CS61B Midterm 2

Asymptotics

Notation	Meaning	Family members	
Big Theta $\Theta(N^2)$	Order of growth is EQUAL TO. Usually qualified by on average, worst case, etc.	$\frac{1}{2}N^2$, $2N^2$, N^2+30N	most precise
Big O $O(N^2)$	Order of growth is LESS THAN OR EQUAL to.	$5N^2$, $\log(N)$	upper bound only
Big Omega $\Omega(N^2)$	Order of growth is GREATER THAN OR EQUAL to	$2N^2$, N^3, 2^n	lower bound only
Tilde $f(n) \sim g(n)$	$\lim_{N\to\infty} \frac{f(n)}{g(n)} = 1$	$42N\log(N) + 50N$, $42N\log(N)$	both upper & lower bound

Analysis Tips

We analyze performance for LARGE N & ignore lower and constant multiplicative terms.

For Loops: Sequence usually takes form $\Theta(N)$ or $\Theta(\log(N))$

Nested For Loops: Series usually takes form $\Theta(N)$ or $\Theta(N^2)$

Arithmetic series: $1+2+3+4\ldots = \frac{n(n+1)}{2} = \Theta(N^2)$

Geometric series: $1+2+4+8\ldots = 2n-1 = \Theta(N)$

Recursion: Draw out recursion tree. Calculate work done at each level. Substitute in the height of the tree.

Ex: `return f(n/2) + f(n/2)`
$1+2+4\ldots+2^{\log_2 N} = N$

Ex: `return f(n/2) + g(n) + f(n/2)`
Each level: N Height: $\log_2 N$ = $N\log N$

Ex: `return f(n/8) + g(n) + f(n/2);`
Each level: $(\frac{5}{8})^L$ Height: $\log_2 N = N$

Ex: `return f(n-1) + f(n-2);`
Each level: 2^L $L=N$ $= 2^N$

Reminders

- $\log_x N \in \Theta(\log_y N)$
- $2^{\log_2 N} \in \Theta(N)$
- $2^{2N} = (2^2)^N = 4^N$
- array construction is $\Theta(N)$

$\log_a b = \frac{\log b}{\log a}$

Empirical Analysis

Assume $R(N) \sim aN^b$

N	time
500	0.08s
1000	1.22s
4000	5.15s
8000	43.60s

$$\frac{R(8000)}{R(4000)} = \frac{a\,8000^b}{a\,4000^b}$$

$$\frac{43.6s}{5.15s} = \left(\frac{8000}{4000}\right)^b$$

$$8.47 = 2^b$$

$$b = \log_2 8.47 = 3.08$$

$5.15s = a(4000)^{3.08}$

$a = 4.14 \times 10^{-11}$

$R(N) \sim 4.14 \times 10^{-11} N^{3.08}$ s

Pseudocode

```
public boolean isConnected {
    return find(p) == find(q)
}
private find(int p, int q) {
    while (p != parent[p])
        p = parent[p];
    return p;
```

```
public class WeightedQuickUnion {
    public WeightedQuickUnion(int N) {
        parent = new int[N];
        size = new int[N];
        for (int i = 0; i < N; i += 1) {
            parent[i] = i;
            size[i] = 1;
        }
    }
```

```
public void connect(int p, int q) {
    int rootP = find(p);
    int rootQ = find(q);
    if (size[rootP] < size[rootq]) {
        parent[rootp] = rootq;
        size[rootq] += size[rootp];
    } else {
        parent[rootq] = rootp;
        size[rootp] += size[rootq];
    }
```

Amortized Cost example (inserting into ArrayList)

Insert #:		0	1	2	3	4	5	6	7	8
Total Cost:		1	3	5	1	9	1	1	1	17
Amortized Cost:		5	5	5	5	5	5	5	5	5
Change in Potential:		4	2	0	4	-4	4	4	4	-12
Potential:	0	4	6	6	10	6	10	14	18	6

Choose a_i as low as possible to keep potential > 0. Since potential > 0, amortized analysis guarantees the operation is at most a_i

Disjoint Sets

Methods connect(int p, int q)
isConnected(int p, int q)

Uses Discover if a path exists between things quickly & efficiently

Implementations (from worst to best)

- **QuickFind** uses an array id[] of length N to store the set each object belongs to. To connect p and q, we set every object in p's set to be in q's set.
  - connect $\Theta(N)$
  - isConnected $\Theta(1)$

  NOT Possible:

0	1	2	3	4	5	6	7	8
3	1	4	4	4	4	3	4	4

  (3 must've been updated. All other 3's would've updated too)

- **QuickUnion** uses an array id[] of length N to store parent of object i. A find() method climbs ladder of parents until it reaches an object whose parent is itself (i == id[i]). To connect p and q, we set parent of p to the parent of q.
  - connect $\Theta(N)$ in worst case
  - isConnected $\Theta(N)$ in worst case

