CSI 2110 Cheat Sheet

1 Analysis

We analyse algorithmns using the Big Oh notation.

A big Oh of O(n) means that the worst case running time of the algorithm is n.

We say f(n) is O(g(n)) if there exist positive constants n_0 and c such that $f(n) \leq cg(n) \forall n \geq n_0$.

Ex. Prove the following:

1. $2n^2 + 10n^3$ is O(n^3) We need to find the constants c and n0.

$$2n^2 + 10n^3 \leq 2n^3 + 10n^3 \leq 12n^3 \implies c = 12, n_0 = 1$$

Since the constants exist, then it is proven.

Big Omega is similar but it is the best case running time.

Big theta is the exact time. An algorithm of $\Theta(n)$ ALWAYS takes n time.

2 Stacks

A stack is a first in last out data type.

We can `push()`, `pop()` to add or remove to the top of the stack. There is also `isEmpty()`, `pek()` (returns but does not remove the top) and `size()`.

3 Queues

Queues are first in first out data types.

Elements are *enqueued* at the read, and *dequeued* from the front.

This can be inplemented using a **circular array, or a linkedlist**.

4 Deques

A double ended queue (Deque) can be inserted at the front or rear of the queue, and can also be removed at the front or end of the queue.

A **doubly linked list** works well for this.

5 Lists/Sequence

This is a collection of elements in linear order.

5.1 Array List

An array list can get from any index `get(i)`, set any index `set(i,e)`, add at any index `add(i,e)`, or remove at any index `remove(i)`.

We want to implement the array lists using an **extendable array** (double or increment the size when it gets full)

5.2 Positional List

A positional list is a list where we can get the previous or next element given a certain element in the list `addBefore(p,e)`, `addAfter(p,e)`. We can also add first or last `addFirst()`, `addLast()`.

We implement positional lists using **linked lists**.

5.3 Sequence

This provides all operations from the positional list and array list and has bridge methods as well such as `atIndex(i)` (returns position), `indexOf(p)` (returns index).

This can be implemented using arrays or linked lists.

With **arrays**, the index based ops will be fast, but the positional ones will be slow.

With **linked lists**, the positional ops will be fast, but the index ones will be slow.

6 Priority Queues

A priority queue is a queue where each element has a priority (key value pair).

We can `insert(k,v)` in any order, and we can `removeMin()` or just view `min()`.

If we use a **sorted sequence**, inserting takes O(n), and removing is O(1).

If we use an **unsorted sequence**, inserting takes O(1), and removing is O(n).

6.1 Heaps

A Heap is a **complete** binary tree that stores a collection of key value pairs.

We can do either a **min heap** (min at top), or a **max heap** (max at top).

With the min heap, the parent of a node is always greater than the children.

We implement these using a **priority queue**.

To remove from a heap, we remove the top element, then take from the bottom. Then we need to **downheap** until the heap is fixed. (Downheaping means swap with smallest child until child is bigger)

To add on, we add at the first empty spot at the bottom, and then **upheap**. (Upheaping means swap with parent until parent is smaller).

6.2 Heap Construction

We use bottom up heap construction to build a heap. This takes O(n).

For this, we start at **bottom right of heap**, then do a downheap on that parent. Then we go to the right and do it again. When that level is done, go up a level starting on the right. We will end up downheaping each element.

7 Maps

A map stores and retrieves data based on a **unique key**. We have `get(k)`, `put(k,v)`, `remove(k)`.

A **sorted map** also has `firstEntry()`, `lastEntry()`.

We can use an **ordered sequence** where searching takes O(logn), and inserting/removing take O(n).

We can also use a *better* **BST** (ideally a balanced one such as AVL) where the worst case search is O(n), but best case is O(logn). Inserting and deleting also depend on search so they are the same.

8 Trees

A tree has **internal nodes** (nodes with at least 1 child) and **external nodes** or Leafs (node without children).

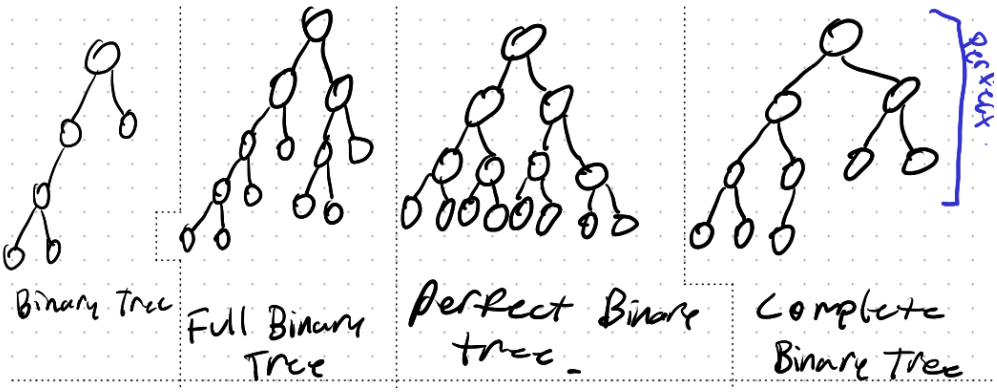
A tree is called *binary* if each node has at most 2 children.

