

# CSE 332

## Data Abstractions

# Welcome to CSE 332!



# Outline

- 1 Administrivia
- 2 A Data Structures Problem
- 3 Review of Stacks & Queues

## Course Material

- “Classic” Data Structures/Algorithms
- Rigorously analyze efficiency
- When to use each type of data structure
- Sorting
- Dictionary ADT
- Parallelism and Concurrency
- ...

## CSE 143 vs. CSE 332

- Client of Priority Queue vs. Implementor of Priority Queue
- Linked Lists vs. Graphs
- BST vs. Balanced BST
- Merge Sort vs. Advanced Sorting
- X vs. Parallelism

During the course, we will...

- Implement many different data structures
- Discuss trade-offs between them
- Rigorously analyze the algorithms that use them (math!)
- Be able to pick “the right one for the job”
- Experience the purposes and headaches of multithreading

After the course, you will be able to...

- make good design choices as a developer, project manager, or system customer
- justify and communicate your design decisions

**This is the course where you stop thinking like a “Java Programmer” and start thinking like a Computer Scientist!**

## Resources

- Section every week!
- Lots of office hours!
- Piazza!

**Asking for help is not a sign of weakness; it's a sign of strength.**

## Course Website

<http://cs.uw.edu/332>

## Grading

- 35% programming projects, 25% exercises, 20% midterm, 20% final
- **four** tokens

## Partner Projects

- All three programming projects are “partners projects”
- If you want to work alone, you must petition to do it.
- Please sign up for one of the options TODAY!

## Textbook

Data Structures and Algorithm Analysis in Java (3rd edition) by Weiss

Do what helps you most.

...but **active learning** has been proven to result in better performance.



Choose a **data structure** and **algorithms** to solve the following problem:

## Prefix Sums

**Input:** An array `arr` of size  $n$ .

**Methods:**

- `arr.sum(i)` should return  $\sum_{k=0}^i \text{arr}[k]$
- `arr.update(i, value)` should update the value of the array at index  $i$  with `value`.

Then, analyze how good your solution is.

## Naïve Implementation

**Structure:** The input array.

**`arr.sum(i)`:** Loop from 0 to  $i$  adding up the elements.

**`arr.update(i, value)`:** Update index  $i$  with `value`.

How good is it? `sum` is  $\mathcal{O}(n)$ ; `update` is  $\mathcal{O}(1)$ .

Suppose we know `update` is going to happen very rarely but `sum` will happen a lot. Can we do better?

## Prefix Sums

**Input:** An array `arr` of size  $n$ .

**Methods:**

- `arr.sum(i)` should return  $\sum_{k=0}^i \text{arr}[k]$
- `arr.update(i, value)` should update the value of the array at index  $i$  with `value`.

Then, analyze how good your solution is.

## Another Try (?)

**Structure:** An array, partials of partial sums (e.g. `[3, 1, 9]` → `[3, 4, 13]`) .

**`arr.sum(i)`:** Return `partials[i]`.

**`arr.update(i, value)`:** Update every index from  $i$  to the end by adding the difference between old and new.

How good is it? `sum` is  $\mathcal{O}(1)$ ; `update` is  $\mathcal{O}(n)$ .

## Which is better?

## First Solution

- sum is  $\mathcal{O}(n)$
- update is  $\mathcal{O}(1)$

## Second Solution

- sum is  $\mathcal{O}(1)$
- update is  $\mathcal{O}(n)$

**This is a design trade-off!**

The answer is sometimes the left and sometimes the right.

## The left is better when. . .

- sum is rare; update is frequent
- We aren't allowed to change the data structure (or we're not allowed extra space).

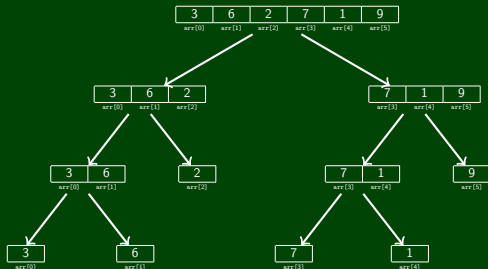
## The right is better when. . .

