**Module 1 – Basic Data Structures**

**1.1 Arrays and Linked Lists**

Arrays:

* Contiguous block of memory, elements are the same size
* Reading and writing are constant-time (O(1)) operations
* Adding to end is constant time
* Adding elements in the start or the middle are O(n)

Singly-linked lists:

* Head 🡪 Node = value + pointer to next node 🡪Node …
* Adding to front: constant-time (allocate and update 2 pointers)
* Pop and return also constant-time
* At the tail order(n) unless you have a tail pointer, then constant time
  + Except popping the last value – still have to go to the head and walk all the way down!
* Similarly adding after a node is easy, adding before a node is tricky

Doubly-linked lists

* Next and prev pointers – constant time to do operations at the end too

Arrays vs linked lists:

* Arrays are random access: constant time read and write for any element
* Linked lists allow constant time addition at the front (and back or anywhere else, if doubly linked)
* For linked lists, elements need not be contiguous in memory

**1.2 Stacks and Queues – abstract data types**

Stacks: push(), top() (return) and pop(), isEmpty()

* Can implement with an array, appending and keeping track of num elements – all constant time operations
* Can implement with linked list too (to get around max allocated space of array, also wasted space of pre-allocated array) – generally use PushFront() if singly linked list – no a priori limit on size or wasted space, but more memory overhead per used element
  + Also all constant time operations
* Last in first out (LIFO), queue

Queues: enqueue(), dequeue(), empty()

* FIFO data type, i.e. for servers
* Implement with a linked list, enqueue adds at tail, dequeue pops from head
* With an array, dequeueing from front would be order(n) in a naïve implementation
  + Instead, maintain a read and write index for first and last in – then change the read index (start of the array) instead of shifting all elements up in the array
  + At the end of the allocated array, write index will wrap back around to the front
  + Will error when read = write – need buffer of at least one index to ensure read and write are distinct (can only have read and write index = 1 🡪empty()=true)

**1.3 Trees**

Binary search tree: at most two children at each node, root node is >= all left children and <= all right children

* Like binary search in a sorted array
* Root = top node, child is directly below parent, notion of ancestors, descendants, etc.
* Leaf = node with no children, leaf has height=1
* Level = 1 + num edges between root and node

In general trees, ach node has key, children pointers if exist, optionally parent pointers

Binary trees: at most two: key, left, right

Procedures: height, size (can define these recursively)

Tree traversal:

1. Depth first – use a stack
   1. InOrderTraversal() – will print elements in order **if it’s a binary search tree** as defined above – only makes sense for binary trees
   2. PreOrderTraversal(), PostOrderTraversal() – well-defined for general trees
   3. Each call to a procedure invokes another frame on the stack – saving info about where we are in the tree, on the stack
2. Breadth first – use a queue
   1. Enqueue and dequeue/print, one level at a time

Python notes:

* Python lists are dynamic arrays, so accessing items is constant time
  + Looking up items is order(n) though
  + Sets/dicts use hash tables so lookup is constant time
* When items are appended, the array has to be resized – when this happens, python gives it some extra space so the next few calls don’t require further resizing (how much?)

**Module 2 – Dynamic Arrays and Amortized Analysis**

**2.1 Dynamic arrays**

Dynamic arrays – allocate and determine size at run time

* What if you don’t know the size to allocate at runtime?
  + Store a pointer to a dynamically allocated array – can allocate a new array if needed, copy, redirect pointer
* Therefore a dynamic array is distinct from a dynamically allocated array that can’t be resized once created
* Operations:
  + get, set for particular indexes – constant time as with static arrays
  + add to end, remove elements, etc.
* need to store pointer, capacity and current size
* Run times:
  + Get, set = constant time
  + PushBack (append) is worst case O(n), if new array has to be allocated and all n elements copied
  + Remove is O(n) to shift elements

**2.2-4 Amortized analysis: aggregate method, banker’s method**

Worst-case cost could be overly conservative, i.e. in dynamic arrays, adding elements is usually constant time but O(n) when resizing is needed

