Data Structures

structure	**add**	**remove**	**contains**
array	$O(N)$, $\Omega(1)$	$\Theta(1)$	$O(N)$
arraylist	$O(N)$, $\Omega(1)$	$O(N)$, $\Omega(1)$	$O(N)$
linkedlist	$\Theta(1)$	$\Theta(1)$	$O(N)$
stack, queue	$\Theta(1)$	$\Theta(1)$	$O(N)$
heap	$O(\log(N))$, $\Omega(1)$	$O\log(N)$	$O(N)$
hashmap	$O(N)$, $\Theta(1)$	$O(N)$, $\Theta(1)$	$O(N)$, $\Theta(1)$
BST	$O(N)$, $\Theta(\log(N))$	$O(N)$, $\Theta(\log(N))$	$O(N)$, $\Theta(\log(N))$
BTree	$\Theta(\log(N))$	$\Theta(\log(N))$	$\Theta(\log(N))$
skiplist	$O(N)$, $\Theta(\log(N))$	$O(N)$, $\Theta(\log(N))$	$O(N)$, $\Theta(\log(N))$
Point of Confusion: O was used for worst case; Θ for best / overall; if there's only Θ, worst = overall.			

Note:

- Cost of array insertion can be amortized to $\Theta(1)$ if array is resized to 2 * array.length everytime array is full: $\Theta(N) \cdot (1/N) = \Theta(1)$.

Sorting Asymptotics (source: CSM review slides for Fa17 MT2)

sort	runtime (best)	runtime (worst)	stable	notes
insertion	$\Theta(N)$	$\Theta(N^2)$	✔	fast for small / almost-sorted data (less than log N inversions)
selection	$\Theta(N^2)$	$\Theta(N^2)$	no	should avoid using
merge	$\Theta(NlogN)$	$\Theta(NlogN)$	✔	fastest stable-comparison sort
quick	$\Theta(NlogN)$	$\Theta(N^2)$	no	fastest comparison sort; improbable worst case
heap	$\Theta(NlogN)$	$\Theta(NlogN)$	no	
counting	$\Theta(N+R)$	$\Theta(N+R)$	✔	alphabet keys only
LSD	$\Theta(WN+WR)$	$\Theta(WN+WR)$	✔	required to be stable to work
MSD	$\Theta(WN+WR)$	$\Theta(WN+WR)$	✔	
$N = \text{num elements}, W = \text{longest-width key}, R = \text{size alphabet (radix)}$				
Point of Confusion: if best case $\Theta(f(n))$ and worst case $\Theta(g(n))$, then overall $\Omega(f(n))$ and $O(g(n))$; and vice versa.				

Asymptotics (source: wikipedia.org/wiki/Big_O_notation)

Big-Omega	**Big-O**	**Big-Theta**
$f(n) \in \Omega(g(n))$ iff $\exists k > 0 \; \forall n$ s.t. $f(n) \geq k \cdot g(n)$	$f(n) \in O(g(n))$ iff $\exists k > 0 \; \forall n$ s.t. $f(n) \leq k \cdot g(n)$	$f(n) \in \Theta(g(n))$ iff $f(n) \in \Omega(g(n))$ and $f(n) \in O(g(n))$
$\lim_{n \to \infty} \frac{f(n)}{g(n)} = k > 0$	$\lim_{n \to \infty} \frac{f(n)}{g(n)} = k < \infty$	$\lim_{n \to \infty} \frac{f(n)}{g(n)} = k$

Relation between Big-Omega and Big-O:

- If $f(n)$ is in $O(g(n))$, then $g(n)$ is in $\Omega(f(n))$.
- If $f(n)$ is in $\Omega(g(n))$, then $g(n)$ is in $O(f(n))$.

Asymptotics Cont.

Useful Sequences

Arithmetic:

$$1 + 2 + 3 + \cdots + (N-1) + N = \sum_{i=1}^{N} i \in \Theta(N^2)$$

Geometric:

Note:
There are two different types of Geometric sequences: converging and diverging. The former has a runtime of $\Theta(N)$ and the latter has a runtime of $\Theta(p^N)$.

$$1 + 2 + 4 + \cdots + (N/2) + N = \sum_{i=0}^{\log(N)} 2^i \in \Theta(N)$$

$$1 + 2 + 4 + \cdots + 2^{N-1} + 2^N = \sum_{i=0}^{N} 2^i \in \Theta(2^N)$$

If the first one seems foreign, it's because it is more often seen as
$N + (N/2) + \cdots + 4 + 2 + 1$

Both of them have general extensions, where previously $p = 2$.