- sum is frequent; update is rare
- We're more concerned with the time complexity of sum than our space efficiency

Consider the following input array:

arr:	3	6	2	7	1	9
	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]

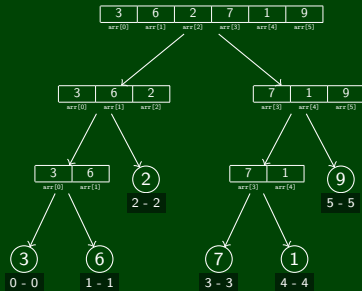
Let's get fancy now. Consider the following split of the array:



Consider the following input array:

arr:	3	6	2	7	1	9
	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]

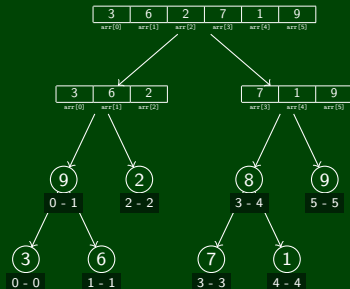
Let's build up a tree that stores **partial sums** in each node. Start at the leaves:



Consider the following input array:

arr:	3	6	2	7	1	9
	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]

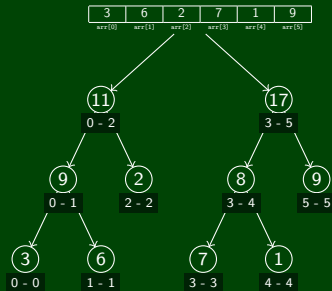
Let's build up a tree that stores **partial sums** in each node. Now go one level up:



Consider the following input array:

arr:	3	6	2	7	1	9
	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]

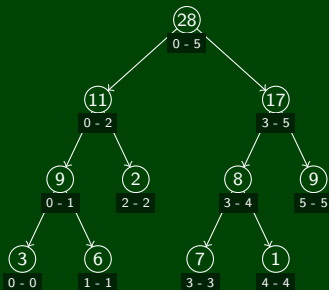
Let's build up a tree that stores **partial sums** in each node. And another...



Consider the following input array:

arr:	3	6	2	7	1	9
	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]

And finally, we get:



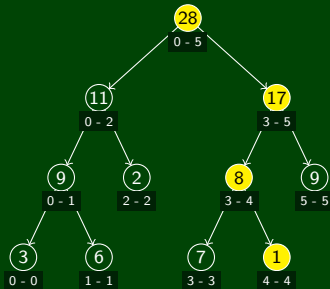
Let's use **THIS** as our data structure.

(For reference, this data structure is called a **Segment Tree**.)



Consider the following input array:

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]
3	6	2	7	1	9

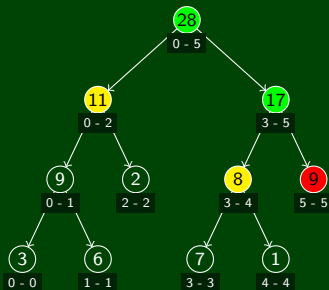


### How Do We Write update?

Walk up the tree from the leaf node that represents the index we're updating. Change each node accordingly.

What is the complexity of update?

**It's  $\mathcal{O}(\log n)$ , because the tree is balanced.**



## How Do We Write `sum(i)`?

```

1 sum(i) = sum(i, root);
2 sum(i, node) {
3   if (node.range is completely outside (0, i)) {
4     return 0;
5   }
6   else if ((0, i) contains something not in node.range) {
7     return sum(i, node.left) + sum(i, node.right);
8   }
9   else {
10    return node.value;
11  }
12 }
  
```

See above for `sum(4)`.

This is  $\mathcal{O}(\log n)$ , btw.

While trying to solve this problem, we did the following things:

- Considered an algorithmic problem and attempted to solve it
- Chose data structures and algorithms to solve the problem (duh...)
- Analyzed code for runtime
- Considered trade-offs between different implementations
- Learned a new data structure which helped us solve the problem **much better than before**
- Ran into analyzing a recursive runtime

One thing we didn't consider (but that we will later!) was how to solve the problem **if we had multiple processors**.

**This course is about learning fundamental data structures and algorithms to help you solve Computer Science problems.**

**Excited yet?** Okay...what if I told you this is an interview question?

## Definition (Abstract Data Type [ADT])

An **Abstract Data Type** is a **mathematical model** of the properties necessary for a data structure to be a particular data type. To put it another way, an ADT specifies what a data type **is** and the valid **operations** on it.

## Definition (Data Structure)

A **Data Structure** is a particular implementation of an ADT.

ADT	Data Structure	Implementation
Stack	ArrayList	java.util.Stack
Stack	LinkedList	-
Queue	LinkedList	java.util.LinkedList

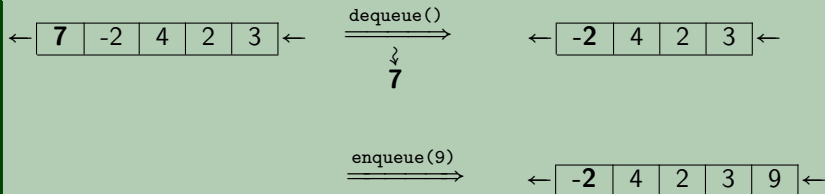
## Queue ADT

enqueue( <b>val</b> )	Adds <b>val</b> to the queue.
dequeue()	Returns the <b>least-recent</b> item not already returned by a dequeue. (Errors if empty.)
peek()	Returns the <b>least-recent</b> item not already returned by a dequeue. (Errors if empty.)
isEmpty()	Returns true if all inserted elements have been returned by a dequeue.

