

Lecture 24: Final Review

Kaleel Mahmood

Department of Computer Science and Engineering

University of Connecticut

Reminder to self: Take 10 minutes to give class time to do the set survey.

What is the format of the final?

- 4 Questions (although some may have multiple parts).
- Closed book, closed notes. No computer. No calculator. Human brain only.
- For Syntax: we will not deduct points if you miss something simple like). However everything should be written in Scheme syntax. E.g. No Python “for” loops are allowed. If you need to loop use recursion.

*Is the final
cumulative?*



- A heavy focus is going to be on the latter part of the course.
- 3 out of the 4 questions will deal with material from the second half of the course.
- Does that mean you can forget everything you learned at the start?
NO! Everything builds on itself.

How hard is the final?

- To figure this out answer the following questions:
 1. Did you do the homework independently and understand the questions?
 2. Can you write Scheme code on paper in a reasonable amount of time?
 3. Did you pay attention in lecture?
 4. Does everything make sense on the lecture slides?

What is the focus of the final?



- Scheme is being phased out so we will not focus heavily on the Scheme language specifics.
- This course could have been taught in ANY developed programming language for a reason: the fundamentals of coding transfer between languages.
- Focus on the fundamentals: data structures, assigning and determining variable values, pointers, objects etc.

Now its time to go back in time and review...



Going through the material lecture by lecture

1. Lecture 13 - Sorting Part 2
2. Lecture 15 - Sorting Part 3
3. Lecture 16 - Binary Search Trees and Magic
4. Lecture 17 - Tree Removal and Heaps
5. Lecture 18 - Huffman Coding, ADT and Stacks
6. Lecture 19 - Mutable Data in Scheme
7. Lecture 20 - Object Oriented Design in Scheme
8. Lecture 21 - Introduction to Pointers
9. Lecture 22 - Queues with Pointers
10. Lecture 23 - Streams

- What you should understand:

Lecture 13: Sorting Part 2

Basic Idea Behind Selection Sort

1. Find the minimum element in an unsorted list.
2. Remove that element from the unsorted list.
3. Put that element in a new sorted list.
4. Repeat steps 1-3 on the unsorted list until no elements remain.
5. Return the new sorted list.

```
(define (smallest l)
  (define (smaller a b) (if (< a b) a b))
  (if (null? (cdr l))
      (car l)
      (smaller (car l) (smallest (cdr l)))))
```

```
(define (remove v elements)
  (if (null? elements)
      elements
      (if (equal? v (car elements))
          (cdr elements)
          (cons (car elements)
                (remove v (cdr elements))))))
```

Selection Sort in Scheme

```
(define (selSort l)
  (define (smallest l)
    (define (smaller a b) (if (< a b) a b))
    (if (null? (cdr l))
        (car l)
        (smaller (car l) (smallest (cdr l)))))
  (define (remove v l)
    (if (null? l)
        l
        (if (equal? v (car l))
            (cdr l)
            (cons (car l) (remove v (cdr l))))))
  (if (null? l)
      '()
      (let* ((first (smallest l))
              (rest (remove first l)))
        (cons first (selSort rest)))))
```

Code for finding the
smallest number in a list

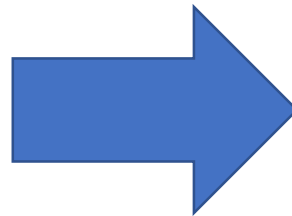
Code for removing an
element from a list

Code for doing selection
sort using smallest and
remove functions

Using let*

- In Scheme can we write this expression?

```
1 | (let* ((x 5)
2 |      (y 5)
3 |      (z (+ x y))))
4 | (+ x y))
```



```
Welcome to DrRacket, version 8.3 [cs].
Language: R5RS; memory limit: 128 MB.
10
>
```

Going through the material lecture by lecture

1. Lecture 13 - Sorting Part 2
2. Lecture 15 - Sorting Part 3
3. Lecture 16 - Binary Search Trees and Magic
4. Lecture 17 - Tree Removal and Heaps
5. Lecture 18 - Huffman Coding, ADT and Stacks
6. Lecture 19 - Mutable Data in Scheme
7. Lecture 20 - Object Oriented Design in Scheme
8. Lecture 21 - Introduction to Pointers
9. Lecture 22 - Queues with Pointers
10. Lecture 23 - Streams

- What you should understand:
 1. How to find the minimum/maximum in a list in Scheme.
 2. How to remove an element from a list and return a new list without the removed element.
 3. Don't need to worry about `let*`.

Lecture 15: Sorting Part 3

How to find the smallest crab using accumulators?



- **The smallest crab problem**: I give you three buckets. A bucket of crabs and two empty buckets. You also get a scale. If you can find the smallest crab I will give you an A in the course. If you don't find the smallest crab I will give you an F.
- The constraints: If you let any crabs escape you also fail. I expect you to return to me 3 buckets at the end. One bucket contains the smallest crab. The other bucket contains all remaining crabs. The last bucket should be empty.
- If you take a crab out of the bucket and leave it on the beach it will run away (then you fail).

How to find the smallest crab using only three buckets?