Aggregate method:

* Amortized cost = cost(n operations) / n
* N calls to PushBack(), say we start with 2 elements and double for each resize
* Ci = 1 + (i-1), if (i-1) is a power of 2, 1 otherwise

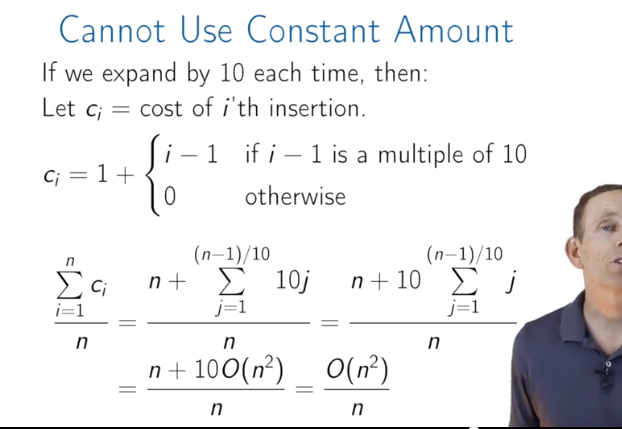
Banker’s method:

* Charge extra for cheap operation, save extra charge conceptually and use to “pay for” expensive operations
* N calls to PushBack():
  + Charge 3 for each insertion, 1 token is raw cost for each
  + If resize needed, use token on each element that has to move, place one token on newly inserted element and one token on element capacity/2 elements prior
  + Notion of prepaying future moves – deferred payment (mental accounting only)
  + For this dynamic array, we charged 3 for each insertion to “account for” all moves
    - So since every insertion costs 3, this is O(1) = constant time amortized cost

Physicist’s method:

* Notion of a potential function
  + phi(h0) = 0
  + phi(ht) >= 0
* amortized cost for operation t:
  + ct + phi(ht) – phi(ht-1)
  + choose phi so if cost is small, potential increases for later work, if cost is large, potential decreases as we pay for the work
  + summing over all costs and canceling out, sum over c’s + phi(hn) – phi(h0)
  + which is greater than or equal to sum over c’s since phi(hn) is nonneg.
* N calls to PushBack()
* Let phi(h) = 2\*size – cap
  + Can verify is a potential function
* If not resizing, ci + 2\*(size\_i – size\_i-1) = 3
* If resizing, size\_i-1 = cap\_i-1 = “k”
  + Phi(h\_i-1) = 2\*k – k = k
  + Phi(h\_i) = 2\*(k+1) – 2k since adding one element and doubling capacity = 2
  + Amortized cost = size(i) + 2 – k = k+1 + 2 – k = 3, again

Summary of amortized analysis:

* What if we used a different growth factor, i.e., multiplicative factor = works fine
* Can’t use a constant amount
  + If you use aggregate method e.g. for adding 10 whenever array is full (multiples of 10 are reached), then the amortized cost is O(n2)/n = O(n) 

**Module 3 Priority Queues**

Basic operations:

* Insert(p)
* ExtractMax()

Naïve implementation: e.g. in array or doubly-linked list

* Insert(k): add k to end – constant time
* ExtractMax(): O(n) because need to scan the whole array

Could try storing in a sorted array

* ExtractMax() is now constant time
* But, Insert(k) is now O(n) due to shifting (although the searching part is O(log n) using binary search)

Could try using a sorted doubly-linked list:

* ExtractMax(): constant time
* Insert(k): changing the pointers is constant time, but the position finding is O(n) since you can’t use binary search on a list
  + Side note – binary search is O(log n) since the depth of the equivalent binary comparison tree is floor(log2(n)+1)

