Data Structures

Linked Lists

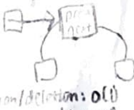
SList
- size 
- next

Runtime: insertion/deletion: O(1); indexing/searching: O(n)

DLists
- size
- next
- previous

Runtime: insertion/deletion: O(1); indexing/searching: O(n)

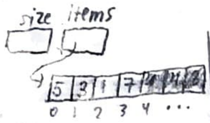
DLists w/ Sentinal / sentinental
- sentinal
- size
- next
- previous



Runtime: insertion/deletion: O(1); indexing/searching: O(n)

• LinkedLists have O(n) get. It must travel through from the beginning

ArrayLists
- addLast
- getLast
- removeLast
- get(int i)



- resizing arrays
• Literally an array with resizing.
• Fast get methods.

	ArrayList	LinkedList
get	O(1)	O(n)
add	O(1)*	O(1)
remove	O(n)	O(n)
contains	O(n)	O(n)
insert	O(n)	O(n)

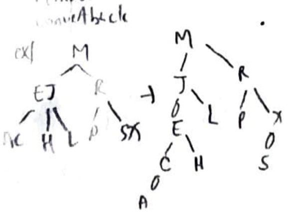
*amortized

LLRB Tree (basically 2-3 tree but converted)

When you want to add to LLRB,
1) first transform to 2-3 tree
2) add to 2-3 tree : smaller goes down left. make that red.
3) transform back to LLRB

remove
- convert to 2-3 BST
- remove
- convert back

ex/



For HashMap's, calculate hashcode for key only. K.hashcode();

Runtime

add	O(LogN)
remove	O(LogN)
get	O(LogN)
contains	O(LogN)

Heap
- Top down sorted binary tree structure used for queues (priority).
- Guaranteed to be a complete tree
- binary min heap: Binary Tree that is complete and pop
  - min heap = Every node is less than are = children
  - complete = Missing items only in bottom row, for left sided

add:
1) add to the end (bottom leftmost of heap)
2) compare item to parent, swap if need be. Repeat. Approach 1

• implements PQ's Approach 2: Array to store

deleteMin:
1) swap the last item with root
2) sink way down by swapping with smaller values.

Runtime — amortized, resizes

add	O(LogN)
remove Top	O(logN)
get Top	O(1)

Externally Chained HashMap/HashSet
• auto-resizing array of LinkedLists.

Add
1) get hashcode of item
2) Mod hashcode by array size
3) Iterate through LinkedList to check if there
4) If present replace
5) if not present, add to end of LinkedList
6) resize if Load Factor is exceeded

Remove
1. get hashcode of item
2. Mod hashcode by arraysize
3. Iterate through LinkedList to find
4. Remove that node from LinkedList

Get: compute hashcode and then looks into corresponding bucket — then looks for that item.

Runtime* — O(1) amortized

add	O(1)
remove	O(1)
get	O(1)
contains	O(1)

* runtimes are constant if hashcode distributes well.

Resizing = Θ(N)

Tree Traversals

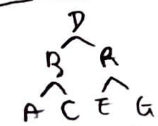
Depth First Traversal

PreOrder: print node, preOrder(left), preOrder(right) — DBACREG
- good for printing directory lists

InOrder: inorder(left), print(key), inorder(right) — ABCDERG

PostOrder: postOrder(left), postOrder(right), print(key) — ACBEGRD

example tree



or add to fringe (Queue), while (fringe>0) dequeue, print, add all children

Level Order Traversal
- left to right, top to bottom
- implement through iterative deepening

```
for(int i=0; i<height; i++){
    visitLevel(T, i, action)}
VisitLevel(Tree T, int level, action toDo){
    if T=null
        do Nothing
    if level==0
        print T.key
    else visitLevel(T.left, level-1)
         visitLevel(T.right, level-1)
```

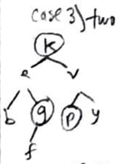
InRange Searching
searching for items in given range
Θ(LogN+R) where R is # matches

Vanilla BST (no guarantee of bushiness)