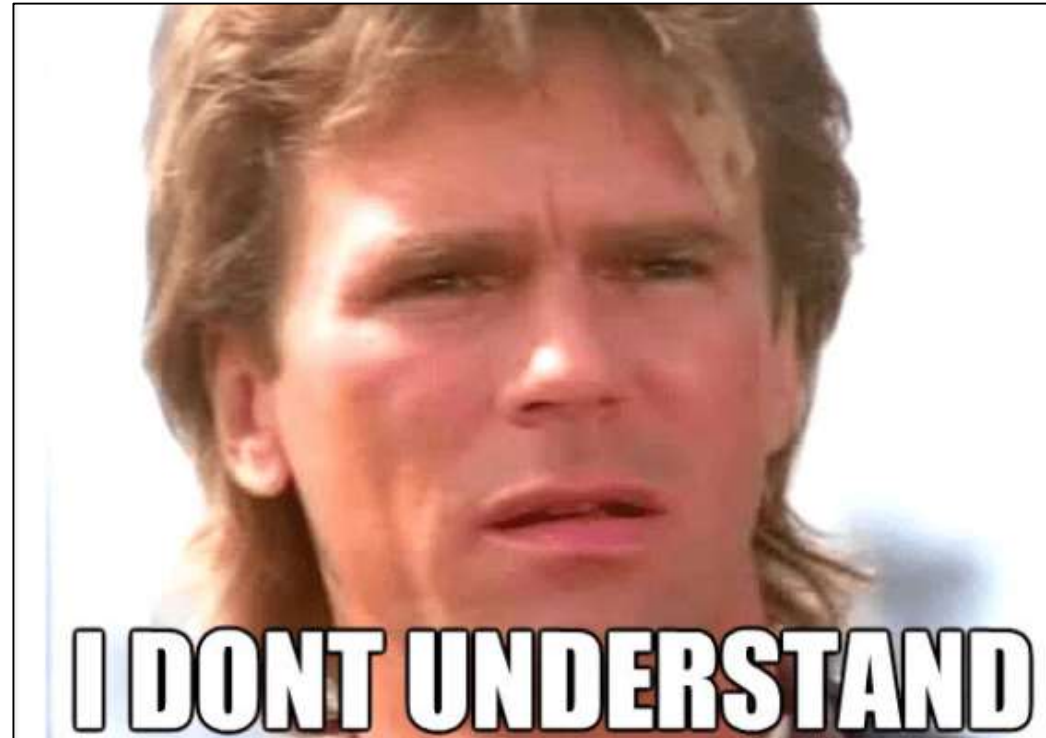
- Denote the bucket with the crabs we have already looked at that are NOT the smallest as the “clean” bucket.



A SOLUTION USING ACCUMULATORS

```
(define (alt-extract elements)
  (define (extract-acc smallest dirty clean)
    (cond ((null? dirty) (make-pair smallest clean))
          ((< smallest (car dirty)) (extract-acc smallest
                                                    (cdr dirty)
                                                    (cons (car dirty)
                                                          clean))))
    (else (extract-acc (car dirty)
                        (cdr dirty)
                        (cons smallest clean)))))
  (extract-acc (car elements) (cdr elements) '()))
```


But why would we program using an accumulator?



- It is faster? No.
- Is it an easy design pattern to understand and re-use? Maybe.
- Is it tail recursive? Yes!

Another Sorting Algorithm: Quicksort

Basic idea:

1. Choose a random element in the list.
2. Use this element as a pivot and divide the list into two parts.
3. Repeat step 1 with each of the two parts of the list.
4. Continue steps 1 through 3 until you reach lists that **ONLY** have one element (base case).