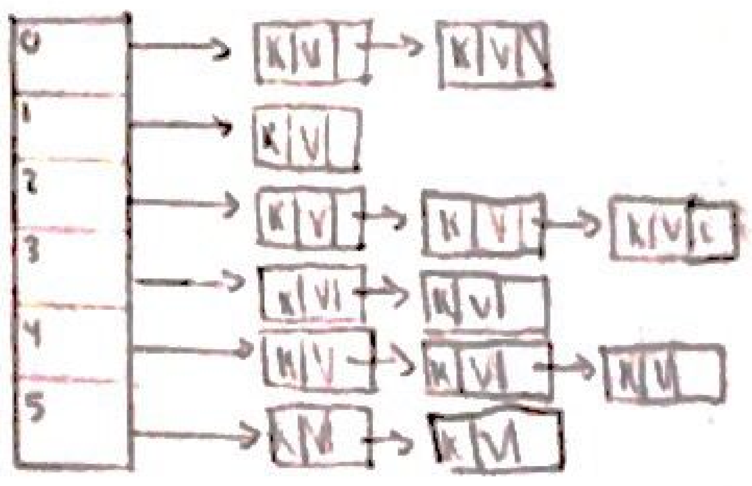
  Height best: 1
  Height worst: N−1

- **Weighted QuickUnion** Same as QuickUnion, but connect(p,q) changes the parent of the smaller tree (by # of nodes, not height) to the parent of the larger tree. Requires a new size[] array that's updated every connection. Tree heights are at most log N.
  - connect $\Theta(\log N)$ in worst case
  - isConnected $\Theta(\log N)$ in worst case

- **Weighted QuickUnion with Path Compression** When find is called, every node along the way is set to the parent index.
  - connect $\Theta(\log^* N)$ in worst case
  - isConnected $\Theta(\log^* N)$ in worst case

  $\}$ max 5 for any reasonable N

Hash Sets / Hash Tables

M=6 

Contains and insert operations are $\Theta(L)$, where L is the load factor = N/M. With external chaining with resizing, $L \in \Theta(1)$ amortized. Thus contains and insert are amortized constant time. (not guaranteed; assuming good spread)

Hash Functions

Must always return same hash for same objects, and should have infrequent collisions. Should override this method whenever equals is overridden.

Hashing collections:

```
int hash = 1;
for (Object item : collection) {
    hash = hash * 31;   <- some odd #, not power of 2
    hash += item.hashCode();
}
```

Hashing Recursive Structure:

```
if (this == null) return 0;
return this.value.hashCode() +
  31 * this.left.hashCode() +
  31 * 31 * this.right.hashCode();
```

Index into array usually computed as follows:

```
int index = (hashCode() & 0x7FFFFFFF) % M
                 ^ set sign bit to 0
```

Note that the multiplier shouldn't equal M. This is because the index will be dependent on the least significant entry; in the case of Java's String hashCode(), this is the last character in the string.

Uses Good for memoization. Can be used to solve 3Sum in $\Theta(N^2)$ time. Insert all elements into a hashmap. For each i and j (nested for loops), check if hashmap contains x - (A[i] + A[j])

Min/Max Heaps

A tree that obeys:

- Min/Max Heap Property: Every node is ≤/≥ (for min/max respectively) to its children.
- Complete: Missing items (if any) at bottom level and as far left as possible.

* We index the nodes level-order.

* Any BST can be represented as an array of length 2^n

Insert: Add to end of heap. Swim up hierarchy, swapping values until reaching correct place.

* To delete any node, same procedure. Just swimup and sink.

Delete min/max: Put last item into the root. Swim down hiearchy, swapping values w/ smaller (if min)/larger (if max) of 2 children.

Implemented as an array of keys, offset by 1, because of completeness.

- add $\Theta(\log N)$
- getSmallest $\Theta(1)$
- removeSmallest $\Theta(\log N)$

(amortized due to resizes)

leftChild(i) = 2i
rightChild(i) = 2i+1
parent(i) = floor($\frac{i}{2}$)

Uses 1/3 as much memory as BST, and allows duplicates.

Priority Queue API

```
public interface MinPQ<Item> {
    public void add(Item x);
    public Item getSmallest();
    public Item removeSmallest();
    public int size();
}
```

Best implemented with a binary heap.

Tree Traversals

Depth First Traversals $\Theta(N)$

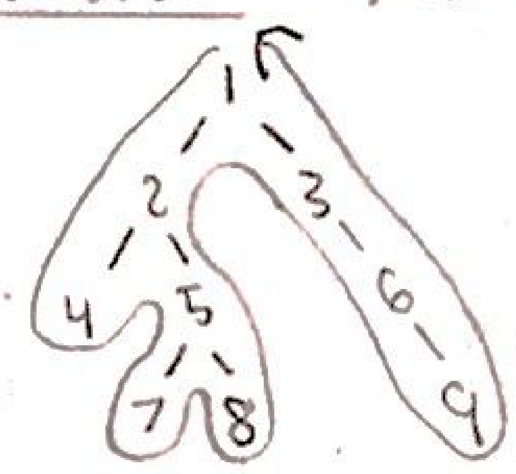
- Pre-order: Action, L, R
- In order: L, Action, R
- Post-order: L, R, Action

Walk from top of graph, going counterclockwise

- Shout when we pass left
- Shout when we pass bottom
- Shout when we pass right

good for file size finding

(not a BST)



