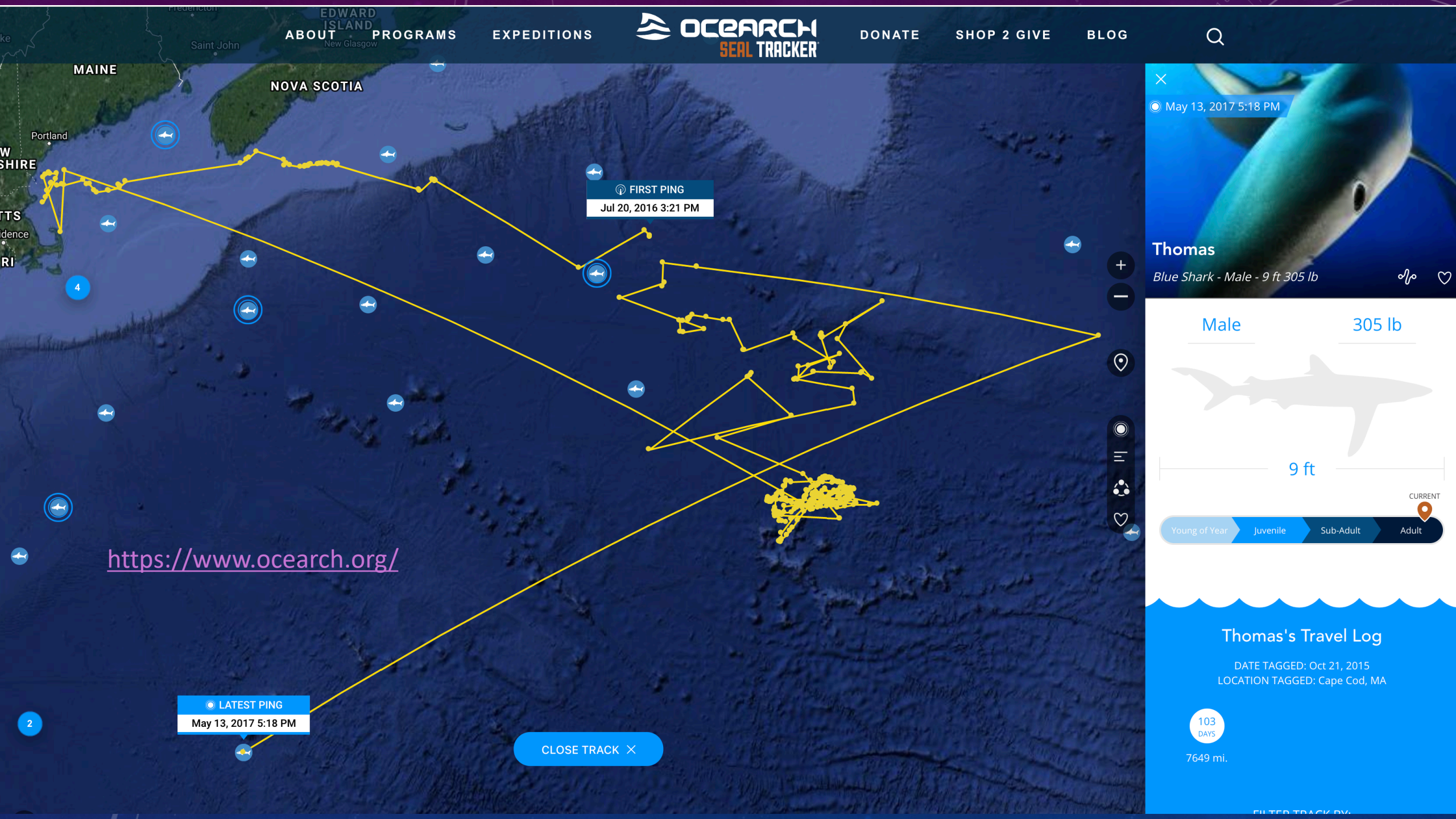
DATA STRUCTURES AND ALGORITHMS

WHERE ARE ALL THE SHARKS?!


