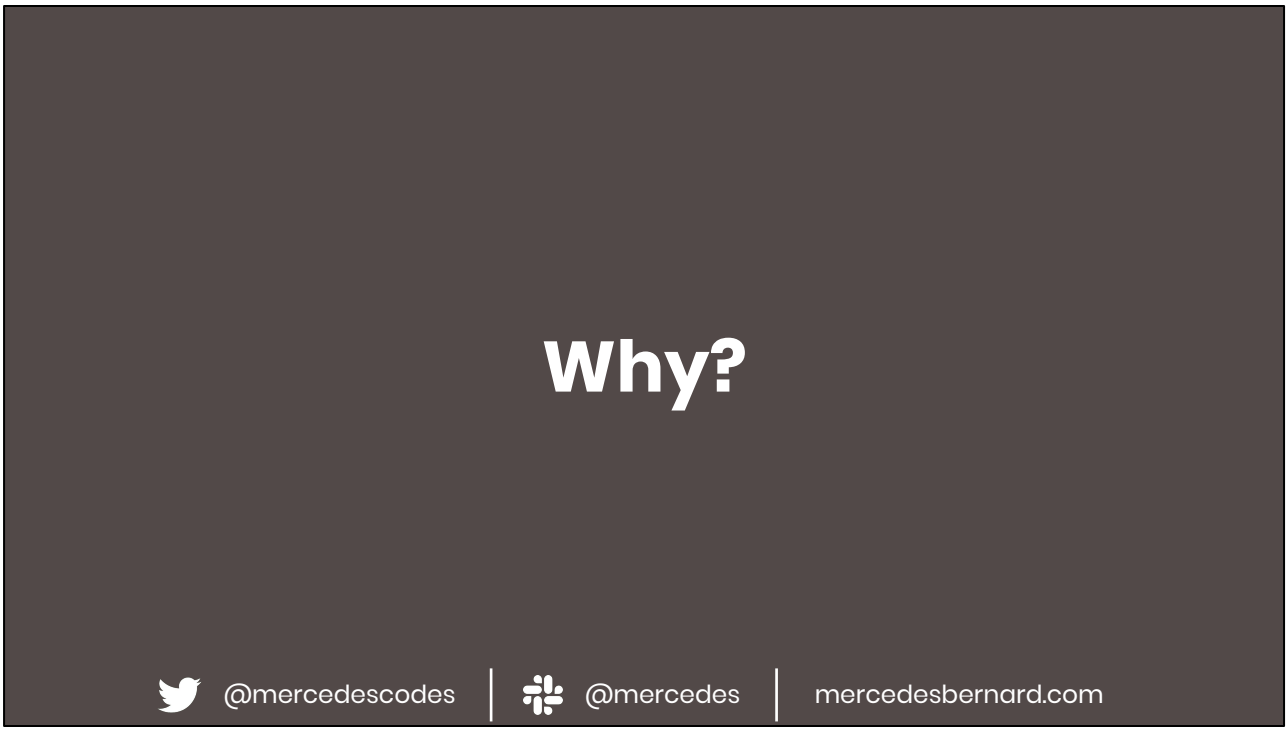
Fun, Friendly Computer Science

 @mercedescodes |  @mercedes | mercedesbernard.com

Hello! Today we're here to talk about Fun, Friendly Computer Science. This talk is going to be computer science quick hits. We're going to cover 7 topics in about 30 minutes. So because we don't have very much time, this talk will mainly focus on fundamental and introductory object-oriented programming concepts. But there's a whole world of other CS things to learn if you want to explore more.

If you have some familiarity with loops, array, and classes, you'll be able to follow along with this talk!

My name is Mercedes Bernard. My pronouns are she/her. And I'm a senior software engineer and engineering manager with a digital consultancy in Chicago called Tandem.



My background is in traditional, CS. I have a BS in CS. But the longer I'm in this industry, the more I realize that there is a lot that I learned that I never use. If you came into software from a non-traditional path, you may never have had the chance to learn this stuff. But you'll still be interviewed on it.

My goal with this talk is to show you that these topics that are used in interviews and sometimes used for gatekeeping aren't intimidating and also aren't really that important because a) you probably already know it and just don't have the words to explain it and b) you really don't use it very often.

You'll walk away from this talk with a high level understanding of a bunch of different topics as well as metaphors that you can use to explain them and examples that you can refer to later if you need them.

If you already know everything in this talk, that's great! But you'll probably still find something useful in explaining this to those you mentor or teach.

If you are someone who likes to reference slides or speaker notes while I'm talking, I've posted the slides for this talk here. I also tweeted out a link right before I got started so you can also find it on my Twitter profile.

There will be code samples in this talk and you can find them all in this repo. Be sure to check out the commits because each commit corresponds to one of the topics we'll cover here today.

The project runs and there's a simple UI that you can interact with if you are someone who learns by doing more than reading code.

Agenda

1. Concepts

- Big O notation
- Set theory
- Recursion

2. Data structures

- Linked list
- Stack
- Queue
- Tree

Concepts

Big O Notation

@mercedescodes | @mercedes | mercedesbernard.com

Cooking with ratios is when you don't memorize recipes but instead memorize the ratios of ingredients. The amount of ingredients you need changes *in proportion* to how much of the food you want to make. For example, cupcakes follow a 4:3:2:1 ratio.

Big O Notation measures relative complexity of a function or algorithm. Most often this is measuring running time but it could also be used to measure memory consumption, stack depth, and other resources. We're going to focus on running time since in an interview, that's what they're usually asking about. The actual input size is unimportant because we want to measure the proportional complexity of the logic.

This measurement is language/hardware/time agnostic and is relative to the code's input size.

We also talk about it according to worst case scenario. Sometimes in different sort and search algorithms you get lucky and the collection is nearly sorted or the object is near the beginning of your iteration but when we're talking about complexity, we want to plan for the worst case scenario.

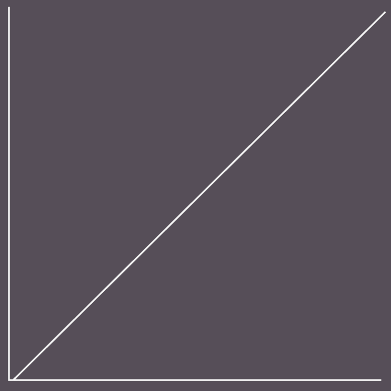
O(1)

```
def combine_butter_and_sugar(self, batches):
  steps = []
  butter = {
    "ingredient": self.recipe_ratios["butter"],
    "amount": batches * self.recipe_ratios["butter"]["number"]
  }
  sugar = {
    "ingredient": self.recipe_ratios["sugar"],
    "amount": batches * self.recipe_ratios["sugar"]["number"]
  }

  steps.append(self.beat_with_mixer([butter, sugar], 3))
  steps.append("Combine butter and sugar: O(1)")
  return "<br/>".join(steps)
```

@mercedescodes | @mercedes | mercedesbernard.com

O(1) = Constant running time regardless of input size

O(n)

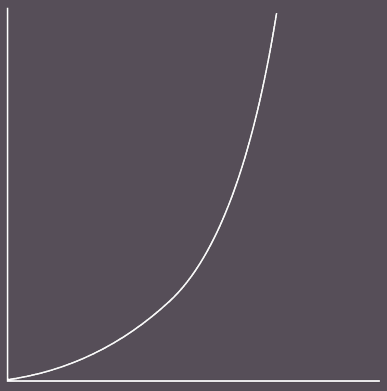


```
def add_eggs(self, batches):
  steps = []
  one_egg = { "ingredient": self.recipe_ratios["eggs"],
"amount": 1 }
  butter_mixture = { "ingredient": "butter mixture" }

  amount = batches * self.recipe_ratios["eggs"]["number"]
  for _ in range(amount):
    step = self.beat_with_mixer([one_egg, butter_mixture], 1)
    steps.append(step)

  steps.append("Added eggs: O(n)")
  return "<br/>".join(steps)
```

