Data Structures

Data Structures

Objectives:

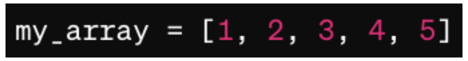
- understand the inner workings of common data structures
- be able to write out the data structure when needed
- recognise which data structure can be used in a problem

Why we learn data structures?

- Programming revolves around data structures. They are a foundational pillar of building software.
- Improve your skills – with programming in general – the language you use and how languages work on abstract level.

What are Arrays?

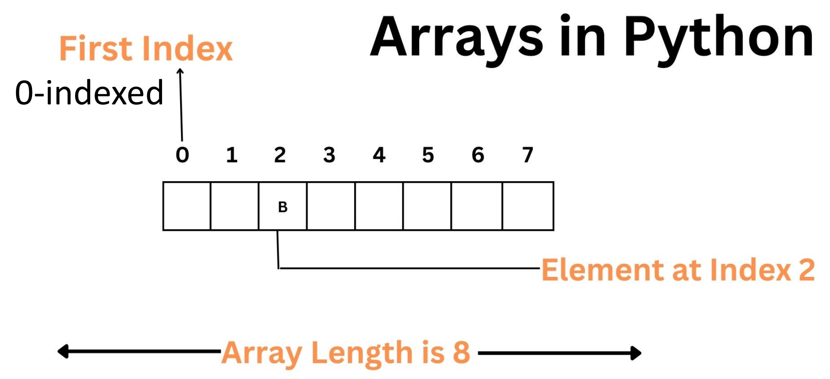
What are Arrays? Arrays are fundamental data structures in programming, used to store multiple values in a single variable. For example:



```
my_array = [1, 2, 3, 4, 5]
```

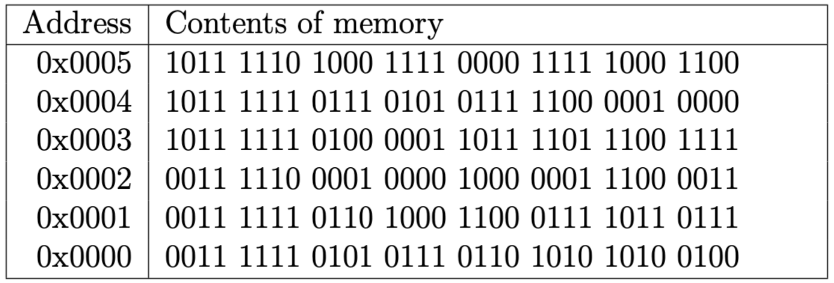
Using Arrays: The datatype of lists can be any. Whether the elements of the array have to be the same type however depends on the language. Most commonly though the use-case for arrays is working with the same datatypes, such as an array of numbers or strings.

Real-world example (1):
- An engineer at Apple working on image processing algorithms for iPhone cameras might use arrays over files to represent pixel data. Storing images this way makes it easier and faster to perform operations.



Real-world example (2):
- A front-end developer at Google working on Gmail might use arrays to manage email threads or conversations. They might even store it as a list of JSON objects with info like author, timestamp, message, receiver…

Arrays in Memory



Address	Contents of memory
0x0005	1011 1110 1000 1111 0000 1111 1000 1100
0x0004	1011 1111 0111 0101 0111 1100 0001 0000
0x0003	1011 1111 0100 0001 1011 1101 1100 1111
0x0002	0011 1110 0001 0000 1000 0001 1100 0011
0x0001	0011 1111 0110 1000 1100 0111 1011 0111
0x0000	0011 1111 0101 0111 0110 1010 1010 0100

Types of Memory:

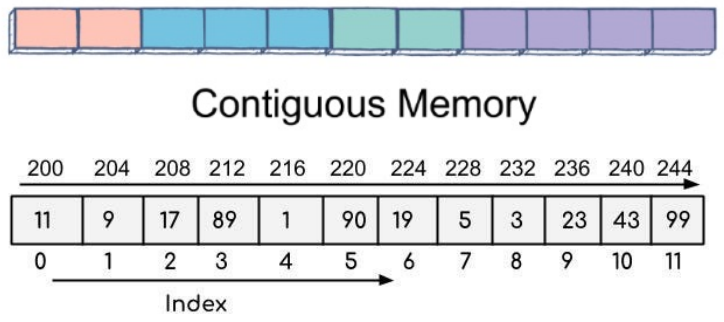
- **Primary Memory:** Stores data being actively used. It includes Random Access Memory (RAM), which is volatile (loses its contents when power is turned off), and cache memory, which is faster but smaller in size.
- **Secondary Memory (Storage)**: This type of memory includes devices like hard drives, solid-state drives (SSDs), and optical drives. Secondary memory is non-volatile and used for long-term storage of data and programs.

How does Memory work in computers? All data in computers is represented using binary digits (bits), which are either 0 or 1.
8 bits grouped together = 1 byte. Bytes are the fundamental units of storage and manipulation in computers.

Accessing Memory: Each byte in memory has a unique address (hexadecimal), allowing the CPU to access and manipulate the data.

Memory management unit (MMU): translates virtual addresses (used by programs) to physical addresses (used by the hardware). The CPU communicates with the MMU to access memory locations based on virtual addresses, and the MMU handles the translation to physical addresses.

Storing Arrays in Memory: stored in contiguous memory, meaning its elements are stored one after another in memory. The memory address of the array is typically the address of its first element.

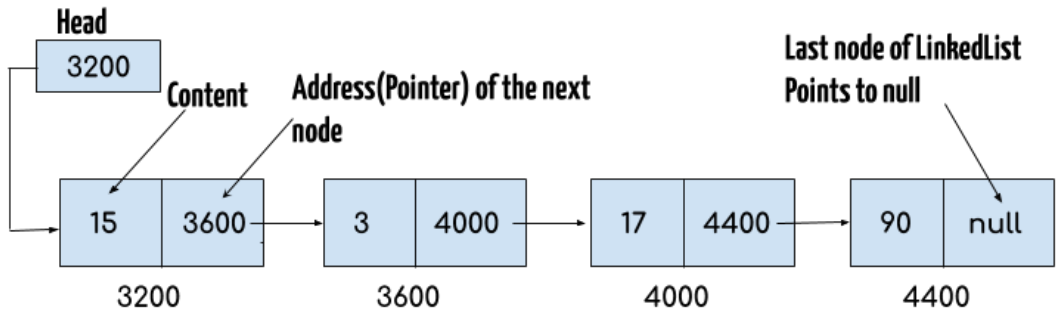


What are Linked Lists?

Head: The entry point into a linked list. The head is not a separate node, but the reference to the first node.

What are Linked Lists? Data structure used to store a sequence of elements. Unlike arrays, which store elements contiguously in memory, a linked list consists of nodes (a block of memory), where each node contains the data as well as a reference (or link) to the next node in the sequence (which is not necessarily contiguous).

Benefits over arrays: Typically to add to arrays which do not have contiguous space, the computer (specifically the MMU) will move the array to a whole new memory address.
A linked list is not contiguous and so it allows for efficient insertion and removal of elements from any position in the list. A linked list only requires updating the links, rather than shifting elements as in an array.


