Trevor Swan / CSDS233 / Midterm 10/15/24

Encapsulation → Access Modifiers

Arrays

- Constant Access to indexes O(1)
- | 0x800 | 0x804 | 0x808 | 0x812 | Adjacent Memory
- Fixed Size
- Sorted (operations / complexity)
  - (i) O(n) Addition / Removal
    O(log n + n) = O(n)
    Find Index, Shift
  - (ii) O(log n) Search w/ Binary Search
    Cut Array in half every iteration

Linked List

- Dynamic → Easy Addition/Removal
- Links Nodes together
- O(n) Access to get elements
- Addition Removal Requires only updating pointers
- Memory is not stored adjacently

Nodes

Node(Element, Node next)
Element
Node next

List

List(Node head)
Node head
size()
add(), remove()

Doubly LL have ports to next & prev

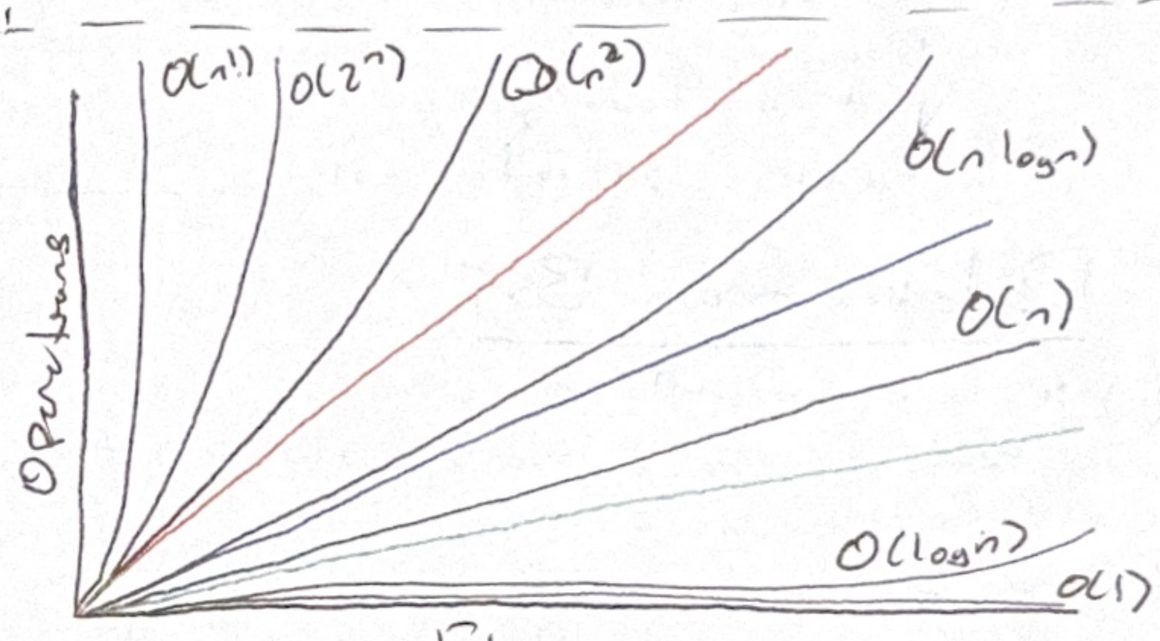
Big-O: $T(N) = O(f(N))$: $\exists\, c, n_0 > 0$

s.t. $T(N) \leq c \cdot f(N)\ \forall N \geq n_0$

(i) $T(N) = O(f(N) + c) \Rightarrow T(N) = O(f(N))$
(ii) $T(N) = O(c * f(N)) \Rightarrow$ ″
(iii) $T(N) + c = O(f(N)) \Rightarrow$ ″
(iv) $T(N) * c = O(f(N)) \Rightarrow$ ″

Factor out / Ignore any scalar quantities!

Always cancel out the non-dominant term when you can!!