@mercedescodes | @mercedes | mercedesbernard.com

O(n) = Running time proportional to input size and running time increases linearly

$O(n^2)$

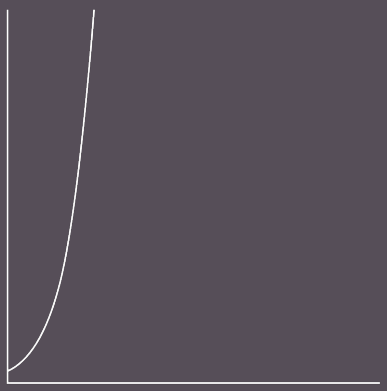


```
def combine_flour_mixture_and_milk_and_butter_mixture(self,
batches):
  steps = []
  # abbreviated for slides

  steps.append(self.beat_with_mixer([butter_mixture,
flour_mixture], 1))

  for _batch in range(batches):
    for _portion in range(batches):
      steps.append(self.beat_with_mixer([butter_mixture, milk],
1))
      steps.append(self.beat_with_mixer([butter_mixture,
flour_mixture], 1))

  steps.append("Slowly combined milk, flour mixture, and butter
mixture: O(n^2)")

  return "<br/>".join(steps)
```

@mercedescodes | @mercedes | mercedesbernard.com

O(n^2) = Running time proportional to the square of the input size. This is common in nested iterations.

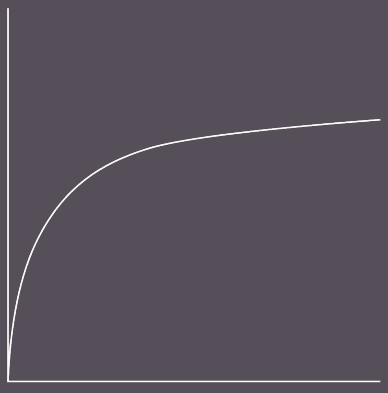
$O(2^n)$



```
def fibonacci_frosting(self, batches):
  number_to_frost = self.calculate_fibonacci_number(batches)
  return f"Iced the fibonacci number {number_to_frost} to all
of the cupcakes: O(2^n)"

def calculate_fibonacci_number(self, number):
  if number <= 1:
    print("Fibonacci base case!")
    return number
  else:
    print(f"Adding fibonacci numbers {number} and {number -
1}")
    return self.calculate_fibonacci_number(number - 1) +
self.calculate_fibonacci_number(number - 2)
```


O(2^n) = Running time grows exponentially with the size of the input. For example, calculating Fibonacci recursively

O(logn)



Divide and conquer algorithms such as binary search

O(log n) = This is kinda the opposite of O(2^n).

Big O Uses

@mercedescodes | @mercedes | mercedesbernard.com

Interviews
Comparing the performance of 2 possible solutions
Having a shared language