ABOUT
PROGRAMS
EXPEDITIONS
OCEARCH
SEAL TRACKER
DONATE
SHOP 2 GIVE
BLOG
MAINE
NOVA SCOTIA
Portland
Saint John
New Glasgow
EDWARD ISLAND
FIRST PING
Jul 20, 2016 3:21 PM
LATEST PING
May 13, 2017 5:18 PM
CLOSE TRACK
https://www.ocearch.org/
May 13, 2017 5:18 PM
Thomas
Blue Shark - Male - 9 ft 305 lb
Male
305 lb
9 ft
CURRENT
Young of Year
Juvenile
Sub-Adult
Adult
Thomas's Travel Log
DATE TAGGED: Oct 21, 2015
LOCATION TAGGED: Cape Cod, MA
103
DAYS
7649 mi.

WHAT'S THE PROBLEM?

- We are continually receiving files with information about sharks
- We want to answer these questions:
  - Where was the shark at a particular time?
  - Where does the shark go most often?
  - What is the most likely way an shark gets from point A to point B?
  - If an shark is at point A, where does it usually go next?

```
1  Thomas
2  2019-06-07 11:28:22.897527,122.3320708,7.6062095
3  2019-06-08 06:28:22.897527,122.3320708,7.6062095
```

NAME
TIME,LONGITUDE,LATITUDE

WHAT'S COVERED

- Maps
- Lists
- Queues
- Stacks
- Trees
- Graphs
- Search
- Sort
- Graph search

WHY AND WHY

- Custom behavior for your application
- You can’t use certain libraries
- Understanding libraries and debugging
- You have a job interview

MAPS

Operations

- Put
- Get
- Delete

Collisions

When two values have the same key

```
>>> map = {}
>>> map['one'] = 1
>>> map['two'] = 2
>>> print(map)
{'two': 2, 'one': 1}
>>> map['two'] = 3
>>> print(map)
{'two': 3, 'one': 1}
>>> print(map['three'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

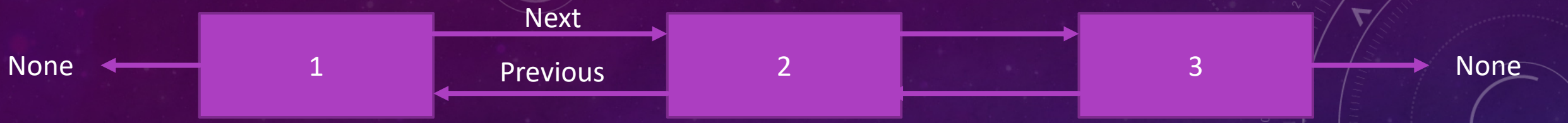
- Different add rules: A new item is added to a custom data structure following specific rules
- Different existence rules: Items that don’t exist return None instead of an exception
- Key = animal name
- Value = set of structures holding animal data

```
class AnimalTracker:
    def __init__(self):
        self.animal_map = {}

    def add_file(self, file):
        animal = file.get_subject()

        if(animal in self.animal_map):
            files = self.animal_map[animal]
            files.add_new_file(file)
        else:
            self.animal_map[animal] = FileTuple()

    def get_animals(self):
        return list(self.animal_map.keys())

    def get_animal(self, subject):
        if subject in self.animal_map:
            return self.animal_map[subject]
        else:
            return None
```

```
>>> list = []
>>> list.append(1)
>>> print(list)
[1]
>>> list.insert(1,2)
>>> print(list)
[1, 2]
>>> list.remove(1)
>>> print(list)
[2]
```

LISTS

- Ordered sequence
- **Single linked:** traverse only in one direction
- **Double linked:** traverse in two directions
- **Circular:** the last item points to the beginning
- **Operations:** add, remove, get