Pre: 1 2 4 5 7 8 3 6 9
In: 4 2 7 5 8 1 3 6 9
Post: 4 7 8 5 2 9 6 3 1

Level Order Traversal $\Theta(N)$ full tree, $\Theta(N^2)$ spindly tree

Iterative Deepening

```
public void levelOrder(Tree T, Action do) {
    for (int i = 0; i < T.height(); i += 1) {
        visitLevel(T, i, do);
    }
}
public void visitLevel(Tree T, int level, Action do) {
    if (T == null) return;
    if (level == 0) do.visit(T.key);
    else {
        visitLevel(T.left(), level - 1);
        visitLevel(T.right(), level - 1);
    }
}
```

```
public class MinPQ<Key implements Comparable> {
    private Key[] pq;
    private int N;
    public MinPQ(int size) {
        pq = (Key[]) new Object[size];
        N = 0;
    }
    public Key min() {
        return pq[1];
    }
    public void insert(Key x) {
        N += 1;
        pq[N] = x;
        swim(N);
    }
    public Key removeMin() {
        swap(1, N);
        Key min = pq[N--];
        sink(1);
        pq[N+1] = null;
        return min;
    }

    private void swim(int k) {
        while (k > 1 && pq[k/2] >
               pq[k]) {
            swap(k, k/2);
            k = k/2;
        }
    }

    private void sink(int k) {
        while (2*k <= N) {
            int j = 2*k;
            if (pq[j] > pq[j+1]) j+
            if (pq[k] < pq[j]) brea
            swap(k, j);
            k = j;
        }
    }
}
```

Binary Search Tree

Uses Can be used to implement an ordered map/set. When balanced, max/min are $\Theta(\log N)$, range queries are $\Theta(\log N + Q)$, and iteration is simple. Also decent as a priority queue.

Best height: (int) $\log_2 N$
Worst height: $N-1$

BST property

For every node X in the tree, every key in the left subtree is less than X's key and every key in the right subtree is greater than X's key. No duplicates allowed w/o modification.

Insertion (for a BST set)

```
public BST insert(BST T, Key k) {
    if (T == null) return new BST(k);
    else if (k < T.entry()) T.left = insert(T.left, k);
    else if (k > T.entry()) T.right = insert(T.right, k);
    return T;
}
```

Deletion

Case 1 (No children): Delete node
Case 2 (1 child): Move child up
Case 3 (2 children): Find leftmost node (min) or rightmost value (max) in right or left subtree, respectively. Replace node with this found node. Then delete the found min or max.

worst case = $\Theta(h+1)$

Delete 20

remove node, combine children, move up

On random data, insertion & deletion is $\Theta(\log N)$
On sorted data, insertion & deletion is $\Theta(N)$

B Trees (specifically 2-3 Trees)

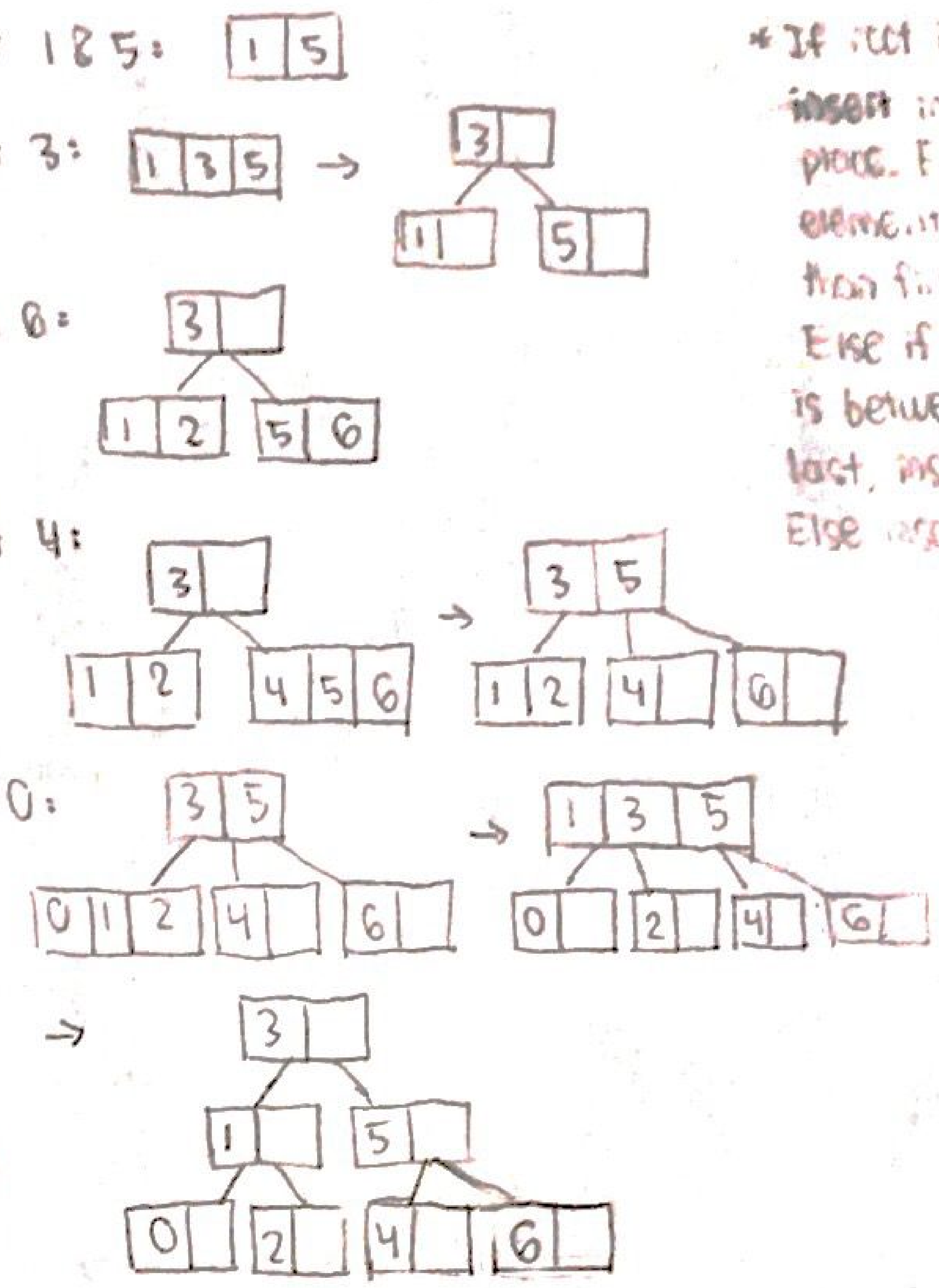
A B tree of order $M=3$ consists of nodes with at most $M-1$ data elements and at most M children. They are perfectly balanced, with height between $\log_M(N)$ and $\log_2(N)$. Thus insertion & deletion is $\Theta(\log N)$.