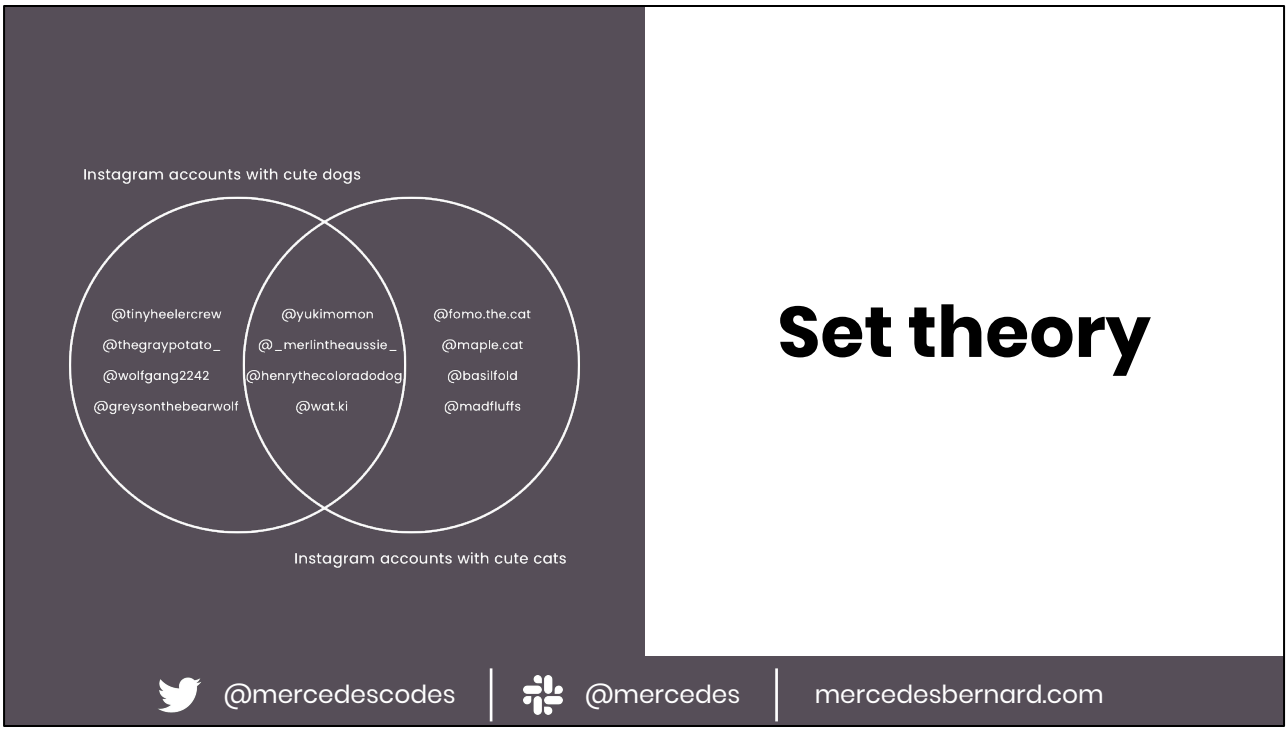
Agenda

1. Concepts

- ~~Big O notation~~
- Set theory
- Recursion

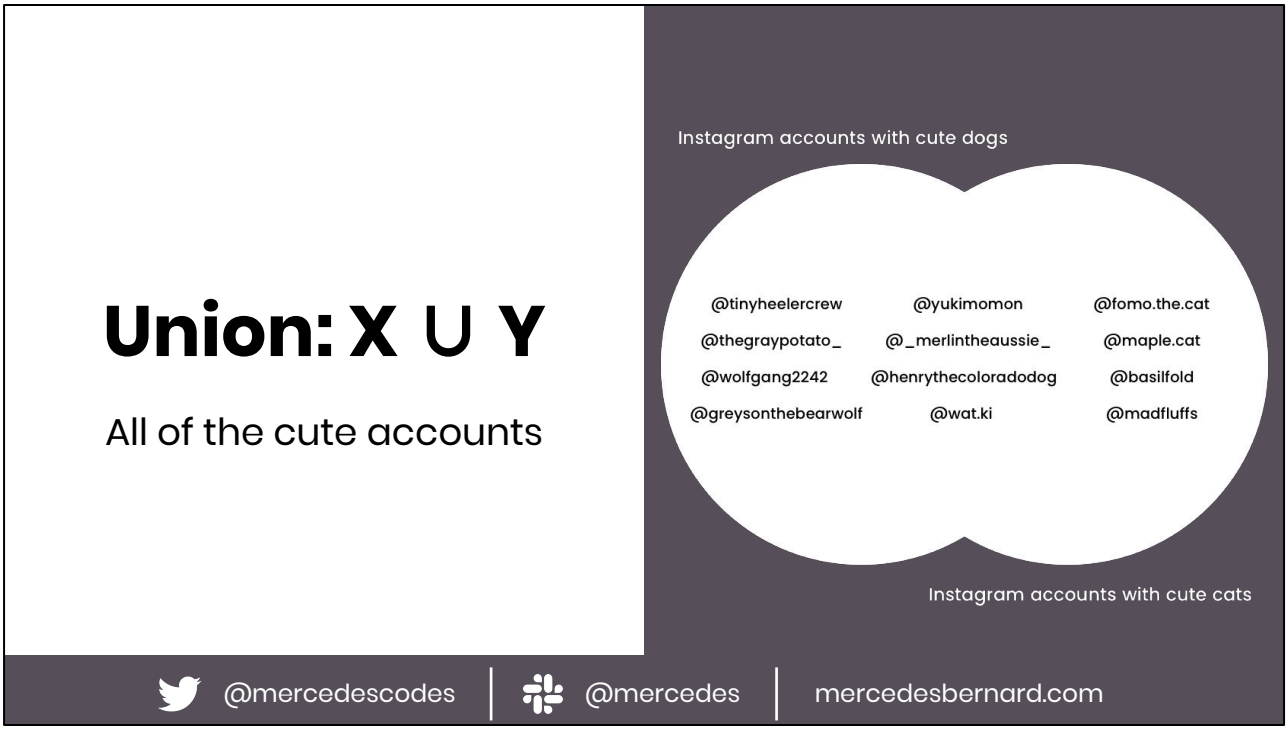
2. Data structures

- Linked list
- Stack
- Queue
- Tree



Venn diagrams are a great way to think about set theory. A set is a data structure similar to arrays or lists but it is an unordered collection of objects with no duplicates.

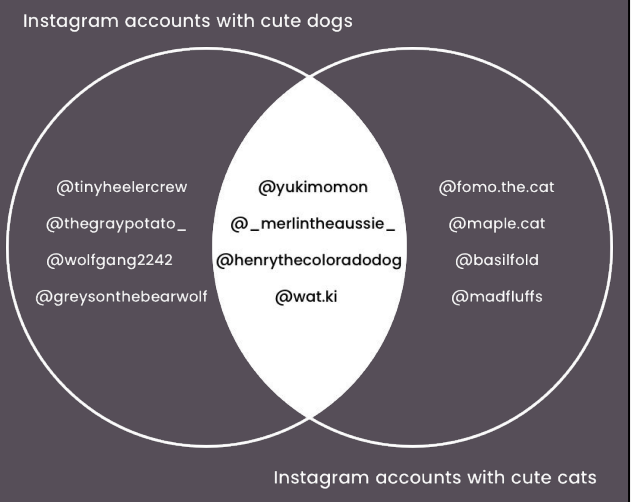
Sets are more interesting for what we can do on them. These operations form the basis of set theory.



Union - X ∪ Y The stuff that exists in X OR Y

Intersection: X ∩ Y

Accounts featuring cute dogs and cats

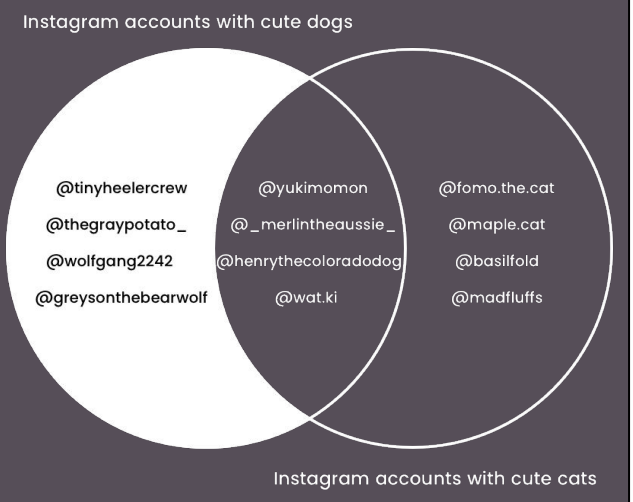


@mercedescodes | @mercedes | mercedesbernard.com

Intersection X ∩ Y - The stuff that exists in X AND Y

Difference: X - Y

Accounts featuring only cute dogs (no cats allowed)

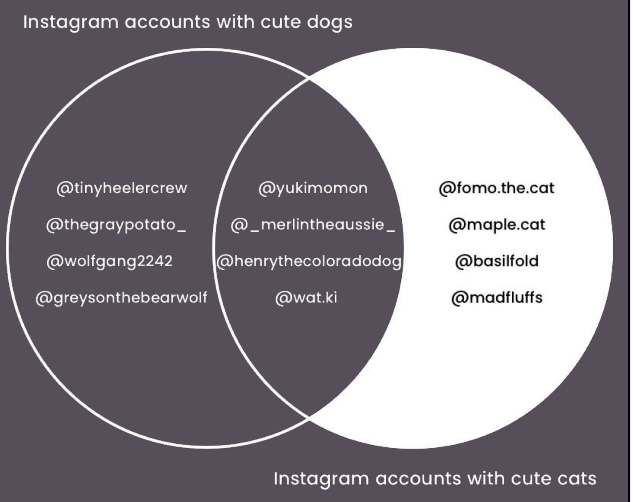


@mercedescodes | @mercedes | mercedesbernard.com

Difference - X - Y The stuff that only exists in X

Relative Complement: Y \ X

Accounts featuring only cute cats (no dogs allowed)



Relative complement Y \ X (same as Y - X) The stuff that only exists in Y

Symmetric Difference: X △ Y

Accounts featuring either dogs or cats, but not both (no cross species friendships here)

@mercedescodes | @mercedes | mercedesbernard.com

Symmetric difference (disjunctive union) X △ Y
Same as (X ∖ Y) ∪ (Y ∖ X).
The stuff that exists in only X and the stuff that exists in only Y but none of the stuff that exists in both

Set Theory Uses

@mercedescodes | @mercedes | mercedesbernard.com

Set theory is the foundation of relational databases
Website filters

Agenda

1. Concepts

- ~~Big O notation~~
- ~~Set theory~~
- Recursion

2. Data structures

- Linked list
- Stack
- Queue
- Tree

Recursion

@mercedescodes | @mercedes | mercedesbernard.com

Russian nesting dolls are a great metaphor for recursion, because each doll is the same except for its size. The dolls continue to open until you get to the smallest child which does not open. When you reach the smallest child, your reverse the process, closing each doll one by one in reverse order.

Recursion is the process in which a function calls itself directly or indirectly. And the function that is doing this calling of itself is called a recursive function. Everything that you do recursively you can also do in a loop.

When writing a recursive function, we don't want it to continue calling itself infinitely so we have to set up a condition where it exists the nesting and returns a finite value. This is called the base case. The smallest doll in Russian nesting dolls is like the base case.

