Data Structures Review

Quiz 2

Quadratic Sorts

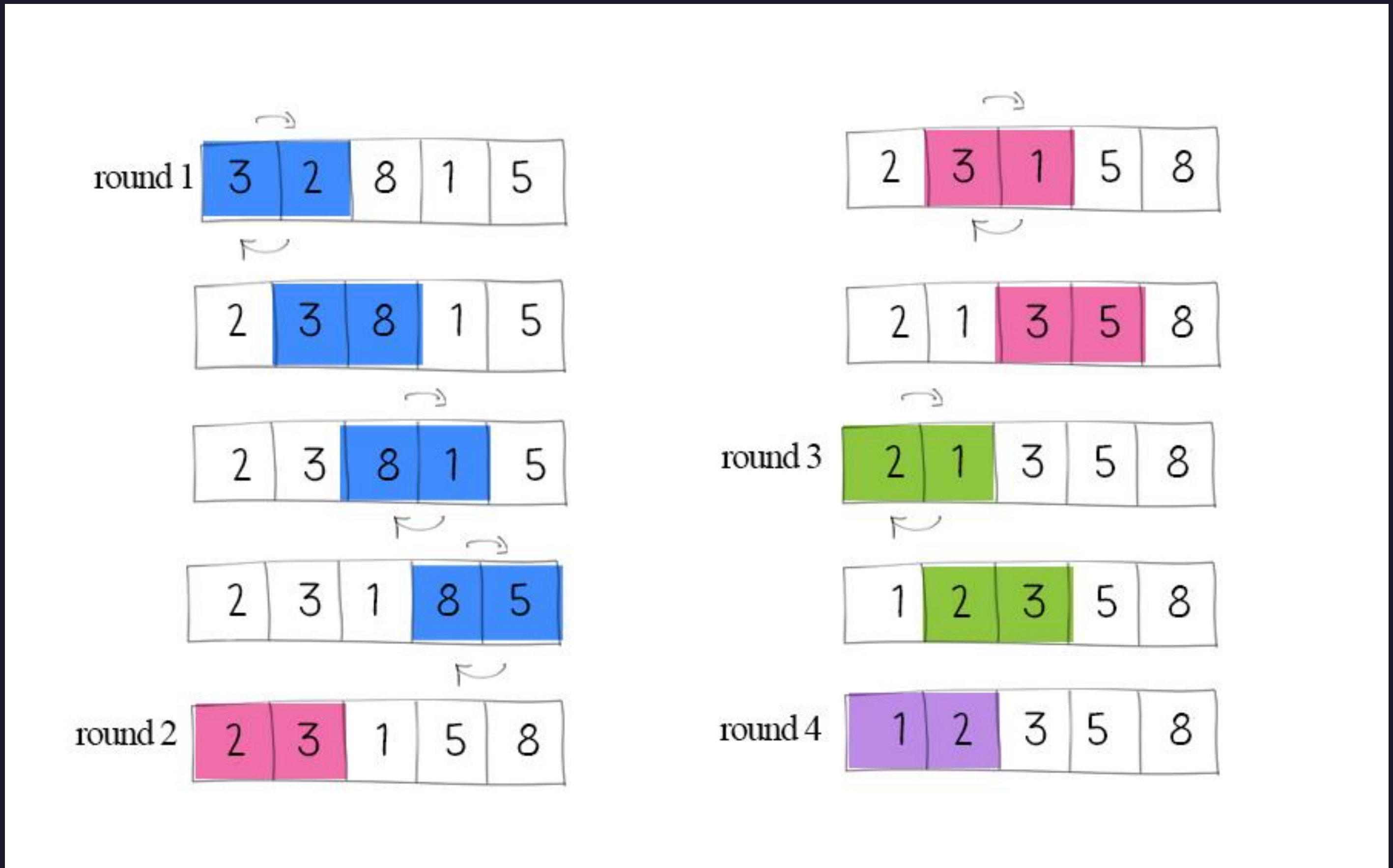
Selection Sort (selecting the smallest element)

13	46	24	52	20	9	13 > 9 ==> swap
9	46	24	52	20	13	46 > 13 ==> swap
9	13	24	52	20	46	24 > 20 ==> swap
9	13	20	52	24	46	52 > 24 ==> swap
9	13	20	24	52	46	52 > 46 ==> swap
9	13	20	24	46	52	Sorted array

Insertion Sort (insert next element into sorted list)

8	7	5	9	1	6	2	4	3
7	8	5	9	1	6	2	4	3
5	7	8	9	1	6	2	4	3
5	7	8	9	1	6	2	4	3
1	5	7	8	9	6	2	4	3
1	5	6	7	8	9	2	4	3
1	2	5	6	7	8	9	4	3
1	2	4	5	6	7	8	9	3
1	2	3	4	5	6	7	8	9

Bubble Sort (compare adjacent elements)



Practice Problem

Exercise Suppose we have the following array contents after the **third pass** of the outer loop of some quadratic sorting algorithms meant to put the array in ascending order:

$$3, 5, 7, 8, 2, 9, 4, 10, 15, 20$$

Which sorting algorithm could be operating on this array?