No duplicates in BST

add:
1) start at root do comparison
2) if smaller go left
3) if bigger go right
4) when null leaf encountered, add there

deletion: 3 cases:
- case 1) no children: just sever the link (cut)
- case 2) one child: just move pointer to its child
- case 3) two children: (hibbard deletion):
  1) find new root node
  2) must be maintaining bst property
  3) pick node that is rightmost of left subtree or leftmost of right subtree, connect its children



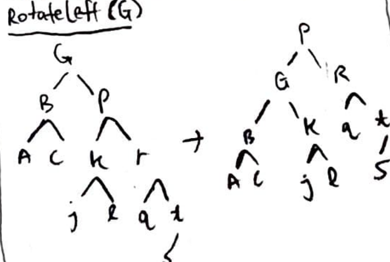
properties: left keys are less than, right keys are greater than.

Runtime:

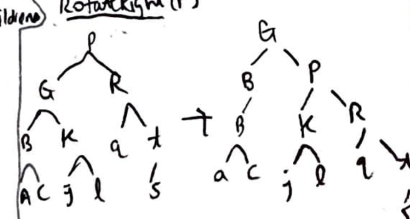
add	O(N)
remove	O(N)
get	O(N)
contains	O(N)

Tree Rotations (to make more bushy)

RotateLeft (G)



RotateRight (P)



QuadTree

(use two compares)

WeightedQuickUnion
• Tree structure (represented by arrays of numbers that store info about which numbers are part of the same set. Sizes of each set tracked.

Constructor
1) instantiate array with N numbers
2) set each number's parent to itself

Connect
1) get first number's parent. if this parent is not its own parent repeat.
2) get second number's parent. If the parent is not its own, repeat
3) set the root of the smaller set to the root of the bigger set.

isConnected
1) get both nodes' root.
2) check if the parents are same

FindRoot
1) get number's parent
2) if the number's parent is itself, done
3) else repeat.

Runtime

Constructor	O(N)
connect	O(LogN)
isConnected	O(LogN)
findRoot	O(LogN)

(Union) = connect

2-3-(4) BST (Guaranteed self balancing B-Tree)

2-3-4 trees: may have up to 3 items in each node, may have 2, 3, 4 children for each node

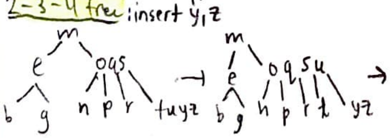
2-3 tree: may have up to 2 items in each node, may have 2 or 3 children for each node

add:
1) iterate through tree by doing comparisons as usual until reaching leaf node.
2) insert new item into leaf node, where it should be
3) check if node is "too juicy"
4) if too juicy, take out left center, put it into parent node.

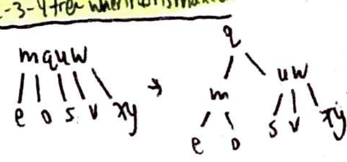
Runtime

add	O(LogN)
remove	O(LogN)
get	O(LogN)
contains	O(LogN)

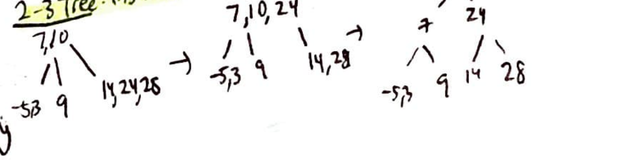
2-3-4 tree: insert y, z



2-3-4 tree when root is maxed



2-3 Tree: insert 28



Stacks

- crippled list: can only operate on item at the top
- pop removes -peek
- push adds

add
adds to the top
remove
removes to the top.

all runtime (O(1))

- Last in First Out or First in Last Out: (FILO)

Queue

- can ... ly read or remove item at the front of queue
- c... ly add item to the back of queue

All methods O(1)

FIFO.

usea... BreadthFirst searches

Dequeue

... know when
... insertion at the front & back

Sorting

SelectionSort

1) Find ... smallest item
2) S... this item to the front and fix it (literally swap)
3) R... the unfixed items

InsertionSort (In-place)

* First item is fixed, start i on second and switch until happy.

1) designate item i as the traveling item
2) swap it backwards until traveler is in the right place among all previously examined items.
3) use ... pointer as the ... to current spot of traveling item.