When you're writing a recursive function, the base case is usually the easiest place to start.

```
def count(self):
  return self.count_nested_dolls(self.big_doll)

def count_nested_dolls(self, doll):
  child = doll.open()

  # base case
  if child is None:
    return 1

  return self.count_nested_dolls(child) + 1
```

Recursion Uses

 @mercedescodes |  @mercedes | mercedesbernard.com

Navigation paths on a website

Agenda

1. Concepts

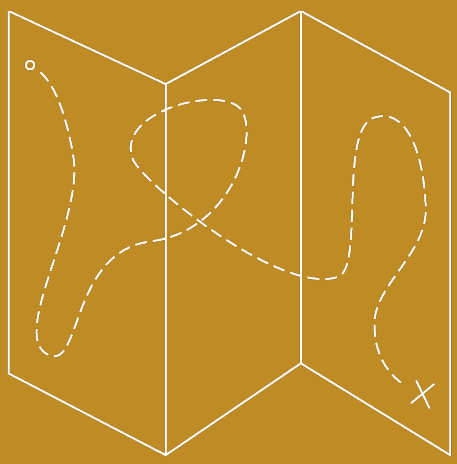
- ~~Big O notation~~
- ~~Set theory~~
- ~~Recursion~~

2. Data structures

- Linked list
- Stack
- Queue
- Tree

Data Structures

@mercedescodes | @mercedes | mercedesbernard.com



Linked List

@mercedescodes | @mercedes | mercedesbernard.com

When you are following a scavenger hunt, you start with the first clue and must follow each clue sequentially one at a time. Because you have no way of knowing where the nth clue is when you start, you must follow the hunt from clue to clue to clue.

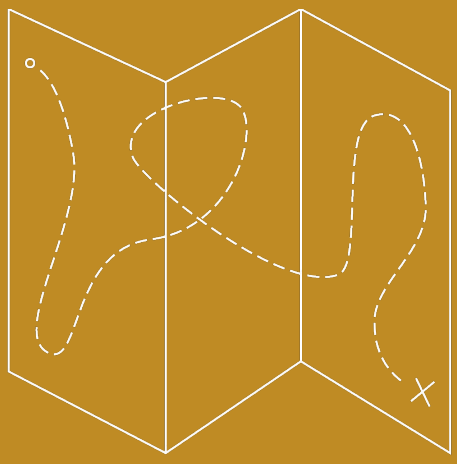
A scavenger hunt is a good metaphor for a linked list. A linked list is a data structure characterized by sequential data access and no random access. This is unlike arrays. In arrays, because all of the data is stored in contiguous locations in memory, you can do simple addition and subtraction to find the memory location of the data at the nth node. For example, array[3] is just 3 memory locations from array[0]. Linked lists on the other hand, do not need contiguous memory allocated. Each node in a linked list points to where the next node is located.

Because of the memory allocation, a linked list will have no wasted space because it can grow and shrink dynamically whereas traditionally an array always needs to be allocated to maximum size (or copied to a new array if it needs to grow). We tend to forget this when working with Javascript since JS Arrays behave more like ArrayList.

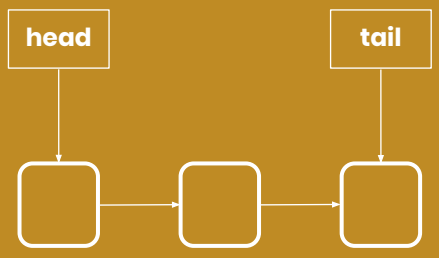
However, because a linked list node needs to store its data and a reference to the next node, it does take up a bit more space than an array of the same data.

Linked lists have fast insertion and deletion if its at the head or tail. But to insert into the middle of a linked list is O(n) because we need to loop through the list to the place we want to insert, update the previous nodes pointer to the inserted node and update

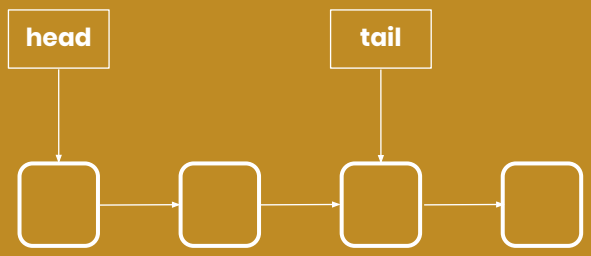
the inserted nodes pointer to the next node.



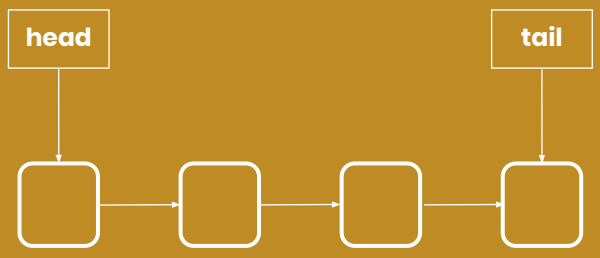
```python
def add(self, data):
  new_node = LinkedListNode.objects.create(data=data)
  if self.head is None:
    self.head = new_node
  else:
    self.tail.next = new_node
    self.tail.save()

  self.tail = new_node

  self.save()
  return new_node
```



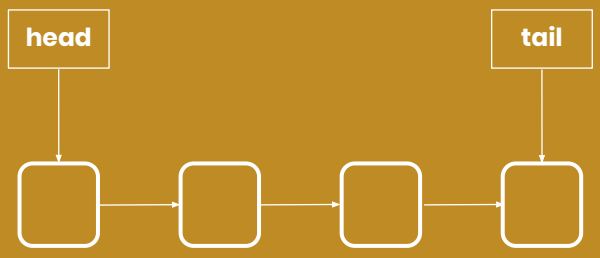

```
def add(self, data):
  new_node = LinkedListNode.objects.create(data=data)
  if self.head is None:
    self.head = new_node
  else:
    self.tail.next = new_node
    self.tail.save()

  self.tail = new_node

  self.save()
  return new_node
```



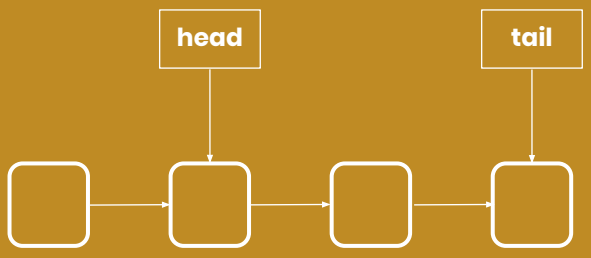

```
def add(self, data):
  new_node = LinkedListNode.objects.create(data=data)
  if self.head is None:
    self.head = new_node
  else:
    self.tail.next = new_node
    self.tail.save()

  self.tail = new_node

  self.save()
  return new_node
```



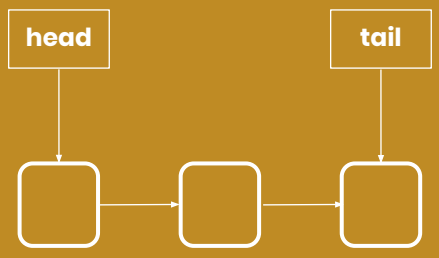

```
def add(self, data):
  new_node = LinkedListNode.objects.create(data=data)
  if self.head is None:
    self.head = new_node
  else:
    self.tail.next = new_node
    self.tail.save()

  self.tail = new_node

  self.save()
  return new_node
```



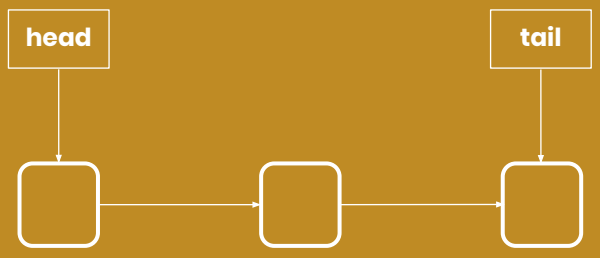

```
def remove(self):
  if self.head is None:
    return self.head

  removed = self.head
  self.head = self.head.next
  self.save()
  removed.next = None
  removed.save()
  return removed
```