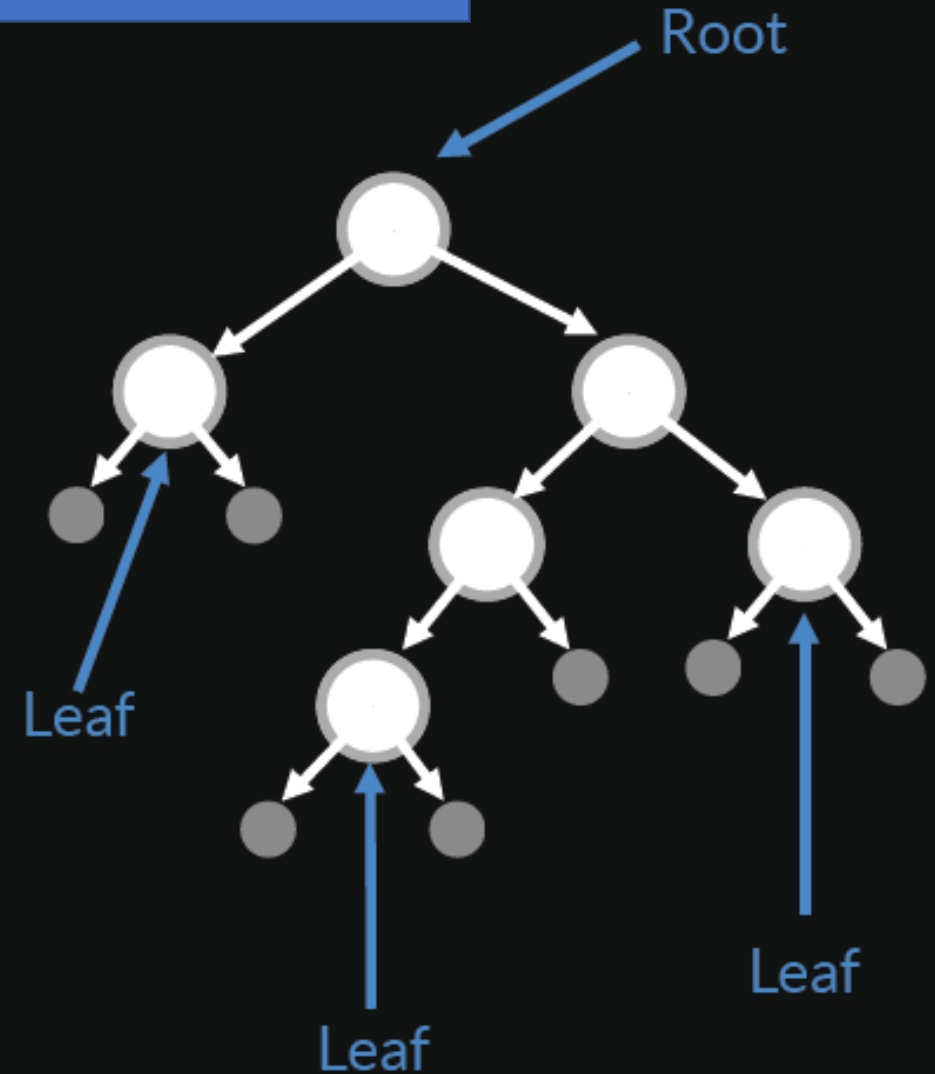
Going through the material lecture by lecture

- ~~1. Lecture 13 - Sorting Part 2~~
2. **Lecture 15 - Sorting Part 3**
3. Lecture 16 - Binary Search Trees and Magic
4. Lecture 17 - Tree Removal and Heaps
5. Lecture 18 - Huffman Coding, ADT and Stacks
6. Lecture 19 - Mutable Data in Scheme
7. Lecture 20 - Object Oriented Design in Scheme
8. Lecture 21 - Introduction to Pointers
9. Lecture 22 - Queues with Pointers
10. Lecture 23 - Streams

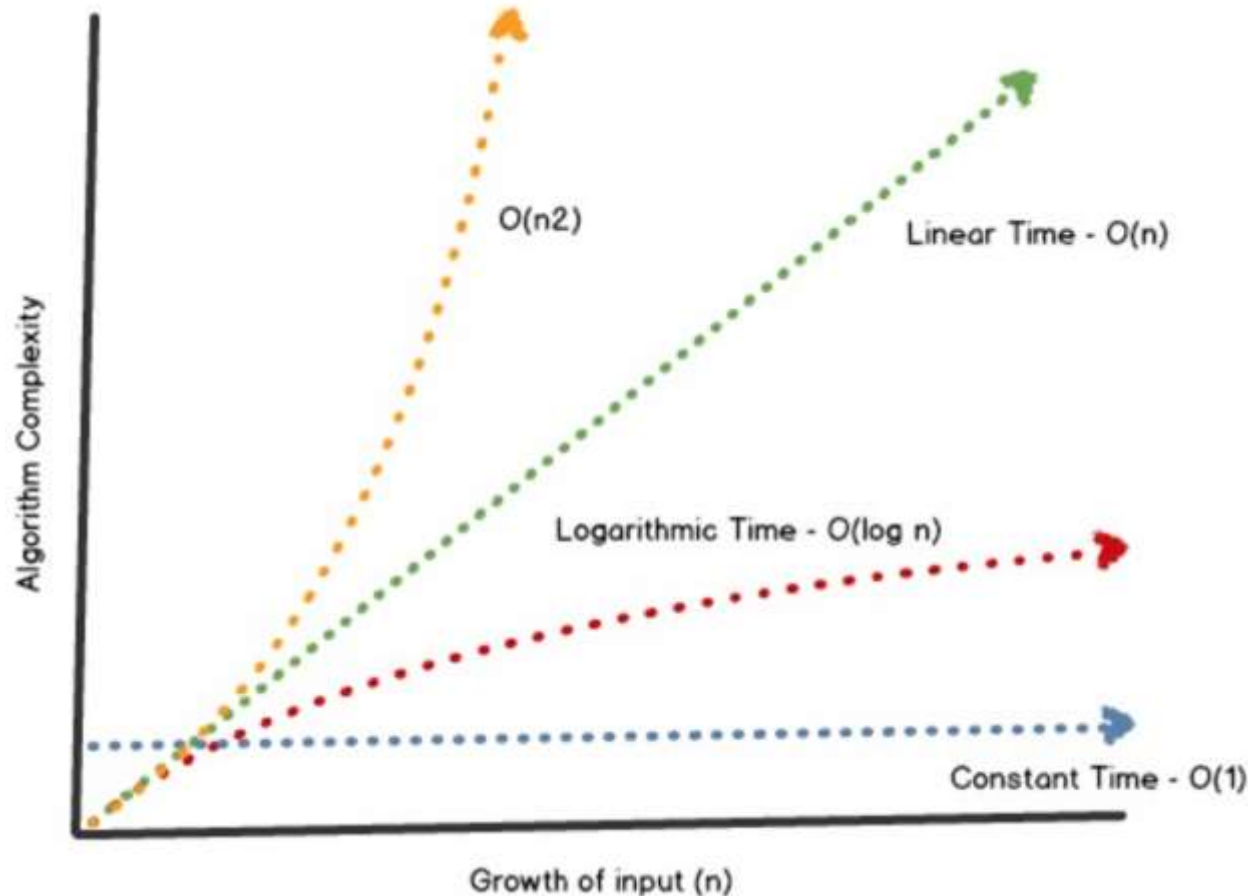
- What you should understand:
 1. How to use accumulators to find the minimum element in a list.
 2. What is the purpose of tail recursion.
 3. Don't need to worry about the implementation details of quicksort (but may be important in future classes).