Head
3200
Content
Address(Pointer) of the next node
Last node of LinkedList Points to null
15
3600
3
4000
17
4400
90
null
3200
3600
4000
4400

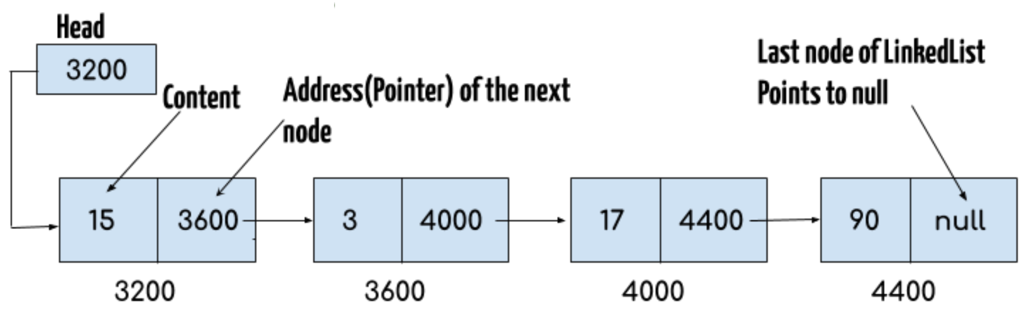
Real-world example (1): An engineer at Spotify might use a linked list to build a playlist, where each node is a song.

Real-world example (2): Users of WhatsApp are able to delete messages, so a linked list could be used to store chat history. Updating the linked list with new/deleted messages and maintaining the chronological order.

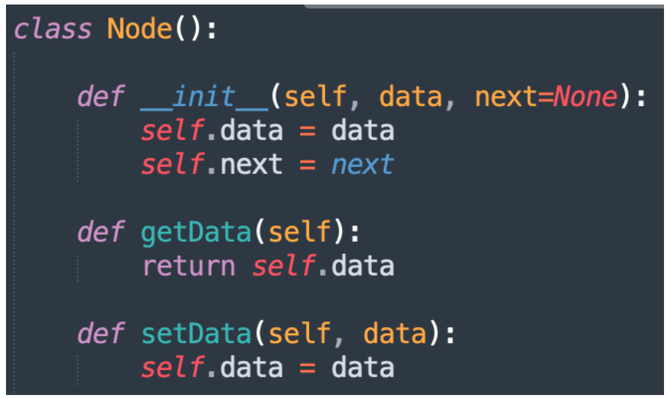
Linked Lists in Code

Basic linked list operations you should know:

- Insert element at end of list
- Insert element at head of list
- Insert element at middle of list
- Delete given element
- Delete the first element
- Return given element

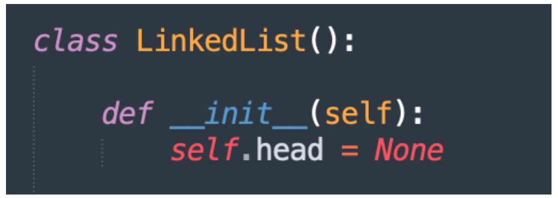

Head
3200
Content
Address(Pointer) of the next node
Last node of LinkedList Points to null
15
3600
3
4000
17
4400
90
null
3200
3600
4000
4400

Node:

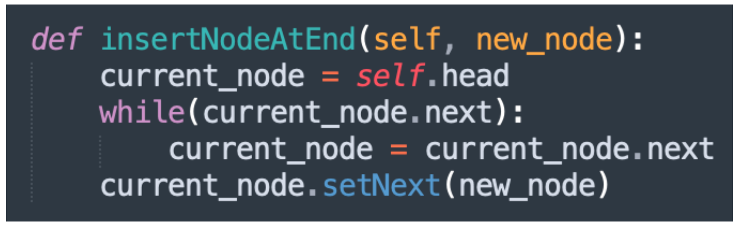


```
class Node():

    def __init__(self, data, next=None):
        self.data = data
        self.next = next

    def getData(self):
        return self.data

    def setData(self, data):
        self.data = data
```

Linked List:



```
class LinkedList():

    def __init__(self):
        self.head = None
```

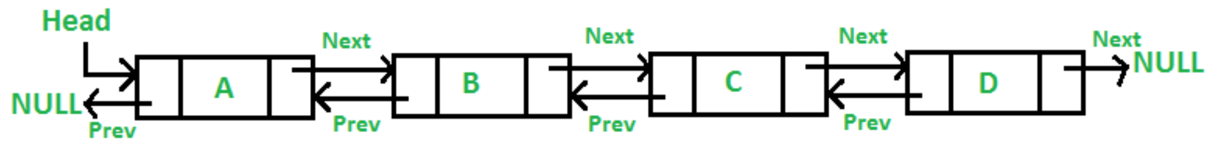


```
def insertNodeAtEnd(self, new_node):
    current_node = self.head
    while(current_node.next):
        current_node = current_node.next
    current_node.setNext(new_node)
```

What are Doubly Linked Lists?

What are Doubly Linked Lists? A doubly linked lists is simply a linked list where each node points to the previous node, as well as the next node, while storing some data.

Benefits over linked lists: The ability for bi-directional traversal results in more efficient operations e.g. deletion of a node O(1) vs O(N).



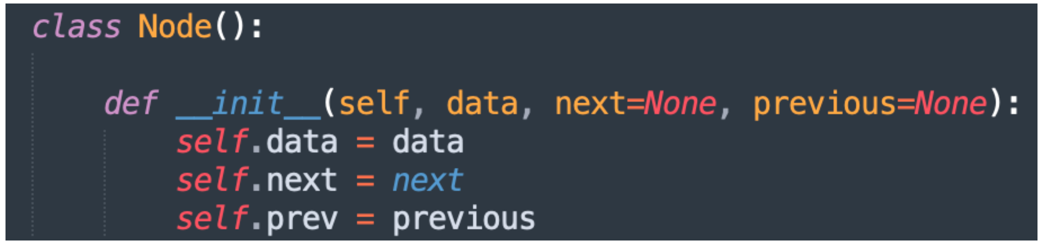
Real-world example (1): Browser history management. A web browser may use a doubly linked list to easily go back and forth through the history.

Real-world example (2): Operating systems might use a doubly linked list to manage file system directories. Each node in the list represents a file or directory, with pointers to both the previous and next files/directories

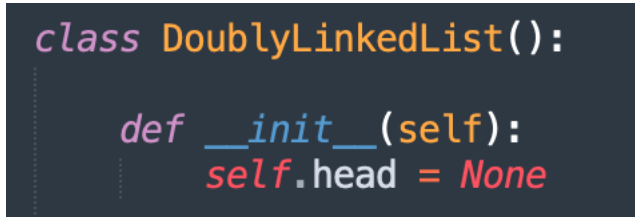
Doubly Linked Lists in Code

Basic doubly linked list operations you should know:

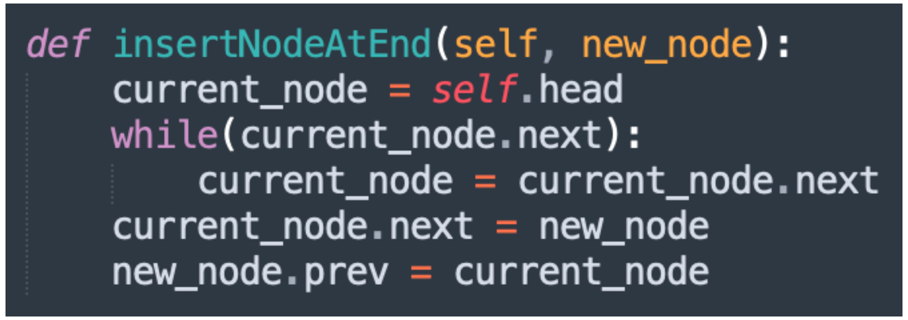
- Insert element at end of list
- Insert element at head of list
- Insert element at middle of list
- Delete given element
- Delete the first element
- Return given element



```python
class Node():

    def __init__(self, data, next=None, previous=None):
        self.data = data
        self.next = next
        self.prev = previous
```