A) bubble (up) sort
B) (min) selection sort
C) insertion sort
D) none of these

Linear Data Structures

Stack, Queue, Linked Lists

Array List

ARRAY LIST

- Space used is O(n)
- Size(), isEmpty(), get(), and set() are O(1) operations
- Add() and remove() run in O(n)
- When adding to full array, grow!

GROW():

- Replace the array $k = \log_2 n$ times
- What is $T(n)$?
  - $n + 1 + 2 + 4 + 8 + \ldots + 2^k = n + 2^{k+1} = 2n - 1$
  - O(n)
- The amortized runtime is O(1)!!!!

Stack

STACK STUFF

- LIFO (Last in, First out) Data structure
- Push() Inserts element to the top of the stack
- Pop removes(/returns) the last inserted element
- Top() returns last inserted element

ARRAY BASED IMPLEMENTATION

- We can use an array to implement a stack
- Add elements from left to right
- Keep track of the index of the top element!!
- Popping an element → Decrement the top pointer
- Pushing an element → increment the stack pointer and insert.

Stack

ARRAY IMPLEMENTATION

Operation	How?	Time
`push`	add to end: `arr[numElement++]`	$O(1)$
`pop`	delete from end: `numElement--`	$O(1)$
`top`	return last: `arr[numElement - 1]`	$O(1)$
`empty`	check if `numElement == 0`	$O(1)$

Stack

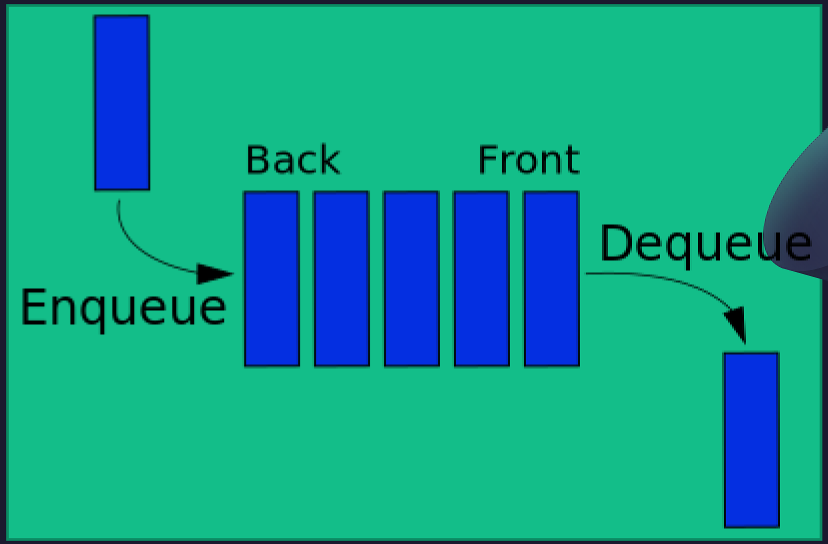
LINKED LIST IMPLEMENTATION

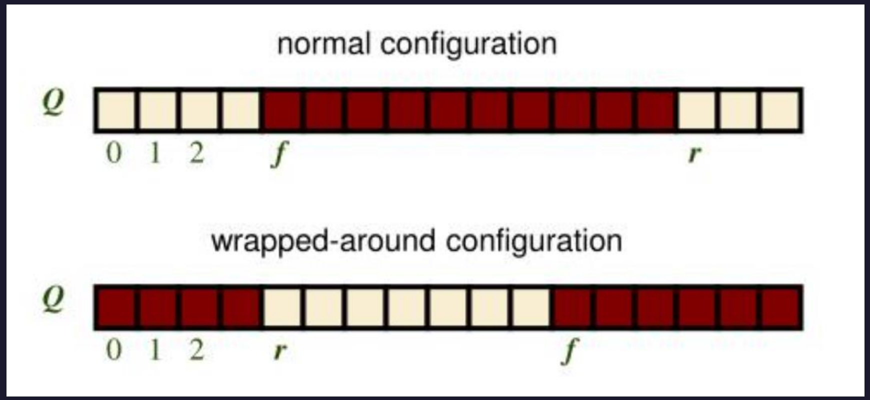
Operation	How?	Time
`push`	prepend the list and update the `head`	$O(1)$
`pop`	delete from front: `head = head.next`	$O(1)$
`top`	return `head.data`	$O(1)$
`empty`	check if `head` is `null`	$O(1)$