Inversions

if a[i] > a[j] and i < j

ex) [1, 2, 3, 4]:

Heapsort (In-Place)

1) Max-heapify the array (inplace)
2) while the heap is not empty, throw max element to the end of the array

detailed version
1) put the array in the heap but no sink. Just as you see it.
2) then, start from bottom-right, and sink each value.

Heapification

- add all the array values as it is to the heap. Then sink from bottom-right, Right to Left.
- add the max value to the end of the array, delete it, swap with bottom right (last) and sink down. Swap with larger of the two children.
- Repeat but consider only the array from the non-finished ones at the end.

MergeSort

1) split items into roughly 2 even pieces
2) merge sort each half, recursively.
3) compare sorted halves by comparing values from each sort, and putting in smaller of the two values to a third array.

QuickSort - 3scan

1) Partition on leftmost item
2) Quicksort left half
3) Quicksort right half

Partitioning
- ... all items greater than the pivot to the right,
- ... all items less than the pivot to the left.
- ... fix the pivot where its supposed to be.

QuickSort - Tony Hoare In Place

1) create L and G pointers at left and right ends
2) L is friend to small items and hates large or equal items
3) G is a friend to large items and hates small or equal items
4) walk pointer towards each other, stopping on a hated item
   - when both pointers stop, swap and move pointers by one.
5) when pointers cross, you're done walking.
6) swap pivot with G.

Pivot is leftmost. * L starts at second, G at last

CountingSort

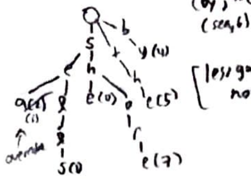
1) count # of times each key appears
2) use the counts to create an array of starting indices
3) iterate through the list and place the keys based off of starting indices.
4) iterate through original array and put the values in the final array and increment the starting index position of the counts array of starting position.

- good if $n \geq R$ where R is the # of keys in the alphabet/whatever code.
  good for big collections of items from some alphabet.

* Dont use if # digits >> # elements in array

Regular Trie

- insert (she (0)), (sells, 1) (sea, 2) (by, 4) (the 5) (sea, 6) (shore, 7)



[less goes to left not shown in diagram]

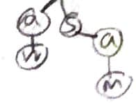
RadixSort LSD - counting sort

1) Sort each digit independently rightmost digit to left
2) Treat empty spaces as less than other characters.
   - count sort each column and keep doing it
   - Good for highly similar that are long strings
   - Bad for highly dissimilar that are long strings. Because mergesort knows the strings are different just by checking the leftmost digit

RadixSort MSD - Counting Sort

1) sort each digit from leftmost
2) run MSD on each subgroup.
- counting sort from left to right
- Good for highly dissimilar because then subgroups are smaller

Trie: a digit-by-digit set representation



- good for prefix matching, approximate matching

algorithm for longestPrefixOf
1) check each digit in turn, walking down tree
2) Keeping track of the most recent blue thing
3) At some point you find the next isnt there then
4) return the most recent blue thing.

1) Array based Trie.
2) BST based Trie
3) Ternary Search Trie.
   - assign a character to each node
   - give each node 3 links
     1) Left link if keys next character < node's character
     2) middle if keys ... == node's character
     3) right ... > node's character →

TST
3) insert sam, sad, sap, same, a, awls

Runtime for contains

	Amortized Average	Worst	Best (miss)	Memory
hash Table	O(L) X			NL
BST		LogN	1	NL
Trie (arrayMap)		L	1	NLR
Trie (hashMap)		L	1	NL
Trie (TreeMap)		LLogR	1	NL
TST		NL	1	NL

N keys, L digits per key, R alphabet size, A miss means the key isnt present

Hoffman Coding for Compression

1) assign each character to a node with relative frequency as weight
2) Take two smallest nodes and merge them into a supernode with weight as sum
3) Repeat until tree

Hoffman Compression ...

1) count relative freq.
2) build encoding array and decoding trie
3) write decoding trie to output.txt
4) write codeword for each symbol to output.txt

Radix over Compare: alphabet is well defined.

