Jangmin Song

Professor Lawson

DCS 229

April 22, 2024

Final Portfolio

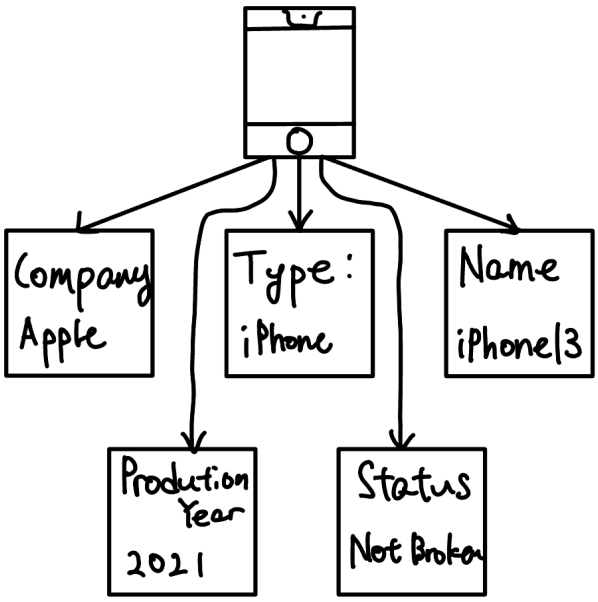
1. OOP & Classes

OOP (Object-Oriented Programming) is a programming model that is based on objects and classes. If you are using a phone as an example, there are multiple different kinds of phones with different parts such as companies, color, series, production year, camera, chips, size, etc. By defining this class, all the information is gathered that explains the phone. Easy way to think about OOP and classes is to think about the big object, and go specifics. There is no big O analysis.

Codes below show the difference between not using the class vs using the class.

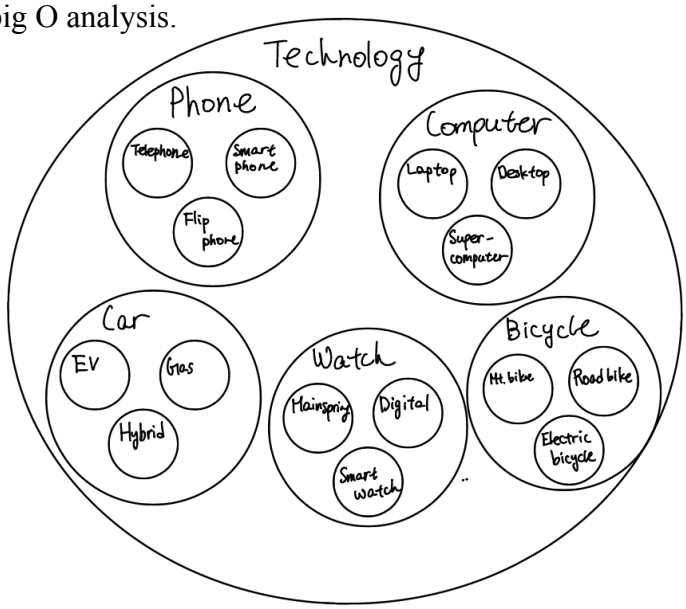
```python
class Phone:
    def __init__(self, company, type, name, production_year, broken):
        self.company = company
        self.type = type
        self.name = name
        self.production_year = production_year
        self.broken = broken
    def getCompany(self) -> str:
        return self.company
    def getType(self) -> str:
        return self.type
    def getName(self) -> str:
        return self.name
    def getProductionYr(self) -> str:
        return self.production_year
    def Broken(self) -> str:
        if self.broken:
            return 'Broken'
        else:
            return 'Not broken'
    def __str__(self):
```

```
        return f"Phone informaiton with class: \n Company: {self.getCompany()}
\n Type: {self.getType()} \n Name: {self.getType()} \n Production Year:
{self.getProductionYr()} \n {self.Broken()}"
def main():
    company = 'Apple'
    type = 'iPhone'
    name = 'iPhone13'
    production_year = '2021'
    print(f'Phone informaiton without class: \n Company: {company} \n Type:
{type} \n Name: {name} \n Production Year: {production_year} \n Not broken \n')
    phone1 = Phone('Apple', 'iPhone', 'iPhone13', '2021', False)
    print(phone1)
main()
################################################################################
Output:
Phone informaiton without class:
 Company: Apple
 Type: iPhone
 Name: iPhone13
 Production Year: 2021
 Not broken
Phone informaiton with class:
 Company: Apple
 Type: iPhone
 Name: iPhone
 Production Year: 2021
 Not broken
```