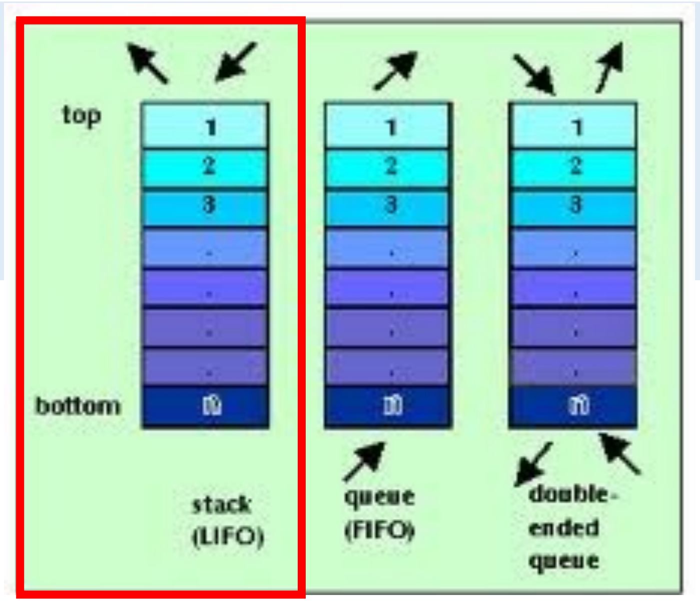


```python
class DoublyLinkedList():

    def __init__(self):
        self.head = None
```



```python
def insertNodeAtEnd(self, new_node):
    current_node = self.head
    while(current_node.next):
        current_node = current_node.next
    current_node.next = new_node
    new_node.prev = current_node
```

What are Stacks?



What are Stacks?

- A stack is data structure that uses a last-in, first-out (LIFO) principle. The last element added will be the first element removed, like a stack of plates.

Most basic stack operations:

- **Push:** add an element to the top of the stack
- **Pop:** you remove an element off the top of the stack.
- **Peek:** return the item from the top of the stack without removing it.

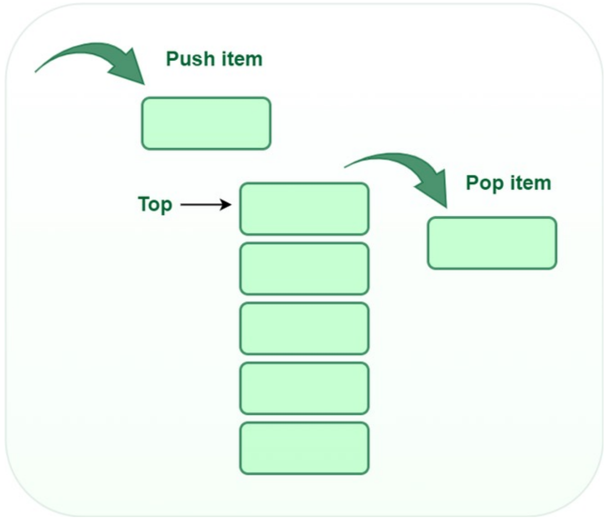
Real-world example (1): Browser history management. A web browser may use a stack to easily go back and forth through the history. Important to note – we also mentioned a doubly linked list for this problem. A problem can have multiple solutions, with different data structures used. In this case a stack may be easier to implement but we lose reference to the next page when we pop.

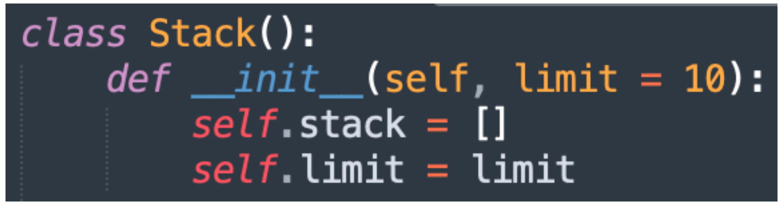
Real-world example (2): Programming language **c**ompilers to parse syntax. For example keeping track and scope of a function or code by validating opening/closing brackets.

Stacks in Code

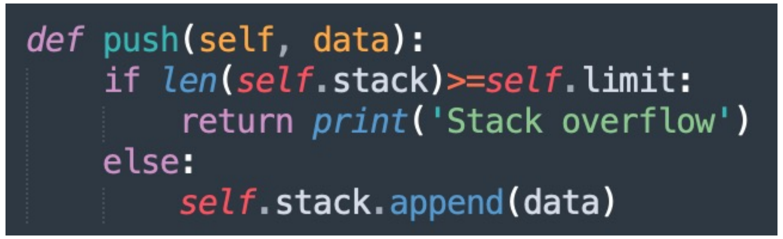
Basic stack operations you should know:

- Push
- Pop
- Peek
- Check is empty

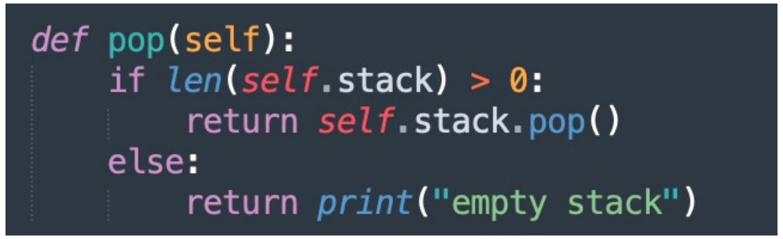




```python
class Stack():
    def __init__(self, limit = 10):
        self.stack = []
        self.limit = limit
```

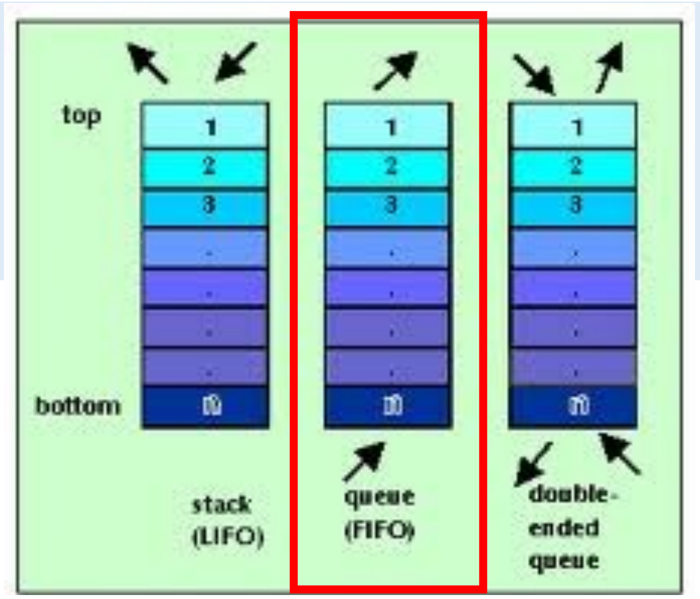


```python
def push(self, data):
    if len(self.stack)>=self.limit:
        return print('Stack overflow')
    else:
        self.stack.append(data)
```



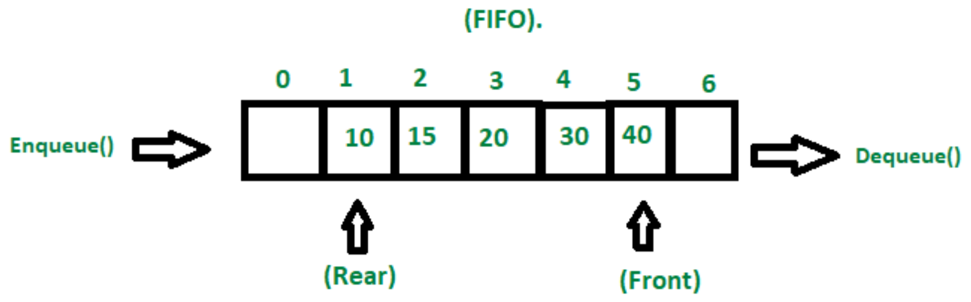
```python
def pop(self):
    if len(self.stack) > 0:
        return self.stack.pop()
    else:
        return print("empty stack")
```

What are Queues?



What are Queues?