Helpful Equations

$n = a^r$

n: Total # of function calls
a: # of equal-sized pieces broken into per call
r: max number of recursive calls

$\log(ab) = \log a + \log b$, $\log\left(\frac{a}{b}\right) = \log a - \log b$

$\log_b(x) = \frac{\log_k(x)}{\log_k(b)}$, $\log N < N^c$ $c > 0$ always!

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$, $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$, $\sum_{i=0}^{n} 2^i = 2^{n+1}$

Recursion

- Method which invokes itself
- Must break input into smaller parts
- Must specify a base case
- Function calls go onto call stack

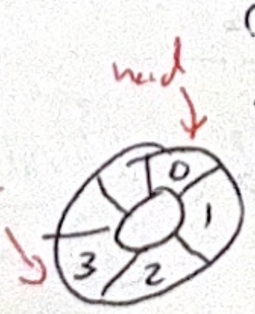
Example: sum(n) = sum(n-1) + n

Stacks (stack of plates or books)

- First-In-Last-Out (FIFO)
- push(), pop(), peek(), isEmpty/isFull

Queue Line for banking

- First-In-First-Out (FIFO)
- DLL is most efficient
- LL is fine
- Circular Array is Great too




```
public boolean add(val) {
  if(isFull()) return false;
  rear = (rear+1) % arr.length
  arr[rear] = val; items++; }
```

```
public int remove()
  if(isEmpty()) throw err;
  int value = array[head]
  head = (head+1) % arr.length
  items--; return val; }
```

Complete Graph: Length of path between any 2 vertices is exactly 1.

Tree Terminology

Leaf: Node w/o children
Depth: Num edges from node → root
Height: Max num Depth
Binary Tree: Each Node has 0-2 children
BST: All Node's in left subtree are less than its key, right subtree has greater than or equal to key
↳ In order → Ascending

Binary Tree Operations

height ↓
All take time to search O(h)

Common search for Node & its parent

```
... while (trav != null)
ref:  parent = trav
      if (key < trav.key)
          trav = trav.left
      else
          trav = trav.right
} if (trav == null) return null; ...
```

parent=null
trav=root

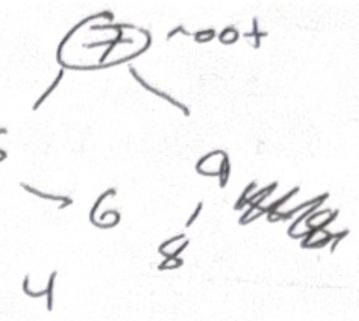
Deletion from BST (Cases)

(i) x has no children
Remove x from Tree by setting parent reference to null

(ii) x has one child
Set x's parents reference to x's child

(iii) x has two children
1) Find Y := leftmost node in X's right subtree (smallest node)
2) Copy Y's key to X, not messing w/ tree child
3) Delete Y using case 1 or case 2

Binary Tree Traversals



Pre-order Traversal

Recursive

(i) Root → (ii) Left → (iii) Right

7 5 2 4 6 9 8

Post-order traversal

Recursive

(i) Left → (ii) Right → (iii) Root

4 2 6 5 8 9 7

In-order Traversal

(i) Left → (ii) Root → (iii) Right
Recursive

2 4 5 6 7 8 9

Level-order Traversal

Breadth-First
Queue

top → bot
left → right

7 5 9 2 6 8 4

Deletion Implementation

```
public T delete(key) {
    while (trav != null && trav.key != key)
        trav = key < trav.key ? trav.left : trav.right;
    ...
} if (trav == null) return null;
  else T removed data = trav.data;
  deleteNode(trav, parent); return removed Data;

void dNode(toDelete, parent) {
  if (1 or 0 children)
      toDelete child = null
      = left if non null
      = right if non null
  if (toDelete == root)
      root = toDelete child;
  else if (toD.key < parent.key)
      parent.left = toDelete child;
  else
      parent.right = toDelete child;
  else
      Node replParent = toDelete;
      Node repl = toDelete.right
      while (repl.left != null) {
          replParent = repl;
          repl = repl.left;
      }
      toDelete.key = repl.key;
      toDelete.data = repl.data;
      deleteNode(repl, replParent)
```