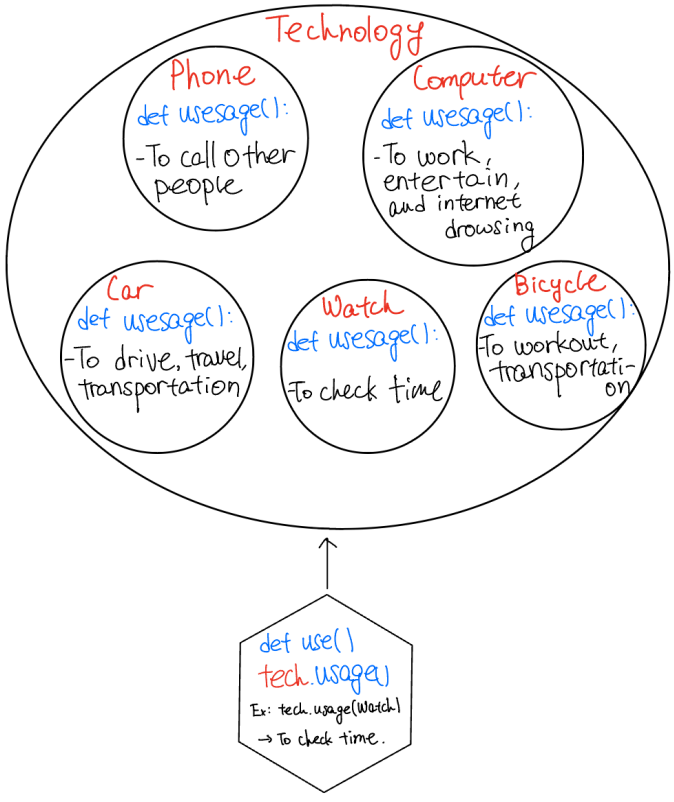


2. Inheritance & Polymorphism

Inheritance goes more deeply into a class, and from the bottom, it goes upwards. For example, the iPhone is a phone, and a phone is a technology. In other words, the phone is a subclass of Technology, and the iPhone is a subclass of the phone. This is only one example. There are other things that can be branched out from technology, such as laptops, cars, Artificial Intelligence, and etc. There is no big O analysis.



Polymorphism is similar to Inheritance, however, objects of different classes can be treated as instances of a common superclass and it can respond to the same method call in different ways. For example, by making `def usage(any_class)->None: return any_class.usesage()`, the usage of each technologies can be written under each technologies, rather than writing everything in `usage()` function. There is no Big O Analysis.



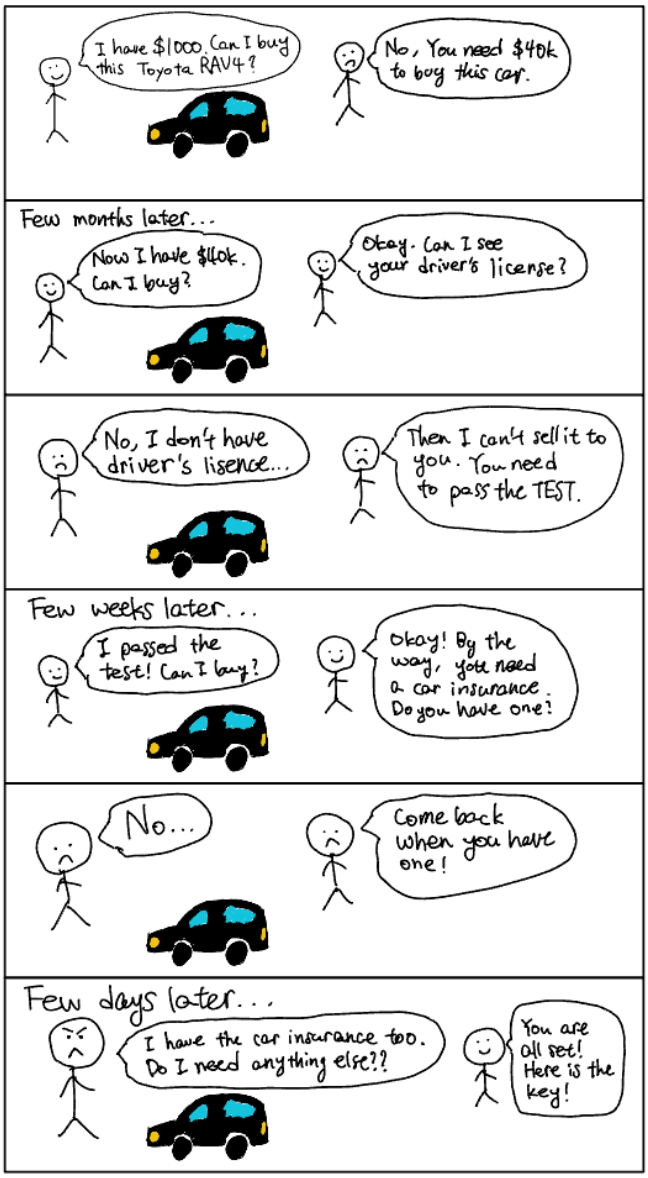
Code:

```python
class Technology:
    def __init__(self, name):
        self.name = name
    def usage(self):
        raise ValueError("Subclasses must implement this method")
class Phone(Technology):
    def __init__(self, name):
        self.name = name
    def usage(self):
        return f"{self.name} is used for communication"
class Laptop(Technology):
    def __init__(self, name):
        self.name = name
    def usage(self):
        return f"{self.name} is used for work, entertainment, and browsing"
class Car(Technology):
    def __init__(self, name):
        self.name = name
    def usage(self):
        return f"{self.name} is used for transportation"
# Polymorphism
def describe_usage(tech):
    return tech.usage()
def main():
    iPhone = Phone("iPhone", "13")
    macbook = Laptop("MacBook")
    Toyota = Car("Toyota")
    #Inheritance
    print(iPhone.usage())
    print(macbook.usage())
    print(Toyota.usage())
    #Polymorphism
    print(describe_usage(iPhone))
    print(describe_usage(macbook))
    print(describe_usage(Toyota))
main()
################################################################################
Output:
#Inheritance
iPhone is used for communication
MacBook is used for work, entertainment, and browsing
Toyota is used for transportation
```

```
#Polymorphism
iPhone is used for communication
MacBook is used for work, entertainment, and browsing
Toyota is used for transportation
```