InsertionSort: 1) when input is small 2) almost sorted

Quicksort over Mergesort if
1) want faster overall runtime
2) better memory usage
3) dont care about stability
4) many duplicates, 3 way parti...

Trie over Hash:
1) when we want to look for near misses (spell che...)
2) autocomplete or other string operations
3) when we expect to have mismatches early on in a string
4) to save space for certain pathological inputs with large degree of overlap at the front of strings
5) when we want to support ordered operations
6) implement alphabet sorts.

Heaps over LLRB:
1) uses less memory
2) easier to implement/understand
3) handles duplicates well
4) stepping stone for heapsort

Stable		Runtime	Best	Worst	Notes
N	Heapsort	Θ(NlogN)*	Θ(N)	Θ(NlogN)	Bad caching
N	Selection Sort	Θ(N²)	Θ(N²)	Θ(N²)	terrible
Y	Insertion Sort	Θ(N²)* Θ(N+K)	Θ(N)	Θ(N²)	good if inversions is small, good if small list
Y	Mergesort	Θ(NLogN)*	Θ(NLogN)	Θ(NLogN)	Best stable sort
No depends	QuickSort	Θ(NlogN)*	Θ(NLogN)	Θ(N²)	Best comparison based sort in practice
Y	Counting Sort	Θ(N+R)		Θ(N+R)	Avoid compares. Good if n≥R
Y	RadixSort LSD	Θ(W(N+R))		Θ(W(N+R))	
Y	RadixSort MSD	Θ(W(N+R))		Θ(W(N+R))	
	Shells Sort	Θ(N Log N)	Θ(N)		

Data Structures II

Graphs

- vertices, nodes, adjacents, paths, cycle: path whose first/last node is the same
- nnected: all vertices are connected

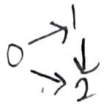
Implementing Graphs

Rep 1: Adjacency Matrix:

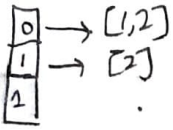
	0	1	2	3
0		1	0	
1	1	0	0	
2	0	0	0	
3				

Rep 2: Edgesets: hashsets

{(0,1), (0,2), (1,2)}



Rep3: Adjacency List



An arraylist of arraylists;

Tree is a graph with no cycles.

Run Times

	addEdge	for (w: adj(v))	print graph	hasEdge	space usage
adjacency matrix	$\Theta(1)$	$\Theta(V)$	$\Theta(V^2)$	$\Theta(1)$	$\Theta(V^2)$
list of Edges	$\Theta(1)$	$\Theta(E)$	$\Theta(E)$	$\Theta(E)$	$\Theta(E)$
adjacency list	$\Theta(1)$	$\Theta(1)$ to $\Theta(V)$	$\Theta(V+E)$	Θ(degree(v))	$\Theta(E+V)$

When would you use matrix over list?

- when the graph is very dense or where the number of edges $|E|$ approaches $|V^2|$

Graph Traversals

Depth First Traversal

- guranteed to reach every node that is reachable
- runs in $O(V+E)$ [finds a path from s to every reachable vertex]

1) start somewhere
2) mark it
3) for each unmarked adjacent, set edge to.

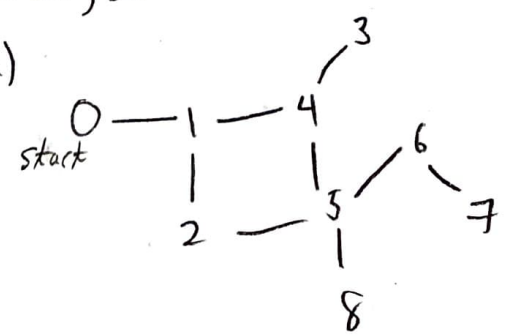
uses a stack. literally go deep before wide.

- DFS PreOrder: return value as you traverse: 0 1 2 5 4 3 6 7 8
- DFS PostOrder: return once at the end, then backtrack: 3 4 7 6 8 5 2 1 0
- Level Order: order of increasing distance from s: 0 1 4 2 5 3 6 8 7

↑ anything on the same level can be permuted.

uses a stack

ex)



Worst case runtimes

BFS	DFS	Dijkstra's	A*	Topological Sort
$\Theta(V+E)$	$\Theta(V+E)$	$\Theta((V+E)\log V)$	$\Theta((V+E)\log V)$	$\Theta(V+E)$

Topological Sorting

- Reverse Postorder.
  - start at node with no incoming direction
  - works only for directed graphs.