```
class DoubleLinkedListNode:
    def __init__(self, file, previous=None, next=None):
        self.data = file
        self.previous = previous
        self.next = next
```

The example is using lists to hold sets of data files for each animal

```
def get_at_index(self, index):
    # This is an empty list
    if self.first_node == None:
        return None

    count = 0
    current = self.first_node

    while count < index and current != None and current.next != None:
        current = current.next
        count = count + 1

    # If the list is not long enough to contain 'index', return None
    if count == index:
        return current
    else:
```

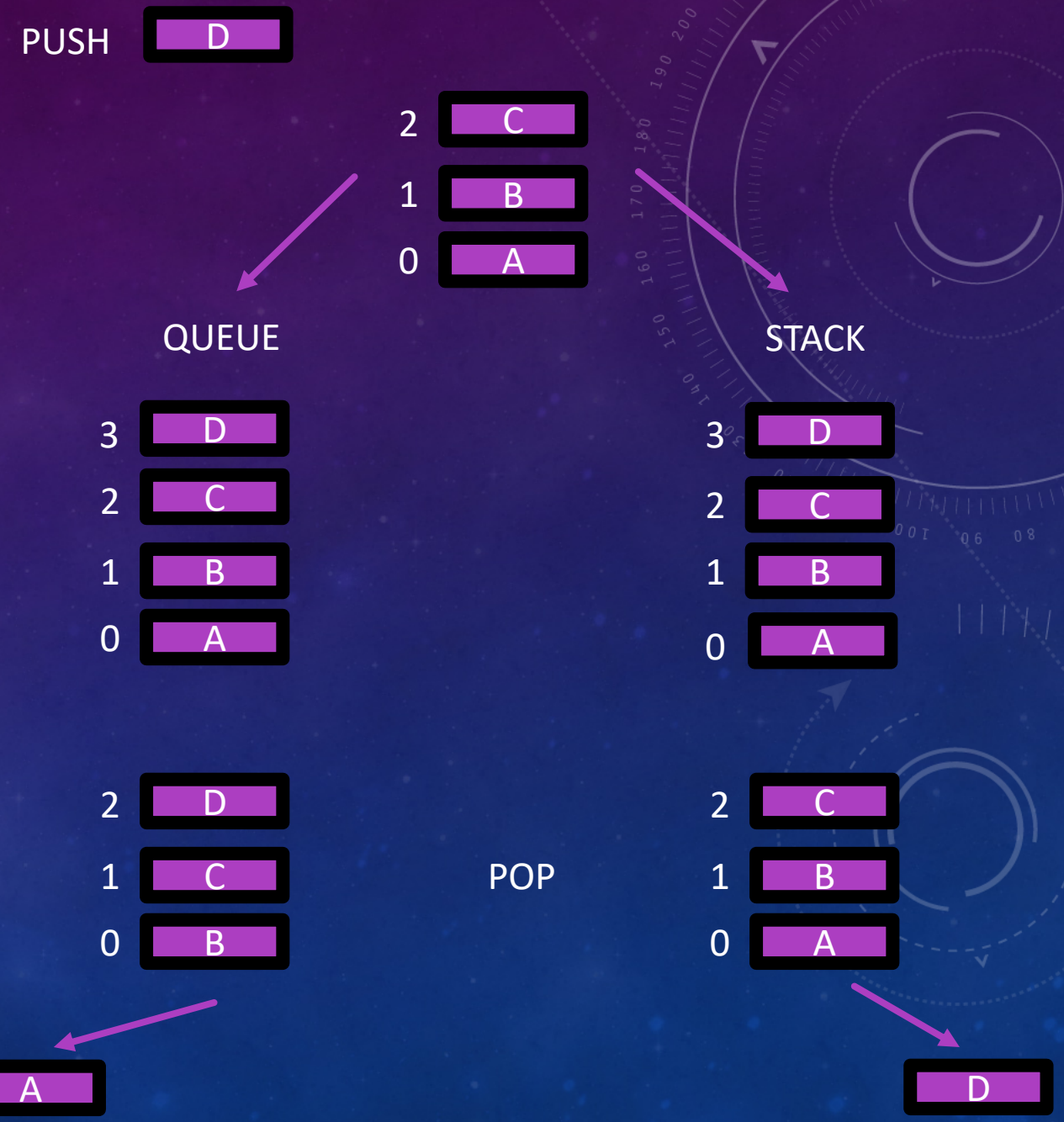
```
def insert_at_index(self, index, data):
    if index == 0:
        return self.insert_first()