3. TDD & Pytest

TDD (Test Driven Development) is a way to test the code. By writing the test, and writing the code that passes the test, we can move on to developing the software. In TDD, 3 big steps are taken. First is to write the test. The next step is to write the code that does not fail the test. Lastly, clean up the code but still pass the test, and repeat the steps until the project is completed. Pytest makes the tests easier to write in Python. There is no Big O analysis.



Codes that Pytest fails:

```
Python
def is_even(num):
    return num % 2 == 0
def main():
    assert is_even(2) == True
    assert is_even(3) == True
    assert is_even(0) == True
main()
################################################################################
Output:
=================================== ERRORS ====================================
_________________ ERROR collecting pytest_final_portfolio.py _________________
pytest_final_portfolio.py:9: in <module>
    main()
pytest_final_portfolio.py:6: in main
    assert is_even(3) == True
E   assert False == True
E    +  where False = is_even(3)
=========================== short test summary info ============================
ERROR pytest_final_portfolio.py - assert False == True
!!!!!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!!!!!
=============================== 1 error in 0.04s ===============================
```

Codes that Pytest pass:

```
Python
def is_even(num):
    return num % 2 == 0
def main():
    assert is_even(2) == True
    assert is_even(3) == False
    assert is_even(0) == True
main()
################################################################################
Output:
============================= test session starts ==============================
platform darwin -- Python 3.11.5, pytest-8.0.0, pluggy-1.4.0
rootdir: /Users/songjangmin/Desktop/Academics 23-24/Data Structures & Algorithms
collected 0 items
============================ no tests ran in 0.01s =============================
songjangmin@bobcat-111-154 Data Structures & Algorithms %
```

4. Fixed-Size Arrays & Dynamic Sizing

Fixed-size array is when the number of arrays is fixed when the array is made. Dynamic sizing is when elements can be added or removed to the array. For example, the list of numbers is the fixed-size array. However, as you append or remove, the array becomes a dynamic array. However, when appending the data, there are different ways of increasing the capacity, and the efficiencies are significantly different.

Big O Analysis:

Methods:	Fixed-Size:	Dynamic Size:
Finding an element by the value	O(n)	O(n)
Finding an element by the index	O(1)	O(1)
Appending an element	O(1)	O(1+growth) If the array reaches the max cap, then 1 + size of old array
Removing an element	O(n)	O(n)
Finding the size	O(1)	O(1)

Codes:

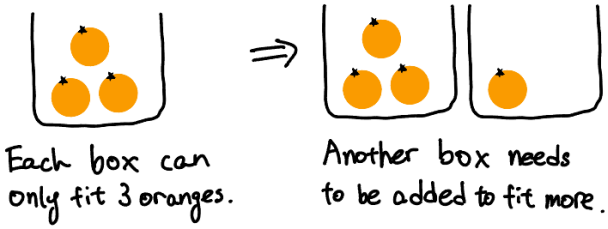
```python
class MyList:
    __slots__ = ('_n', '_capacity', '_array')
    # class-level variables for stats
    _resizes: int = 0  # total number of resizes
    _copies:  int = 0  # number of array-to-array copies during resizing
    @classmethod
    def resetStats(cls) -> None:
        MyList._resizes = 0
        MyList._copies = 0
    @classmethod
    def getStatsDict(cls) -> dict:
        return {"resizes" : MyList._resizes, \
                "copies"  : MyList._copies}
    def __init__(self):
        self._n        = 0  # number of actual elements currently in the list
        self._capacity = 1  # default MyList capacity
        self._array    = self._make_array(self._capacity)
    def _make_array(self, capacity: int) -> 'ctypes array':
```