T is a 2-3 tree iff the following holds:
T is a node with left subtree L, middle subtree M, and right subtree R, and data elements a and b.
- L, M, R are equal height
- a > each element in L and ≤ each element in M
- b > each element in M and ≤ each element in R.

Downsides: Difficult implementation, conversion of node types necessary

2-3 Tree Insertion Example (1, 5, 3, 6, 2, 4, 0)

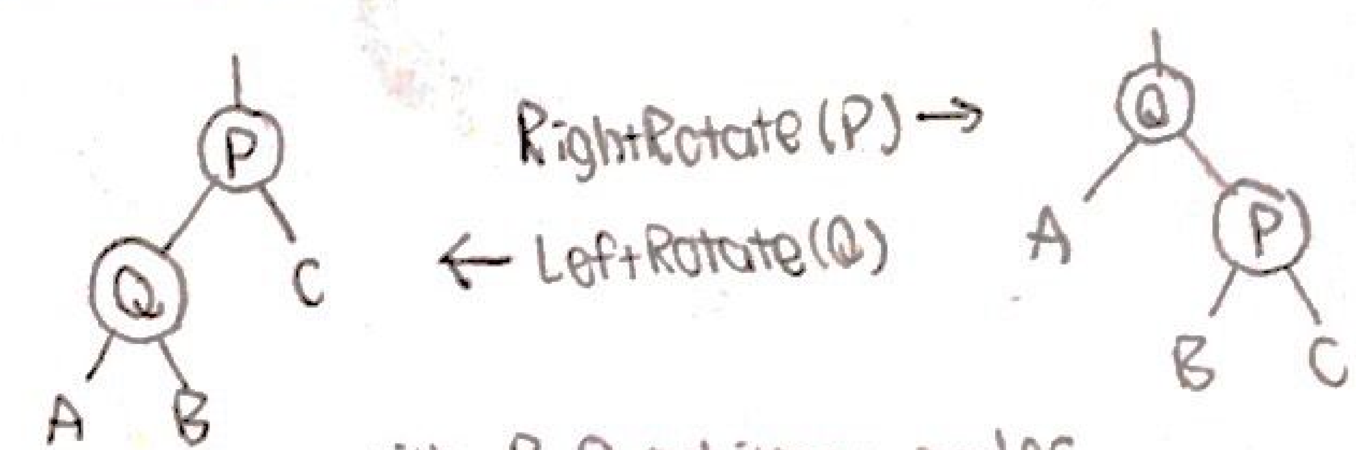
Insert 1 & 5:
Insert 3:
Insert 6 & 2:
Insert 4:
Insert 0:



* If root is leaf, insert into sorted place. Else if element is less than first, insert(T.[illegible] Else if element is between first & last, insert(T.middle). Else insert(T.right)

Tree Rotation

Invertible, $\Theta(1)$ operation



with P, Q arbitrary nodes & A, B, C arbitrary subtrees

Left-Leaning Red-Black Tree

* Guaranteed fastest for any data set

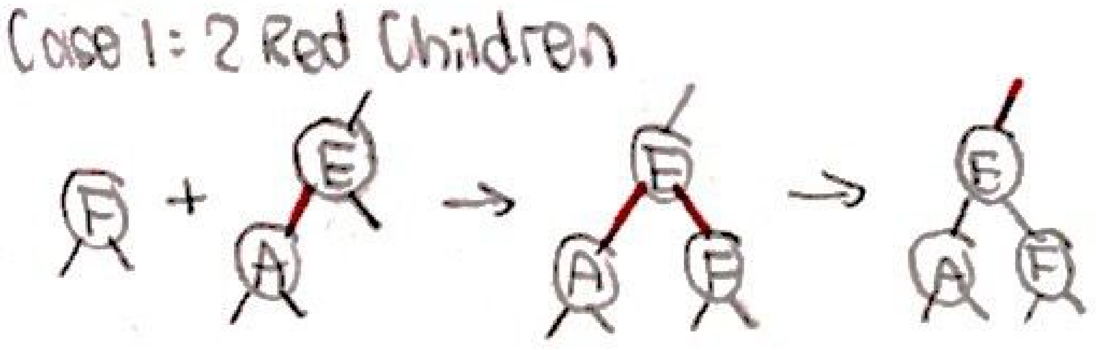
A BST such that
- No node has 2 red links
- Every path from root to leaf has same number of black links
- Red links occur on the left