Lecture 16: Binary Search Trees

- The top of the tree is the **root**.
- The **children** of a node are the roots of the trees to which it points.
- A **leaf** is a node with no children (so it points to two empty trees).

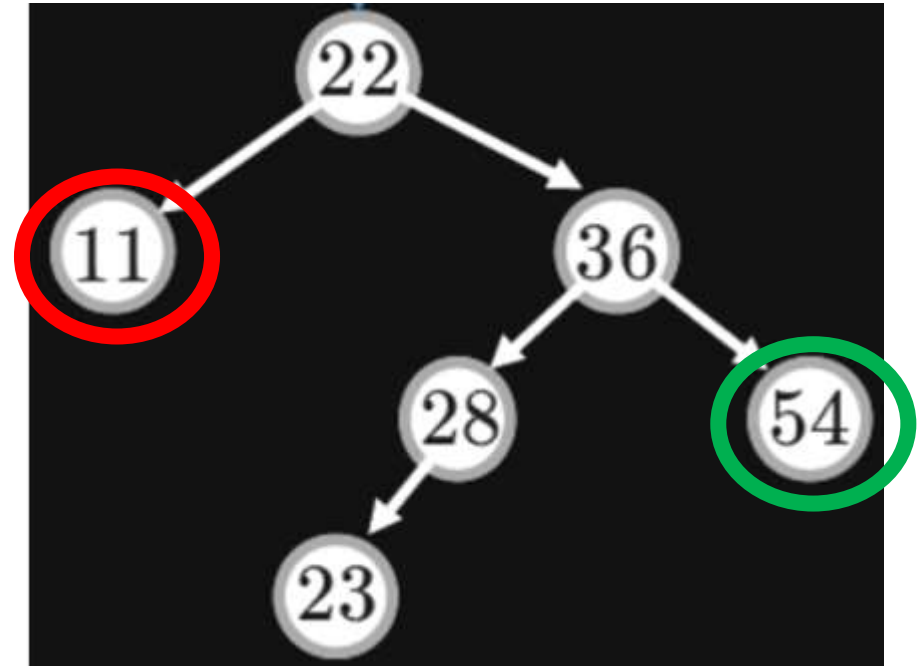
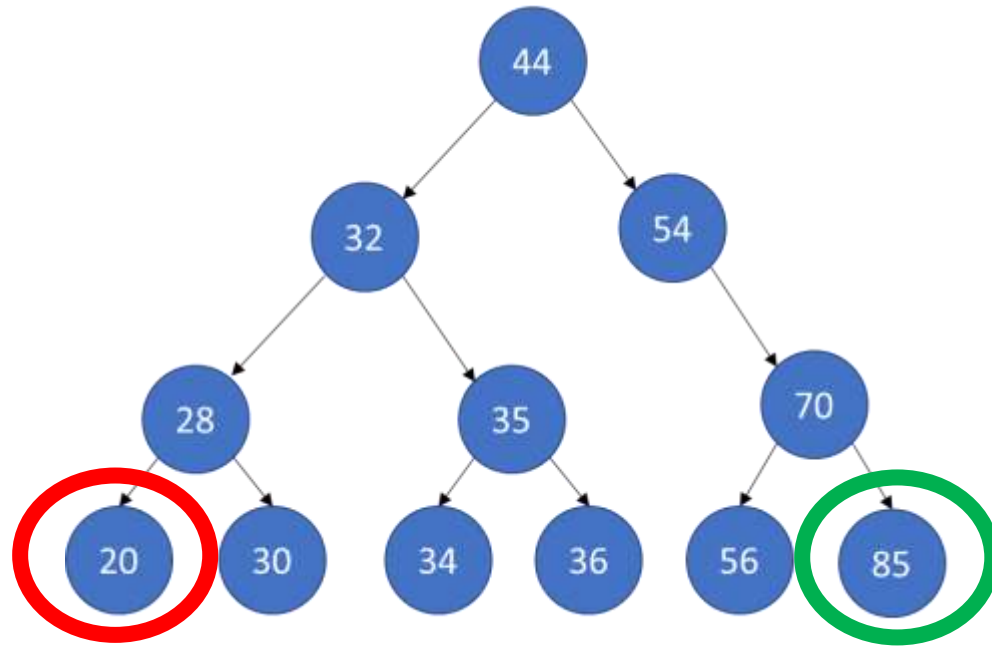


Again, why is $\log(n)$ depth for size n elements important?