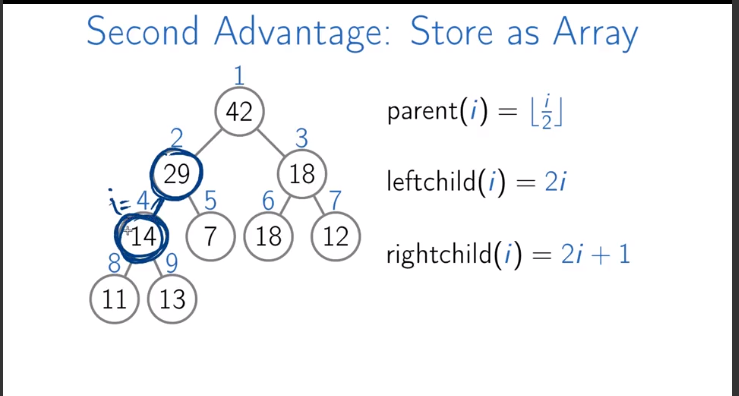
**Binary Heaps**

Binary max-heap: binary tree (each node has 0,1, or 2 children) where value of each node is at least the value of its children; left child bigger than right child (is this always true?)

Operations:

* GetMax(): the max value is always at the root of the tree
* Insert(p): may require sifting up (SiftUp = swap with parent) if a value cannot be added to a leaf node: swap with parent node and make comparison with new parent
  + “invariant”: the binary heap property is violated on at most one edge at a time
  + Running time is O(tree height)
* ExtractMax(): replace root with a leaf node and SiftDown ( = swap with child, using the left branch since left child > right child, so heap property will be satisfied at root node now)
* ChangePriority(k, p): requires either SiftUp or SiftDown
  + Running time at most O(heap height)
* Remove(k): change priority to infinity, then call SiftUp, then call ExtractMax()

Most operations are O(tree height) – need to keep trees shallow

* Binary trees are complete if all levels are filled except last which should be filled left to right
* Has height at most O(log n) – minimum possible height – see proof in slides
* Also, can store as an array
* 
  + Parent(i) = floor(i/2)
  + Leftchild(i) = 2i
  + Rightchild(i) = 2i+1
* So can compute parent, leftchild and rightchild on the fly
* BUT: need to keep the tree complete to reap these benefits – how?
  + SiftUp(k) and SiftDown(k) don’t change shape of tree
  + Only Insert(k) and ExtractMax() change the shape of the tree (these change the number of elements)
  + To maintain, e.g., use the rightmost leaf to replace root node when doing ExtractMax(), then all the necessary SiftDown() operations
* Summary:
  + Fast: all operations O(log2 n) or faster
  + Space efficient: relationships between nodes stored implicitly based on array index
  + Easy to implement – not many lines of code

**HeapSort using Priority Queues**

* O(n log n) – asymptotically optimal for comparison based sort
* Requires more space to store priority queue
* Turning array into heap:

Size = n

For i=floor(n/2) to 1 :

SiftDown(i)

* + This works because each call heapifes the given subtree
  + Complexity O(n log n) since SiftDown is called ~ n/2 times
* HeapSort = BuildHeap(A), then swap root with end leaf and reduce heap size by one, do for all elements
  + “in-place heap sort”
  + Worst-case n log n vs average case n log n for quicksort – often try quicksort and terminate if not fast, switch to heapsort
* More about BuildHeap complexity: in fact, most calls to heapify nodes in unsorted array are much less than O(log n) since the subtree depth is small
  + Sum (k=1,inf) { 1/ 2^k} = 1 – can see pictorially by dividing 1 into diminishing fractions
  + Can prove sum {k/2^k} similarly – see slides
  + Thus, BuildHeap is actually linear time
  + BUT, doesn’t help overall algorithm complexity since we still have to ExtractMax() n-1 times
    - But if we only need to sort last k elements we can solve in O(n) time if k is not too large
    - PartialSort: BuildHeap(A) = O(n), then for I = 1:k, ExtractMax()
    - Running time = O(n + k\*log n)
    - If k <= n/log n, then run time is O(n)
* Can make d-ary trees
  + Height is reduced to log\_d(n)
  + But SiftDown runtime complexity increases to O(d \* log\_d(n))
* Big picture notes to remember from summary/quiz:
  + The reason Insert and ExtractMax work the way they do is to maintain the shape of the tree so it stays complete and we can enjoy the advantages of low runtime complexity for operations

**Disjoint Sets**

3 operations:

* MakeSet(x): create singleton set {x}
* Find(x): return ID of set containing x
* Union(x,y): merge two sets containing x and y

Naïve implementations:

Consider sets of integers 1,…n; use smallest element as set ID.