    node = self.get_at_index(index)
    if node == None:
        return False

    # Create a node to take the place of the old node at get_at_index
    # It will point to node's previous but use node as the next
    new_node = DoubleLinkedListNode(data, node.previous, node)

    node.previous.next = new_node
    node.previous = new_node

    return True
```

STACKS AND QUEUES

- Lists with specific add and remove rules
- Operations:
  - push (add),
  - pop (remove),
  - peek (show the next value)

(LIFO) Stack: push always adds at index 0 and pop always removes at index 0

(FIFO) Queue: push always add to the end and pop always removes at index 0



```
>>> queue = []
>>> queue.append(1)
>>> queue.append(2)
>>> queue.append(3)
>>> print(queue)
[1, 2, 3]
>>> queue.pop(0)
1
>>> print(queue)
[2, 3]
```

```
>>> stack = []
>>> stack.append(1)
>>> stack.append(2)
>>> stack.append(3)
>>> print(stack)
[1, 2, 3]
>>> stack.pop()
3
>>> print(stack)
[1, 2]
```

STACKS AND QUEUES IN PYTHON

STACK/QUEUE

```
class Queue:
    def __init__(self):
        self.linked_list = DoubleLinkedList()

    def push(self, item):
        self.linked_list.insert_last(item)

    def pop(self):
        first = self.linked_list.get_first()
        self.linked_list.remove_first()
        return first

    def peek(self):
        return self.linked_list.get_first()

    def get_count(self):
        return self.linked_list.get_length()
```

```
class Stack:
    def __init__(self):
        self.linked_list = DoubleLinkedList()

    def push(self, item):
        self.linked_list.insert_first(item)

    def pop(self):
        first = self.linked_list.get_first()
        self.linked_list.remove_first()
        return first

    def peek(self):
        return self.linked_list.get_first()

    def get_count(self):
        return self.linked_list.get_length()
```

The example is using stacks and queues to hold new files. If we are interested in a shark's recent location data first, we use a stack. If we are interested in historical data first, we use a queue.

SORTING

- Insertion Sort: for each item, insert it into the sorted list where it fits in order
- Merge Sort: Split the list in half and sort each half before merging. For a list of size 1, it is sorted.
- Quicksort: Pick a number n from the list. Put all numbers less than n on the left and greater on the right. Quicksort the left and right then merge. For a list of size 1, it is sorted.
- For this example, we want to take each received file and sort it so that we can more easily search the data and also construct a timeline of the shark's path

```
def insertion_sort(split_lines):
    # Note: this does duplicate space needed
    sorted = [None] * len(split_lines)
    sorted[0] = split_lines[0]
    split_lines = split_lines[1:]

    sorted_length = 1

    for data in split_lines:
        timestamp = data[0]
        last_sorted = sorted[sorted_length - 1][0]
        current_sorted_index = sorted_length - 1

        while last_sorted > timestamp and current_sorted_index
            sorted[current_sorted_index + 1] = sorted[current_s
            current_sorted_index = current_sorted_index - 1
            if current_sorted_index >= 0:
                last_sorted = sorted[current_sorted_index][0]

        sorted[current_sorted_index + 1] = data
        sorted_length = sorted_length + 1