We can traverse a binary tree using preorder, inorder, or postorder:

1. Preorder means visit Root, then Left, then Right
2. Inorder means visit Left, then Root, then Right
3. Postorder means visit Left, then Right, then Root

A binary tree is called **full** if each internal node has exactly 2 children.

A binary tree of height h is called **complete** if the tree of height h-1 is perfect, and the rest of the nodes are added starting on the left.



We have some statistics about a full binary tree:

n is number of nodes, e is number of external nodes, i is number of internal nodes.

$$e = i + 1 \qquad n = 2e - 1$$

These do NOT apply to a general binary tree.

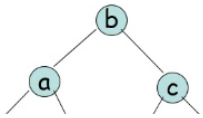
All operations have O(1) except search which depends on the height of the tree

8.1 AVL

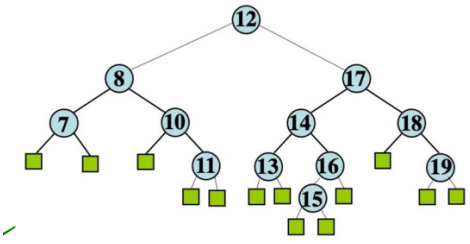
An AVL tree is a BST where each **internal node's subtrees heights can differ by at most 1**. This means the **height is O(logn).**

To insert or delete, we do the same thing as with a BST, except if any change renders the tree unbalanced, we need to rebalance the tree.

To rebalance, we go to the first node (from the bottom) that is unbalanced, then we do the inorder traversal from it down to the longest side naming the first node visited a, then b, then c. Then we align abc like this with their corresponding subtrees.



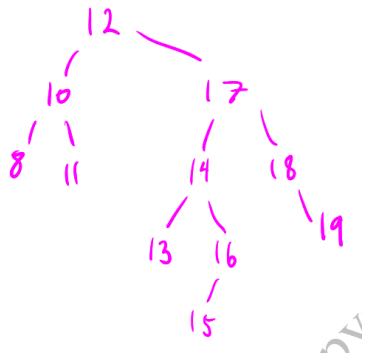
Ex. Remove 7 from this tree.



When removing 7, we see that the 8 is unbalanced. So we do inorder traversal of it and

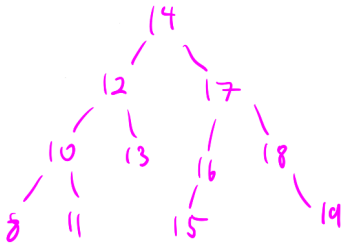
its longest side (10, 11). We assign 8a, 10b, 11c.

So we rearrange like:



It is still unbalanced at the 12 node. So we do the inorder traversal there down the longest branch (ending with 15)

12a, 14b, 17c



Now we are done. The tree is balanced.

8.2 2-4 Trees

A 2-4 tree is a **non binary search tree** where each node has 2, 3, or 4 children (and therefore 1, 2, or 3 keys) and all **external nodes have the same depth.**

When we insert, we go down the tree to find where in the key should go. If this causes 5 keys in a node, we split (usually using the third key which gets sent to the parent).

When we delete, we remove the item to delete and replace with inorder successor. If this causes an underflow, we borrow from the parent and fuse the two children together or if the sibling has extra, we take from the sibling, put into the parent, and then steal from the parent.

9 Hash Maps

We take a problem using a very large table, and use a **function** to map all data points to a smaller table.

To resolve collisions, we have a few methods:

1. Chaining is creating a linked list at each spot in the hash table with items with duplicate hashes.
2. Linear probing is we will go to the next empty spot linearly (h+0, h+1, h+2, h+3).
3. Quadratic probing will go to the next spot squared (h+0, h+1, h+4, h+9)
4. Double Hashing means we have a second hash function that we apply (h(k)+0d(k), h(k)+1d(k), h(k)+2d(k) for $h_j(k) = [h(k) + j^2 d(k)] mod N$

When we are using any of the methods 2-4, and we delete an item, we cannot just delete the item since we need a record that there was an item there. So we change it to AVAILABLE. This means not to stop searching for a key (since the key could be further along), but we can put a new key there.