* Union operation is O(n), MakeSet it constant time
* Need efficient data structure for merging – try linked list
  + Singly linked list with tail pointer; use tail as set ID; then just change tail of one to point to head of the other; ID is updated automatically 🡪 Union is now constant time
  + The ID is also well-defined; always return tail of list as ID
  + Downsides:
    - Find(x) is now O(n) since you have to traverse the list to find its tail
    - Union(x,y) only works in constant time if you can get tail of list with x and tail of list with y in constant time – might need to store more info
* Try linked tree – nearly constant amortized time for all operations

**Efficient Implementations of Disjoint Sets**

When creating union, change pointer of tail of one list to the tail of another: a rooted tree, whose root is the ID

Need to know parent: have an array parent[i] which returns parent or i if root

Parent array just contains the parent index of each element

Operations:

* MakeSet – parent[i] = I, constant time;
* Find(i): while i ~= parent[i]:

i = parent[i]

return i

* + Runtime is O(tree height)
* Union(x, y):
  + Choose one tree to hang on other tree’s root; want to keep our trees shallow: **“union by rank”** heuristic
  + Should store a rank array to keep track
  + Rank of the tree only increases when the ranks are equal (otherwise the rank doesn’t increase because the longer tree was set to root anyway, so no rank update needed)
* Invariant: rank = height of tree at node
* Lemma: height of any tree is at most log2(n), which follows from:
  + Any tree of height k in a forest has at least 2^k nodes
    - Proof by induction: initially, tree of height 0 has 1 node, 2^0 = 1. A tree of height k results from merging two trees of height k-1: 2^k-1 + 2^k-1 = 2^k
    - And, n >= 2^height 🡪 height <= log n
* Union by rank 🡪 Union and Find are O(log n)
* **Path Compression:** when finding a root for an element, we also find roots of any elements traversed – should save this information: reattached all these nodes directly to the root
  + Find(i) implemented recursively, doing the reattachment along the way
  + Iterated log n, log\* n, is number of times to apply log to n to get a result <= 1
  + Log \* n bounded by about 5 for all practical values (2^65536)
* Think about log2 operation as height of a complete binary tree of n nodes
* Claim: with Union by rank and path compression: for a sequence of m operations including n calls to MakeSet, total runtime is O(m log\* n)
* Amortized time of a single operation is O(log \*n) and log\* n is at most 5 for all practical values, so we are at nearly constant time for all operations

**Analysis – proof of nearly constant runtime**

Note: maintaining separate rank value, when we use path compression, height <= rank

Also, root node of rank k still has at least 2^k nodes in its subtree since the nodes stay in the subtree

Thus height is distinct from rank.

Properties:

* There are at most n/2^k nodes of rank k: each node of rank k has at least 2^k nodes. So, if there were more than n/2^k nodes of rank k, there would be more than n nodes which is a contradiction
* For any node i, rank[i] < rank[parent[i]]
* Once an internal node, always an internal node (non-root)

Complexity of m operation:

Union boils down two calls to Find plus constant time operations (link and possibly change rank) – so just look at calls to Find operation

* Boils down to edges traversed in a find operation

**Module 4: Hash tables and hash functions**

**Direct addressing and chaining**

Didn’t take notes

**Hash Functions**

**Phone book problem**: store phone numbers and their corresponding names

* can store all phone numbers using direct addressing: array of size 10^L
  + operations run in constant time while memory usage is O(10^L )
* can try chaining: hash function with cardinality m
  + store chains in each cell of array
  + chain Name[h(int(P))] in list corresponding to h(int(P))
  + O(n+m) memory to store list
  + Operations in O(c+1) where c is length of longest list
  + Want small m and small c – try to distribute chains evenly
    - area code is bad example
* Good hash functions:
  + Deterministic
  + Fast to compute
  + Distribute keys well
  + Have few collisions