    return sorted
```

INSERTION SORT

MERGE SORT

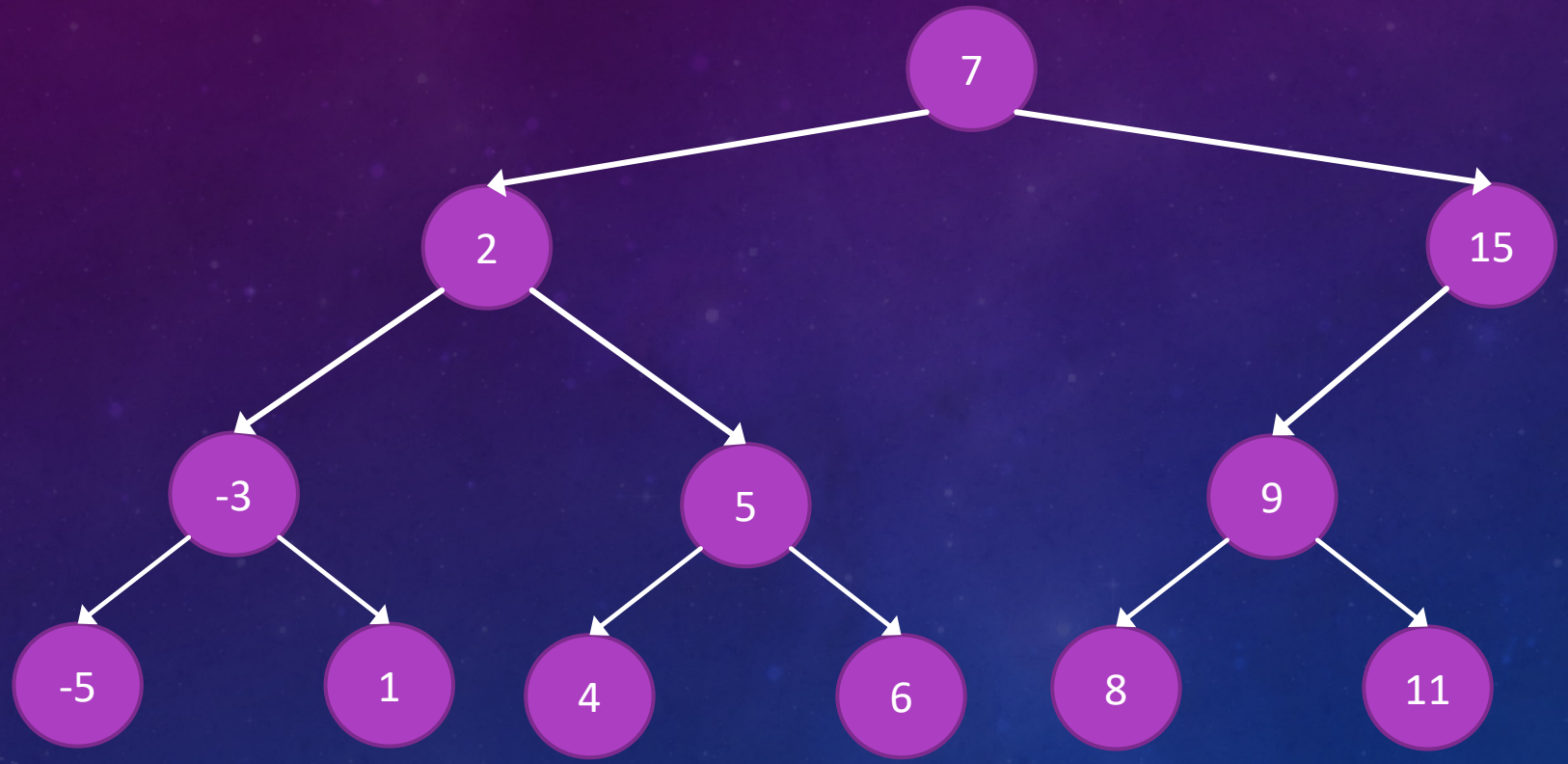
```python
def merge_sort(lines):
    length = len(lines)

    if length > 1:
        left = lines[0:int(length / 2)]
        right = lines[int(length / 2):]
        left = FileSort.merge_sort(left)
        right = FileSort.merge_sort(right)
        return FileSort.merge(left, right)
    else:
        return lines
```