10 Graphs

A graph has a set V of all **vertices**, and a set E of pairs of vertices called **edges**.

We know that:

$$\sum deg(v) = 2m$$

where m is the number of edges, n is number of vertices, and the degree of vertex v is the number of edges coming out of it.

A graph is connected if one can get from any vertex to any other vertex along the edges. Basically the graph does not have multiple isolated parts.

10.1 DFS

Depth First Search is a graph traversial algorithm.

We start at a random vertex, and then go to the next vertex, then the next one, then the next one, then when we reach the end, we **backtrack** to visit all the ones we missed.

We can use a **stack** to implement this. We go to the first vertex, get all edges, and push to the stack. Then we pop and follow that edge.

If that edge goes to a new vertex, we then repeat. If it leads to an already visited vertex, we then pop the next edge. Do this until the stack isEmpty.

Its complexity is $O(m + n)$ where the number of edges m can be up to n^2 if the graph is an adjacency list. If it is an adjacency matrix, it is always $O(n^2)$.

10.2 BFS

Breadth First Search is a graph traversial algorithm.

We start at a random vertex and then visit **all adjacent nodes**. Then of all the adjacent nodes visited, we visit all of their adjacent nodes. This creates a **layer** structure.

We use a **queue** to implement this. We go to the first vertex and add all edges to a queue,

then we dequeue, go to that vertex, and if not visited, queue all of its edges. Then we dequeue the next edge and repeat.

Its complexity is the same as DFS.

10.3 Shortest Path

A weighted graph is a graph where all edges have a weight assigned to them.

10.4 Dijkstra Algorithm

All vertices start with **distance of infinity**. We start at a vertex A and assign distance 0.

Then we take all of its incident edges and their weights, and then update all adjacent vertices with their new weights if new weights are better (weight of edge + weight of current verticy).

Now that vertex is in the cloud (done with it). Then we take the smallest weight edge of the entire graph.

We are done once all vertices are in the cloud.

We use a priority queue for this (min heap).

This is O(mlogn) for a heap, or $O(n^2)$ for an unsorted sequence.

10.5 MST

10.5.1 PrimJarnik

This is almost identical to Dijkstra except rather than keeping track of the whole shotest distance at each vertex, we will just check at vertex V going to K, if VK is shorter than the weight of K. (just weight of edge, do not consider weight of current vertex).

Runs in O(mlogn).

10.5.2 Kruskal Algorithm

With Kruskal, initially each vertex is its own cluster. Then we pick the lowest weight edge and check if each vertex it connects are in different clusters, if so, connect them, if not, move on.

We do this until we run out of edges.

This works in O((n+m)logn)

11 Sort

These are lots of sorting algorithms:

Type	Worst Case	Best Case	Average Case
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion	$O(n^2)$	O(n)	$O(n^2)$
Bubble	$O(n^2)$	O(n)	
Heap	O(nlogn)	O(nlogn)	O(nlogn)
Merge	O(nlogn)	O(nlogn)	O(nlogn)
Quick	$O(n^2)$		O(nlogn)

11.1 Mergesort

Mergesort is a **non in place algorithm** that breaks up the array into 2 parts, each of which gets sorted, and then merges them together.

This is a recursive algorithm. It gets broken down until there is just 1 element in each side, then it **merges** those 2 together.

11.2 Quicksort

Quick sort will choose a **pivot**, then sort the array by putting all elements **smaller than the pivot on one side**, and all **bigger than pivot on the other side**. Then we quicksort each side, finally we put both sides together.

This can be **in place**.

11.3 Bucket Sort

Bucket sort puts all items into a **fixed number of buckets** such as sorting by first letter of the name, there are 26 buckets.

If there are 2 things that go in the same bucket, we need to ensure to keep the same relative order. This is because Bucket Sort is a **stable** algorithm.

If we have n items and N buckets, it takes O(n+N).

11.4 Radix Sort

Radix sort is a bucket sort that can run more than one time. If we are sorting abc, aac, bba, aad,

We sort once by the first letter to get: abc, aac, aad, bba

Then we sort by the second letter to get: aac, aad, abc, bba

Then by the third letter to get: aac, aad, abc, bba

11.5 Quadratic Sorts

Insertion sort has a sorted part of the array, and then inserts an element from the unsorted part into the sorted part by comparing with each sorted element.

Selection sort looks for the minimum value then puts into index 0. Then looks for the

minimum value from 1-n and puts that into 1, then looks for minimum of 2-n and puts into 2...

Bubble sort goes though the array and swaps while the lower valus is smaller than the higher value. This means each iteration the max goes to the last spot in the array.