Quicksort has O(n^2) worst, but adding a random pivot changes this to O(n log n) – looks this up

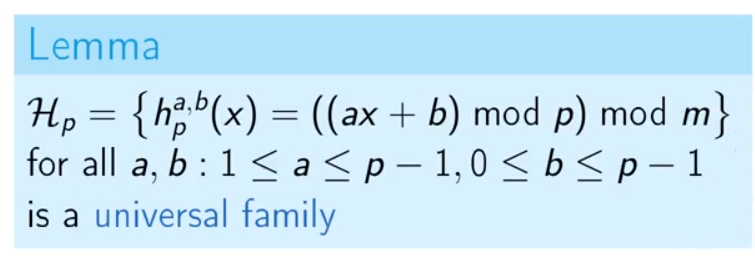
Hashing uses randomization similarly

**Concept of universal family of hash functions:**

* For a given cardinality, if for any two keys, the probability of collision is “small” – less than 1/m where m is cardinality
* Lemma that if h is from a universal family, operations run on average time O(1+alpha): where alpha is load factor (num keys stored / size of table) – which is usually constant time
* Choosing cardinality of hash table m: ideally load factor should be between 0.5 and 1
* What if you don’t know in advance how many keys (n) you’ll be storing?
  + Can borrow ideas from dynamic arrays – resize it by x2 when alpha exceeds some value and do a “rehash”
  + You have to choose a new hash function whenever you do that, of higher cardinality
  + Rehashing is o(n) but amortized running time is still constant, since rehashing becomes rare as size increases

**Hashing integers**

* Where p is a prime number:



* P\*(p-1) cardinality
* IN the phone book example, need to pick p > 10^L, since if it’s less then there will be some key pairs that have the same hash value for ALL hash functions in the family, and our lemma says they should produce collisions only for at most 1/m hash functions for any given pair in order to have a “universal family”
* Pickng a hash function from the universal family means picking two random values independently for a and b in the above
* O(m) memory and constant time on average for wisely selected m

**Proof on upper bound for expected chain length**

* Upper bound of 1/alpha on expected chain length
* Amortized constant run time for all operations
* See proof on paper 🡪 big picture is that you should pick cardinality correctly for your hash function in order to get these benefits

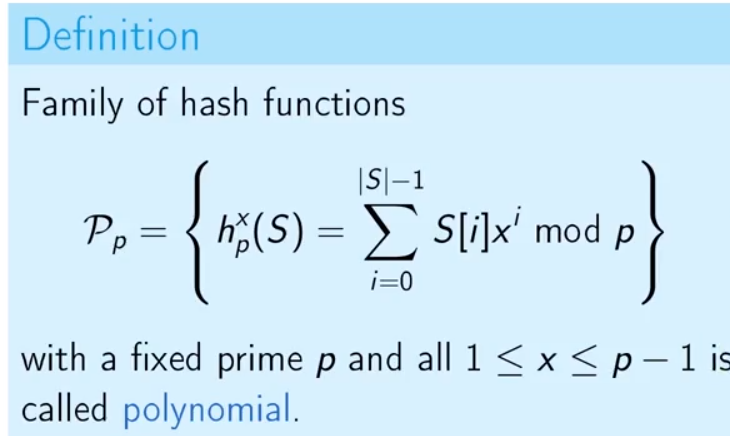
**Proof for universal family for integers**

See video: easy to follow proof that hash collisions have probability <= 1/m for universal families (up to integer p-1), defined in lemma up above