$$1 + p + p^2 + \cdots + (N/2) + N = \sum_{i=0}^{\log(N)} p^i = N \cdot \frac{p}{p-1} \in \Theta(N)$$

$$1 + p + p^2 + \cdots + p^{N-1} + p^N = \sum_{i=0}^{N} p^i = \frac{p^{N+1} - 1}{p - 1} \in \Theta(p^N)$$

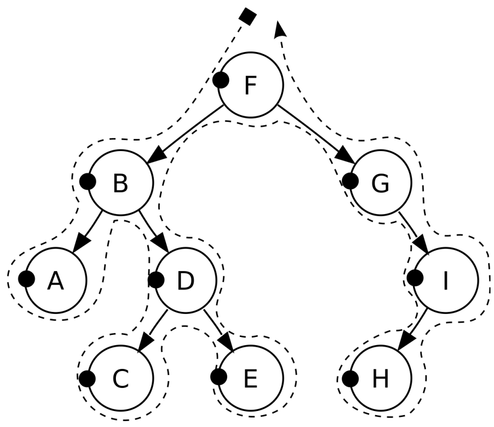
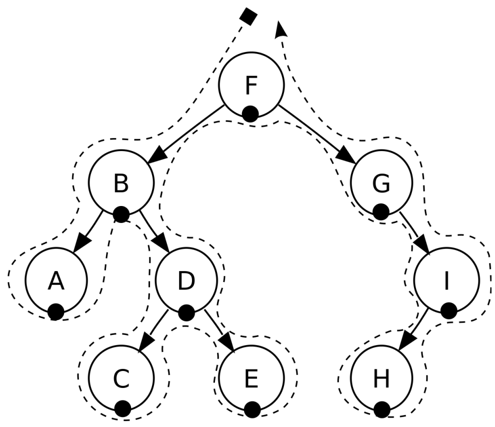
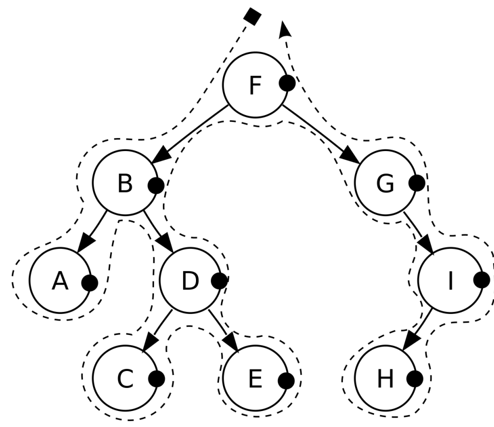
Stirling's Approximation:

$$\log(1) + \log(2) + \log(3) + \cdots + \log(N-1) + \log(N) = \log(N!) \in \Theta(N \log(N))$$

Or more generally,

$$\log(n!) \in \Theta(n \log(n))$$

Tree Transversals (DFS) (source: wikipedia.org/wiki/Tree_traversal)

pre-order	in-order	post-order
		
curr -> left -> right	**left -> curr -> right**	**left -> right -> curr**
F, B, A, D, C, E, G, I, H	**A, B, C, D, E, F, G, H, I**	**A, C, E, D, B, H, I, G, F**

dotted-line trick: if you place a dot on the left / bottom / right side of each node like above, you will find the pre-order / in-order / post-order transversals, respectively.

Note:
- pre-order / in-order / post-order are terms that <u>only apply to DFS traversal</u>
- pre-order / in-order / post-order are also the orders in which you print the value of a node:

pre-order	in-order	post-order

`while (!end) {`
`    print(node.value)`
`    recursive-call(left)`
`    recursive-call(right)`
`}`
	`while (!end)`
	`    recursive-call(left)`
	`    print(node.value)`
	`    recursive-call(right)`
	`}`
		`while (!end) {`
		`    recursive-call(left)`
		`    recursive-call(right)`
		`    print(node.value)`
		`}`

Heaps

Note:

- Heaps are always maximally bushy $h(N) = \log_k(N+1)$, and follow either of two constraints:
  - <u>min-heap</u>: root is smallest element; parent must have a smaller value than its children and their children.
  - <u>max-heap</u>: root is biggest element; parent must have a bigger value than its children and their children.

- Two ways to make a valid heap:
  - <u>reverse-order bubble down</u>: start from bottom and swaps child with parent if heap condition is not met; constructs multiple small heaps and connects them
  - <u>level-order bubble up</u>: start from root and swaps parent with child if heap condition is not met; constructs a new heap by repeatedly adding elements, correct heap along the way

- An array can be used to implement a heap.
  - For a binary heap with root indexed at 0: parent(n) = (n - 1) / 2; left-child(n) = 2 * n + 1; right-child(n) = 2 * n + 2.
  - For a k-nary heap with root indexed at 0: parent(n) = (n - 1) / k; i-th child(n) = k * n + i.