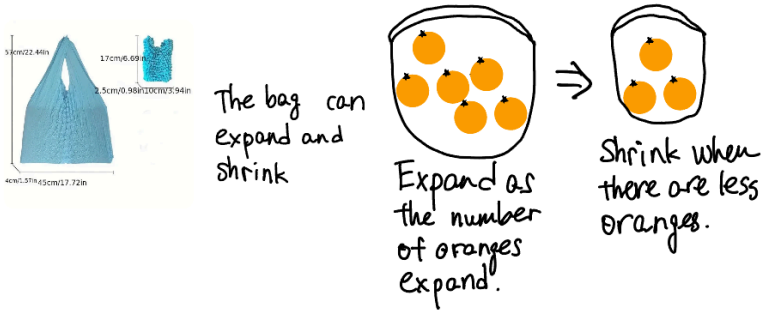
```
        ''' private method to reserve space for a low-level array of the
            given capacity
        Parameters:
            capacity: integer size of the array
        Returns:
            a ctypes low-level array
        '''
        ArrayType = (capacity * c_int)  # defined in ctypes
        return ArrayType() # create and return an array of py_object of size
capacity
    def append(self, item: T) -> None:
        ''' appends the given item to the array, increasing the capacity of
            the array as necessary
        Parameters:
            item: type-T element to append to the array
        Return:
            None
        Raises:
            TypeError exception if type of item does not match types in array
        '''
        if isinstance(item, type(self._array[0])):
            if self._n == self._capacity:
                self._resize(self._capacity * 2)
                # self._resize(round(self._capacity * 1.5))
                # self._resize(math.ceil(self._capacity * 1.25))
                # self._resize(self._capacity + 1024)
                # self._resize(self._capacity + 4096)
                # self._resize(self._capacity + 16384)
            self._array[self._n] = item
            self._n += 1
        else:
            raise TypeError(f'type of item does not match types in array')
###############################################################################
Result of these methods (not the output of these codes):
#self._resize(self._capacity * 2)
average entry: 599902.43
average resizes: 300.47
 average copies: 12281430.90
   average time: 0.31sec
# self._resize(round(self._capacity * 1.5))
average entry: 599902.43
average resizes: 503.93
 average copies: 24663886.30
   average time: 0.36sec
```

```
# self._resize(math.ceil(self._capacity * 1.25))
average entry: 599902.43
average resizes: 856.80
 average copies: 43997591.90
   average time: 0.43sec
# self._resize(self._capacity + 1024)
average entry: 599902.43
average resizes: 856.80
 average copies: 43997591.90
   average time: 0.44sec

# self._resize(self._capacity + 4096)
average entry: 599902.43
average resizes: 856.80
 average copies: 43997591.90
   average time: 0.43sec
# self._resize(self._capacity + 16384)
average entry: 599902.43
average resizes: 609.57
 average copies: 228336771.70
   average time: 1.10sec
```

By running this `append()` function with different ways of adding the capacities in `MyList` class and time how long each method takes to add significantly big numbers, we can find out the most efficient way. The most efficient way among these methods was the capacity that doubles.
Here is the example with drawing:



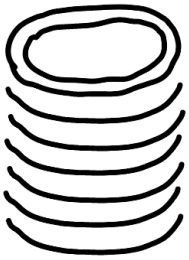


5. Stack ADT

Stack ADT is one way of adding and removing the element from the list. Just like a stack of objects, an element has to be added (pushed) to the top of the list, and the element that is in the top of the list can be deleted (popped).

Big O Analysis:

Methods:	Big O:
push()	O(1)
pop()	O(1)
top()	O(1)
is_empty()	O(1)
__len__()	O(1)



I'm sorry about the bad drawing, but here is the stack of the plate. When trying to add another plate, you have to add it to the top (push). If you want to remove the plate, you have to remove it from the top (pop). If you want to remove the plate that is in the middle, you have to remove the plates above (pop) until you get to the plate, and put the plates removed back on again (push).

In this code, I am using a list, however, think about it as a stack. The element in front of the list is the top of the stack, and the last element of the list is the bottom of the stack.

```python
def push(stack: list, item) -> list:
    return stack.insert(0,item)
def pop(stack) -> list:
    if len(stack) == 0:
        raise 'stack is empty'
    else:
        return stack.remove(stack[0])
def main():
    stack = []
    push(stack,1)
    push(stack,2)
    push(stack,3)
    push(stack,4)
```

```
    push(stack,5)
    print(stack)
    pop(stack)
    print(stack)
main()
################################################################################
Output:
#push
[1]
[2, 1]
[3, 2, 1]
[4, 3, 2, 1]
[5, 4, 3, 2, 1]
#pop
[4, 3, 2, 1]
[3, 2, 1]
```