- A stack is data structure that contains elements and uses a first-in, first-out (FIFO) principle. The first element added to the queue will be the first element removed, like a queue at the cinema.



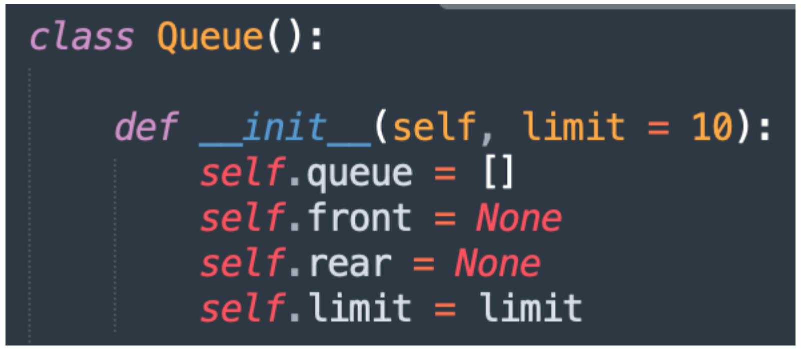
Real-world example (1): An AWS engineer working on web servers might use a queue to handle requests, allowing requests to be processed fairly in-order, without overloading the server.

Real-world example (2): Task processing in NodeJS. When async operations are triggered they are added to the event queue. The event loop checks this queue and processes the events one-by-one in a FIFO order.

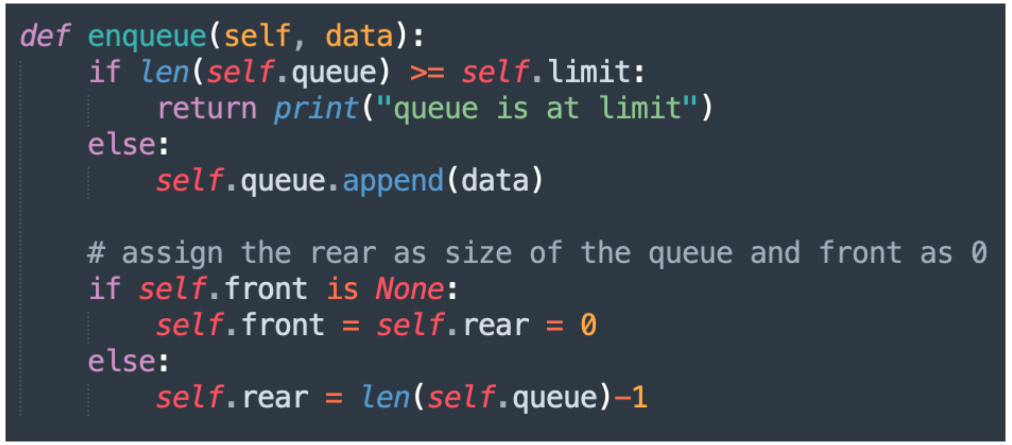
Queues in Code

Basic queue operations you should know:

- Enqueue( ): insert element at the end of the queue
- Dequeue( ): remove an element from the front of the queue
- Check if the queue is empty
- Return the first element from the queue without removing it
- Linked list as a queue



```python
class Queue():

    def __init__(self, limit = 10):
        self.queue = []
        self.front = None
        self.rear = None
        self.limit = limit
```



```python
def enqueue(self, data):
    if len(self.queue) >= self.limit:
        return print("queue is at limit")
    else:
        self.queue.append(data)

    # assign the rear as size of the queue and front as 0
    if self.front is None:
        self.front = self.rear = 0
    else:
        self.rear = len(self.queue)-1
```

What is a Double-ended queue (Deque)?

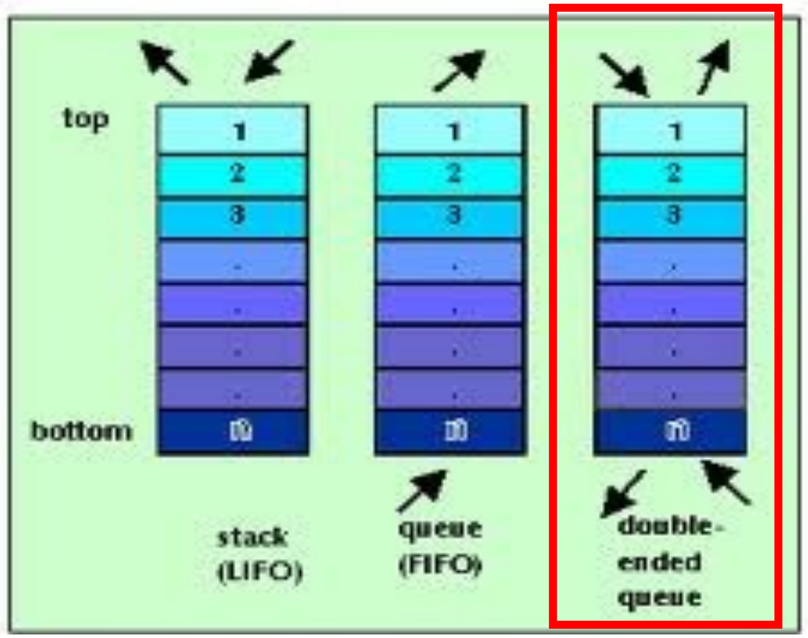
What are Deques?

- A data structure that contains elements and allows for insertion and deletion from both front and back of the structure.

Ways to implement a priority queue:

- Using a list of tuples

Real-world example (1): Maintaining a cache. For example a cache of recently visited websites on a CDN. New sites visited have their content added to the front of the deque. And the website not visited in a while, at the back of the deque is removed.



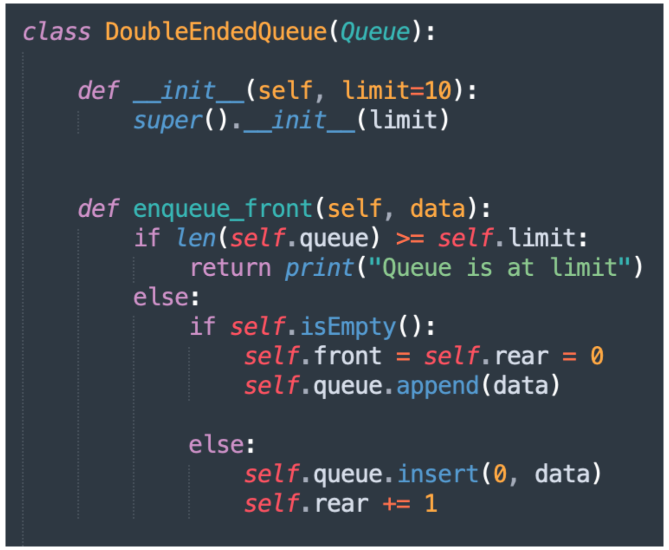
Real-world example (2): We can expand on the browser example. A deque can be better used to manage the history of visited pages as well as the navigation. New pages are added to the front, and the user can move forward or backward through the history, while effectively popping unused pages from the back to manage the available memory.