Breadth First Search (wide before deep)

- guaranteed linear time
1) Initialize a queue (opposite of stack), starting with vertices, and mark it.
2) Fringe (until empty):
   - remove vertex V from queue (oldest)
   - for each unmarked neighbor of V, mark, add to queue, set EdgeTo V.
- Good for finding shortest paths
  - runs in V+E time and V space.

Runtime: $\Theta(V+E)$ runtime.

- gives 2 for 1 deal; reachability / shortest paths. $O(V)$ space

DFS vs BFS

Both traverse entire graphs in different order

DFS good for topological sort

BFS for shortest paths

Both for existance of a path

Dijkstra's Algorithm (doesn't work w/ negative)

goal: to find shortest path from source to: 1) target 2) every vertex

doesn't work w/ negatives because it assumes paths get heavier. And it "unrelaxes" things that are the shortest so it doesn't compare the alternative path. It takes out the visited in the fringe so we can't update it again.

Fact: solution to every direction is a tree because every node has one parent, choose parent with shortest distance.

Shortest Paths Tree has V-1 edges

Relaxation: replacing things with better stuff

1) add start node to PQ
2) remove/pop start from PQ, add its neighbors
3) remove the node that's closest from source
4) add all its neighbors / repeat
5) relax all edges / update the values

Fringe: Priority Queue

Runtime	# operations	cost per operation	total cost
PQ insertion	V	$O(\log V)$	$O(V \log V)$
PQ delete-min	V	$O(\log V)$	$O(V \log V)$
PQ decrease priority	E	$O(\log V)$	$O(E \log V)$

overall: $O(V^* \log V + V^* \log V + E^* \log V) = O(E \log V)$ assuming $E > V$

A*

to a single target.

1) start node add it to PQ, with the PQ as heuristic + distance from source
2) pop start, add neighbors with its heuristic.
3) Basically Dijkstra's but with different PQ compare.

Minimum Spanning Tree

- cheapest way of connecting all vertices
  1) Prim's Algorithm
  2) Kruskal's Algorithm

Prim's Algorithm (works with negative)

- use priority queue

1) start from some Node, mark node
2) repeatedly add shortest edge from the marked nodes
3) repeat until V-1 edges

Prim's and Dijkstra algorithm are the same, except:

- Dijkstra's considers distance from source
- Prim's considers distance from tree" - add to PQ

Runtime

assume binary heap based PQ

	# ops	cost per	total
PQ add	V	$O(\log V)$	$O(V \log V)$
PQ delMin	V	$O(\log V)$	$O(V \log V)$
PQ decreasePriority	$O(E)$	$O(\log V)$	$O(E \log V)$

overall: $O(E \log V)$ assuming $E > V$

Kruskals Algorithm (works with negative)

1) mark all gray (edges)
2) consider edges in increasing weight
3) Add to MST unless it makes cycle
4) Repeat until V-1 edges

algorithm

1) use PQ, add to PQ all edges
2) use WQU to check for cycles.

Runtime

Operation	# times	Time per Op	Total time
insert	E	$O(\log E)$	$O(E \log E)$
deleteMin	$O(E)$	$O(\log E)$	$O(E \log E)$
union	$O(V)$	$O(\log^* V)$	$O(V \log^* V)$
isConnected	$O(E)$	$O(\log^* V)$	$O(E \log^* V)$

Overall: add all total: if preso... then $O(...)$ $O(E^*$

Shortest Paths and MST algorithm summary

Problem	Algorithm	Runtime if $E > V$	Notes
Shortest Paths	Dijkstra's	$O(E \log V)$	Fails for (-) w
MST	Prim's	$O(E \log V)$	analogous to Dijkstra's
MST	Kruskal's	$O(E \log E)$	uses WQ
MST	Kruskal's with pre-sorted edges	$O(E \log^* V)$	uses W