiple keys.
  - Another important consideration is whether the algorithm sorts *in place*, i.e., does not have to allocate too much extra memory.
  - Finally, we might consider how well the algorithm performs on arrays that are already sorted, or mostly sorted.
- The sorting algorithm you choose may depend on what you expect the data to look like, e.g., is it "almost sorted."
- The basic operations performed in sorting are comparison and exchange.
- The relative cost of these operations may also help determine the type of sort that is most appropriate.

Elementary Sorting Methods

- Most straightforward sorting algorithms have a running time in
- *Selection sort* is perhaps the easiest to understand.

- *Bubble sort* is another simple algorithm.

- These simple algorithms are called *nonadaptive* because the sequence of steps is not affected by the initial ordering of the list.
- The number of operations is $\Theta(n^2)$ regardless of the input.
- However, these algorithms are fine for small lists.

More Sophisticated Algorithms

- We can improve slightly on the performance of selection and bubble sort by using a closely related variant called *insertion sort*.
  - Insertion sort performs poorly in many cases and still takes $\Theta(n^2)$ in the worst case.
  - However, it is efficient for lists that are almost sorted.
  - Insertion sort can be performed in place and is stable.
- In Lab 2, we saw a variant of a more sophisticated recursive algorithm called *quick sort*, which is based on partitioning.
  - Quick sort is also $\Theta(n^2)$ in the worst case, but is $\Theta(n \lg n)$ on average.
  - However, it is unstable and can result in a large call stack and poor performance if not implemented carefully.
- In Lecture 4, we saw *merge sort*.
  - Merge sort is asymptotically optimal and stable.
  - However, it cannot be performed in place.

Priority Queues and Sorting

- Before performing a more in-depth analysis of these methods, we introduce one more called *heap sort*.
- Heap sort is optimal and can be performed in place, but is unstable.
- Heap sort is based on construction of a *priority queue*.
  - A priority queue is a data structure for maintaining a list of items that have associated *priorities*.
  - The usual operations are

- Note that any implementation of a priority queue can be used to sort a list of items.

Trees

- A *tree* is a set of items organized into a hierarchical structure (think of a family tree).
- When organized in this way, we call the items *nodes*.
- Each node has a single designated *parent* and one or more *children*.
- There is a single designated node, called the *root*, with no parent.
- Any node with no children is called a *leaf*.
- Any node with children is called *internal*.
- A tree in which all nodes have 2 or fewer children is called a *binary tree*.
- Storing a list of items in a tree structure allows us to represent additional relationships among the items in the list.
- Trees occur naturally in many applications.

Binary Tree Data Structures

- To store a tree, we need a data structure supporting three basic operations.

- This allows us to *traverse* the tree and perform other operations on it.
- The *level* of a node in the tree is the number of recursive calls to `parent()` needed to reach the root.
- The *depth* of the tree is the maximum level of any of its nodes.
- A *balanced tree* is one in which all leaves are at levels k or $k-1$, where k is the depth of the tree.

Data Structures for Storing Trees

- There are two primary data structures used for storing trees.
- Array
  - The root is stored in position 0.
  - The children of the node in position i are stored in positions $2i + 1$ and $2i + 2$.
  - This determines a unique storage location for every node in the tree and makes it easy to find a node's parent and children.
  - Using an array, the basic operations can be performed very efficiently.
  - If the tree is unbalanced or dynamic, a linked list may be better.
- Linked List
  - In a linked list, each item is stored along with explicit pointers to its parent and children.
  - This allows for easy addition and deletion of nodes from the tree.

Heaps

- A *heap* is a balanced binary tree with additional structure that allows it to function efficiently as a priority queue.
- The additional structure needed to support these operations is that the record stored at each node has a higher priority than either of its children.
- Any node with this property is said to satisfy the *heap property*.
- Consider a tree in which all nodes except for the root have the heap property.
- We can easily transform this into a tree in which every node has the heap property (how?).
- This operation is called `heapify()`.
- By calling `heapify()` on each node, starting at the lowest level and working upward, we can transform an unordered binary tree into a heap.

Operations on a Heap

- The node with the highest priority is always the root.
- To delete a record
- To add a record
- Note that we can change the priority of a record in a similar fashion.

Heap Sort

- Suppose the list of items to be sorted are in an array of size n.
- The heap sort algorithm is as follows.

- Why is this algorithm correct?
- How do we analyze the running time?