Double-ended Queues in Code

Basic deque operations you should know:

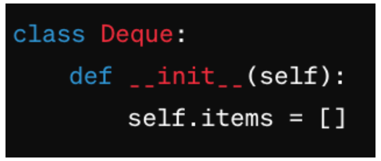
- Add element front and back
- Remove element front and back
- Peek at the front and back element
- Find and return an element in the list

Using inheritance from Queue class:

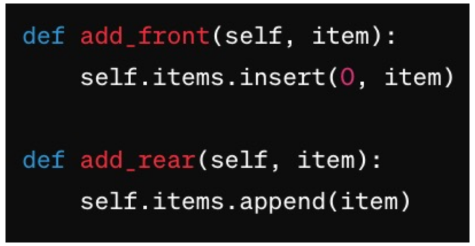


```python
class DoubleEndedQueue(Queue):

    def __init__(self, limit=10):
        super().__init__(limit)


    def enqueue_front(self, data):
        if len(self.queue) >= self.limit:
            return print("Queue is at limit")
        else:
            if self.isEmpty():
                self.front = self.rear = 0
                self.queue.append(data)

            else:
                self.queue.insert(0, data)
                self.rear += 1
```

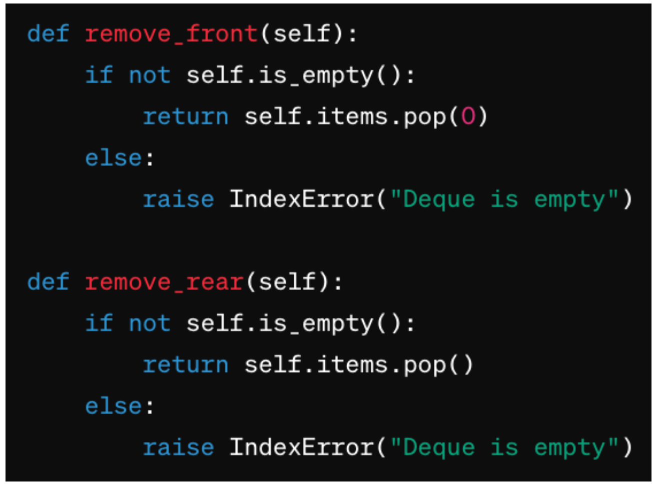
Creating a new class:



```python
class Deque:
    def __init__(self):
        self.items = []
```



```python
def add_front(self, item):
    self.items.insert(0, item)

def add_rear(self, item):
    self.items.append(item)
```



```python
def remove_front(self):
    if not self.is_empty():
        return self.items.pop(0)
    else:
        raise IndexError("Deque is empty")

def remove_rear(self):
    if not self.is_empty():
        return self.items.pop()
    else:
        raise IndexError("Deque is empty")
```

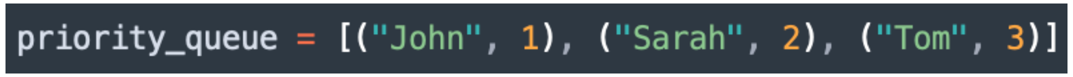
What are Priority Queues?

What are Priority Queues?

- A data structure that contains elements and operates under a highest priority out first principle.

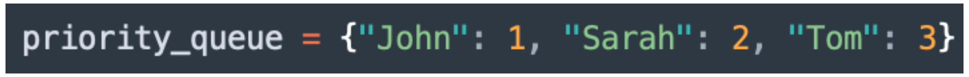
Ways to implement a priority queue:

- Using a list of tuples (lightweight implementation, can be ordered, slower operations – have to traverse list, can have duplicate priorities)



```python
priority_queue = [("John", 1), ("Sarah", 2), ("Tom", 3)]
```

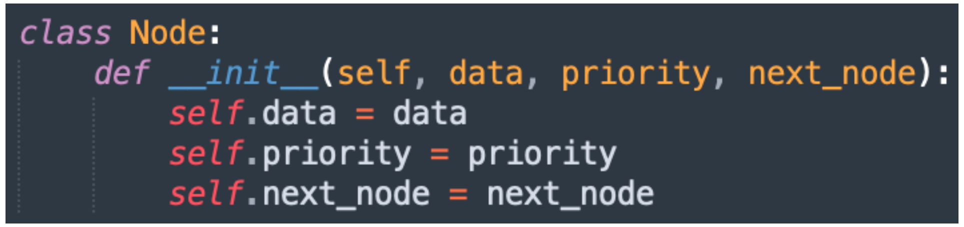
- Using a dictionary (unique priority keys, efficient lookup - can check min of keys, lack of ordering)



```python
priority_queue = {"John": 1, "Sarah": 2, "Tom": 3}
```

Ways to implement a priority queue:

- Using a linked list (elements can easily be added in ordered position, dynamic (non-contiguous) space in memory, slower lookup - requires traversal for some operations, can have duplicate priorities)



```python
class Node:
    def __init__(self, data, priority, next_node):
        self.data = data
        self.priority = priority
        self.next_node = next_node
```

- Using a heap (section later).

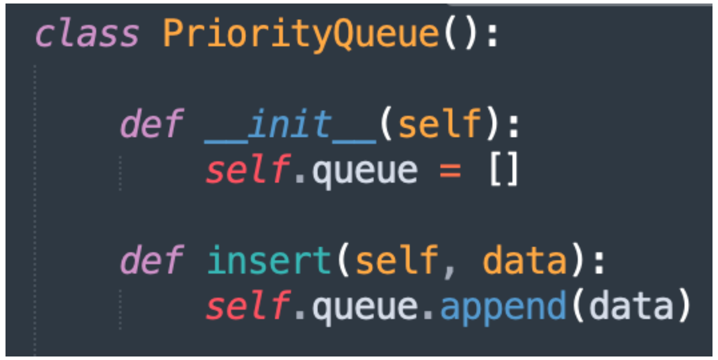
Real-world example (1): An engineer for Amazon might use a priority queue to fulfil orders based on shipping urgency.
Real-world example (2): The Facebook feed might use a priority queue based on relevancy scores to choose the order to display content.

Priority Queues in Code

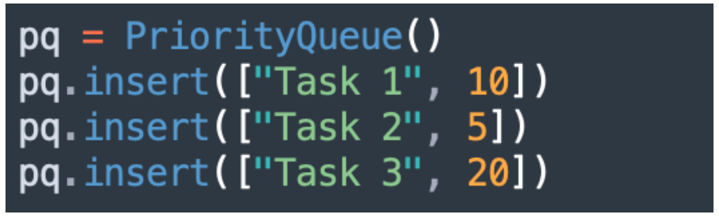
Each problem may require a different implementation of a priority queue. Fundamentally the logic of the operations are pretty similar between implementations

Basic priority queue (using list of tuples) operations you should know:

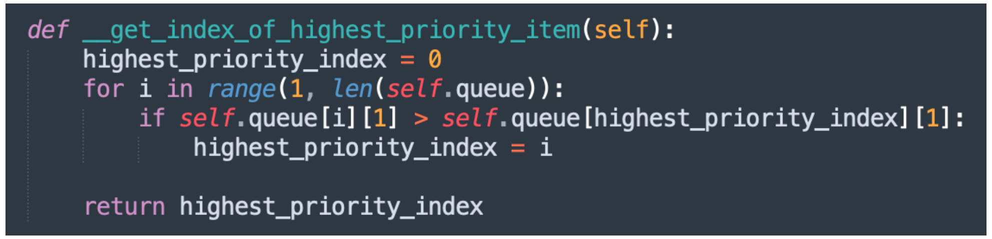
- Insert element with given priority
- Remove element with highest priority
- Peek at the highest priority
- Update a priority



```python
class PriorityQueue():

    def __init__(self):
        self.queue = []

    def insert(self, data):
        self.queue.append(data)
```

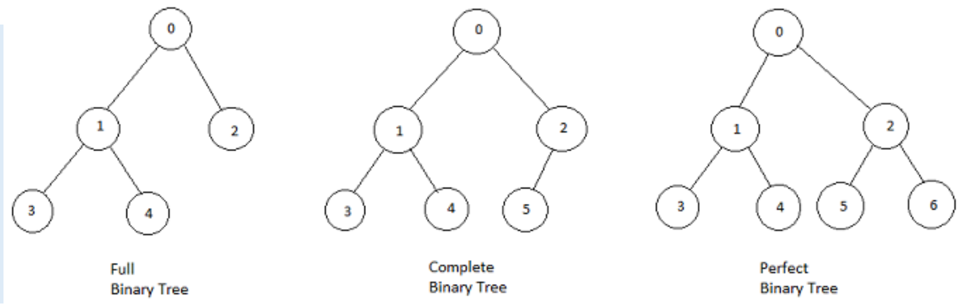


```python
pq = PriorityQueue()
pq.insert(["Task 1", 10])
pq.insert(["Task 2", 5])
pq.insert(["Task 3", 20])
```



```python
def __get_index_of_highest_priority_item(self):
    highest_priority_index = 0
    for i in range(1, len(self.queue)):
        if self.queue[i][1] > self.queue[highest_priority_index][1]:
            highest_priority_index = i

    return highest_priority_index
```

What are Binary Search Trees?