Max height $\Theta(2\log N)$
- To get worst case, stuff as many 3 nodes on the left

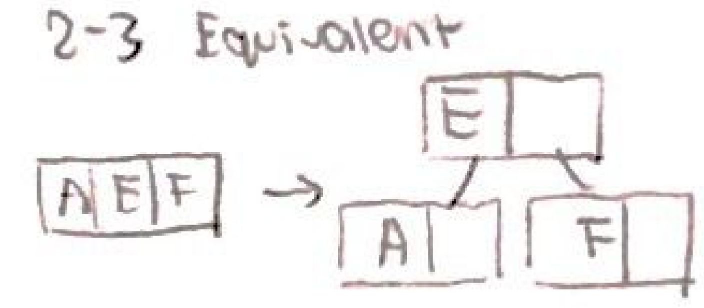
There exists an isometry between 2-3 trees & LLRB trees. They mimic 2-3 tree behavior using color flipping and tree rotation. Easier to implement & constant factor better than B-trees.

Resolving Violations

Case 1: 2 Red Children



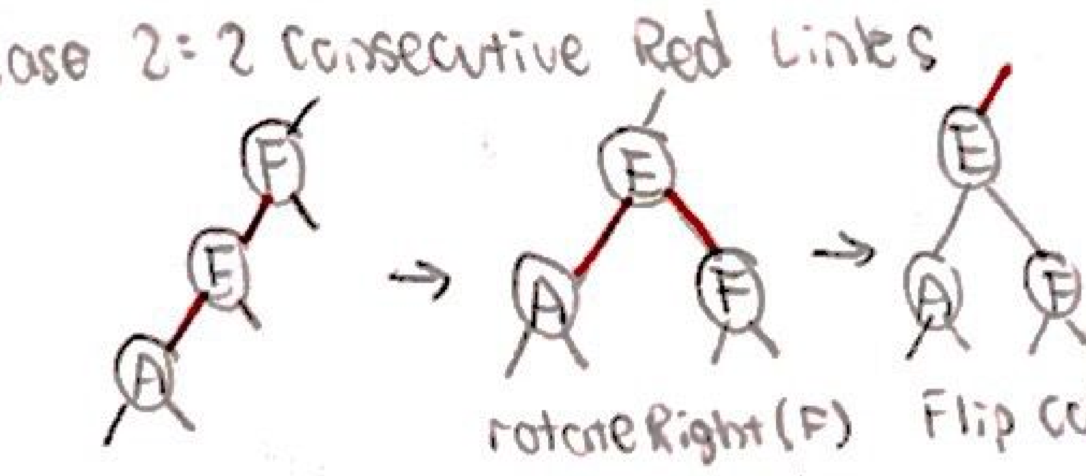
2-3 Equivalent



* Red links represent connected nodes in 2-3 Tree.

Insertion & Deletion $\Theta(\log N)$ guaranteed

Case 2: 2 Consecutive Red Links



Case 3: Left-Red-Right-Red

Clever Uses of Multiple Data Structures

Level Order Traversal using queue

1) Create empty queue q
2) Node = root
3) while (node != null)
   a) action on node.key
   b) enqueue node's left then right child to q
   c) dequeue a node from q and assign to node

Median finder ADT using 2 heaps

Left = max heap
Right = min heap

Insert x into Left if x < root of Left, else insert right. If after inserting Left.size() > 1 + Right.size(), then Right.add(Left.RemoveMax()). Else if Right.size() > Left.Size(), then Left.add(Right.RemoveMin()).

The median is the root of Left.

Huffman Coding

Compress: 1) Calculate frequencies of all symbols in file.
2) Assign each symbol to a node. Take two nodes with smallest relative frequency and merge them into supernode. Repeat until everything in a binary trie. Write trie and codewords to compressed file.

Decompress: Read in binary trie. Use longestMatchingPrefix() to walk through trie, outputing symbols when appropriate.

Implement Max Heap with Min Heap (or vice-versa)

For every insert operation negate the number. For any removeMax()/removeMin() operation, negate it again.

Min() method to a Max Heap (or vice-versa)

Maintain index to min, updating every insert. The index will still be correct after delete operations.

QuickSelect: Find k^{th} smallest element in array. Not used in practice, but could theoretically guarantee $\Theta(N\log N)$ for quicksort by picking median for pivot. Idea is to partition until pivot is in desired array location, discarding 1 half of each partition at a time.