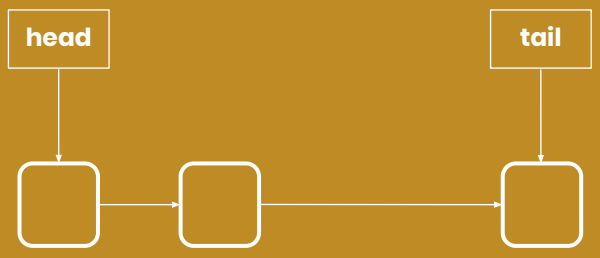

```
def remove(self):
  if self.head is None:
    return self.head

  removed = self.head
  self.head = self.head.next
  self.save()
  removed.next = None
  removed.save()
  return removed
```



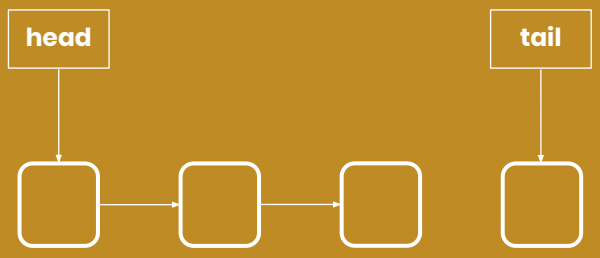

```
def remove(self):
  if self.head is None:
    return self.head

  removed = self.head
  self.head = self.head.next
  self.save()
  removed.next = None
  removed.save()
  return removed
```



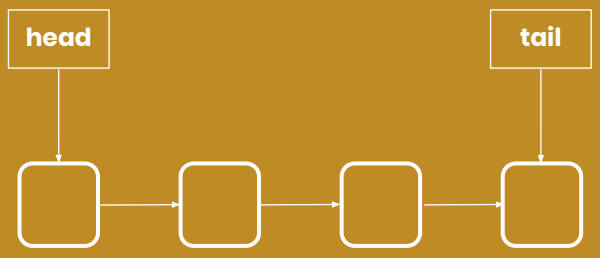

```python
def insert(self, data, index):
  if index < 0:
    return None

  new_node = LinkedListNode.objects.create(data=data)

  if index == 0:
    new_node.next = self.head
    self.head = new_node
    self.save()
  else:
    current = self.head
    previous = None
    i = 0

    while current is not None and i < index:
      previous = current
      current = current.next
      i += 1

    # found where to insert the new node
    if current is not None:
      previous.next = new_node
      new_node.next = current
      previous.save()
      new_node.save()
    elif previous == self.tail:
      self.add(data)

  return new_node
```




```
def insert(self, data, index):
  if index < 0:
    return None

  new_node = LinkedListNode.objects.create(data=data)

  if index == 0:
    new_node.next = self.head
    self.head = new_node
    self.save()
  else:
    current = self.head
    previous = None
    i = 0

    while current is not None and i < index:
      previous = current
      current = current.next
      i += 1

    # found where to insert the new node
    if current is not None:
      previous.next = new_node
      new_node.next = current
      previous.save()
      new_node.save()
    elif previous == self.tail:
      self.add(data)

  return new_node
```




```
def insert(self, data, index):
  if index < 0:
    return None

  new_node = LinkedListNode.objects.create(data=data)

  if index == 0:
    new_node.next = self.head
    self.head = new_node
    self.save()
  else:
    current = self.head
    previous = None
    i = 0

    while current is not None and i < index:
      previous = current
      current = current.next
      i += 1

    # found where to insert the new node
    if current is not None:
      previous.next = new_node
      new_node.next = current
      previous.save()
      new_node.save()
    elif previous == self.tail:
      self.add(data)

  return new_node
```




```
def insert(self, data, index):
  if index < 0:
    return None

  new_node = LinkedListNode.objects.create(data=data)

  if index == 0:
    new_node.next = self.head
    self.head = new_node
    self.save()
  else:
    current = self.head
    previous = None
    i = 0

    while current is not None and i < index:
      previous = current
      current = current.next
      i += 1

    # found where to insert the new node
    if current is not None:
      previous.next = new_node
      new_node.next = current
      previous.save()
      new_node.save()
    elif previous == self.tail:
      self.add(data)

  return new_node
```

Linked List Uses

@mercedescodes | @mercedes | mercedesbernard.com

When you don’t know the size of the data ahead of time and you don’t need efficient random access.

Implementing your own stack or queue… but would you really? Probably not since most languages and frameworks have these data structures built for you already. We don’t want to reinvent the wheel.

Blockchain

Doubly linked list
Playlist

Agenda

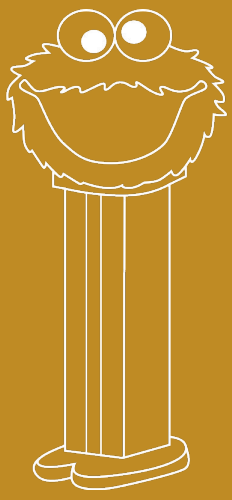
1. Concepts

- ~~Big O notation~~
- ~~Set theory~~
- ~~Recursion~~

2. Data structures

- ~~Linked list~~
- Stack
- Queue
- Tree

Stack

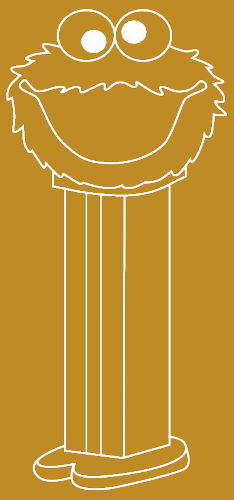


@mercedescodes | @mercedes | mercedesbernard.com

A PEZ dispenser is wonderful visualization of a stack. When you fill it with candy, you are pushing the candy from the top down and when you eat the candy, you pop a piece of the top of the literal stack of pieces one at a time.

A stack data structure behaves the same way. Stacks are characterized by last in, first out (LIFO) data access. When implemented as a linked list, adding and removing from the stack are O(1).

```
def push(self, data):
  new_node = LinkedListNode.objects.create(data=data)

  if self.head is not None:
    new_node.next = self.head
    new_node.save()

  self.head = new_node
  self.save()
  return new_node

def pop(self):
  popped = self.head
  self.head = self.head.next
  self.save()
  popped.next = None
  popped.save()
  return popped
```



Stack Uses

@mercedescodes | @mercedes | mercedesbernard.com

When you want to enforce LIFO data access.
Browser back button
Undo/redo feature
Checking for balanced delimiters in a string (parentheses, quotes, HTML tags, etc)

Agenda

1. Concepts

- ~~Big O notation~~
- ~~Set theory~~
- ~~Recursion~~

2. Data structures

- ~~Linked list~~
- ~~Stack~~
- Queue
- Tree

Waiting in line, or as the British call it queuing up, is a literal visualization of a queue. The first person in line is the first person out of line. And new folks joining the line (should) always join at the back.

A queue data structure behaves the same way. Queues are characterized by first in, first out (FIFO) data access. When implemented as a linked list, enqueuing and dequeuing from the queue are O(1).

```
def enqueue(self, data):
  new_node = LinkedListNode.objects.create(data=data)
  if self.head is None:
    self.head = new_node
  else:
    self.tail.next = new_node
    self.tail.save()

  self.tail = new_node

  self.save()
  return new_node

def dequeue(self):
  dequeued = self.head
  self.head = self.head.next
  self.save()
  dequeued.next = None
  dequeued.save()
  return dequeued
```

Queue Uses

When you want to enforce FIFO data access.
Processing jobs asynchronously but in the order they were scheduled

Agenda

1. Concepts

- ~~Big O notation~~
- ~~Set theory~~
- ~~Recursion~~

2. Data structures

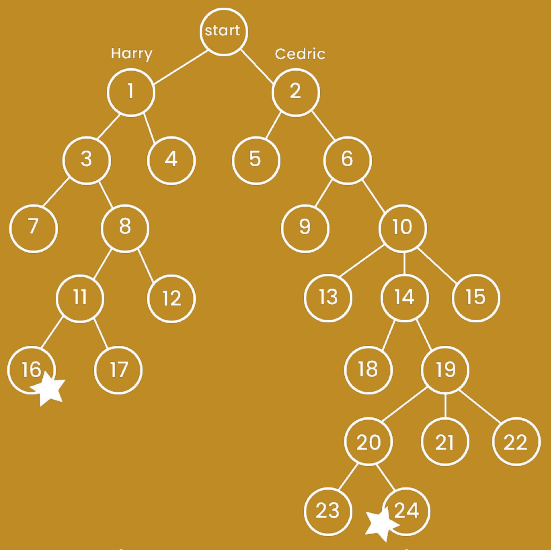
- ~~Linked list~~
- ~~Stack~~
- ~~Queue~~
- Tree

If you're a big Harry Potter fan, you probably enjoyed the end of the fourth book that featured a giant maze full of magical obstacles. The first person to the center of the maze won the Triwizard cup. When you are completing a maze, you are often faced with choices about whether to go left or right or in some cases continue straight down the center path.