- If the most common operation is membership query, a list will take n time for n elements.
- A tree will take $\sim \log(n)$ time for n elements. This means the tree scales better than a basic list.

Finding the smallest and largest elements in a binary search tree



- It should be obvious that the leftmost element (circled in red) in a binary search tree is always the smallest.
- Likewise it should be obvious that the rightmost element (circled in green) in a binary search tree is always the largest.

Going through the material lecture by lecture

- ~~1. Lecture 13 - Sorting Part 2~~
- ~~2. Lecture 15 - Sorting Part 3~~
3. **Lecture 16 - Binary Search Trees and Magic**
4. Lecture 17 - Tree Removal and Heaps
5. Lecture 18 - Huffman Coding, ADT and Stacks
6. Lecture 19 - Mutable Data in Scheme
7. Lecture 20 - Object Oriented Design in Scheme
8. Lecture 21 - Introduction to Pointers
9. Lecture 22 - Queues with Pointers
10. Lecture 23 - Streams

- What you should understand:
 1. How to insert an element into a binary search tree while maintaining the structure/order.
 2. How to find the min/max value in a binary search tree.
 3. How to traverse a binary search tree to look for an arbitrary element.
 4. What kind of operations does the binary search tree have that are faster than a traditional list?

Lecture 17: Tree Removal and Heaps

Summary of Tree Removal

1. The node has no children: Just replace the node with the empty list.
2. The node only has one child: Connect the child to the rest of the tree.
3. The node has two children: Connect the root of the left tree to the smallest element in the right tree.

OR replace the node to be removed with the smallest element in the right tree (which will be a leaf) and remove the smallest element in the right tree (which will follow case 1).

ANOTHER NATURAL TREE STRUCTURE

- A *heap* is a tree, with numbers stored at the nodes, with a different property:

Heap Property: The value of any node is smaller than that of its children.

- Notice that it is easy to determine the minimum element of a heap
 - it's always the root!
- If we can find a way to remove the minimum element and retain the heap property, we could use a heap for sorting. How?
 - I. Build a heap,
 - II. Repeatedly extract the minimum element.

Going through the material lecture by lecture

- ~~1. Lecture 13 - Sorting Part 2~~
- ~~2. Lecture 15 - Sorting Part 3~~
- ~~3. Lecture 16 - Binary Search Trees and Magic~~
- 4. Lecture 17 - Tree Removal and Heaps**
5. Lecture 18 - Huffman Coding, ADT and Stacks
6. Lecture 19 - Mutable Data in Scheme
7. Lecture 20 - Object Oriented Design in Scheme
8. Lecture 21 - Introduction to Pointers
9. Lecture 22 - Queues with Pointers
10. Lecture 23 - Streams

- What you should understand:
 1. How to remove an element from a tree and understand how to rebuild the tree after an element has been removed.
 2. Heaps – How to get the minimum element.
 3. Heaps – How to restructure the heap after the minimum has been removed.
 4. Heaps- How do we insert elements in a heap to build a balanced heap.

Lecture 18: Huffman Coding, ADT and Stacks

TRIES (OR PREFIX TREES)

- We have focused on binary trees, where each node has 0, 1, or 2 children.
- In principle, you can operate with trees of various branching factors.
- As an example, we consider *tries*, also called *prefix trees*, a data structure often used to maintain dictionary entries of fixed length.

Abstract Data Types: Buying a Dog



- Let's say you want to buy a dog. Almost all potential dog buyers assume the following:
 1. The dog can bark.
 2. The dog can walk.
 3. The dog can eat.
 4. The dog can be petted (very important!)

Stacks

- A container of objects, similar to a stack of coins or Pez dispenser.
- Objects can be inserted at any time.
- Only the top (last placed object) can be removed. Last-in-first out (LIFO).
- Pushing – An object is added to the top of the stack.
- Popping – Removing the top object (the last one added) from the stack.



FRONT

Queues

BACK



- A container of objects, similar to a line in a grocery store.
- Elements are inserted at the back and removed at the front. E.g. If you are in line first, you get served first.
 - FIFO- First in, first out.
 - Enqueue - Insert an element at the back.
 - Dequeue - Remove the element at the front.

Going through the material lecture by lecture

- ~~1. Lecture 13 - Sorting Part 2~~
- ~~2. Lecture 15 - Sorting Part 3~~
- ~~3. Lecture 16 - Binary Search Trees and Magic~~
- ~~4. Lecture 17 - Tree Removal and Heaps~~
5. Lecture 18 - Huffman Coding, ADT and Stacks
6. Lecture 19 - Mutable Data in Scheme
7. Lecture 20 - Object Oriented Design in Scheme
8. Lecture 21 - Introduction to Pointers
9. Lecture 22 - Queues with Pointers
10. Lecture 23 - Streams

- What you should understand:
 1. Don't need to worry about Huffman coding.
 2. Abstract data types are VERY important because they are common across programming languages.
 3. Know EXACTLY how a stack and queue works.
 4. Know the operation times for each operation in a stack and queue. Order for removal and addition of elements.