Best: $\Theta(N)$ Avg: $\Theta(N)$ Worst: $\Theta(N^2)$

Sorting Lower Bound Need at least $\log(N!)$ to sort N items with comparisons. Since $\log(N!) \in \Theta(N\log N)$, we can say best possible comparison algorithm $\in \Omega(N\log N)$

Sorting Algorithm Summary

Algorithm	Stable?	Best	Avg	Worst	Space	Notes
Selection Sort	N	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(1)$	
Naive heapsort	N	$\Theta(N)$ identical only	$\Theta(N\log N)$	$\Theta(N\log N)$	$\Theta(N)$	Bad caching performance
In-place heapsort	N	$\Theta(N)$ identical only	$\Theta(N\log N)$	$\Theta(N\log N)$	$\Theta(1)$	Bad caching performance
Mergesort	Y	$\Theta(N\log N)$	$\Theta(N\log N)$	$\Theta(N\log N)$	$\Theta(N)$	Fastest stable sort
Insertion Sort	Y	$\Theta(N)$ sorted	$\Theta(N^2)$	$\Theta(N^2)$ reversed	$\Theta(1)$	Best for small or sorted data; $\Theta(N+K)$ where K is # of inversions.
Quicksort	N* depends on partitioning	$\Theta(N\log N)$	$\Theta(N\log N)$	$\Theta(N^2)$	$\Theta(\log N)$	Fastest sort
Non-comparison sorts						
Counting Sort	Y	$\Theta(N+k)$	$\Theta(N+k)$	$\Theta(N+k)$	$\Theta(N+k)$	k = size of alphabet. Suitable for N > k
LSD Radix Sort	Y	$\Theta(W(N+k))$	$\Theta(W(N+K))$	$\Theta(W(N+K))$	$\Theta(N+k)$	W = length of longest key.
MSD Radix Sort	Y	$\Theta(N+k)$	$\Theta(W(N+K))$	$\Theta(W(N+K))$	$\Theta(N+Wk)$ W recursive levels	

Counting Sort

Let k be the range of input values. Create array of size k to store counts. Count occurrences of each item in array. Iterate through counts array to calculate starting indices of each item. Lastly put items into new array, checking and incrementing target index each time. Good for sorting large collections of items belonging to an alphabet << N.

LSD Radix

Good for similar strings, integers

From right to left, run counting sort on each digit/character. This works because counting sort is stable. For nonequal key lengths, we must pad. Inefficient for dissimilar string sorting or W >> avg length.

MSD Radix

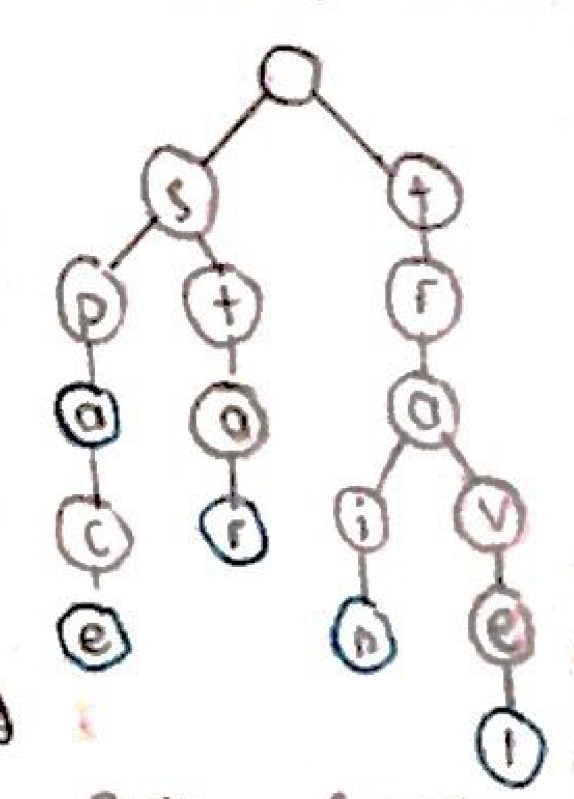
Good for strings

Run counting sort on leftmost digit/character. Recursively call MSD sort on each resulting partition at the next index.

Tries

A set/map data structure implemented as a tree of (usually) String keys. Main use is to support character-based operations like prefix matching (by gathering all children that are words). Child links implemented implicitly; for optimal memory usage, should use a Map<Char, Node> rather than char-indexed array.

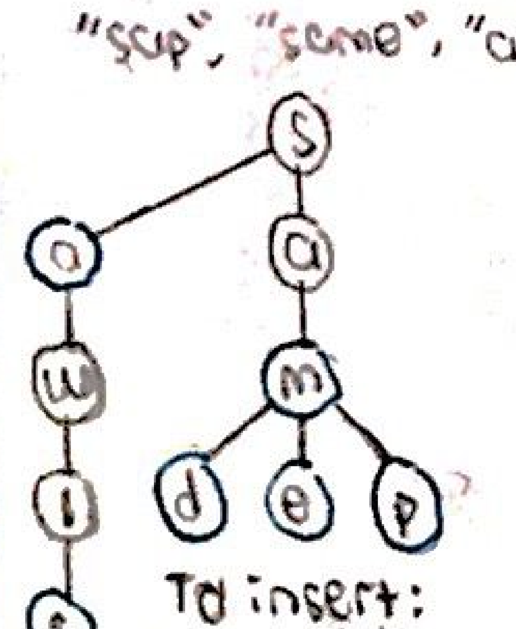
Insert "spa", "space", "star", "train", "travel"



Ternary Search Trie

*slower, but better memory

Insert "sam", "sad", "sap", "same", "a", "awls"



To insert:
i = 0
if key[i] == node: go mid, i++;
else: if key[i] < node: go left
if key[i] > node: go right