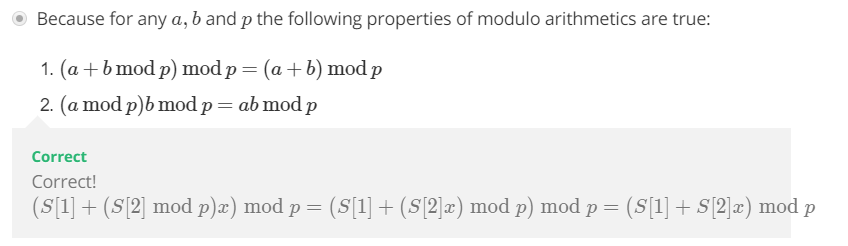
**Hashing Strings**

Convert each character to ASCII or Unicode, and choose large enough prime number p

Polynomial family of hash functions, P(p)



Note (this just shows why you can compute it using a sum that multiples previous term… see video):



Ex: PolyHash in Java

For any two strings of length at most L+1, choosing a hash function from the polynomial family with parameter p, probability of hash collision is at most L/p since there are at most L solutions to the Lth degree polynomial (and we take mod p of this) – can tune p

Running time of hash procedure only actually depends on the length of the string, S

Cardinality fix:

Want to have a hash function of cardinality m (size of our hash table), not p which might be very large

Idea: use polynomial hashing to get a number, then apply integer hashing from family with cardinality m

* As long as the same hash functions are used, this is deterministic
* Lemma: probability of collision is at most 1/m + L/p (not a universal family, but still good!)
* Can prove this is still O(m)
* For big enough p, c = O(1+alpha)
* Computing hash function is O(S): S is string length, but we can determine this if they are bounded by some constant, so still a known constant time operation

**String comparison**

Naïve approach would just go character by character by character and compare – O(T\*P) where T is text length and P is length of string being searched

Rabin-Karp algorithm:

* Idea: hash strings and compare value to decide if worth calling AreEqual(P,S) function
* Use a polynomial hash family with some big p
* Estimate run time:
  + False alarms occur with at most probability len(P)/p –> pick p >> len(T)\*len(P)
  + Hash(P): in O(len(P))
    - Total time is O(len(P)\*len(T)) since always called
  + AreEqual: O(len(P)) … when called
    - Total time is now O(q\*len(P)) – q is num times P is actually found
  + Total run time is O(len(T)len(P)) – same as naïve – but can be improved!

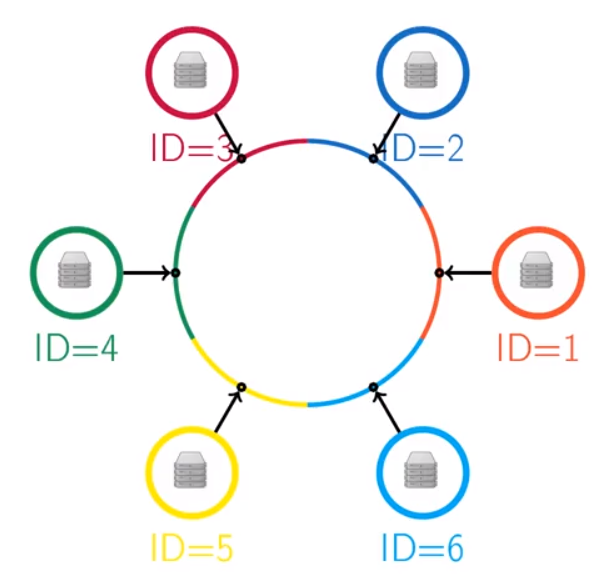
Optimization of Rabin-Karp with precomputation:

* Idea: polynomial hashes of two consecutive substrings T are very similar and the consecutive one can be computed from the prior one in constant time
* Comes from the fact that we’re using a polynomial hash function so each shift requires multiplication by x plus addition/subtraction of one term respectively
* Overall runtime for precomputation is O(len(T) + len(P))
* Overall average runtime for Rabin-Karp with precomputation:
  + O(len(P)\*(q+1) + len(T)) << O(len(P)\*len(T))