Lecture 19 : Mutable Data in Scheme

- The `set!` changes the value of a variable in the assignment in which it is called.
- A simple example:

```
1 > (define a 3)
2 > a
3 3
4 > (set! a 4)
5 > a
6 4
7 > (set! a 5)
8 > a
9 5
```



$a \mapsto 5$

Then we started to introduce objects...

Basic Idea: We can use the environments in Scheme like “walls” to protect variables from manipulation

- The basic definition

```
(define (new-account initial-balance)
  (let ((balance initial-balance))
    (lambda (withdrawal)
      (if (> withdrawal balance)
          "Insufficient funds"
          (begin
             (set! balance (- balance withdrawal))
             balance))))))
```

This function is returned;
it has access to balance

Going through the material lecture by lecture

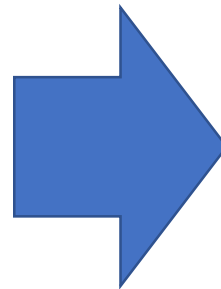
- ~~1. Lecture 13 - Sorting Part 2~~
- ~~2. Lecture 15 - Sorting Part 3~~
- ~~3. Lecture 16 - Binary Search Trees and Magic~~
- ~~4. Lecture 17 - Tree Removal and Heaps~~
- ~~5. Lecture 18 - Huffman Coding, ADT and Stacks~~
- 6. Lecture 19 - Mutable Data in Scheme**
7. Lecture 20 - Object Oriented Design in Scheme
8. Lecture 21 - Introduction to Pointers
9. Lecture 22 - Queues with Pointers
10. Lecture 23 - Streams

- What you should understand:
 1. How to use the set! function.
 2. Be very familiar with how to manipulate/set/check variables in different environments.
 3. Skills like keeping track of variables and values transfers across programming languages. Remember the spiderman examples?

Lecture 20 : Object Oriented Design in Scheme

Bad Code: Anyone can change the balance, no private variables.

```
(define balance 100)
(define (withdraw f)
  (if (> f balance)
      "Insufficient funds!"
      (begin (set! balance
                    (- balance f))
              balance)))
```



Better Code: balance can only be changed using the withdraw function

```
(define (new-account initial-balance)
  (let ((balance initial-balance))
    (lambda (withdrawal)
      (if (> withdrawal balance)
          "Insufficient funds"
          (begin
             (set! balance (- balance withdrawal))
             balance))))))
```

Getting multiple public functions:

```
(define (new-account initial-balance)
  (let ((balance initial-balance))
    (define (deposit f)
      (begin
        (set! balance
              (+ balance f))
        balance))
    (define (withdraw f)
      (if (> f balance)
          "Insufficient funds"
          (begin
            (set! balance
                  (- balance f))
            balance)))
    (define (bal-inq) balance)
    (lambda (method)
      (cond ((eq? method 'deposit) deposit)
            ((eq? method 'withdraw) withdraw)
            ((eq? method 'balance-inquire) bal-inq)))))
```

The methods

Public Methods
to manipulate
private objects

Dispatcher that
uses tokens to
determine the
method call

USING THE FIRST ORDER DISPATCHER

- Note that the dispatcher now returns the requested *function*.

```
> (define my-account (new-account 500))  
> (my-account 'deposit)  
#<procedure:deposit>  
> ((my-account 'deposit) 200)  
700  
> ((my-account 'withdraw) 250)  
450  
> ((my-account 'balance-inquire))  
450
```


Going through the material lecture by lecture

- ~~1. Lecture 13 - Sorting Part 2~~
- ~~2. Lecture 15 - Sorting Part 3~~
- ~~3. Lecture 16 - Binary Search Trees and Magic~~
- ~~4. Lecture 17 - Tree Removal and Heaps~~
- ~~5. Lecture 18 - Huffman Coding, ADT and Stacks~~
- ~~6. Lecture 19 - Mutable Data in Scheme~~
- 7. Lecture 20 - Object Oriented Design in Scheme**
8. Lecture 21 - Introduction to Pointers
9. Lecture 22 - Queues with Pointers
10. Lecture 23 - Streams

- What you should understand:
 1. How to build your own objects in Scheme.
 2. What is the importance of the function dispatcher?
 3. How to use the dispatcher and tokens to call methods.
 4. Why do we make variables in the object private?
 5. How do we modify variables if they are private?

Lecture 21 : Introduction to Pointers

```
x = 5
```

```
y = x
```

```
x = 3
```

```
print(y)
```

```
b1 = BankAccount()
```

```
b2 = b1
```

```
b1.addBalance(500)
```

```
b2.checkBalance()
```

- Pointers are used to access a certain place in memory.
- When dealing with a primitive, using the equality symbol you get a **NEW** pointer and a **NEW** place in memory with the value copied over.
- When dealing with objects, using the equality symbol gets you a **NEW** pointer to the **SAME** place in memory.