Find Node and delete then return deleted node

Swap Data

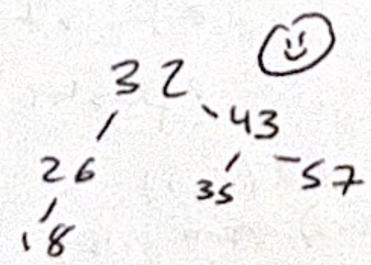
In-order Successor

recursively delete

AVL Trees

→ Balance: Bal(N) = height(N's right subtree) − height(N's left subtree)
(N is root)

- Leaf Node Balance is always 0
- One Leaf: Height = 0
- Empty Tree: Height = −1



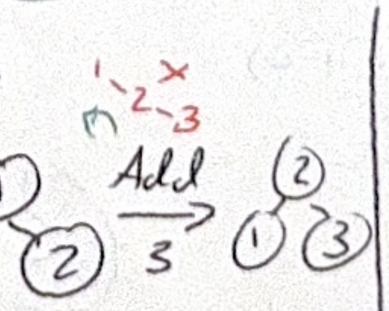
2 new fields

```
private Node parent;
private int Balance;
```

- Balance of all Nodes are all 0, or ±1
  ↳ Balance ≤ |1|

Insertions

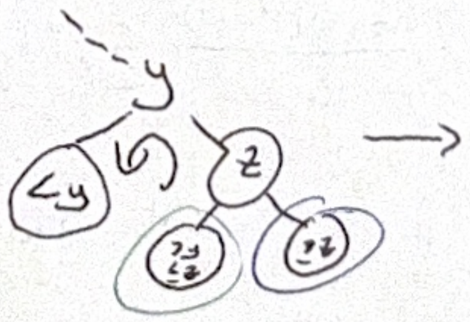
Empty → Add 1, 2 → Add 3

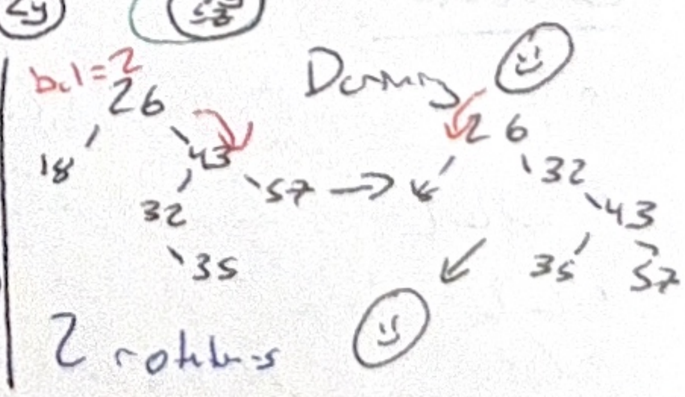


Rotations

6 ptr updates → Constant Time

1 (i) Right child of parent of y
2 (ii) Right child & parent of y
2 (iii) Left child & parent of z
1 (iv) Parent of left child of z





Selection Sort: VecArray, find smallest and move it to position n

1. let variable i=0
2. Loop through arr using i (i through the)
   (a) For i to end of array, loop through to find the smallest element, call its position k
   (b) Swap elements of i and k
   (c) Now elements 0 to i are sorted
3. Once i reaches end of loop, fully sorted

TCA: $O(n^2)$, each loop is a nested loop

Insertion Sort: n is case length of sorted subarray

1. Let i=1, Assumption is subarray from [0, i) is sorted
2. Loop through arr using i
   (a) Examine element at i, call it k
   push k back until the element before is ≤ k and after is ≥ k
   (b) Part i swaps k with previous element until found position, so its loop results in a sorted subarray of [0, i+1)
3. Once i reaches end of loop, done!