Modeling these choices can be done using a tree data structure. Unlike linked lists, stacks, or queues, trees are useful to model hierarchical data. Think file systems or organization charts. In a tree data structure, there is a root node that has 1 to many children (in much the same way that a linked list stores a reference to the next node in the list).

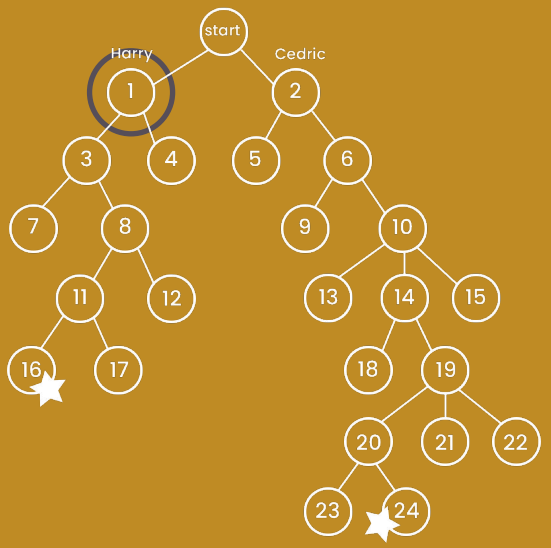
To access other nodes in the tree, you must traverse through the nodes.

Harry Potter & the Triwizard Maze



@mercedescodes | @mercedes | mercedesbernard.com

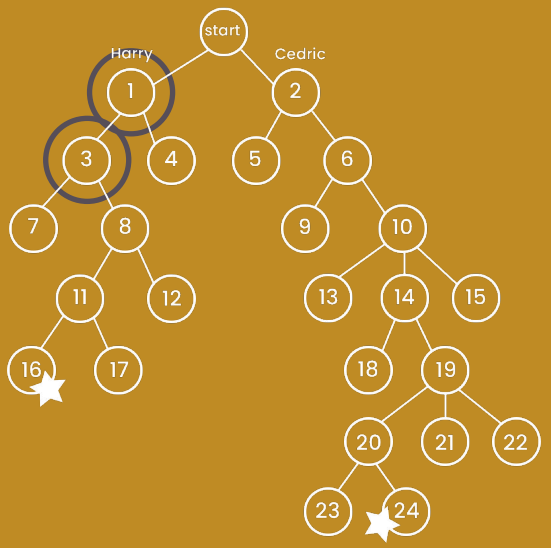
Breadth First Search



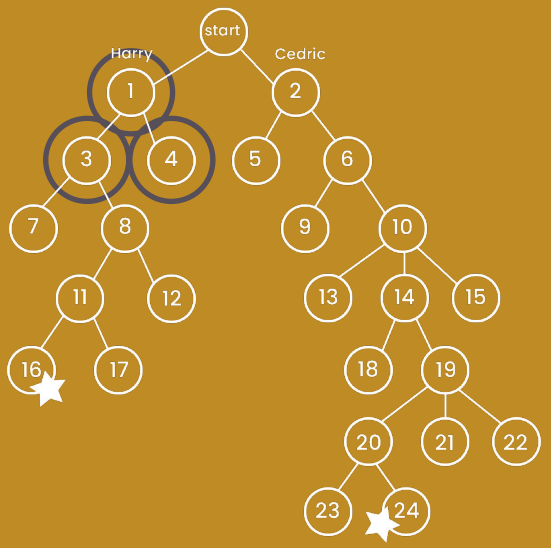
@mercedescodes | @mercedes | mercedesbernard.com

Breadth first search is when you traverse the nodes one level at a time, visiting all the nodes on level 1, then level 2 and so on. In the case of a maze, BFS is the worst way to get to the center quickly.