```python
def merge(left, right):
    result = [None] * (len(left) + len(right))
    left_index = 0
    right_index = 0
    result_index = 0

    # you can do this with nested while instead of this way
    while left_index < len(left) and right_index < len(right) and result_index < len(result):
        if(left[left_index][0] < right[right_index][0]):
            result[result_index] = left[left_index]
            result_index = result_index + 1
            left_index = left_index + 1
        else:
            result[result_index] = right[right_index]
            result_index = result_index + 1
            right_index = right_index + 1

    if left_index < len(left): # copy the rest of the left
        while left_index < len(left):
            result[result_index] = left[left_index]
            result_index = result_index + 1
            left_index = left_index + 1

    if right_index < len(right): # copy the rest of the left
        while right_index < len(right):
            result[result_index] = right[right_index]
            result_index = result_index + 1
            right_index = right_index + 1

    return result
```

```
def quick_sort(lines):
    if len(lines) <= 1:
        return lines

    # this will split into two new lists for undestandability
    # this would be done in place normally
    pivot = lines[-1]
    less = []
    greater = []

    for i in range(0, len(lines) - 1):
        if pivot[0] < lines[i][0]:
            greater.append(lines[i])
        else:
            less.append(lines[i])

    return FileSort.quick_sort(less) + [pivot] + FileSort.quick_sort(greater)
```

QUICKSORT

[BINARY SEARCH] TREES



- Vocabulary:
  - Node
  - Edge
  - Parent
  - Child
  - Root
  - Leaf
- Trees are:
  - Acyclic
  - Directed

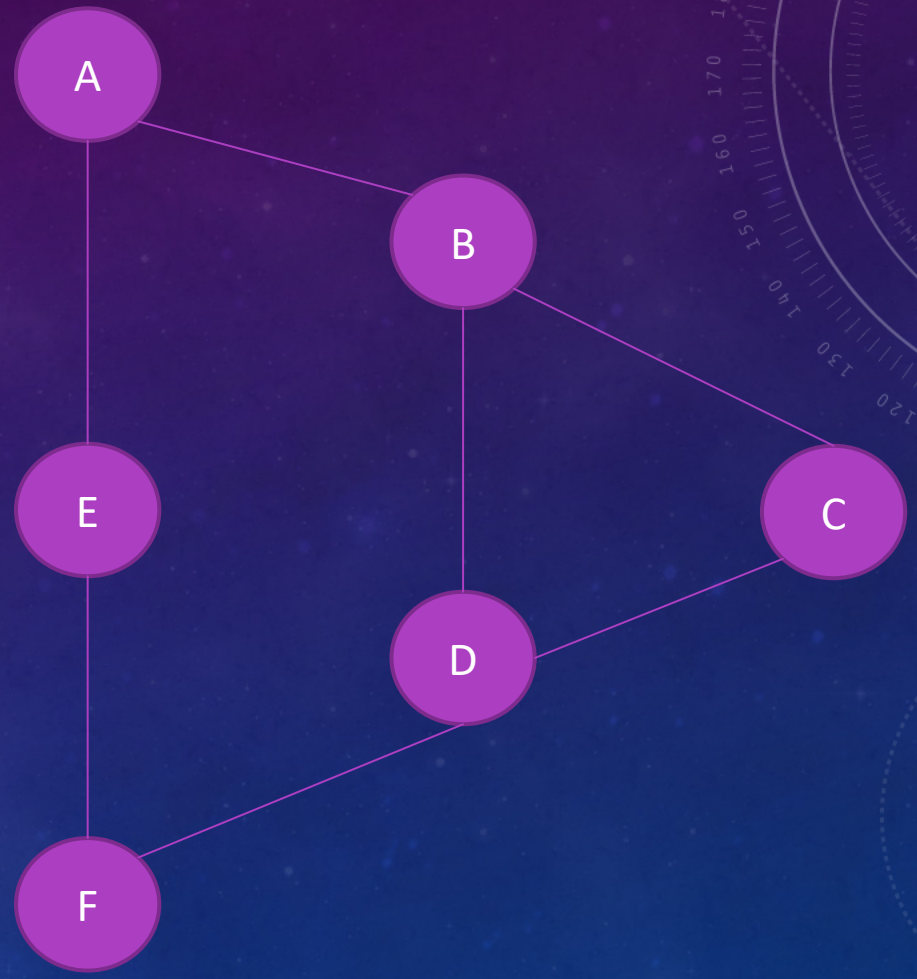
```
class BinarySearchNode:
    def __init__(self, file, parent = None, left = None, right = None):
        self.data = file
        self.parent = None
        self.left = None
        self.right = None