Not a pointer example but the stack code is important

```
(define (make-stack)
  (let ((S ' ()))
    (define (empty?) (null? S))
    (define (top) (car S))
    (define (pop) (let ((top (car S)))
                     (begin (set! S (cdr S))
                            top)))
    (define (push x) (set! S (cons x S)))
    (define (dispatcher method)
      (cond ((eq? method 'top) top)
            ((eq? method 'pop) pop)
            ((eq? method 'push) push)
            ((eq? method 'empty) empty?)))
    dispatcher))
```

Going through the material lecture by lecture

- ~~1. Lecture 13 - Sorting Part 2~~
- ~~2. Lecture 15 - Sorting Part 3~~
- ~~3. Lecture 16 - Binary Search Trees and Magic~~
- ~~4. Lecture 17 - Tree Removal and Heaps~~
- ~~5. Lecture 18 - Huffman Coding, ADT and Stacks~~
- ~~6. Lecture 19 - Mutable Data in Scheme~~
- ~~7. Lecture 20 - Object Oriented Design in Scheme~~
- 8. Lecture 21 - Introduction to Pointers**
9. Lecture 22 - Queues with Pointers
10. Lecture 23 - Streams

- What you should understand:
 1. The difference between primitives and objects when making copies.
 2. What is the difference between doing something like set!

Lecture 22 : Queues with Pointers

- `(define var <expr>)` sets `var` to be the value returned by `<expr>`.
- If the value is an atomic scheme type (number, Boolean, etc.): a new cell with the value is set up & the variable is directed there.

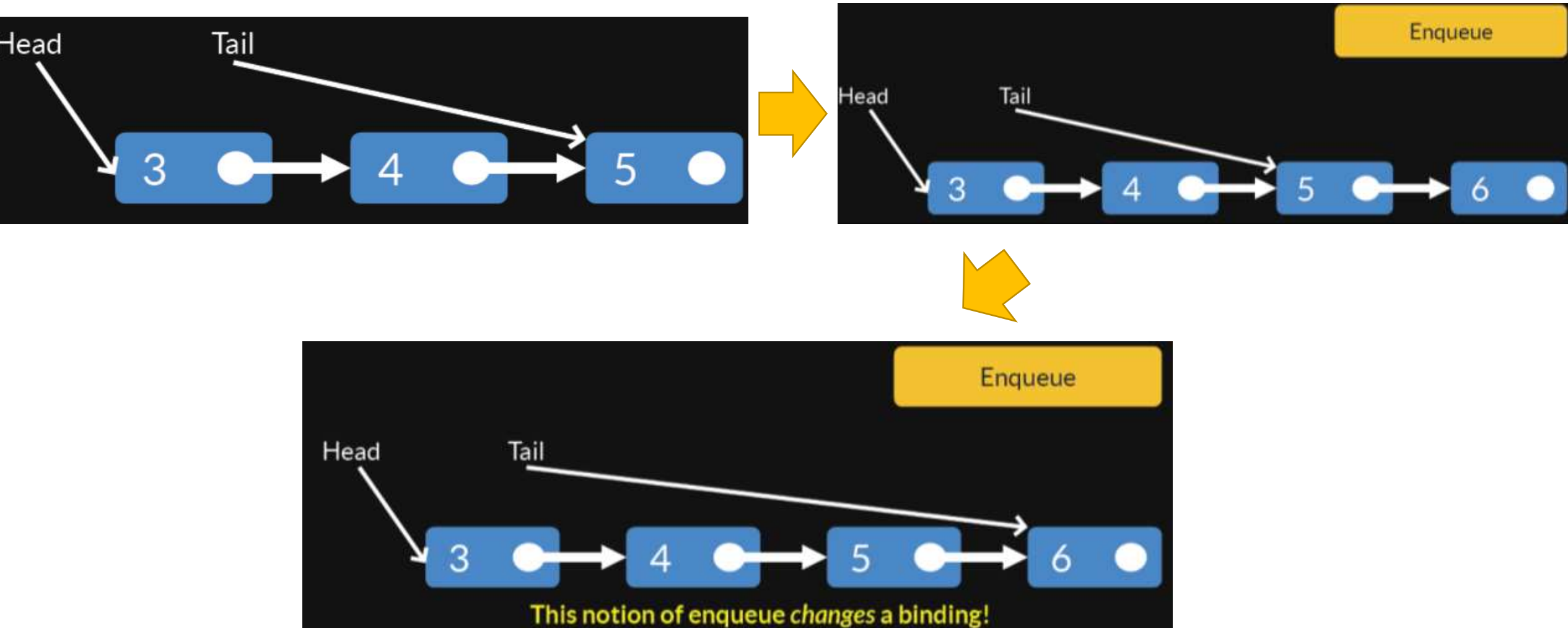
If this is confusing go back and review the example from last lecture where we did the same thing in Python.

```
1 > (define x 0)
2 > (define y x)
3 > x
4 0
5 > y
6 0
7 > (set! x 1)
8 > x
9 1
10 > y
```

Now what is the value of y?