TCA: $O(n^2)$; fewer, less sorted as less pushing back is done

Bubble Sort

1. let i = length of arr - 1
2. Iterate to 0 by decrementing by 1 for each iteration
   (a) Let variable j=0
   (b) Loop j to i, pushing up the largest value all the way to i
3. Once i=0, entire array should be sorted

TCA: $O(n^2)$ due to nested for loop structure w/o $\frac{1}{2}$ checks in insertion

Merge Sort: Divide & Conquer

1. Recursively split array into halves recursively until subarrays are size 1
2. Merge subarrays into sorted subarrays, creating larger subarrays throughout the process

TCA: $O(n \log n)$, takes up more space than quicksort

Merging: Start at beginning of each subarray and append smallest to temp, incrementing temp's index ptr and the subarray's ptr

Bucket Sort (evenly distributed buckets = better)

1. Create an index for the current value, place into a linked list at the index (hash table w/ chains)
2. Sort each bucket w/ insertion sort, recombine to get a sorted array

TCA: $O(M+N)$, where M is number of buckets. Uses much more space, $O(n)$ if data is evenly distributed, $O(n^2)$ if all values end up in the same bucket

Quick Sort: Recursive, set pivot, move all elements less to its left, greater to right

1. Pick Pivot in the given array
2. Call 'Partition', which arranges all values < pivot to its left and all > to its right
3. Restart process for every subarray

Choosing a Pivot: most efficient w/ evenly divided subarrays

(i) Middle Element: Generally good esp for nearly sorted or sorted data
(ii) Pick first or last Element: Not great. Especially bad if data is nearly sorted, will go backwards
(iii) Random: Good for random data
(iv) Median: Choose out of 3 elements, maximizes probability of choosing a good pivot

Partitioning: Organizes values w/ respect to pivot

1. Find elements to swap
2. Swap elements if left subarray ptr is less than right subarray (i.e.) otherwise i has traversed subarray & is already been partitioned

TCA: $O(n \log n)$, but $O(n^2)$ if pivots are poor

Shell Sort: Optimized Insertion Sort

TCA: $O(n^{1.5})$ but can be $O(n^{1.25})$

Heap Sort: Use a heap, remove max

TCA: $O(n \log n)$

Trees vs. Graphs

In theory directed, connected set of nodes, but they are acyclic, meaning no cycles in their paths

Spanning Tree

Subset of connected graph:
(i) Contains all vertices
(ii) Contains a subset of edges that form a tree

Adjacency Matrix

- In each row, cost needed to traverse a neighbor is recorded in its column
- ? = not connected to
- x = row vertex ≡ column vertex

BFS (Queue)

1. Begin @ start vertex
2. Visit all its adjacent vertices
3. Visit all those neighbors neighbors (unvisited only)
4. Visit those neighbors neighbors (unvisited)
5. etc...

DFS (Stack)

1. Begin @ start vertex
2. Proceed as far down as possible, then backtrack to go down new path

Dijkstra's Algorithm (shortest path) $O(E \log_2 V)$

1. Begin @ start node, shortest path is 0, note this, though not traversed
2. Record current node in list of already visited nodes
3. Identify node's neighbors. Note their costs and the total cost to get there
4. For each calculated cost, if it is shorter than the current shortest, save it
5. Select the node with the lowest calculated cost that hasn't been visited already. This is now the current node, restart from step 2
6. All visited => list shows shortest to all nodes

Assumptions: All weights are not negative

Prim's Algorithm (minimum spanning tree)

1. Begin @ an arbitrary start node, add it to list of visited nodes
2. For all nodes in the set of unvisited nodes, find which neighbor takes the least amount to get to, out of the neighbors you haven't been to before
3. Add found neighbor to set of visited and mark down tree edge
4. Go to step 2
5. Repeat until all vertices have been added