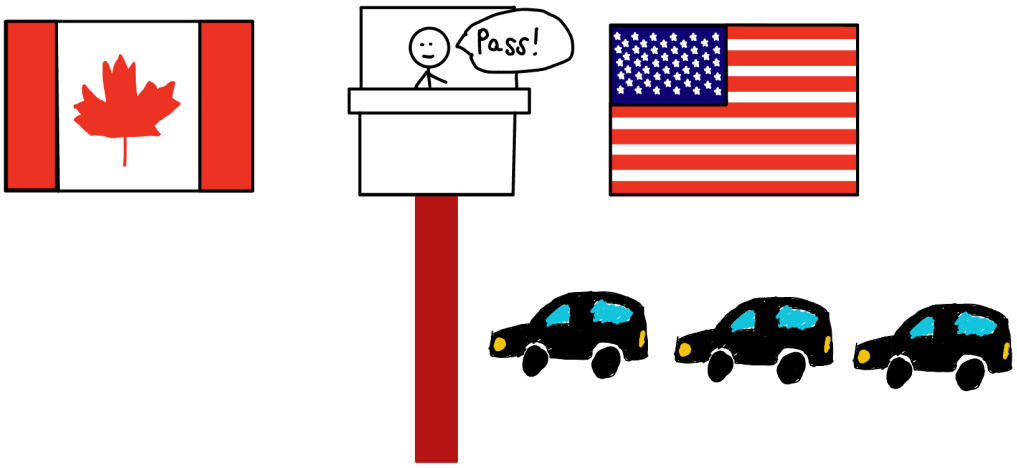
6. Queue ADT

Queue ADT is another way of adding and removing the element from the list. Concepts are a little similar to Stack ADT, however, instead of stacking up, elements get queued behind when pushing. The element that is in front of the queue gets removed when popping.

Big O Analysis with Doubly linked list:

Methods:	Big O:
push()	O(1)
pop()	O(1)
top()	O(1)
is_empty()	O(1)
__len__()	O(1)

Here is the drawing that can be used as an example of a queue. It is the border line between the US and Canada. The last car arrives gets queued to the line, and the first one in the line gets popped from the queue and enters Canada.



In this code, I am using a list, however, think about it as a queue. The element in front of the list is the front of the queue, and the last element of the list is the back of the queue.

```python
def push(queue: list, item) -> list:
    return queue.append(item)
def pop(queue) -> list:
    if len(queue) == 0:
        raise 'queue is empty'
    else:
        return queue.remove(queue[0])
def main():
    queue = []
    push(queue,1)
    push(queue,2)
    push(queue,3)
    push(queue,4)
    push(queue,5)
    print(queue)
    pop(queue)
    print(queue)
main()
################################################################################
Output:
#push
[1]
[1, 2]
[1, 2, 3]
[1, 2, 3, 4]
[1, 2, 3, 4, 5]
#pop
[2, 3, 4, 5]
[3, 4, 5]
```

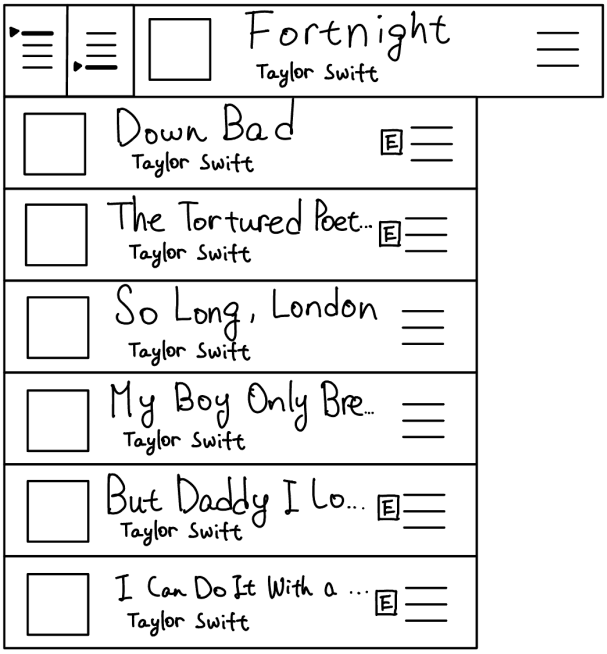
7. Deque ADT

Deque ADT works just like Queue ADT, however, in Deque, elements can be pushed and popped from both sides of the list. An element can be pushed from the left or the right, and the alement can be popped from the left or the right.

Big O Analysis with Doubly linked list:

Methods:	Big O:
push_right()	O(1)
push_left()	O(1)
pop_right()	O(1)
pop_left()	O(1)
top_right()	O(1)
top_left()	O(1)
is_empty()	O(1)
__len__()	O(1)

For an example of Deque ADT, we can use the Apple Music Playlist (This is the top 100: Global Playlist. I am not a Swifty). We can queue (push) the songs to the top and the bottom of the song list. For popping, we can only play (pop) top and the bottom of the playlist.



Code:

```python
def push_right(deque: list, item) -> list:
    return deque.append(item)
def push_left(deque: list, item) -> list:
    return deque.insert(0, item)
def pop_left(deque) -> list:
    if len(deque) == 0:
        raise 'deque is empty'
    else:
        return deque.remove(deque[0])
def pop_right(deque) -> list:
    if len(deque) == 0:
        raise 'deque is empty'
    else:
        return deque.remove(deque[len(deque)-1])
def main():
    deque = []
    push_right(deque, 1)
    print(deque)
    push_right(deque,2)
    print(deque)
    push_left(deque,3)
    print(deque)
    push_left(deque,4)
    print(deque)
    pop_right(deque)
    print(deque)
    pop_left(deque)
    print(deque)
    pop_right(deque)
    print(deque)
main()
################################################################################
Output:
[1]
[1, 2]
[3, 1, 2]
[4, 3, 1, 2]
[4, 3, 1]
[3, 1]
[3]
```