    def find_node_with_timestamp(self, timestamp):
        """ get the file that contains the given timestamp """

        timestamps = self.get_range_from_data()

        if timestamp < timestamps[0] and self.left != None:
            # The search is for a file on the left of this node
            return self.left.find_node_with_timestamp(timestamp)
        elif timestamp > timestamps[1] and self.right != None:
            # The search is for a file on the right of this node
            return self.right.find_node_with_timestamp(timestamp)
        elif timestamp <= timestamps[1] and timestamp >= timestamps[0]:
            return self.data
        else:
            return None
```

For this example, we put our sorted data in a binary search tree and can then get the location data for a specific timestamp

SEARCHING

GRAPHS

- Like a tree: nodes, edges
- More generic: neighbors and undirected
- Path: a set of nodes or edges defining a traversal sequence in the graph
- In this example, we use the time sorted data to create a path of where a shark travels

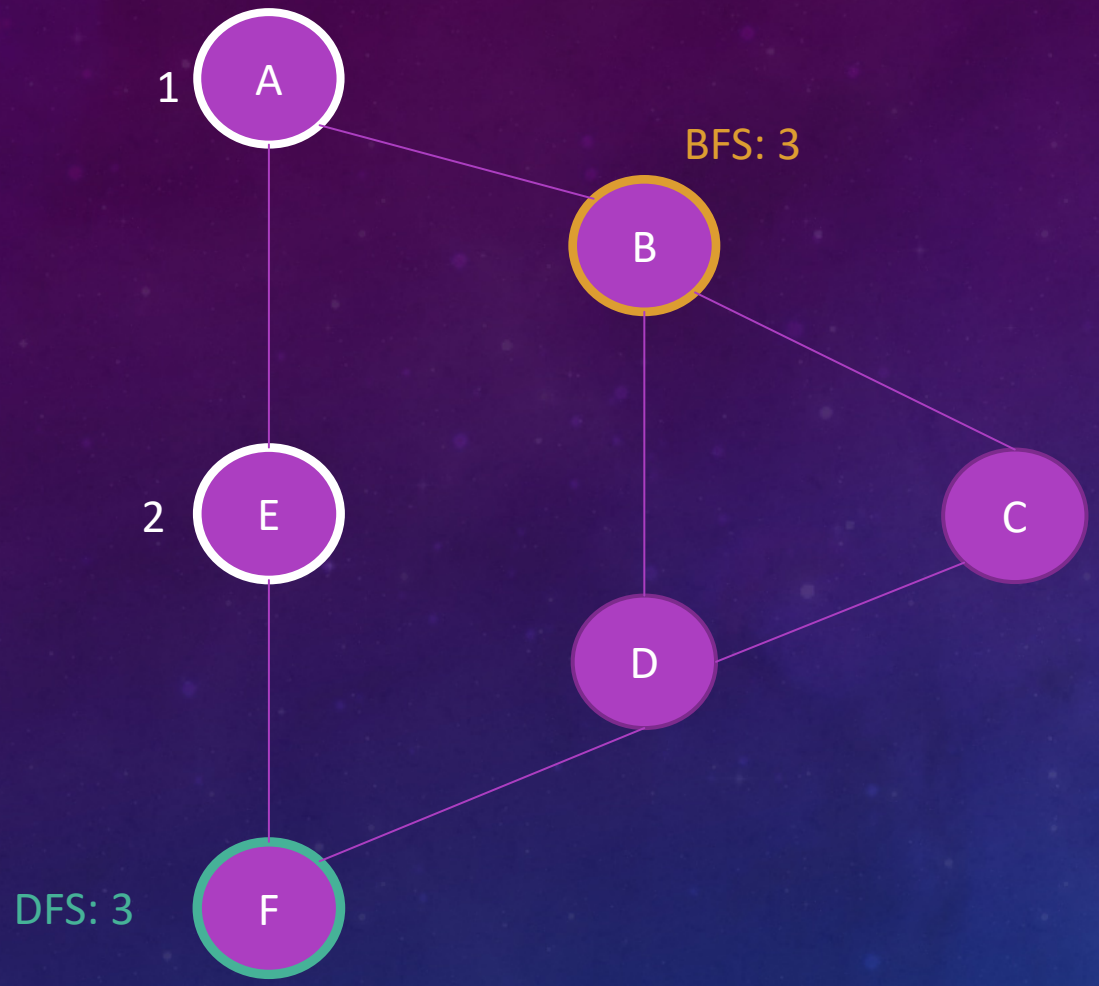