**Hashing in distributed systems** **(e.g. Dropbox, Google Drive)**

Case 1: two users upload the same file, but with different names

* Naïve way: upload file, compare with all uploaded files byte by byte
  + No speed gain, O(NS) to compare with others = HUGE
  + As N grows, total run time of uploads grows as O(N^2)
* Compare hashes of files first as in Rabin-Karp, and only compare files with exact same hash
  + Still have to upload to compare directly, and compare with all N files
* Another idea – choose several hash functions from the same family and compute all hashes to compare – if all match, probably equal
  + Here we can compute hashes locally and only send values for comparison through the network
  + Collisions can happen even for several hashes simultaneously – but extremely rare
  + Still, very rare, and this idea is often implemented in practice since it works well
  + Store file addresses in a hash table with the hash values – so comparison also only requires searching in the hash table using the hash values computed – very fast
* Still, number of files and amount of simultaneous uploads are still way too high for a single hash table 🡪 distributed hash tables:
  + Determine which computer’s hash table to use by computing hash(obj) mod N where N is number of computers
  + Should also store on several computers since they break, etc., need to move items from broken computer to others, etc. – how?
    - Consistent hashing:



* + - If computer is added/removed, items are taken from /added to neighbor
  + Overlay network: each node should know a few neighbors
    - For each key, the node will either have the key, or know some neighbors that are “closer” to the key – can get to the right node quickly

**Module 5 – Binary Trees – AVL trees**

Motivation : local search applications – want to do search over a range or for nearest neighbors

Methods

* RangeSearch
* NearestNeighbors
* Insert
* Delete

Hash Table: Wouldn’t be able to do RS/NN in a hash table, e.g.

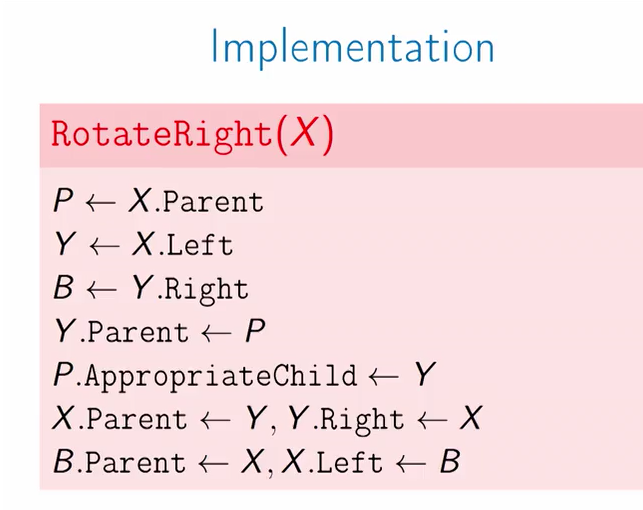
Array: Could do a RS/NN in O(n) but this is slow; Insert and Delete would be O(1)

LinkedList: same as for array

Sorted Array: RangeSearch/NearestNeighbors O(log(n))…(because we can do binary search for the start element)… but updates are O(n)

Need new data structure that can do all of these operations efficiently – sorted arrays came close in that searching is efficient, but updates are not

* Search tree is like a sorted array but easier to insert/delete elements
* Tree Node data type:
  + Key
  + Parent
  + Left child (smaller)
  + Right child (bigger)
* Search tree property: X’s key is larger than any descendant of its left child and smaller than any descendant of its right child
* Methods:
  + Find: find a node with a given key – if a key is not in the tree you still find the location where that key would fit
  + Next: find adjacent elements (i.e. node in tree with next largest key)
    - If you have a right child, go to it then keep going left while you can
    - If no right child, go up a level until you find a key less than N
  + RangeSearch: first find key x, then use Next to add nodes to the list until you get to y
  + Insert: can use Find and add node with key k in the right place
  + Delete is harder: if there is no right child, all nodes are less than N still, so you can just delete it and promote its left child; OTHERWISE, need to find Next(N), which will have no left child (since this would’ve then been Next(N), pop this node up to where X was, and then promote its \*right\* child
* Runtime and notion of balanced trees:
  + Runtime: Find(x) has num operations = O(tree depth); depth can be as bad as n
  + Want to balance trees to keep things O(log n)
  + Insertions and deletions can ruin balance – need to preserve somehow
  + **Rotations:**