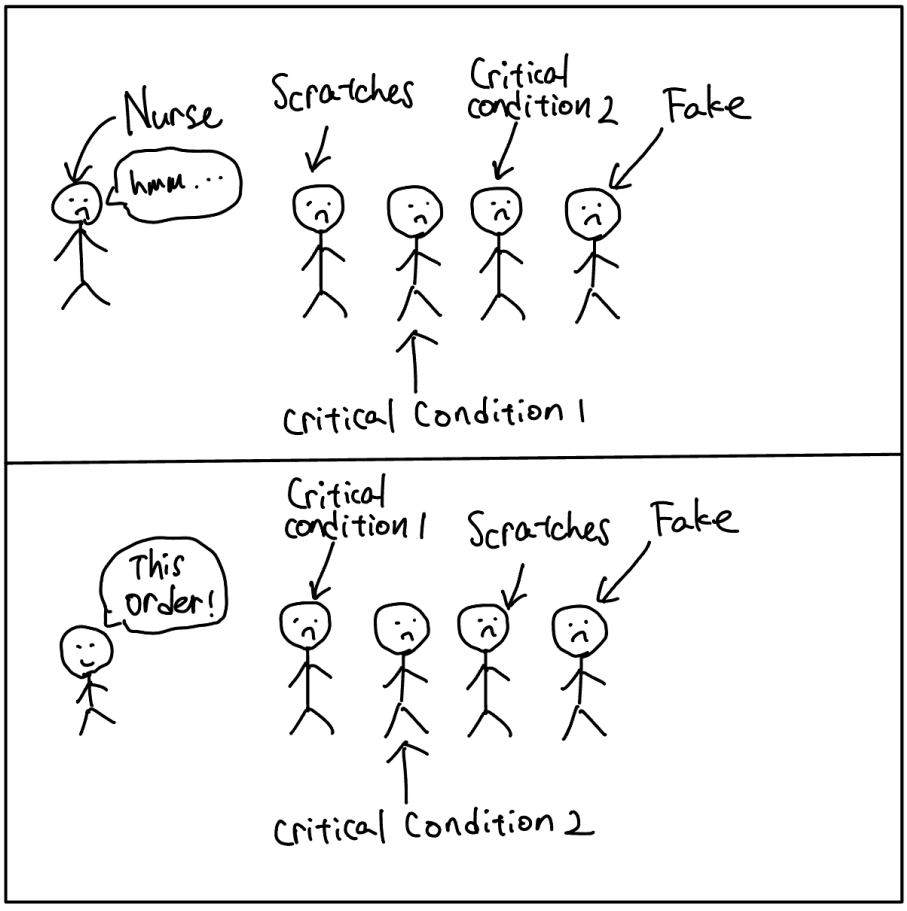
8. Priority Queue ADT

Priority Queue ADT works similarly to Queue ADT except that prioritized elements get popped first. An element can be pushed in any order.

Big O Analysis with Heap:

Methods:	Big O:
insert()	O(log(n))
remove_min()	O(log(n))
min()	O(1)
is_empty()	O(1)
__len__()	O(1)

Think about an emergency hospital. There is already a person who got scratches on her knee waiting in line for the treatment, then a patient who is in critical condition gets queued. A person who is in a critical condition needs to get the treatment (pop) first. A label of priority decides which element to get pushed first.



Here is the code that we used in our lab:

```python
class Entry[K,V]:
    __slots__ = ('key', 'value')
    def __init__(self, priority: K, data: V) -> None:
        self.key  : K = priority
        self.value: V = data
class PriorityQueue[E]:
    __slots__ = ('_container')
    def __init__(self):
        self._container: list[Entry] = list()
    def insert(self, key: K, item: V) -> None:
        '''
        inserts the key item pair into the array
        Parameters:
            key: the int that determines the priority in the arrary
            item: the item being inserted
        Returns:
            None
        '''
        entry = Entry(key, item)
        heapq.heappush(self._container, entry)
    def remove_min(self) -> Entry:
        '''
        removes the minimum key from the array
        Parameters:
            None
        Returns:
            None
        '''
        if self.is_empty():
            raise EmptyError("Priority queue is empty")
        return heapq.heappop(self._container)
def main():
    pq = PriorityQueue()
    # Insert test
    print("insert test:")
    pq.insert(4, 'D')
    print(f"Expected: [(4,'D')]")
    print(f'  Result: {pq}')
    pq.insert(7, 'G')
    print(f"Expected: [(4,'D'), (7,'G')]")
    print(f'  Result: {pq}')
    pq.insert(2, 'B')
    print(f"Expected: [(2,'B'), (7,'G'), (4,'D')]")
```

```
    print(f'  Result: {pq}')
    pq.insert(4, 'P')
    print(f"Expected: [(2,'B'), (4,'P'), (4,'D'), (7,'G')]")
    print(f'  Result: {pq}')
    pq.insert(9, 'A')
    print(f"Expected: [(2,'B'), (4,'P'), (4,'D'), (7,'G'), (9,'A')]")
    print(f'  Result: {pq}')
    print()
    # Remove min test
    print("removal of min test:")
    pq.remove_min()
    print(f"Expected: [(4,'D'), (4,'P'), (9,'A'), (7,'G')]")
    print(f'  Result: {pq}')
    pq.remove_min()
    print(f"Expected: [(4,'D'), (4,'P'), (7,'G')]")
    print(f'  Result: {pq}')
    pq.remove_min()
    print(f"Expected: [(4,'D'), (7,'G')]")
    print(f'  Result: {pq}')
    pq.remove_min()
    print(f"Expected: [(4,'D')]")
    print(f'  Result: {pq}')
    print()
main()
################################################################################
Output:
Expected: [(4,'D')]
  Result: [(4,'D')]
Expected: [(4,'D'), (7,'G')]
  Result: [(4,'D'), (7,'G')]
Expected: [(2,'B'), (7,'G'), (4,'D')]
  Result: [(2,'B'), (7,'G'), (4,'D')]
Expected: [(2,'B'), (4,'P'), (4,'D'), (7,'G')]
  Result: [(2,'B'), (4,'P'), (4,'D'), (7,'G')]
Expected: [(2,'B'), (4,'P'), (4,'D'), (7,'G'), (9,'A')]
  Result: [(2,'B'), (4,'P'), (4,'D'), (7,'G'), (9,'A')]
removal of min test:
Expected: [(4,'D'), (4,'P'), (9,'A'), (7,'G')]
  Result: [(4,'D'), (4,'P'), (9,'A'), (7,'G')]
Expected: [(4,'D'), (4,'P'), (7,'G')]
  Result: [(4,'P'), (7,'G'), (9,'A')]
Expected: [(4,'D'), (7,'G')]
  Result: [(7,'G'), (9,'A')]
Expected: [(4,'D')]
```