```python
class GraphNode:
    def __init__(self, data):
        self.data = data
        self.neighbors = []
        self.count = 0
        self.previous = None # just a shortcut for search
        # previous should be separate from this structure

    def add_neighbor(self, node):
        self.neighbors.append(node)

    def get_neighbors(self):
        return neighbors

    def get_data(self):
        return self.data
```

```python
class Graph:
    def __init__(self):
        self.nodes = {}
```

GRAPHS

- The graph is a map of nodes with key: coordinates and value: node object
- The node object contains information about how many times this location has been visited and where the shark was before and after being at this location
- The previous field will be used in tracing a search path



GRAPH SEARCH

- Search starts with one node and then goes to their neighbors
- Then those neighbors add their neighbors and so on
- When you find your target, you stop
- **Breadth first:** Search the "closest" nodes first
  - Will find shortest path
- **Depth first:** Search the furthest nodes first
  - Will find any path (faster)
- **Cycles:** prevent infinite loops by marking what is visited already

SEARCHING

- The difference between BFS and DFS is a queue or a stack for your neighbors

```
>>> queue = []
>>> queue.append(1)
>>> queue.append(2)
>>> queue.append(3)
>>> print(queue)
[1, 2, 3]
>>> queue.pop(0)
1
>>> print(queue)
[2, 3]
```

```
def path_from_to_dfs(self, source, to):
    # If we don't know about either of these locations, return None
    if source not in self.nodes or to not in self.nodes:
        return None

    if source.data == to.data:
        return [source]

    start = self.nodes[source]
    end = self.nodes[to]

    stack = []
    visited = []
    path = []
    stack.append(start)
    visited.append(start)

    while len(stack) > 0 and end.previous == None:
        current_node = stack.pop()

        for neighbor in current_node.neighbors:
            if neighbor == end:
                # WHen you find the target, set it's path and exit
                end.previous = current_node
                break
            if neighbor not in visited:
                neighbor.previous = current_node
                stack.append(neighbor)
                visited.append(neighbor)
```

THANK YOU

- Slack – Pyladies: Alicia B
- Github: albendz https://github.com/albendz/sideProjects/tree/master/file_manager
- LinkedIn: abendz