Search runtime summary

	Worst	Best	Space
Balanced BST	$\Theta(L\log N)$	$\Theta(L)$	$\Theta(NL)$
Hash Table	$\Theta(L^*)$	$\Theta(L^*)$	$\Theta(NL)$
Trie (array)	$\Theta(L)$	$\Theta(1)$	$\Theta(NLK)$
Trie (Hashmap)	$\Theta(L)$	$\Theta(1)$	$\Theta(NL)$
Trie (TreeMap)	$\Theta(L\log K)$	$\Theta(1)$	$\Theta(NL)$
TST	$\Theta(N)$	$\Theta(1)$	$\Theta(NL)$

log N on average

L = digits in key
K = alphabet size

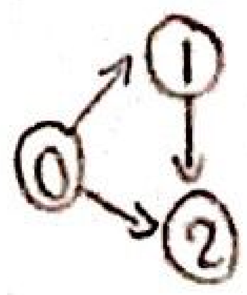
Other uses include spellchecking, longestPrefixOf

CS61B Final Study Guide

Graphs

Adjacency Matrix

	0	1	2
0	0	1	1
1	0	0	1
2	0	0	0



Matrix of ints/booleans indicating prescense of edge between vertices.

Pro: fast edge lookup $\Theta(1)$

Con: $\Theta(V^2)$ memory (symmetric for undirected graphs) and $\Theta(V)$ for .adj(v) lookup

Adjacency List

Array of lists indexed by vertex (or held by node object)

```
0 [ ] → [1,2]
1 [ ] → [2]
2 [ ] → []
```

(for above graph)

Pro: $\Theta(E+V)$ memory, fast iteration, simple

Con: hasEdge(s,t) slower Θ(degree(v))

Depth First Paths

Finds a path from s to every other reachable vertex, visiting each vertex at most once.

```
boolean[] marked;
int[] edgeTo;      ← (parent)

dfs(Graph G, int v) {
    marked[v] = true;
    for (int w: G.adj(v)) {
        if (!marked[w]) {
            edgeTo[w] = v;
            dfs(G, w);
        }
    }
}
```

$\Theta(E+V)$ time

$\Theta(V)$ space

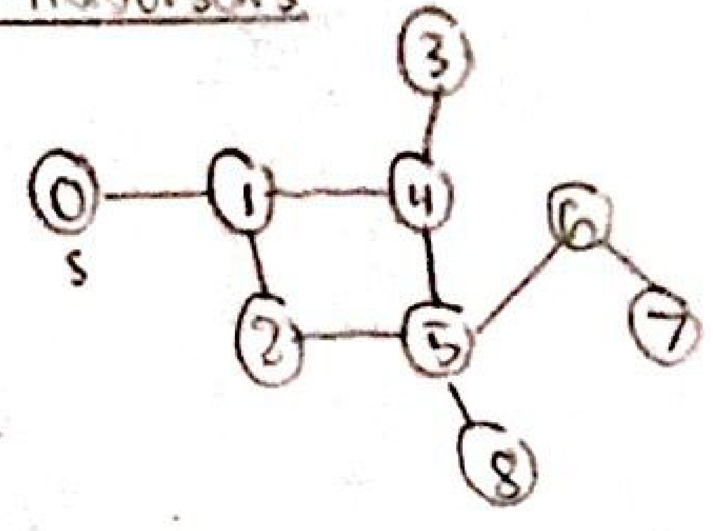
* hasPathTo(int v) return marked[v];
* retrace edges to get path

This illustrates the idea of depth first search, guaranteed to reach every vertex. Can be implemented iteratively using stack instead of queue as in BFS. Must mark as visited after popping, however. (If marked, continue)

Applications:

- Connected Components: partition a graph into connected subgraphs. Run DFS on every vertex if not already marked by a previous DFS. By incrementing a counter each time DFS is run, can partition vertices into distinct subgraphs.
- Cycle detection: return true if during DFS, we encounter adjacent vertex already marked that is not the parent of the node. (For digraphs, use onStack[] array to model nodes on DFS call stack)

Graph Traversals



DFS Preorder: order of DFS calls

0 1, 2 5 4 3 6 7 8

DFS Postorder: order of returns from DFS calls

3 4 7 6 8 5 2 1 0

Level Order / BFS: visit all of node's children before visiting any child's children

Topological Sort

Find an ordering of vertices consistent with directed edges. (only for DAGs)

- Perform DFS traversal from every vertex with indegree 0 (no edges coming in), NOT clearing markings in between traversals. A valid topological sort is given by the reverse of the concatenated DFS postorders.

$\Theta(E+V)$ time, $\Theta(V)$ space

BreadthFirstPaths

Finds shortest path from s to every reachable vertex.

```
boolean[] marked;
int[] edgeTo;        * forms a SPT

bfs(Graph g, int v) {
    Queue<Integer> q = new Queue();
    q.enqueue(v);
    marked[v] = true;
    while (!q.isEmpty()) {
        int v = q.dequeue();
        for (int w: G.adj(v)) {
            if (!marked[w]) {
                q.enqueue(w);
                marked[w] = true;
                edgeTo[w] = v;
            }
        }
    }
}
```

$\Theta(E+V)$ time

$\Theta(V)$ space

Applications:

- Test bipartiteness: starting at any vertex, give 0 label to starting vertex, 1 to its neighbors, 0 to its neighbor's neighbors, etc. If at any point a vertex has neighbors with same label as itself, return false.
- Counting degrees of separation

Shortest Path-Finding with Weighted Edges

For a given start vertex, the shortest paths tree contains V-1 vertices

Dijktra's Algorithm

Shortest path from s to every vertex (w/ weighted edges)

* Note: fails for negative edge weights

Idea is to visit vertices in order of best-known distance from source, relaxing (adding to SPT if better than current distTo) each edge from visited vertex.