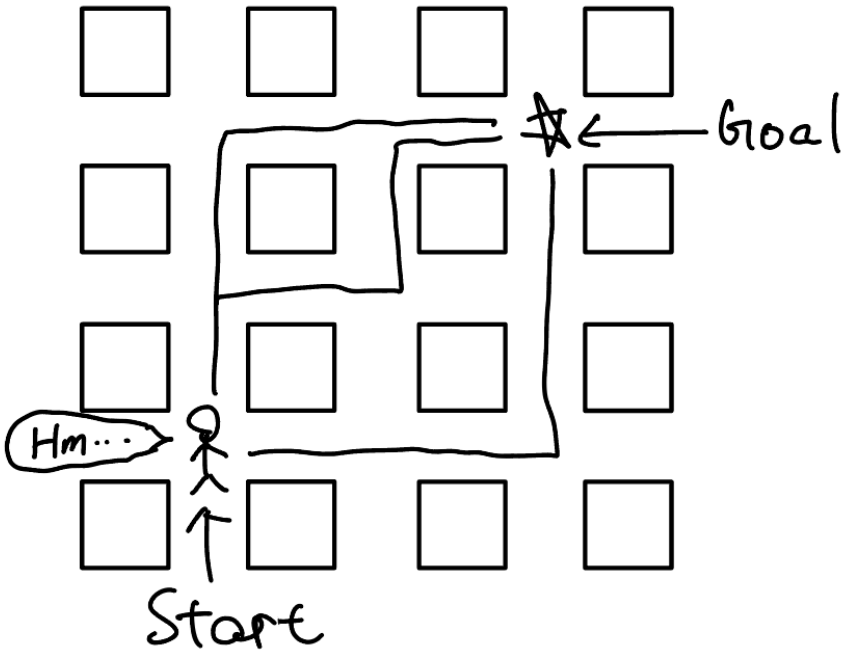
```
Result: [(9,'A')]
```

9. BFS vs DFS vs A*

BFS (Breadth-first search), DFS (Depth-first Search), and A* are the ways of finding the ways from a certain place to the other place. Depth-first search uses stack as their ways to push and pop the possible ways. Breadth first search uses a queue to push and pop. A* uses priority queue as their way to push and pop. By finding out the parent of each cell that was popped, and going backwards from the goal to the start, we can find the way.

Big O Analysis adjacency list ()
BFS: O(n+m) where n is the number of vertices and m is the number of edges.
DFS: O(n+m) where n is the number of vertices and m is the number of edges.
A*: Uses heuristic method, thus it does not explore all of the grid, thus it is generally faster than BFS and DFS.

Imagine a city with a square simple grid-like layout. You start at one intersection and want to reach another intersection.
Here is how each method works.
BFS explores all the grid and finds the best possible way using queue ADT.
DFS explores all the grid and finds the way to the goal using stack ADT. DFS doesn't find the optimal or shortest path to the goal, rather, it explores paths deeply before backtracking.
A* explores limited grids because the prioritized ones (the least costly grids) get popped first.