B-tree

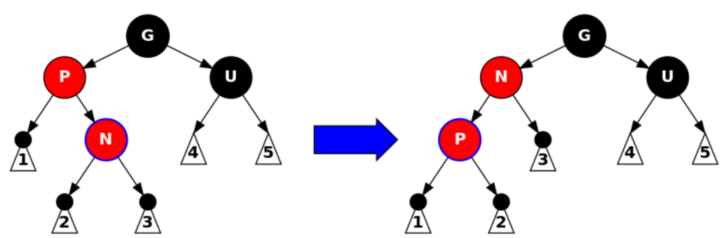
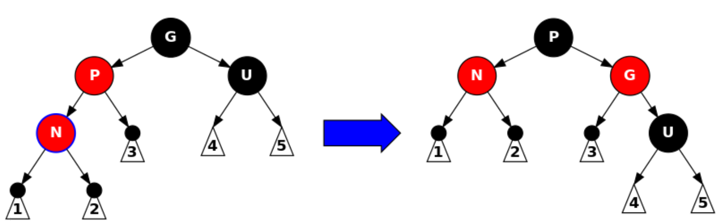
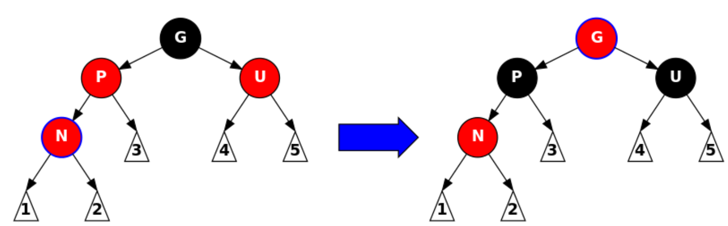
Note:

- A self-balancing tree data structure that keeps data sorted and allows operations (search, access, insertion, deletion) in logarithmic time $O(logn)$
- N-K tree: a tree with [N, K] possible children; maximum number of children is equal to (number of values stored in a node) + 1.
- example - a 2-4 tree is a tree with the possible configurations: 1 value, 2 children; 2 values, 3 children; 3 values, 4 children.

Red-Black Tree

Note:

- A binary B-tree often used in place of other B-trees (as BSTs are easier to implement).
- Nodes with color red indicate same depth in B-tree representation.
- Restrictions:
    - root is black (optional)
    - if a node is red, then both of its children are black
    - new nodes are originally inserted as red
- **4 Operations to balance RBTree**

1 red root => flip color at each level	1 red child => swap to left
[no img found]	
1 red child, 1 red parent, 1 black grand-parent => 2 red children, 1 black parent	2 red children, 1 black parent => 2 black children, 1 red parent
	

Graph Transversals

Asymptotics

transversal	runtime
BFS	$O(\|V\| + \|E\|)$
DFS	$O(\|V\| + \|E\|)$
Dijkstra	$O((\|V\| + \|E\|) \log(\|V\|)$
A*	$O((\|V\| + \|E\|) \log(\|V\|)$
Kruskal	$O((\|V\| + \|E\|) \log(\|V\|)$
Prim	$O((\|V\| + \|E\|) \log(\|V\|)$
$O((\|V\| + \|E\|) \log(\|V\|) = O(\|E\| \log(\|V\|)$	
Point of Confusion: runtime of Dijkstra, A*, Kruskal, and Prim assumes the use of adjacency lists and binary heap; if an adjacency matrix was used, runtime would be $O(\|V\|^2)$.	

Pseudocode for BFS & DFS

BFS: fringe = queue; DFS: fringe = stack

```
transverse(int start, fringe) {
    List seen; fringe.add(start);
    while (!fringe.isEmpty()) {
        node = fringe.pop();
        if (!seen.contains(node)) {
            seen.add(node);
            for (neigh : node.neighbors) {
                fringe.add(neigh);
            }
        }
    }
}
```

Note:

- As with trees, pre-order and post-order:
  - applies only to DFS.
  - depends on the order in which an operation is performed on node: pre-order implies do(node) occurs before the addition of neighbors; post-order implies do(node) after.
- Use BFS to find shortest number of edges (on unweighted graph) to a goal, and or if solution is shallow in graph; DFS to find solutions deep in graph (e.g. does there exists a cycle?)

Note (cont.):
- Surprisingly, reversing a sequence given by post-order DFS on a directed acyclic graph results in a valid **topological sort**. Thus, runtime of finding a topological sort = runtime of DFS.

Dijkstra

1. Assign distance value of 0 to initial node; infinity for all other nodes.
2. Create a set of unvisited nodes ("the frontier").
3. For the current node, consider all of its neighbors and calculate their tentative distance: min(prev_dist, edge_dist + curr_dist).
4. Mark the current node as visited; a visited node will never be checked again.
5. Select the node with the smallest tentative distance from the frontier, and label that as your current node.
6. Repeat steps 3-5 until "the frontier" is empty.