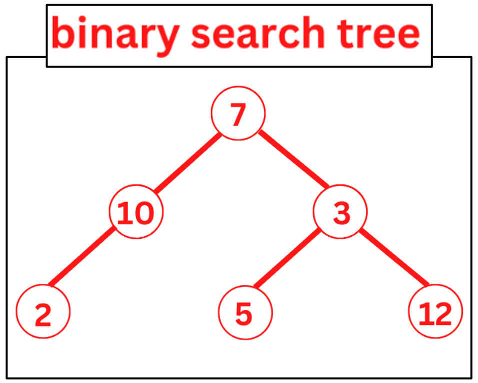
- **Full binary tree**: Every node other than leaf nodes has 2 child nodes.
- **Complete binary tree**: All levels are filled except possibly the last one, and all nodes are filled in as far left as possible.
- **Perfect binary tree**: All nodes have two children and all leaves are at the same level.

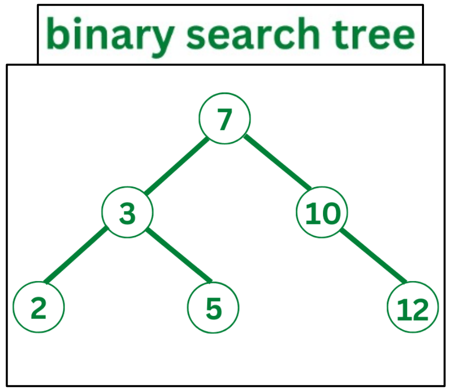
What are Binary Trees?

- A data structure that organises elements into connected nodes, where typically each node has up to two children nodes – the left and right child. The first node in the tree is the root.

What are Binary Search Trees?

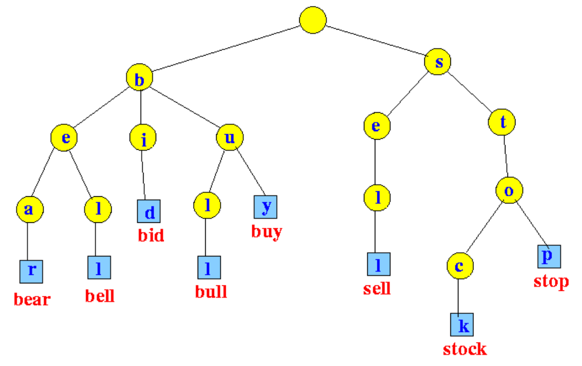
- A binary search tree has the same definition as above with 1 extra line: A BST is a binary tree where each node's left child is smaller in value than the parent, and each node's right child is greater in value than the parent.





Real-world example (1): A company may likely use a binary tree with multiple nodes to organise the employee hierarchy. In companies we can follow the tree all the way up to the CEO.

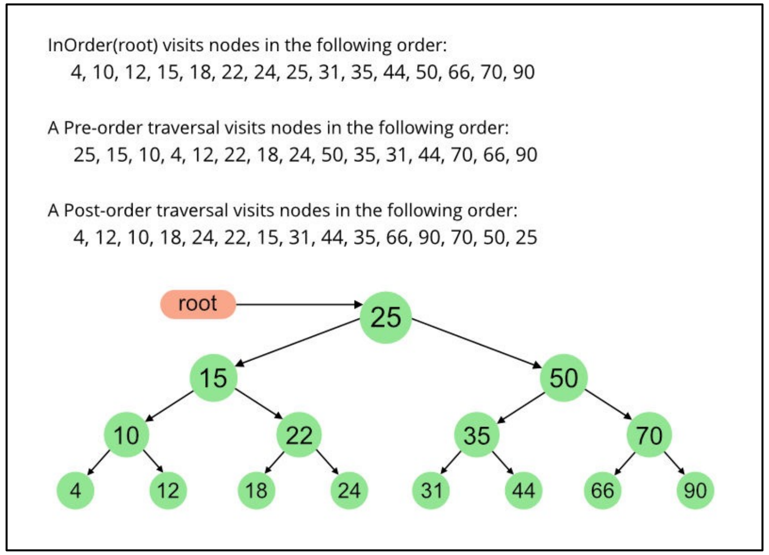
Real-world example (2): We could use a BST to implement an autocomplete in an app.



Binary Search Trees Code

Basic priority queue (using list of tuples) operations you should know:

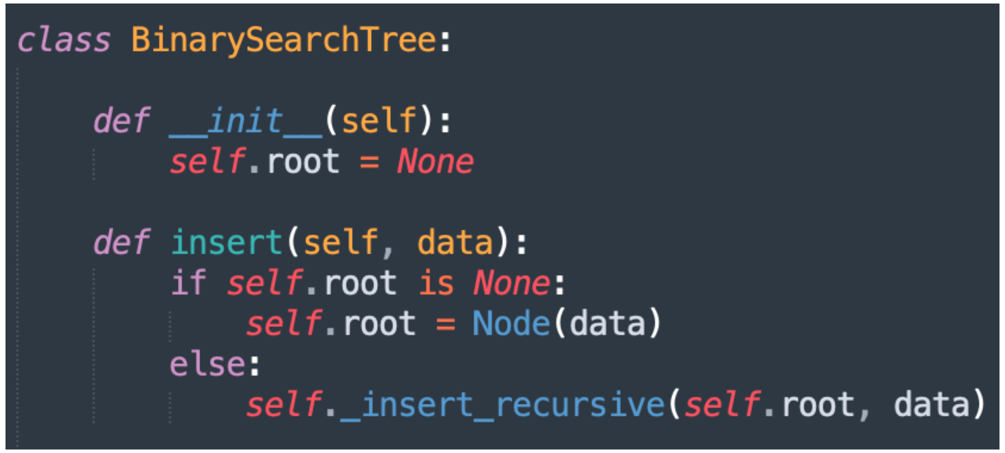
- Insert node
- Search for node
- Find min/max node
- Delete
- Traverse and print



InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

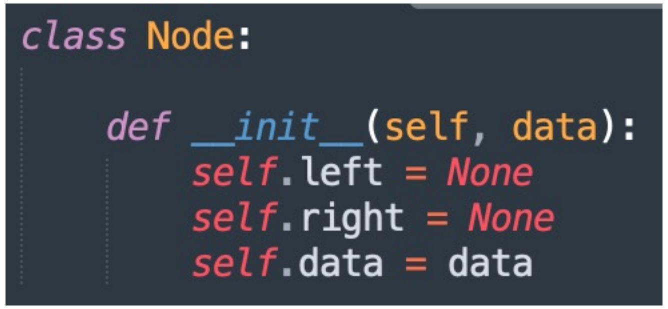
A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



```python
class BinarySearchTree:

    def __init__(self):
        self.root = None

    def insert(self, data):
        if self.root is None:
            self.root = Node(data)
        else:
            self._insert_recursive(self.root, data)
```

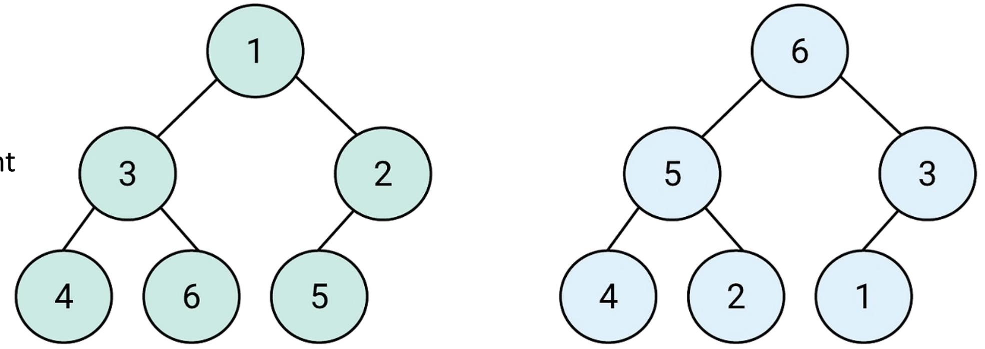
Note: iterative BST operations in Coding Problems section – very similar code however so feel free to use preferred method.



```python
class Node:

    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data
```

What are (Binary) Heaps?

