CPS305 Summary Sheet
Printed 2011-10-24

1 Recursion (ch 3)

- Always a 'base case' (the way out), recursive case(s)
- Tail recursion is when the last instruction executed in the method is the recursive function call
- Types:
  - Divide & Conquer – break up into smaller problems
    - Down by 1 (and Up by 1)
    - Division in halves
  - Last & All But Last, First and All But First

2 – Complexity (ch 6)

Time Complexity (operations/cpu usage)

- Count the number of:
  - operations
  - comparisons
  - loop overhead
  - pointer/array references
  - function calls

Space Complexity (storage/memory usage)

- Count number of variables

Unroll Recurrence Relation:

```
        base: T(0) = 0
        T(n) = 1 + T(n-1)
        =>   = 1 + (1 + T(n-2))
        ** need to make T(n-x) into the base case, so
replace x with whatever is necessary **
             = 2 + T(n-2)
             = n + T(n-n)
             = n + T(0)
             = n + 0      => O(n)
```

Using a Call Tree to determine complexity

- Space: lenght of longest branch
- Time: total number of nodes (see fmla for node count in a perfect binary tree)

3 – Lists (ch2, ch8.1 - 8.4)

Here is a list with a header. The header helps make the list easier to navigate.

```
        [list] -> [header|]->[ 1 |]->[ 2 |] --
                        ^                     |
                        ^                     |
                        ^----------------------
```

Empty list (with header):

```
        [list] -> [header|]----
                        ^      |
                        ^      |
                        ^-------
```

Simple Insert:

```
        ptr = List;
        while (ptr->Link != List && ptr->Info < value)
                ptr = ptr->Link;
        newItem->Link = ptr->Link;
        ptr->Link = newItem;
```

When is a (2-way) LL more space-efficient than an array of MAX size? (I is number of bytes):

LL space: n items + ptrs per item + header + variable for the list
= I*n + 2*p*n + (I + 2*p) + p

Array space: Max*I + index of last item (or sentinal value) + pointer/variable
= Max*I + I + p

So, more efficient to use 2LL when:

I*n + 2*p*n + I + 2*p + p < MAX*I + I + p

=> $n < \dfrac{MAX \cdot I - 2p}{I + 2p}$

Depends on size of storage (I), and how much you want to allocate as the MAX of the array

But basically, for small amounts of data, an array is better (LL's have pointer overhead)

3a - Restricted Linked Lists (ch 7)

- Stacks
  - program function calls
  - computing post-fix math
  - converting in-fix math to post-fix
- Queues
  - printers, server requests, keyboard buf

4 – Trees (ch 9)

Binary Tree

- Sequential
  - stored in an array
  - root = A[1]
  - left child of A[i] = A[i*2]
  - right child of A[i] = A[i*2 + 1]
  - parent of A[i] = A[i/2]
  - A[i] is a if <=> 2*i > n
  - Problematic when right-heavy
- Linked
  - Navigation (pg 362, 363):
    - Level Order
      - Level-by-level, Left to right
    - Pre-order
      - Root, then left, then right
    - In-order
      - Left, then root, then right
    - Post-order
      - Left, then right, then root
  -

Complete Binary Tree:

- All leaves on same lvl or 2 adjacent levels s.t. bottom-most leaves are as far left as possible
- height = FLOOR(Log n) [log base 2]

Binary Search Tree

- Has index values in the nodes
- Left child < parent < right child
- Search/Insert:
  - Navigate left/right as needed
- Delete:
  - If leaf, simple
  - If has 1 child, promote child
  - If has 2 children,
    - 'copy' largest from left or smallest from right
    - delete 'copy' (repeate recursively)

AVL Tree

- Is a Binary Search Tree, but not necessarily Complete
- For every node, the difference in height of the left and right subtree is **+/- 1**
- Rebalancing: See pg 379 of text
- Rebalancing – Outer-heavy:
  - Single Right (or Left) Rotation of the unbalanced node (plus swap one child branch to keep things even ?)
- Rebalancing – Inner-heavy:
  - Double Right (or left) rotation
    - Eg, Dbl right -> left then right