0

Pictorial example of what we want to accomplish with enqueue:



Coded a new Queue Using Pointers

```
(define (make-queue)
  (let ((head '())
        (tail '()))
    (define (value n) (car n))
    (define (next n) (cdr n))
    (define (empty?) (null? head))
    (define (front) (value head))
    (define (enqueue x)
      (let ((new-node (cons x '())))
        (begin
          (if (empty?)
              (set! head new-node)
              (set-cdr! tail new-node))
          (set! tail new-node))))
    (define (dequeue)
      (let ((return (value head)))
        (if (eq? head tail)
            (begin (set! head '())
                   (set! tail '())
                   return)
            (begin (set! head (next head))
                   return))))
    (define (dispatcher method)
      (cond ((eq? method 'empty) empty?)
            ((eq? method 'enqueue) enqueue)
            ((eq? method 'dequeue) dequeue)
            ((eq? method 'front) front)))
    dispatcher))
```

private data

private functions

methods

the dispatcher, returned

We build a Queue with pointers.
Why did we build it this way?



```
(define (make-queue)
  (let ((head '())
        (tail '()))
    (define (value n) (car n))
    (define (next n) (cdr n))
    (define (empty?) (null? head))
    (define (front) (value head))
    (define (enqueue x)
      (let ((new-node (cons x '())))
        (begin
          (if (empty?)
              (set! head new-node)
              (set-cdr! tail new-node))
          (set! tail new-node))))
    (define (dequeue)
      (let ((return (value head)))
        (if (eq? head tail)
            (begin (set! head '())
                  (set! tail '())
                  return)
            (begin (set! head (next head))
                  return))))
    (define (dispatcher method)
      (cond ((eq? method 'empty) empty?)
            ((eq? method 'enqueue) enqueue)
            ((eq? method 'dequeue) dequeue)
            ((eq? method 'front) front)))
    dispatcher))
```



private data

private functions

methods

the dispatcher, returned

A queue built with pointers

	 Array		 Queue
Insertion	$O(n)$	Insertion (Enqueue)	$O(1)$
Deletion	$O(n)$	Deletion (Dequeue)	$O(1)$

Going through the material lecture by lecture

- ~~1. Lecture 13 - Sorting Part 2~~
- ~~2. Lecture 15 - Sorting Part 3~~
- ~~3. Lecture 16 - Binary Search Trees and Magic~~
- ~~4. Lecture 17 - Tree Removal and Heaps~~
- ~~5. Lecture 18 - Huffman Coding, ADT and Stacks~~
- ~~6. Lecture 19 - Mutable Data in Scheme~~
- ~~7. Lecture 20 - Object Oriented Design in Scheme~~
- ~~8. Lecture 21 - Introduction to Pointers~~
- 9. Lecture 22 - Queues with Pointers**
10. Lecture 23 - Streams

- What you should understand:
 1. The difference between define and set!. How to do variable manipulation with both in Scheme.
 2. How to use pointers to make the enqueue and dequeue operations run in constant time.
 3. I won't ask anything about cons vs mcons.

Going through the material lecture by lecture

- ~~1. Lecture 13 - Sorting Part 2~~
- ~~2. Lecture 15 - Sorting Part 3~~
- ~~3. Lecture 16 - Binary Search Trees and Magic~~
- ~~4. Lecture 17 - Tree Removal and Heaps~~
- ~~5. Lecture 18 - Huffman Coding, ADT and Stacks~~
- ~~6. Lecture 19 - Mutable Data in Scheme~~
- ~~7. Lecture 20 - Object Oriented Design in Scheme~~
- ~~8. Lecture 21 - Introduction to Pointers~~
- ~~9. Lecture 22 - Queues with Pointers~~
- 10. Lecture 23 - Streams**

- What you should understand:

1. What is a stream and the associated functions (head, empty rest). Very basic.
2. What thing does a stream object model and under what conditions would a stream data structure NOT be useful?

What about the first half of the course?

- Lecture 1
- Lecture 2 - Scheme and Spiderman
- Lecture 3 - Conditionals and Breaking Scheme
- Lecture 4 - Introduction to Recursion and The Scream
- Lecture 5 - Recursion Part 2 The Space Pen
- Lecture 6 - Let There Be Scheme
- Lecture 7- Lexical Scope
- Lecture 8 - Short Circuit Evaluation, Typing and Lambda Expressions
- Lecture 9 - Tail Recursion and Pairs
- Lecture 10 - Lists and Ghosts

- What you should understand:
 1. Everything in the first half of the course is need to do the second half of the course.
 2. For reviewing the first half of the course focus on tail recursion and setting variable values in different environments (think the spiderman example).



*It was a pleasure to be a part of your educational journey.
Good luck!*

Figure Sources

- <https://i.pinimg.com/originals/54/1c/8a/541c8a42b4bd8c568d75d9627a1c1389.jpg>
- [https://upload.wikimedia.org/wikipedia/en/d/dd/The Persistence of Memory.jpg](https://upload.wikimedia.org/wikipedia/en/d/dd/The_Persistence_of_Memory.jpg)
- <https://pbs.twimg.com/media/E2UtyOvXIAA2Zbh.jpg>
- <https://media.makeameme.org/created/one-does-not-0f3b3dbb6f.jpg>