Data Structures

Introductory ones:

- Linked Lists
  - set of nodes w/ a next pointer.

  - Finding = $O(n)$
    Inserting = $O(1)$
    Deleting = $O(N)$
- Arrays
  - containers that hold a fixed # of objects.
  - Finding: (Given Index) $O(1)$ (Not Given) $O(N)$
    Inserting: $O(N)$ [w/ resizing]
    Deleting: $O(N)$ [need to shift items]
- Stacks
  - First in, last out.
  - Push → put item on top → $O(1)$
    peek → see the top item → $O(1)$
    pop → remove top item → $O(1)$
- Queue
  - First in, First out.

Disjoint Sets

- Quick Find
  - Keep 1 array, indices are index, Values are the parents.

    before call:

1	2	3	4	5
1	2	2	4	4

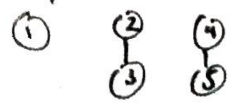
    After union(2,4):

1	2	3	4	5
1	4	4	4	4

  - Union: worst case = $O(n)$
    Find: $O(1)$ * indexing into an array.

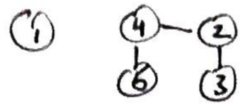
Quick Union:

- Connect parent of one node to parent of other.
- parent: 1 2 2 4 4 :
  index: 1 2 3 4 5



After Union (3, 5)

parent: 1 4 2 4 4 :
index: 1 2 3 4 5



- find: $\Theta(n)$ worst case
  Union: $\Theta(n)$ worst case

  BUT that's only when the "tree" is not bushy.

  when bushy:
  find: $\Theta(\log n)$
  Union: $\Theta(\log n)$

Weighted Quick Union

- Same as Quick Union, but now we make the smaller tree's root go under the larger tree's root, and size is determined by the amount of nodes in the tree.
- ensures that the tree is of $\log(n)$ height.
- Union: $\Theta(\log n)$ worst case.
  Find: $\Theta(\log n)$ worst case.

WQU with Path Compression.

- When find() is called, every node along the way points to the root.
- Making M calls to union and find w/ N objects results in no more than $O(M \log^* N)$. $\log^*$ is at most 5, so,
- Union: almost constant time $\Theta(1)$
  Find: almost constant time: $\Theta(1)$

In Summary:

Algorithms	Union/Connect RT.	Find/isConnected RT.
Quick Find	N	1
Quick Union	Tree height	Tree height
Weighted Quick Union	log n	log n
WQU w/ Path Compression	~ constant	~ constant.

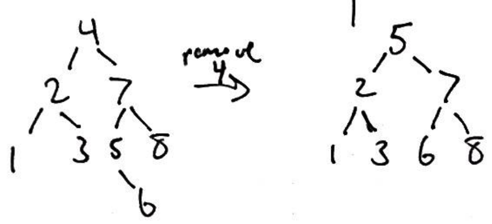
Trees, BSTs.

Trees

- Set of nodes connected by a set of edges. If nodes have 0 children, then considered a leaf.
- Every child of a tree is also a tree
- Constraint: There can only be one path from 1 to another node.
- Node that is not the child of any other is the root.

Binary Search Trees

- Way of organizing comparable nodes.
- Nodes have 0-2 children.
- Children on left always less than parent. Vice versa for right.
- Inserting: O(height of tree) [generally log N]
  Getting: O(height of tree) [generally log N]
- Deleting / Removing
  * A leaf: just remove it
  * A node w/ 1 child: remove it, make parent point to child.
  * A node w/ 2 children:
    Replace the deleted node w/ either the node that is the greatest on its left side or the node that is the least on its right side. When removed, run the code to connect the old parent to the node's child.



- In the worst case, insert will be $\Omega(\log N)$. This is when we have a bushy tree.

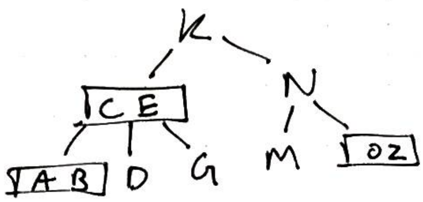
Balanced Search Trees

- Will always have a logarithmic height!

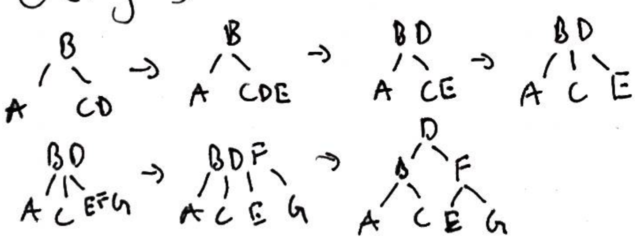
2-3 trees

- Can have a node w/ 2 elements in it.
- Can have 0 or 3 children for 2 element node.
- Can have 0 or 2 children for 1 element nodes.

Example:



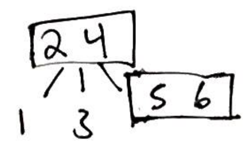
- Finding elements is very similar to Binary Trees.
  - if we run into a "1 node" just proceed normally.
  - if we run into a "2 node," check if we need left, mid, or right value.
  - Recursively repeat.
- Inserting elements:
  - No matter what you insert, it will go into a leaf.
  - If the node is a single node, make it a 2-node & done.
  - If node becomes 3-node, promote middle element.
  - Split children so that parent has 1+ # of items in node
  - If node becomes a 2-node after promotion, then done. otherwise, promote middle element again.
  - keep repeating unless node is the root.
  - If root becomes a 3-node, then middle child becomes new root. Left item= Left child, Right item= Right child.
- All 2-3 trees have log(N) height, so never exceeds $\Theta(\log N)$.



Red-Black Trees

- Each Red-Black tree can be represented by a 2-3 tree.
- R-B trees have log(N) height & easier to implement than 2-3 trees.
- LLRBs have 2 distinct edges.
  - Black edge = parent-child relationship.
  - Red edge = "same node" relationship.
- Rules:
  1) Red links only lean left.
  2) No node has 2 red links.
  3) Every leaf has same amount of black links above it.

is equivalent to



- Finding
  - Same as a BST.
- Inserting into LLRB
  - A node is always inserted w/ red link.
  - If inserting causes a problem, take care of the smaller subproblem which only contains the inserted node's parent & potential other child.
  - If item inserted to left, no problem.
  - If item inserted to right & parent doesn't have a 2nd child, move child in a rotation w/ parent to make link be on left.
  - If item inserted to right & parent has 2 children, perform a color flip where parent's link turns red & child links are black.
- height & runtime = log(N).