Queue

QUEUE STUFF

- Use an array of size N in a circular fashion
- Two variable to keep track of the front and rear
  - Index of the front element
  - Index immediately past the rear element
- Size = (N-f+r) mod N





Queue

LINKED LIST IMPLEMENTATION

Operation	How?	Time
`enqueue`	apend the list and update the `tail`	$O(1)$
`dequeue`	delete from front: `head = head.next`	$O(1)$
`front`	return `head.data`	$O(1)$
`empty`	check if `head` is `null`	$O(1)$

Queue

ARRAY IMPLEMENTATION

Operation	How?	Time
`enqueue`	`data[back] = value` and `back = ++back % length`	$O(1)$
`dequeue`	`front = ++front % length`	$O(1)$
`front`	return `arr[front]`	$O(1)$
`empty`	check if `numElement == 0`	$O(1)$

Linked Lists

SENTINEL NODES

- Dummy nodes at either end
- Head & Tail Pointer
- Removes edge cases

POSITION ADT

- Models the notion of a place within a data structure where a single object is stored.
- Gives a unified view
- One method – get()
- Protect the Node Class

Iterators

ITERATORS

- An object that enables you to traverse through a collection and to remove elements from the collection selectively if desired.
- Fail Fast & Version Numbers

Cooler Data Structures

Sets, Trees, Maps

Sets

SETS

- **Iterable** collection of unique elements
- Insert(), Remove,() Has()
- O(n) – Linked List Implementation
- O(n) – Array (Isn't sorted)
- DO NOT WORRY ABOUT ORDER WHEN ITERATING!

HEURISITICS

- Transpose Sequential Search
  - Search an array or list by checking items one at a time. If the value is found, swap it with its predecessor so it is found faster next time.
- Move to Front
  - moves the target of a search to the head of a list so it is found faster next time.

Ordered Set

SETS

- **Iterable** collection of **ordered** unique elements
  - Generics are bounded by Comparable Interface → Extends Comparable<T>
- O(n) – Linked List Implementation (Find)
- O(Lg n) – Array (Binary Search)
  - Insert and Remove are O(n) → Elements need to be shifted

Binary Search Trees

TREE STUFF

- Leaf – A node at the end of a tree with no children
- Root – node with out a parent
- Depth – number of ancestors deep excluding itself
- Height – Maximum depth of any node

BST STUFF

- In a binary tree, each node has at most two children.
- A BST – (Complete Ordering)
  - All elements to the left of a node are smaller.
  - All elements to the right of a node are larger.

BST Remove

NODE IS A LEAF

- ***Node to be removed is a leaf:*** Simply remove from the tree!

```
              50                            50
           /     \         remove(20)      /   \
         30       70       --------->    30      70
        /  \     /  \                      \    /  \
      20   40   60   80                    40  60   80
```

NODE HAS ONE CHILD

- ***Node to be removed has only one child:*** Copy the child value into the node and delete the child node.

```
              50                            50
           /     \         remove(30)      /   \
         30       70       --------->    40      70
           \     /  \                          /  \
           40   60   80                       60   80
```

BST Remove

NODE HAS TWO CHILDREN

- ***Node to be removed has two children:***

  1. Find the smallest value in the node's right subtree (*in-order* successor).
  2. Copy the value key of the in-order successor to the target node and delete the in-order successor.

```
              50                            60
           /     \         remove(50)     /    \
         40        70      --------->   40      70
                  /  \                            \
                60    80                           80
```

  - Note that the largest value in the left subtree (in-order predecessor) can also be used.