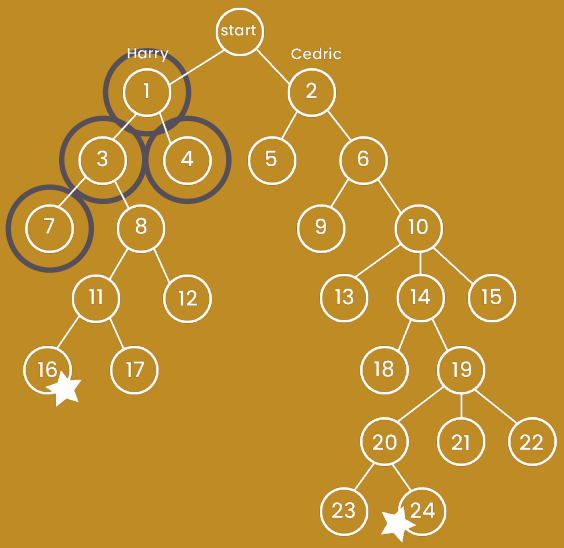
Breadth First Search



Breadth First Search

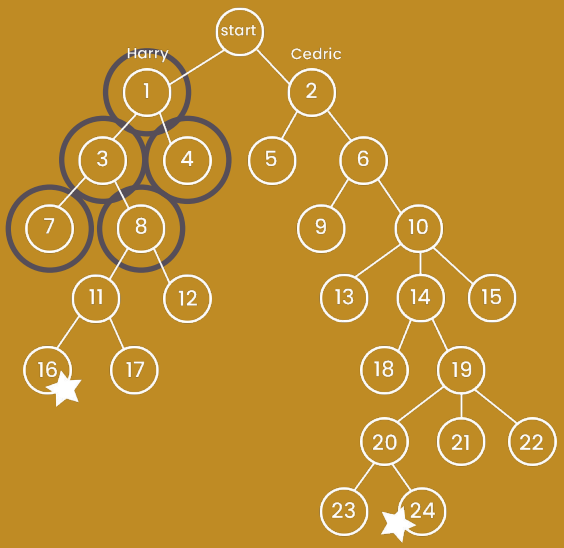


Breadth First Search

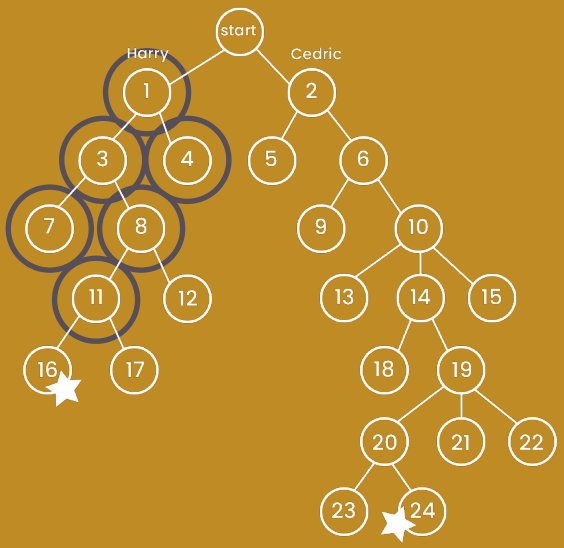


@mercedescodes | @mercedes | mercedesbernard.com

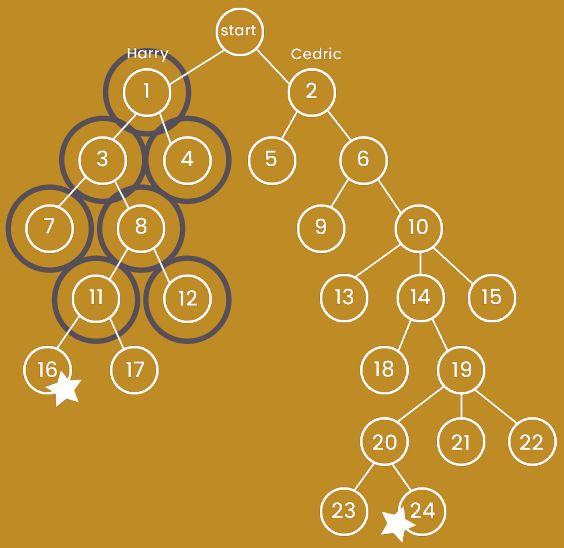
Breadth First Search



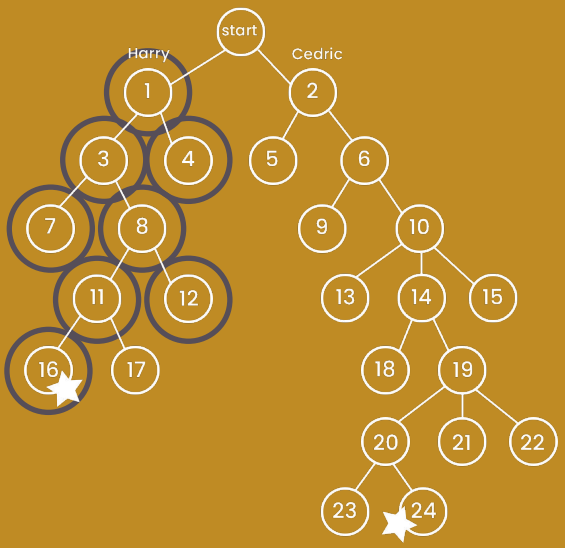
Breadth First Search



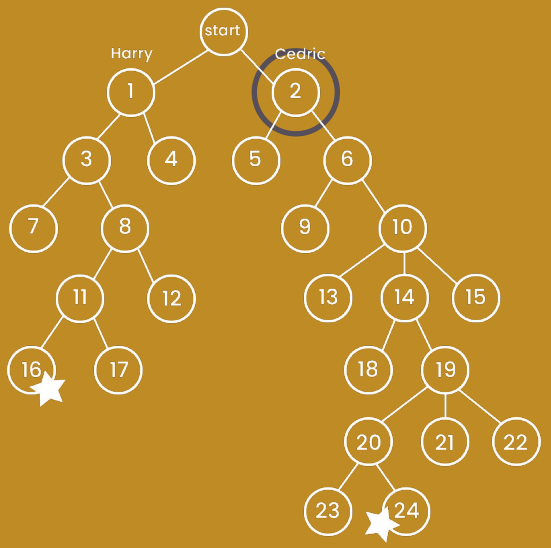
Breadth First Search



Breadth First Search



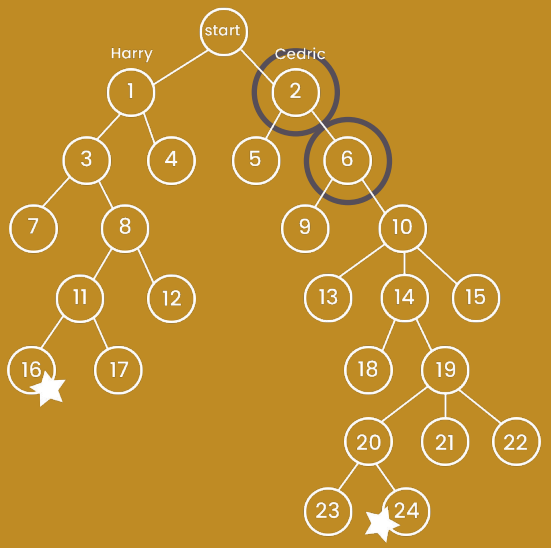
Depth First Search



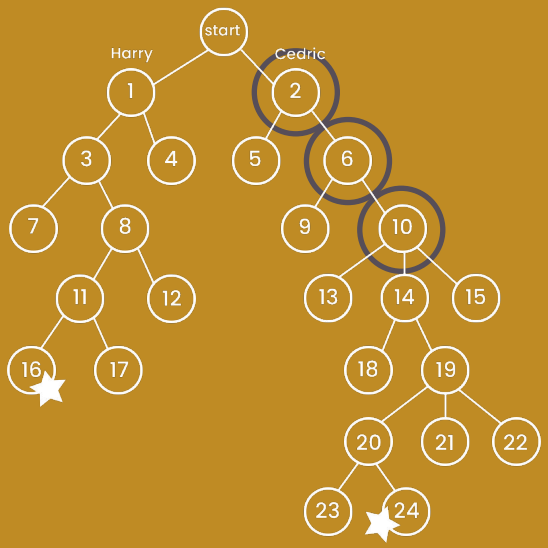
@mercedescodes | @mercedes | mercedesbernard.com

Depth first search is when you traverse the nodes as deep as possible from one side of the tree to the other. In a maze, DFS is the best way to get to the center quickly but you can see here that the villain in the book laid a trap for Harry in the center of the maze and purposefully set up the maze so that even Cedric's best route would be worse than Harry's worst route.