Assumptions
- Graph is connected
- Graph is undirected

$O(E \log_2 V)$

Priority Queue

- FIFO w/ insert, remove, peek
- Elements with high priority are removed first
- Uses a heap to implement

Heaps ← Most ops: O(N) → Balanced

- Complete binary tree, with levels filled left to right
  - ↳ construct/deconstruct arrays using a level-order traversal
- → Left Child of a[i] node is a[2i+1]
- → Right Child of a[i] node is a[2i+2]
- → Parent of node a[i] is a[(i-1)/2] if i≠0
- Max-at-Top, parent node always greater than both of its children
- Min-at-Top, parent < all children

Building a Heap ∈ O(N)

1. Start pos = (numItems-2)/2 ← last parent
2. Sift Down from this position and decrement down position by 1
3. Repeat step 2 until complete ∈ (i ≥ 0)

Naive approach is calling insert on every item ⇒ useless siftDowns

Ex. Max-at-Top Heap ← findMax = O(1)

Removing Maximum ← O(logN)

1. Replace root of heap with last value of heap
2. Call Sift Down on the new out of place node
   (a) check to see if current value is ≤ one of its children
   (b) If so, choose the larger of the children and swap it with the current value ← sifted down 1 level
   (c) Continue process until current value is greater than its children, or reaches bottom of heap

Inserting ← O(logN)

1. Insert new element at last spot in heap
2. Call Sift Up on the new out of place node
   (a) Check to see if current value > its parent
   (b) If so, swap parent with current value ← sifted up 1 level
   (c) Continue until current value is less than parent or is root

Removing Arbitrary Element ∈ O(logN)

last item in heap

1. Compare to be-replaced element with replacing item
   (a) if new item > old item, replace and sift up
   (b) otherwise, replace and sift Down ← on new item

Updating Arbitrary Element ∈ O(logN)

1. Compare previous value to current (updated value)
   (a) if new value > old value, sift up
   (b) if new value < old value, sift Down
   ↳ Follow general rules of removing Arbitrary Element

Hashing

LoadFactor (>) ↑ means harder to probe

- Hash Tables have constant time ops
- Turn data into index via a hash
- Identical hashes => collisions
- Load Factor: (open positions)/(total positions)

Chaining ← Memory overhead

- Store LinkedLists in each slot, add elements to their index's list (or array)

Infinite Probing

- When number of insertions reaches the size of the table, simply indicate that the operation (insertion/deletion) cannot be performed

Rehashing: Two reasons w/ solutions

1. Too many removed flags => Increased time
   Solution: Load factor is assumed to be ok, just create new hash table of same size and copy all occupied slots
2. Load Factor too high => Time to find slots ↑
   Solution: Roughly double size of hash table w/ a new table and copy all elements. New table size should remain prime for ← Probing

Open Addressing ← search for an open spot

(i) Linear Probing: if we map data to an already full index, increment by 1 until an open spot is found

(ii) Quadratic Probing: if mapped to full index, increment by (1, 4, 9, ...) quadratic sequence
↳ works if load factor < 0.50 and table size is a prime number

(iii) Double Hashing: Use two hash functions h1, h2
1. Apply h1 to data for initial index
   (a) If index is empty, place data -> done!
   (b) If index is not, proceed
2. Let h1 be current index
3. Add index obtained from h2(data) to current index, let this be the current index
   (a) If current index is empty, place data -> done!
   (b) If index is not, repeat step 3 until done

- Use 3 states: occupied, empty, and removed
- Removed should be a flag indicating data can be overwritten in that index.

Hash Functions for Strings => Horner's Method

```
int hash = s.charAt(0);
for(i=1; i<s.length(); i++)
  hash = hash*b + s.charAt(i);
```

For constant b:
Java b=31