Tree Traversal

LEVEL-ORDER

- nodes are visited level by level from left to right

PRE-ORDER

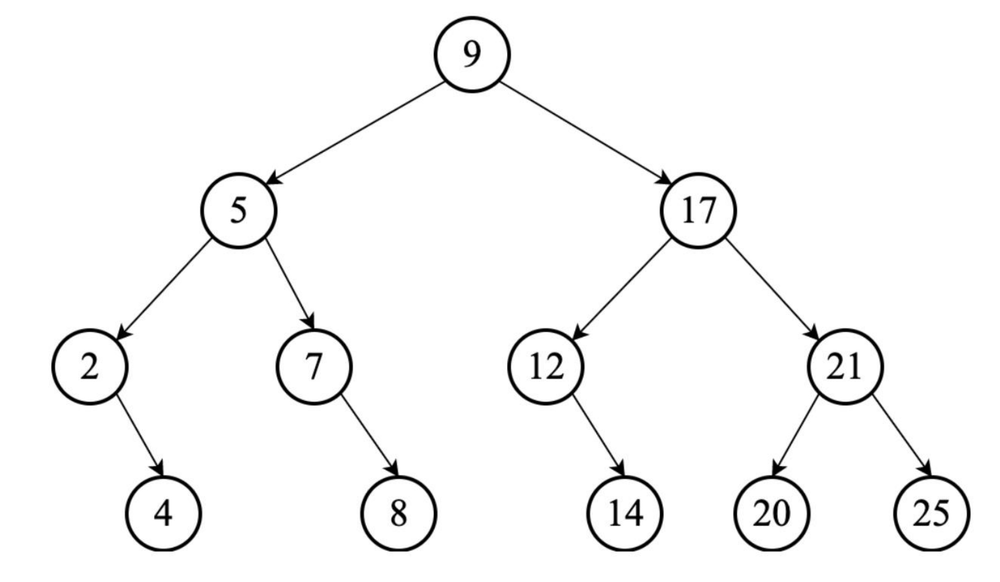
- For every node, visit it, then visit its left subtree, then visit its right subtree.

POST-ORDER

- For every node, visit its left subtree, then visit its right subtree, then visit the node.

IN-ORDER

- For every node, visit its left subtree, then visit the node, then visit its right subtree.



N – number of nodes
E – number of external nodes
I – Number of internal nodes
H - height

Properties

$$E = I + 1$$

$$N = 2e - 1$$

$$H \leq I$$

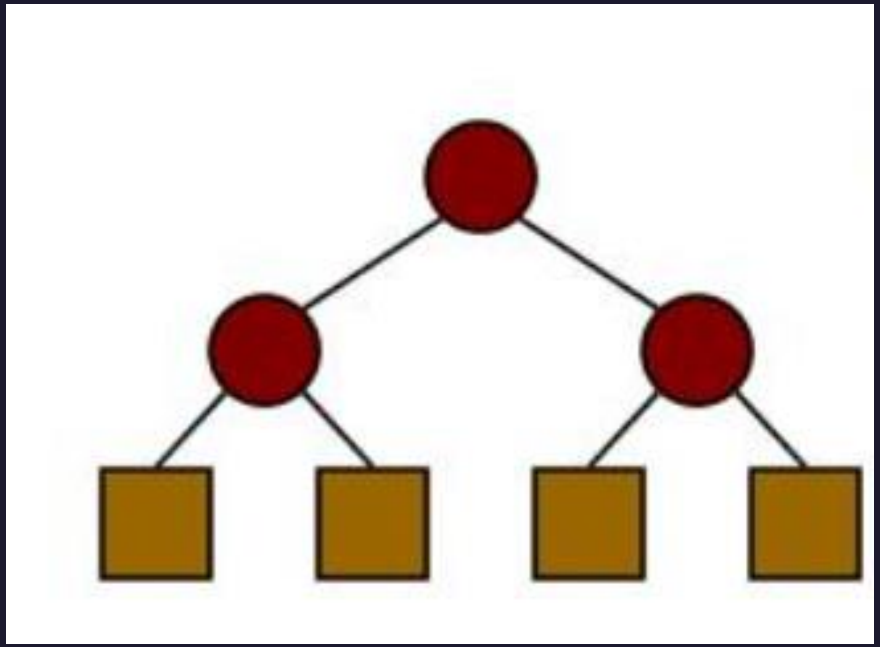
$$H \leq (n-1)/2$$

$$E \leq 2^h$$

$$H \geq \log 2\ e$$

$$H \geq \log_2(n+1) - 1$$

Perfect BST has height O(lg n)



Balanced Binary Search Trees

BBST STUFF

- For any node, the heights of the node's left and right subtrees are either equal or differ by at most one.
- Each node's balance factor is the height of the left subtree minus the height of the right subtree.
  - bf(node) = height(node.left) – height(node.right)
  - Height = for null, 0 for leaf, 1 + max(height of children)

BST STUFF

- In a binary tree, each node has at most two children.
- A BST – (Complete Ordering)
  - All elements to the left of a node are smaller.
  - All elements to the right of a node are larger.

General Study/Prep Tips

- Read the notes!
- Ask questions on Courselore
- Complete the practice midterm
- Do the optional exercises (i.e. implement bubble sort recursively)

Questions?


ABOUT TO START MY DATA STRUCTURES EXAM
PASSED!!
AAAND I ~~FAILED~~
memegenerator.net