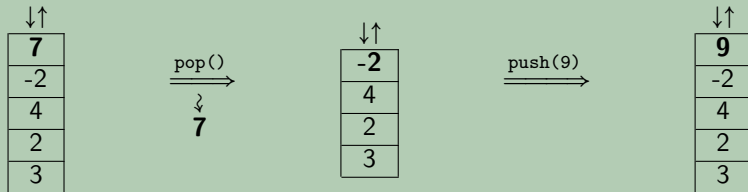
## Stack ADT

push( <b>val</b> )	Adds <b>val</b> to the stack.
pop()	Returns the <b>most-recent</b> item not already returned by a pop. (Errors if empty.)
peek()	Returns the <b>most-recent</b> item not already returned by a pop. (Errors if empty.)
isEmpty()	Returns true if all inserted elements have been returned by a pop.

## Queue Examples



## Stack Examples



ADTs are used to **COMMUNICATE** ideas more easily!

## Parentheses Matching

Given a string of parentheses (i.e. (, ), [, ]), figure out if the parentheses are matched.

**WORST:** A particular implementation in a particular language using the wrong ADT

```
for (int i = 0; i < str.length(); i++) {
    if (str.charAt(i) == '(' || str.charAt(i) == '[') {
        list.add(str.charAt(i));
    }
    else if ((str.charAt(i) == ')' && list.get(list.length() - 1) == '(') ||
             (str.charAt(i) == ']' && list.get(list.length() - 1) == '[')) {
        list.remove(list.length() - 1);
    }
    else {
        throw new Exception();
    }
}
if (list.size() > 0) {
    throw new Exception();
}
```

ADTs are used to COMMUNICATE ideas more easily!

## Parentheses Matching

Given a string of parentheses (i.e. (, ), [, ]), figure out if the parentheses are matched.

**REALLY BAD:** A particular implementation, in a particular language

```
for (int i = 0; i < str.length(); i++) {
    if (str.charAt(i) == '(' || str.charAt(i) == '[') {
        stack.push(str.charAt(i));
    }
    else if ((str.charAt(i) == ')' && stack.peek() == '(') ||
             (str.charAt(i) == ']' && stack.peek() == '[')) {
        stack.pop();
    }
    else {
        throw new Exception();
    }
}
if (!stack.isEmpty()) {
    throw new Exception();
}
```



ADTs are used to COMMUNICATE ideas more easily!

BETTER: Pseudo-code using the right ADT

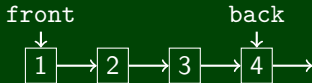
```
for (index in str) {
  if (str[index] is open) {
    put it on the stack
  }
  else if (str[index] is top of stack and it matches the top element) {
    pop the top element off the stack
  }
  else {
    throw error
  }
}
if (stack isn't empty) {
  throw error;
}
```

BEST: High-level description using the right ADT

To match parentheses, loop through the string pushing open parens onto the stack. When we see a close paren, make sure it matches and pop it off. If the stack isn't empty at the end, they don't match.

We can implement the **Queue ADT** using multiple ideas:

## ■ **Linked List Queue Data Structure**



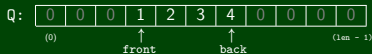
- Empty queue?
- Time complexities?

### Data Structure

```
enqueue(x) {
    back.next = new Node(x);
    back = back.next;
}

dequeue() {
    x = front.item;
    front = front.next;
    return x;
}
```

## ■ **Circular Array Queue Data Structure**



- Empty queue?
- Time complexities?

### Data Structure

```
enqueue(x) {
    Q[back] = x;
    back = (back + 1) % size;
}

dequeue() {
    x = Q[front];
    front = (front + 1) % size;
    return x;
}
```

	LinkedList Queue	CircularArray Queue
Space (in queue)?	No wasted space	Extra (or too little?)
Space (per element)?	Larger	Smaller
Operation Times?	Fast	Fast
Other Concerns?	Never runs out of space	<b>Can</b> run out of space

Why would we ever use a circular array queue?

- In practice, creating new Nodes **can fail**
- Memory allocation can be expensive
- Sometimes, we know in advance what the maximum size of the queue will be (see P1!)

- Hopefully you're excited!
- What is an ADT? What is a Data Structure?
- Understand Stack and Queue ADTs
- Understand Queue implementations

Go to the course website and:

- Read the partners handout and fill out the partners form.
- Finish Pokemon Purple & Gold (???)

Also, the website for petitioning into the course is here:

<http://tinyurl.com/hjl3tpj>