Note:
- **shortest-path tree (SPT)**: a spanning tree such that the path distance from root v to any other u is the shortest path from v to u; this can be constructed using Dijkstra.
- Dijkstra does <u>NOT</u> work for graphs with negative edge weights.
- Implementation-wise, Dijkstra is the same as BFS except with a priority queue instead of a regular queue.

A* Search

Note:

- A* search is essentially the combination of two searches:
  - <u>uniform cost search</u>: same as Dijkstra, except it stops when goal node is reached.
  - <u>greedy search</u>: uses solely a heuristic to find a path -- not necessarily the shortest -- to the goal node
- Thus, A* chooses the next node with the lowest $f(n_1) = h(n_1) + d(n_0, n_1)$, where $n_0 = \text{current node}$, $n_1 = \text{next node}$, and $d(n_0, n_1)$ is the distance from n_0 to n_1.
- For A* to find the shortest path, its heuristic must be consistent.
  - **consistent** (monotone): $\forall n, h(n_0) - h(n_1) \leq d(n_0, n_1)$ and $h(G) = 0$, where $h(G)$ is the heuristic function evaluated for the goal node.
  - A consistent heuristic never overestimates the cost of reaching the goal; in other terms, it is always admissible.
  - <u>admissible</u>: $\forall n, h(n) \leq h^*(n)$, where $h^*(n)$ is the optimal cost of reaching the goal.

Kruskal

1. Sort edges by ascending weights (using a queue).
2. Pop an edge off the queue;
   if adding the edge does not create a cycle, add it;
   else, discard the edge.
3. Repeat until (N-1) edges where N is number of vertices.

Note:

- **minimum spanning tree (MST):** a subset tree within a graph that that connects all the vertices together without any cycles, and whose sum of edge weights is as small as possible; this can be found using Kruskal (or Prim).
- **union find:** a data structure used to implement Kruskal and allows set-operations (add, union, find) in near-constant time
  - **path-compression:** a way of flattening the structure of the union-tree whenever find is called; threads node to root such that distance from root is simply 1.

Prim

1. Initialize a tree with a single vertex.
2. Find the minimum-weight edge which connects the tree to a vertex not yet in the tree. Add it to the tree.
3. Repeat step 2 until (N-1) edges where N is number of vertices.

Bitwise Operators

and x & y = z			or x | y = z			xor x ^ y = z			not ~y = z		
0	0	0	0	0	0	0	0	0	-	0	1
0	1	0	0	1	1	0	1	1	-	1	0
1	0	0	1	0	1	1	0	1	-	-	-
1	1	1	1	1	1	1	1	0	-	-	-

name	**description**	**example**
<< shift left logical	zero-extends new left-bits; = to multiplying by 2^n	**10001 << 1 = 00010**
>> shift right arithmetic	sign-extends new right-bits	**10001 >> 1 = 11000**
>>> shift right logical	zero-extends new left-bits; = to dividing by 2^n	**10001 >>> 1 = 01000**

Note:
Negative numbers are represented by a method known as **two's complement**, where
- the most significant bit (MSB) determines the sign of the number: 0 indicating a non-negative number; 1 indicating a negative number
- the negative complement of a number is: -x = ~x + 1

What about **addition**?

```
add(int x, int y) {

    int carry;
    while (y != 0) {

        // and gets the carry bits
        carry = x & y;

        // xor is also known as no-carry add
            x = x ^ y

        // shift-left gets the carry bits into the proper location
            y = carry << 1;
    }
    return x;
}
```

Hashing

- load factor = (N/k), where $N = \text{num elements}$ and $k = \text{num buckets}$
  - When load factor is exceeded, resize the hash table such that the new size = 2 * current size; this allows for an amortized cost of $\Theta(1)$.
  - When a hash table is resized, all elements must be rehashed and remapped since the hashcode depends on the size of buckets in the table. Cost is amortized.

- A hashcode is considered **valid** if it satisfies the following two properties:
  - If two objects are considered to be equal, then they should have the same hashcode: thus, <u>if you override obj.equals() method, you should also override the obj.hashcode() method</u> -- and vice versa
  - **consistent**: given the same object multiple times, the outputted hashcode should be the same for all instances; thus, a hashcode should not be dependent on a random number generator

- In addition to being valid, a hashcode should try to distribute items as evenly as possible into buckets.
  - This means minimizing the chance of collision.

Java Regex

*****	0 or more
+	1 or more
?	0 or 1
{X}	X times
{X, Y}	X min to Y max (inclusive)

\\\\s	whitespace
\\\\S	non-whitespace
[0-9]	decimal

|	or
()	grouping