Here is the code that we used in one of the labs (only the DFS, BFS, A* parts are shown):

Python

```
def dfs(self) -> Cell | None:
        ''' implements depth-first search (using a Stack)
        Returns:
            a Cell object corresponding to the goal, if a valid path exists
            from the start to the goal;  or None, if no path exists
        '''
        stack = Stack()
        stack.push(self._start)
        self._num_cells_pushed += 1
        while not stack.is_empty():
            current_cell = stack.pop()
            if current_cell == self._goal:
                return current_cell
            neighbors = self.getSearchLocations(current_cell)
            for i in neighbors:
                if str(i.getParent()) == 'None':
                    stack.push(i)
                    i.setParent(current_cell)
                    self._num_cells_pushed += 1
        return None
    def bfs(self) -> Cell | None:
        ''' implements breath-first search (using a queue)
        Returns:
            a Cell object corresponding to the goal, if a valid path exists
            from the start to the goal;  or None, if no path exists
        '''
        queue = Queue()
        queue.push(self._start)
        self._num_cells_pushed += 1
        while not queue.is_empty():
            current_cell = queue.pop()
            if current_cell == self._goal:
                return current_cell
            neighbors = self.getSearchLocations(current_cell)
            for i in neighbors:
                if str(i.getParent()) == 'None':
                    queue.push(i)
                    i.setParent(current_cell)
                    self._num_cells_pushed += 1
        return None
        def a_star(self) -> Cell | None:
        to_explore = PriorityQueue()
        explored = dict()
```

```
        n = self._start
        g_n = 0.0
        h_n = abs((self._goal.getPosition()[0] - self._start.getPosition()[0])
+ (self._goal.getPosition()[1] - self._start.getPosition()[1]))
        f_n = g_n + h_n
        to_explore.insert(f_n, n)
        self._num_cells_pushed += 1
        explored[n.getPosition()] = g_n
        while not to_explore.is_empty():
            e = to_explore.remove_min()
            n = e.value
            if n == self._goal:
                return n
            for m in self.getSearchLocations(n):
                updated_m_cost = f_n + 1
                if m.getPosition() not in explored or updated_m_cost <
explored[m.getPosition()]:
                    g_m = updated_m_cost
                    explored[m.getPosition()] = g_m
                    h_m = abs((self._goal.getPosition()[0] -
m.getPosition()[0]) + (self._goal.getPosition()[1] - m.getPosition()[1]))
                    f_m = g_m + h_m
                    to_explore.insert(f_m, m)
                    self._num_cells_pushed += 1
                    m.setParent(n)
                    explored[m.getPosition()] = updated_m_cost
        return None
################################################################################
Output:
5x5:
| | | | |◆|
| |▒| |▒| |
| | | | |▒|
| | |▒| | |
| | | | | |
|◎|▒| |▒|▒|
dfs:
|★|★|★|★|◆|
|★|▒| |▒| |
|★|★|★|★|▒|
| | |▒|★| |
|★|★|★|★| |
|◎|▒| |▒|▒|
Path length: 16
Number of cells pushed: 22
```

```
bfs:
|★|★|★|★|◆|
|★|▒| |▒| |
|★| | | |▒|
|★| |▒| | |
|★| | | | |
|◎|▒| |▒|▒|
Path length: 10
Number of cells pushed: 22
A*:
| | |★|★|◆|
| |▒|★|▒| |
| |★|★| |▒|
|★|★|▒| | |
|★| | | | |
|◎|▒| |▒|▒|
Path length: 10
Number of cells pushed: 18
```

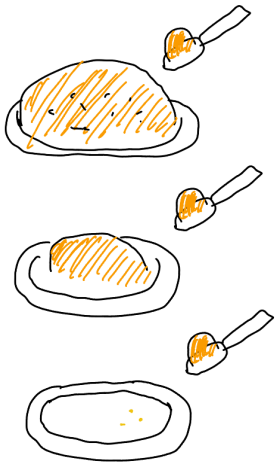
10. Recursion & Merge Sort

Recursion is a function that calls itself in its definition. It's often used to solve problems that can be broken down into smaller, similar problems.

Merge Sort is the most efficient way to sort the order of the list. They divide the list into pairs of two elements, and repairs the order and brings back into the pair. It is an example of recursion.

Big O Analysis for Merge Sort: O(n*log(n))

Example would be eating fried rice. At first, since there are plenty on the plate, each scoop is big, however, as the fried rice gets less, each scoop gets smaller, and it repeats until the end.



Here is the code with the example that I used above about fried rice:

```python
def eat_fried_rice(amount):
    if amount <= 0:
        print("No fried rice left!")
        return
    else:
        print("Eating a scoop of fried rice")
        print(f'Amount left: {amount}')
        eat_fried_rice(amount - 1)
def main():
    initial_amount = 10
    eat_fried_rice(initial_amount)
main()
################################################################################
Output:
Eating a scoop of fried rice
Amount left: 10
Eating a scoop of fried rice
Amount left: 9
Eating a scoop of fried rice
Amount left: 8
Eating a scoop of fried rice
Amount left: 7
Eating a scoop of fried rice
Amount left: 6
Eating a scoop of fried rice
Amount left: 5
Eating a scoop of fried rice
Amount left: 4
Eating a scoop of fried rice
Amount left: 3
Eating a scoop of fried rice
Amount left: 2
Eating a scoop of fried rice
Amount left: 1
No fried rice left!
```