Introduction to Computer Science and Programming in C

Session 21: November 18, 2008

Columbia University

Announcements

- Deergha's Office hours **this week** moved to Tuesday (today) from 6 PM to 8 pm
- Homework 4 is out, due last day of class: December 4 before class
- Final Exam: Tuesday, 12/16, 1:10 pm - 4:00 pm Mudd 233 (our normal room)

Review

- Homework 3 solutions
- Revisiting pointers:
  - Pointers to pointers (multidimensional arrays)
  - Pointers to functions (qsort example)

Today

- Data structures:
  - Linked Lists
  - Binary Trees

Data Structures

- Ways to store data so that computation can be done efficiently
- Most basic: variables, 1-d arrays
- Depending on the computational task, more sophisticated data structures can be helpful, with a tradeoff
- We'll look at two very common data structures

What's Wrong with Arrays?

- Arrays are of fixed size
- We can allocate variable sized arrays, but once they are allocated, the size becomes fixed
- Consider a task where a user inputs as few or as many integers as desired, and we must store them. How do we store them?

Linked Lists

- Store each element in a struct that contains the data and a pointer to the next struct: a **node**
- Keep a pointer to the first node
- Following a linked list is like a scavenger hunt

Linked Lists

- ```
  struct node {
    int data;
    struct node * next;
  };

  struct node *start;
  ```

- How do we add a node at beginning of list?
  - Allocate new node, set **next** pointer to **start**, set **start** to new node.
  ```

Linked Lists

- How do we add a node to the end of the list?
  - Follow pointers to last node, allocate new node, set last node's **next** to new node.
- How do we add in the middle of the list?
  - Set previous node's **next** to new node, set new node's **next** to next node.
- How do we delete a node?

Doubly Linked Lists

- Keep a **next** pointer and a **previous** pointer.
- A little extra work for adding and removing, but allows for faster backtracking.

Binary Trees

- Finding an item in a list or array is usually an O(N) operation.
- We can create a structure that makes it faster (at a cost; a tradeoff)
- We use a tree structure, which is like a linked list, except each node has more than one pointer.

Binary Trees

- Binary tree: Each node has left and right child.
  - Left child is less than, right child is greater than

- ```
  struct node {
     int data;
     struct node *left;
     struct node *right;
  }

  struct node *root;
  ```
  ```

Binary Trees

- Inserting number **x** into a Binary Tree:
    - 0. Start at root
    - 1. If current node is NULL, create new node and set node to **x**
    - 2. Otherwise,
      if **x** >= current node, follow right pointer,
      else follow left pointer. Goto 1.

Binary Trees

- Finding an item **x** in a binary tree:
  - 0. Start at root
  - 1. If current node is **x**, return
  - 2. If **x >=** current node, follow right pointer else, follow left pointer
  - 3. If node is NULL, return “not found”, otherwise goto 1.

Binary Trees

- On average, lookup and insertion take O(log N) time
- But worst case is still O(N)

Reading

- Practical C Programming. Chapter 17