CSC 115 - Notes

**Tree Traversals**

- All handled recursively

1. Pre-order

- visit the node, then both subtrees (left first)

- always starts with the root

2. In-order

- visit the left child, repeat until at bottom

- visit node, visit right

- root somewhere near middle

3. Post-order

- visit both subtrees (left first), then node

- always ends with root

4. Level-order

- start at root, then down

- work straight across levelsß

**Sorts**

Quicksort

* pick pivot, sort all as either < or >
* repeat on all < than pivot, then all > pivot
* recursive, down to base
* O(n log n) expected, O(n2) worst

Selection sort

* look at all n elements, pick smallest
* leave 1st, look at n-1, pick smallest, etc.
* worst, best, expected times all O(n2)

Bubble sort

* compare 1st to 2nd, swap if necessary, compare 2nd to 3rd, etc.
* repeatedly work through list until no swaps necessary
* worst and expected case O(n2), best case O(n)

Insertion sort

* move elements from unsorted to sorted list, like sorting playing cards
* worst and expected case O(n2), best case O(n)

Heap sort

* look at n elements, pick largest
* look at n-1 elements, pick largest, etc.
* worst, best, expected times all O(n2)

Merge sort

* if list 0 or 1, then already sorted
* else cut in half, apply recursively
* reorder when merged back together
* expected, worst and best all O(n log n)

arrays linked lists BST

sorted, unsorted sorted, unsorted worst, expected Hashing

insert(val) O(n) O(1) O(n) O(1) O(n) O(log n) O(1)

remove(val) O(n) O(n) O(n) O(n) O(n) O(log n) O(1)

find(val) O(log n) O(n) O(n) O(n) O(n) O(log n) O(1)

**Binary Search Tree vs Heap**

Tree, general

* any operation traversing a tree is O(log n)

- binary tree has nodes with at most two children

* tree with one element has a height of 1 (but 0 in some other courses/books)
* worst case is height = n elements, because it looks like a stick instead of a tree

Full binary tree

* all nodes full, except possibly the last one at the last level
* max elements for height is n = 2height -1

Binary search tree

* binary tree with additional properties
* value at n > left subtree, < in right subtree
* DEGENERATE (stick-tree) worst case is O(n), height: h = n
* FULL binary search tree worst case (log n)

Heap, Max or Min

* always full, except for possibly last spots
* Max: n > children, root largest
* Min: n < children, root smallest
* worst case add O(log n), remove top O(log n)

Heap, remove root

* remove root, replace with bottom value
* new root -> than children, swap with largest/smallest (min, max heaps)

-> repeat until proper heap again

Heap, add

* always add to last available spot and then reapply heap sort
* it is a heap? compare new to parent, swap if necessary, repeat

Hashing

* “good” function maps values somewhat evenly to hash keys
* open addressing adds directly to array
* collisions for linear probing handled by adding flag, moving to next
* load factor, λ = n/size = (# of items) / (# of places in hash table)
* open addressing should have λ < 0.5
* growing hash table is O(n)

**Queues**

- enqueue - add to end of queue

- dequeue - remove from front of queue

- peek (or front) - look at front, returns value, not removed

- dequeue all - remove all from queue

English |Queue(FIFO) |Big Oh |Stack(LIFO)

--------------------------------------------------------------------

add |enqueue |O(1) |push

remove |dequeue |O(1) |pop

look |peek |O(1) |peek

|dequeue all |O(1)-O(n)\*

(\*non-standard)

- FIFO - First In First Out

- LIFO (/FOLI) - Last In First Out

- Possible error: QueueEmptyException on pop and peek

- Implementation could have QueueFullException if using array|push

**Exceptions**

- ArrayIndexOutOfBounds, ArithmeticException, etc.

-use

try{ //regular stuff }

catch(MyCustomException){ //stuff you expected }

catch(Exception e){ System.err.println(e); //fail stuff }

…

**args** - pass parameters from the command line

System.out.println( "Hello, " +args[0]+ "!" );

cmd > java fileName "Fart MacPoo"

Output "Hello, Fart MacPoo!"

...

**Switch statements** use CASES instead of IFs, e.g.:

int month=8;

switch( month ){

case 1: System.out.println( "Jan" ); break;

case 2: System.out.println( "Feb" ); break;

case 3: System.out.println( "Mar" ); break;

case ... }

...

//Compare strings using boolean \_.equals

if( string1.equals( string2)){ ...}

…

**Convert** String to int

int i = Integer.parseInt( stringThing );

**Convert** Integer to String

String s = Integer.toString( IntegerThing );

...

**Scanner** auto-initialize with \*.getScanner();

try {

PrintWriter out = new PrintWriter(

new BufferedWriter(

new FileWriter( cFileName )));

for( int i= 0; i< addressBook.length; i++ )

out.println( addressBook[i] );

out.close();

} catch (Exception e) {

System.out.println("Fatal Error: " + e);

e.printStackTrace();

return false;

}

return true;

Tree traversal - http://en.wikipedia.org/wiki/Tree\_traversal

http://cslibrary.stanford.edu/110/BinaryTrees.html