* AVL trees: measure of balance so it can be maintained:
  + Height of a node is maximum depth of its subtree
    - Define height recursively: 1 if leaf, 1+ max(n.Left.Height, n.Right.Height)
    - New height attribute on nodes – need to store it and keep track of it
  + AVL property:
    - For all nodes N, abs (N.Left.Height - N.Right.Height) <= 1
    - If this property holds, total tree height is O(log n)
    - And large height implies many nodes (see proof in video)
    - Node of height h has subtree of at least 2^(h/2) which can’t be more than n… so height must be less than 2 log n, O(log n)
  + Insertions with rebalancing: only need to worry about things along the insertion path which is at most O(log n)
    - Insert(k,R) + N
    - Find N
    - Rebalance(N)
  + Rebalance (at least things are never unbalanced by more than 2!):
    - If heights differ by more than 1, if left > right +1, rebalance right, o.w. rebalalnce left
    - Then adjust height N (= 1 + max(left, right)
    - If parent is not null (i.e. not at the root), the rebalance(parent)
  + Fails if left side is too heavy on the right child (“problematic grandchild”)
    - Rotate the grandchild left first, then rotation root node to the right
  + Deletions can also change balance – will decrease height in subtree with successor, whose parent node might now be imbalanced – so, need to rebalance parent of the node that replaced the removed node
  + All of these are O(log n)
* More operations: split and merge
  + Merge: Can easily merge trees where all elements of one are > than all elements of the other – especially easy if you’re also adding an extra node as root between them
  + If no new root, just use largest element of left subtree – but then balance is not preserved
  + Idea: go down side of bigger tree until we find subtree we can merge with the smaller tree
  + Run
  + Split: pick an element and create a tree to either side of it
  + Idea: search for x, merge all left / right subtrees along the path
  + Using the combination of prior AVL type routines, the total runtime for both is O(log n)

**Module 6 – Binary search tree applications**

Applications:

1. Returning order statistics (x largest, median, percentiles)
   * Need a tree where we can easily know how many elements are in each size – store new field size on each node – requires recomputing after rotations, similar to height
   * Can easily design recursive methods that will find various order statistics like the xth smallest element, using the new size field
2. Idea: “Color flips” after some index in an array – would be linear time if you just used an array
   * New use for trees: elements in sorted order, going from bottom left to bottom right
   * Idea: flip elements using merge and split
   * The moral is that trees can be used to store lists, which can be recombined using merge/split operations

Splay trees: another kind of binary search trees

* Motivation: if some items are searched more frequently, can do better than O(log n) search if we put them closer to the root
* Probably don’t know this a priori, so idea is to bring nodes to root as they are queried, i.e. adaptively rearrange tree
* Could just rotate the node up, but this is slow and you might get stuck in a loop with a bad sequence of queries
* Operation called Splay fixes this: zig-zig, zig-zag and zig operations: not totally sure why this avoids loops, but okay
  + Recall overall objective of splay is to bring a node to root in an efficient way
* Generally, calling splay instead of rotate to top tends to keep the tree more balanced, since each call has a balancing effect
  + Amortized cost of doing O(D) work and splaying a node of depth D is O(log n) where n is total number of nodes
  + For this to work, even if you fail to find a node with key k, you have to splay the node closest to k
  + TO delete, splay node and its successor, then remove the node
* To split a tree, splay the node then cut at the root
  + Cutting just involves resetting pointers
* All splay operations are O(log n) amortized time – complexity in final video
* Other interesting properties of splay trees:
  + Can assign weights to nodes
  + Dynamic finger – amortized cost is O(log(D+1)) where D is distance between last access and current access
  + Working set bound: O(log(t+1)) where t is time between accesses of same node
  + Dynamic optimality conjecture: a splay tree is at most a constant factor worse than the best dynamic search tree for a particular sequence
  + So dynamics/structure of access pattern matters