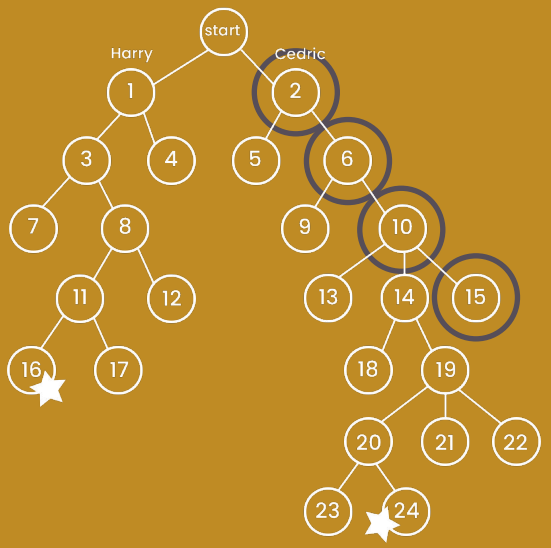
Depth First Search



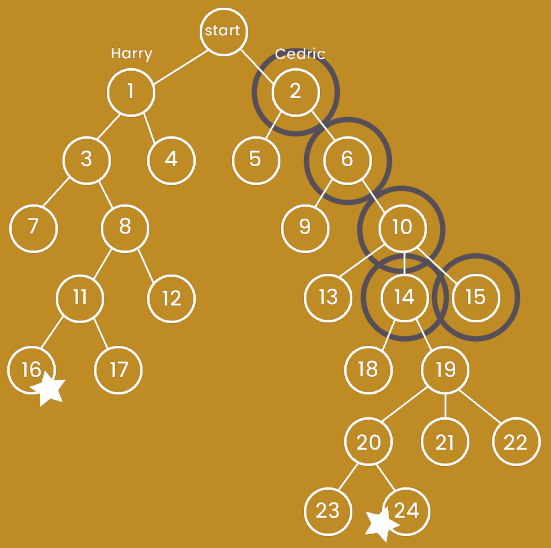
Depth First Search



Depth First Search

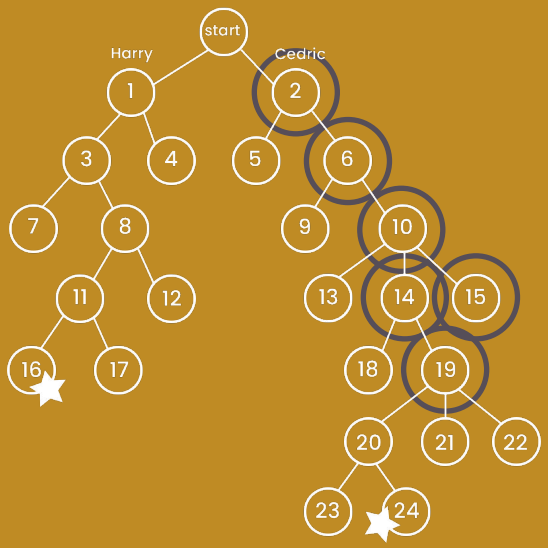


Depth First Search

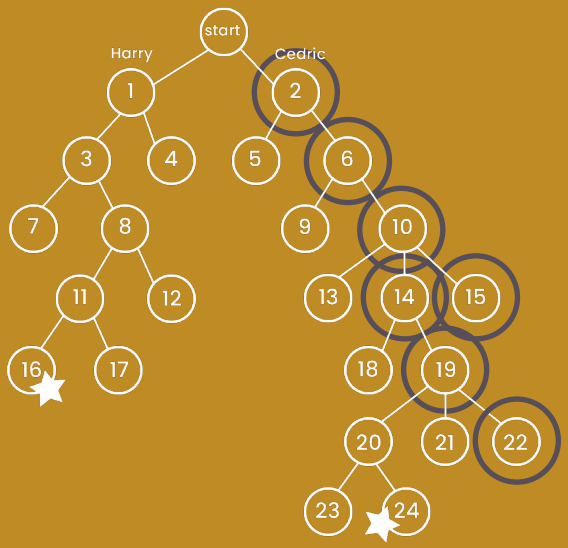


@mercedescodes | @mercedes | mercedesbernard.com

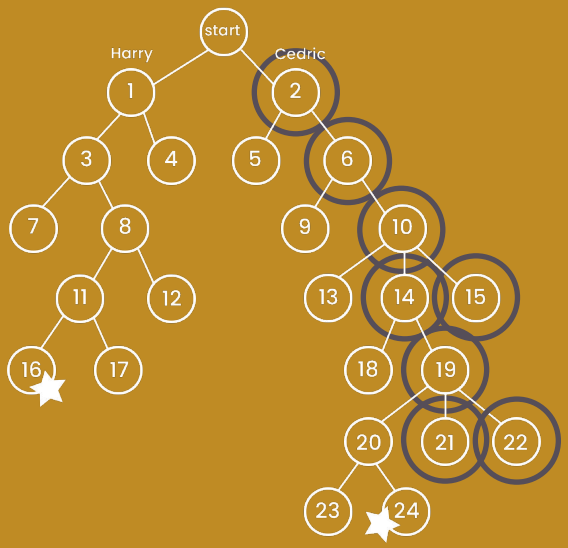
Depth First Search



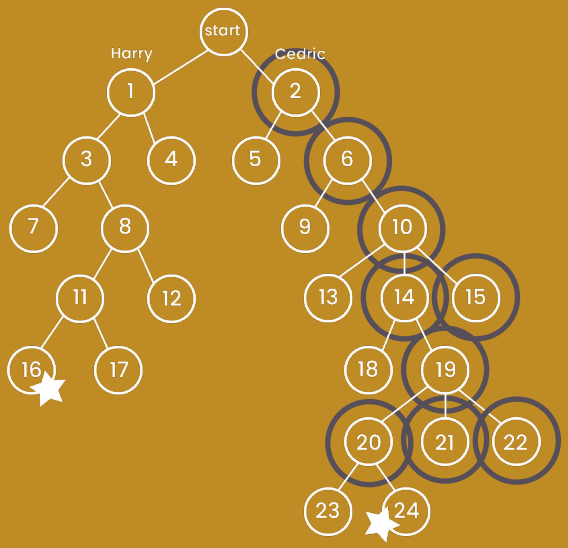
Depth First Search



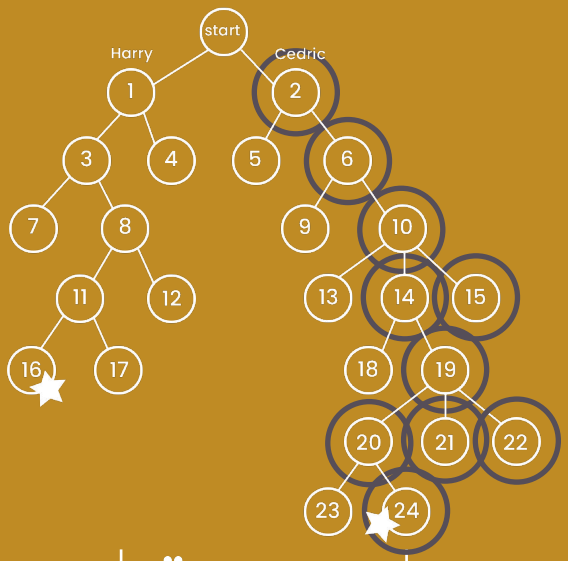
Depth First Search



Depth First Search



Depth First Search



Tree Uses

@mercedescodes | @mercedes | mercedesbernard.com

There are many many types of trees that are out of scope for this talk but the most common you'll hear about are binary trees (of which there are also many types). Binary trees are trees where the nodes have at most 2 children.

Binary trees on their own aren't super useful but there are many types of binary trees that have further constraints making them more useful. For example, a binary search tree.

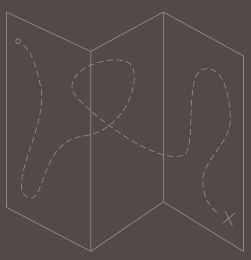
Agenda

1. Concepts

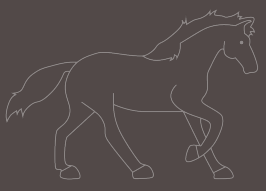
- ~~Big O notation~~
- ~~Set theory~~
- ~~Recursion~~

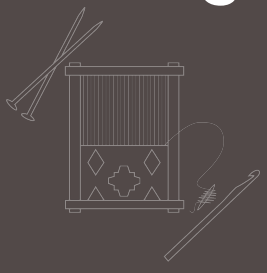
2. Data structures

- ~~Linked list~~
- ~~Stack~~
- ~~Queue~~
- ~~Tree~~

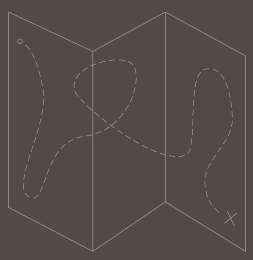



mercedesbernard.com/speaking

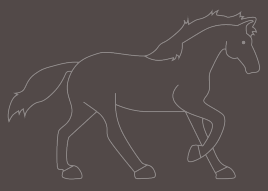


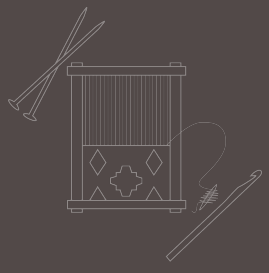



@mercedescodes | @mercedes | mercedesbernard.com

Thank you



@mercedescodes | @mercedes | mercedesbernard.com