N nodes
LogN height



Min heap

Max Heap

What are Heaps?

- A tree-based structure of nodes that satisfies the heap property: Max-heap – every node's value is greater than or equal to each child node individually. Min heap – every node's value is less than or equal to each child node individually.

- To achieve this the heap sorts itself based on new entries by "heapifying up" or "heapifying down" (swapping) parent and children nodes around.

What makes it a binary heap?

- A binary heap is a specific implementation of a heap using a complete binary tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.

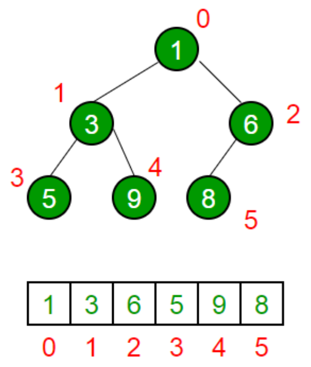
Priority Queue

- Technically binary heaps are another data structure we can use to implementing priority queues. At the abstract level a priority queue is a data type that allows elements to be inserted with priority and retrieved accordingly. A queue and a binary heap are two data structures we can use to implement the priority queue data type

Real-world example (1): A heap could be used in the driver allocation system for Uber. A min-heap of available drivers, sorted by distance to pick-up location. The system can quickly retrieve and assign the closest driver to the new ride request

Real-world example (2): When displaying a user's news feed, the system needs to prioritize and display the most relevant and recent posts first. By using a max-heap to store posts based on their relevance score.

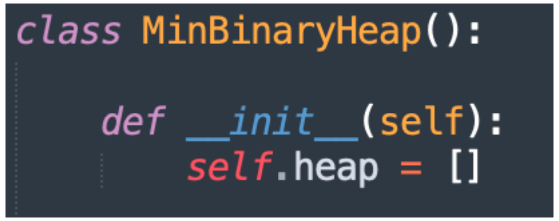
Binary Heaps in Code



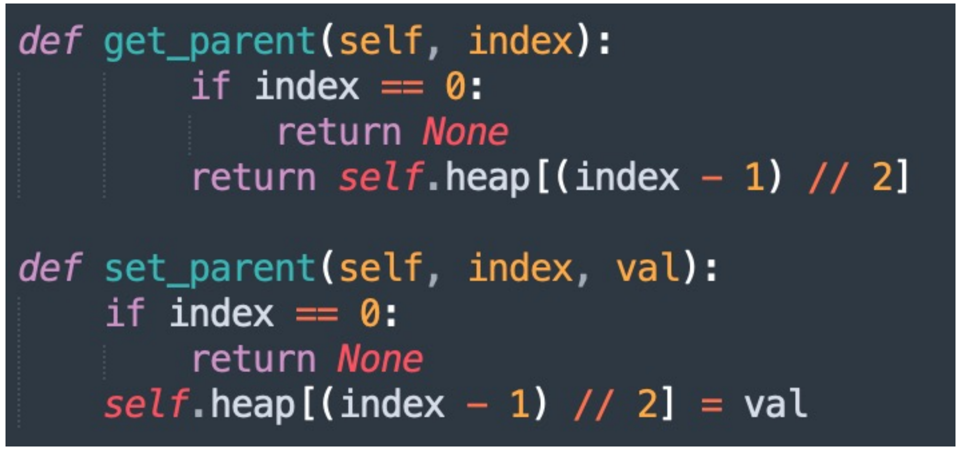
Basic priority queue (using list of tuples) operations you should know:

- Insert element
- Heapify up / heapify down
- Remove min
- Peek min

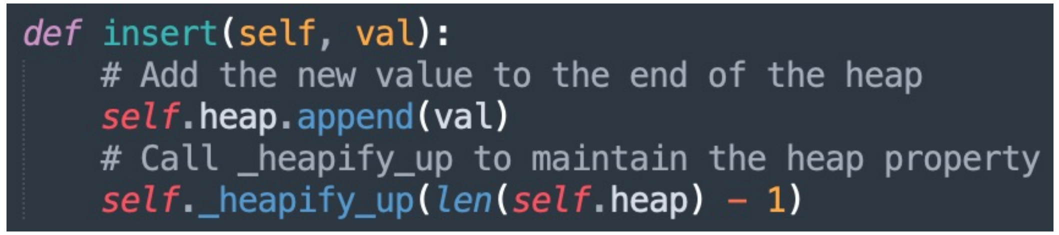
Arr[(i-1)/2]	Returns the parent node
Arr[(2*i)+1]	Returns the left child node
Arr[(2*i)+2]	Returns the right child node



```
class MinBinaryHeap():

    def __init__(self):
        self.heap = []
```

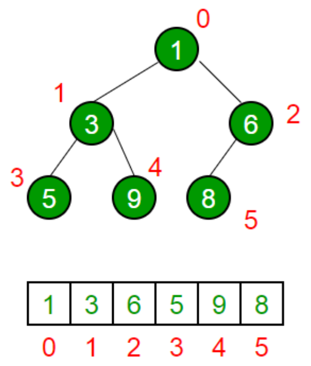


```
def get_parent(self, index):
        if index == 0:
            return None
        return self.heap[(index - 1) // 2]

def set_parent(self, index, val):
    if index == 0:
        return None
    self.heap[(index - 1) // 2] = val
```



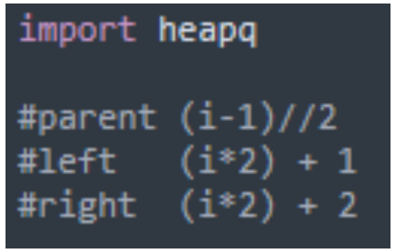
```
def insert(self, val):
    # Add the new value to the end of the heap
    self.heap.append(val)
    # Call _heapify_up to maintain the heap property
    self._heapify_up(len(self.heap) - 1)
```

Binary Heaps in Code (Heapq Library)



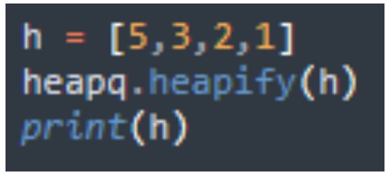
Basic priority queue (using list of tuples) operations you should know:

- Insert element
- Heapify up / heapify down
- Remove min
- Peek min

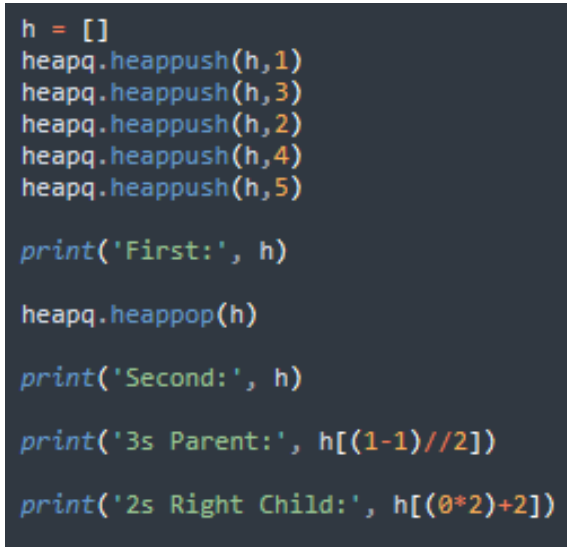


```
import heapq

#parent (i-1)//2
#left   (i*2) + 1
#right  (i*2) + 2
```

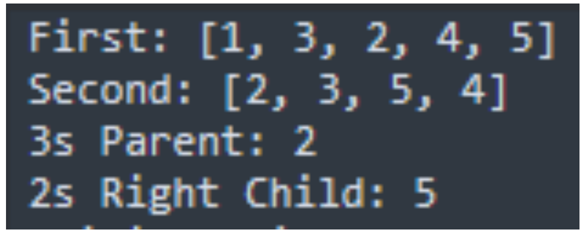
Heapify array



```
h = [5,3,2,1]
heapq.heapify(h)
print(h)
```



```
h = []
heapq.heappush(h,1)
heapq.heappush(h,3)
heapq.heappush(h,2)
heapq.heappush(h,4)
heapq.heappush(h,5)

print('First:', h)

heapq.heappop(h)

print('Second:', h)

print('3s Parent:', h[(1-1)//2])

print('2s Right Child:', h[(0*2)+2])
```



```
First: [1, 3, 2, 4, 5]
Second: [2, 3, 5, 4]
3s Parent: 2
2s Right Child: 5
```

What are Graphs?

What are Graphs?

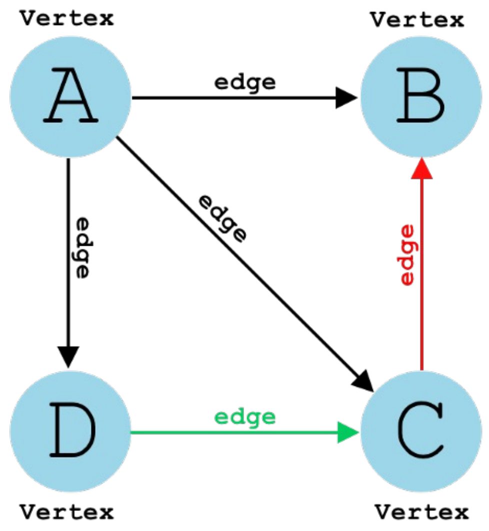
- A data structure consisting of several nodes connected connected by edges (lines). Nodes in a graph are sometimes called vertices.

- To achieve this the heap sorts itself based on new entries by "bubbling up" or "bubbling down" (swapping) parent and children nodes around.

Directed/Undirected

- If the edges that connected nodes are one-way or directed it is a directed graph. Edges that we can traverse in either direction are undirected graphs.

.



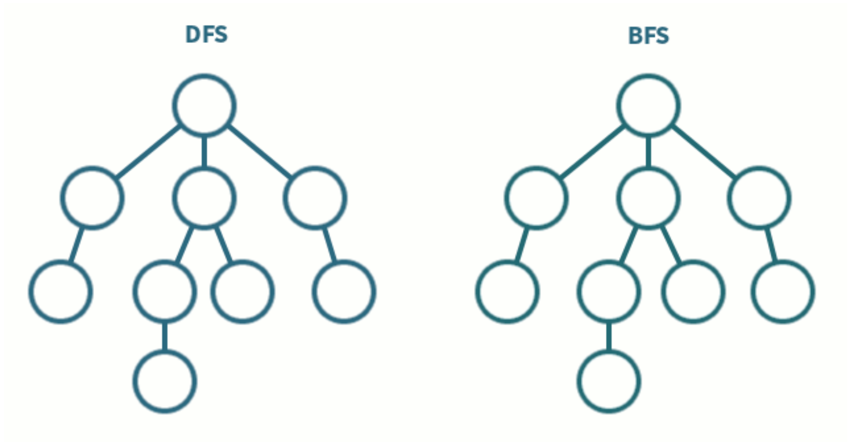
Real-world example (1): In Amazon's delivery network, trucks need to efficiently navigate through a large number of destinations (nodes) and roads (edges) to deliver packages. The problem can be modelled as a graph, using an algorithm (example: Dijkstra) to find the shortest route to all nodes.

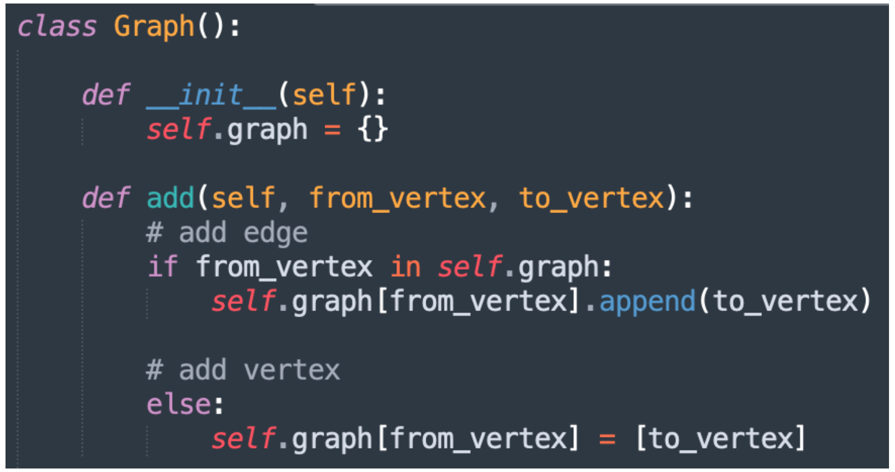
Real-world example (2): Each user in Facebook can be represented as a node in a graph, and friendship connections between users can be represented as edges. By analysing the graph structure, Facebook's recommendation system can suggest new friends.

Graphs in Code

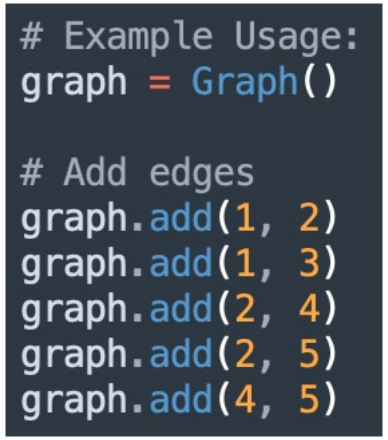
Basic priority queue (using list of tuples) operations you should know:

- Insert vertex
- Remove vertex
- Print graph
- Breadth First Search
- Depth First Search

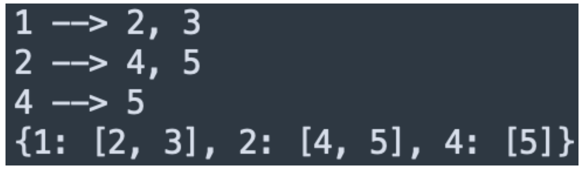




```
class Graph():

    def __init__(self):
        self.graph = {}

    def add(self, from_vertex, to_vertex):
        # add edge
        if from_vertex in self.graph:
            self.graph[from_vertex].append(to_vertex)

        # add vertex
        else:
            self.graph[from_vertex] = [to_vertex]
```



```
# Example Usage:
graph = Graph()

# Add edges
graph.add(1, 2)
graph.add(1, 3)
graph.add(2, 4)
graph.add(2, 5)
graph.add(4, 5)
```



```
1 --> 2, 3
2 --> 4, 5
4 --> 5
{1: [2, 3], 2: [4, 5], 4: [5]}
```