1) Initialize edgeTo[] and distTo[] arrays, with distTo[i] = ∞ for all i ≠ s.

* but we could use marked[] array instead, enqueuing during relaxation

2) Insert all vertices into Priority Queue, ordered by distTo

3) while (!pq.isEmpty())

* no explicit marking; consider it marked by removal from PQ

```
    v = pq.delMin()
    for each edge e in G.adj(v):
        if distTo[dest] > distTo[v] + e.weight:
            distTo[dest] = distTo[v] + e.weight
            edgeTo[dest] = v
            updatePriority(dest, distTo[dest]) if on pq
```

* once a node has been dequeued, the path to that node is guaranteed shortest (but others may change)

Runtime

$O(V\log V + V\log V + E\log V) = O(E\log V)$

$\Theta(V)$ space

A*

Special case of Dijkstra's algorithm for when there is a single target in mind. Essentially the same implementation, except PQ is ordered by distTo[v] + h(v, dest) where h is a hueristic (ie. Euclidean distance). Guaranteed correct as long as hueristic does not overestimate distance. Terminates when goal node is reached for first time. For large graphs, better to have a HashMap<Node, priority> than inserting everything into PQ. Simply check that node is not in map, OR priority is less than currently best known one. (Instead of updating priorities on proj3, I just inserted the new, improved node lol)

Minimum Spanning Trees

A subgraph of an undirected graph that is connected, acyclic (thus a tree), and includes all vertices of minimum total weight. May or may not be the same as SPT of a vertex in the graph. If each edge has a distinct weight, there is a unique MST.

Note: adding constants to edge weights does not affect MST

→ Prims Algorithm $O(E\log V)$

Starting from any arbitrary vertex, repeatedly add the shortest edge that connects a vertex in the current tree to an outside vertex. Essentially same logic as Dijkstra's algorithm, but distTo[] is set to edge weight (dist from tree) rather than edge weight + distance from source.

→ Kruskal's Algorithm $O(E\log V)$ w/ sorted edges, $O(E\log E)$ w/ pq

Consider each edge from least to greatest weight. If adding the edge does not result in a cycle, add it to the MST.

```
while (!pq.isEmpty)
    Edge e = pq.delMin()
    if (!uf.connected(sourceV, destV))
        uf.union(sourceV, destV)
        mst.add(e)
```

Note: These algorithms fail on directed graphs (unlike Dijkstras).

Cut Property: Let S & T be two disjoint sets formed by arbitrarily partitioning the vertices of the graph. The minimum crossing edge for any partition is part of the MST.

Cycle Property: Given any cycle in an edge-weighted graph, the edge of maximum weight in the cycle does not belong to the MST.

Note: Since SPT is a spanning tree, the total weight of the MST is always ≤ any SPT.

DAGSPT Slight improvement over Dijkstra's algorithm for graphs that are DAGs. Simply find a topological ordering of vertices, and then relax the edges in that order. Runtime is $O(E+V)$ and works for negative weights.

DAG Longest Paths Tree

When relaxing an edge, flip sign of inequality. Equivalently, negate the edge weights. Then do DAGSPT

Dynamic Programming LLIS example

*equivalent to finding most edges path in a DAG

```
int[] Q = new int[N]
Q[0] = 1, Q[k] = -∞
for (k=1; k<N; k++)
    for (j=0; j<k-1; j++)
O(N^2)  if (item j < item k and Q[j]+1 > Q[k])
            Q[k] = Q[j] + 1
```

Sorting

Selection Sort

Starting at index 0, repeatedly swap element at current index with the minimum element from current index to N-1, inclusive.

Naive Heapsort

Same as selection sort, but use heap-based PQ. Insert all items into max heap and fill array in reverse.

In-Place Heapsort

1) $O(N) \rightarrow$ Bottom-Up heapify: Treat array as max heap. Delmax (sink) in reverse level order (i.e. reverse iteration). Note: Parent is $\text{floor}(\frac{i-1}{2})$ since no empty index 0. Works because after sinking, tree rooted at k guaranteed heap
2) $O(N\log N)$ Delete max, swapping with last item in the active heap. Sink node down. Size of active heap decreases from N to 0 elements as size of sorted list grows from 0 to N.

Mergesort

If hi ≤ lo return. Else, mergesort two roughly equal halves. Then merge the arrays by initializing indices at the starts and incrementing indices when we take the smaller of the two.

Insertion Sort

Initialize i = 1. Invariant: elements to left of i are sorted. Swap element i left one-by-one until in its proper place, incrementing i each iteration.

Quicksort

Choose a pivot to partition on. Hoare Partitioning: Pivot at lo. Initialize l = pivot + 1, r = hi. Increment l until a[l] ≥ pivot. Decrement r until a[r] < pivot. Swap these elements, and increment/decrement l/r. Repeat until l ≥ r. Swap pivot with r. Recursively partition new subarrays.

Note that worst case (extremely rare) is caused by pivot being least or greatest entry, in which case portion only decreases by 1. Avg case can be achieved by selecting random pivot or shuffling array. In practice, its the fastest sorting algorithm.