Data Structures and Algorithms

UBCO Master of Data Science – DATA 532

Recap…

- Notion of an algorithm and data structures
- Analyze algorithm for time complexity, correctness, and efficiency
- Compare algorithms on the basis on Big O
- Looked at a variety of search algorithms
  - Linear Search
  - Better Linear Search
  - Binary Search

Space Complexity: Big O

- **Time Complexity** cares about how much time it takes for the program to complete the task.
- **Space Complexity** is related to how much memory the algorithm takes with respect to the size.
- Space complexity for these algorithms?
  - Linear Search
  - Better Linear Search
  - Binary Search

Linear Search (Space Complexity)

Linear-Search(A, n)

35	30	19	30	8	12	11	17	2	5
0	1	2	3						N-1

Input:

- *A: an array*
- *n: the given number whose position in A needs to be determine*

Output:

x : Either an index i for which A[i] = x, or -1 (Not Found)

1. Set x to Not-Found
2. For each index i, going from 0 to N-1, in order:
   A. If A[i] = n, then set answer to the value of i
3. Return the value of x as the output

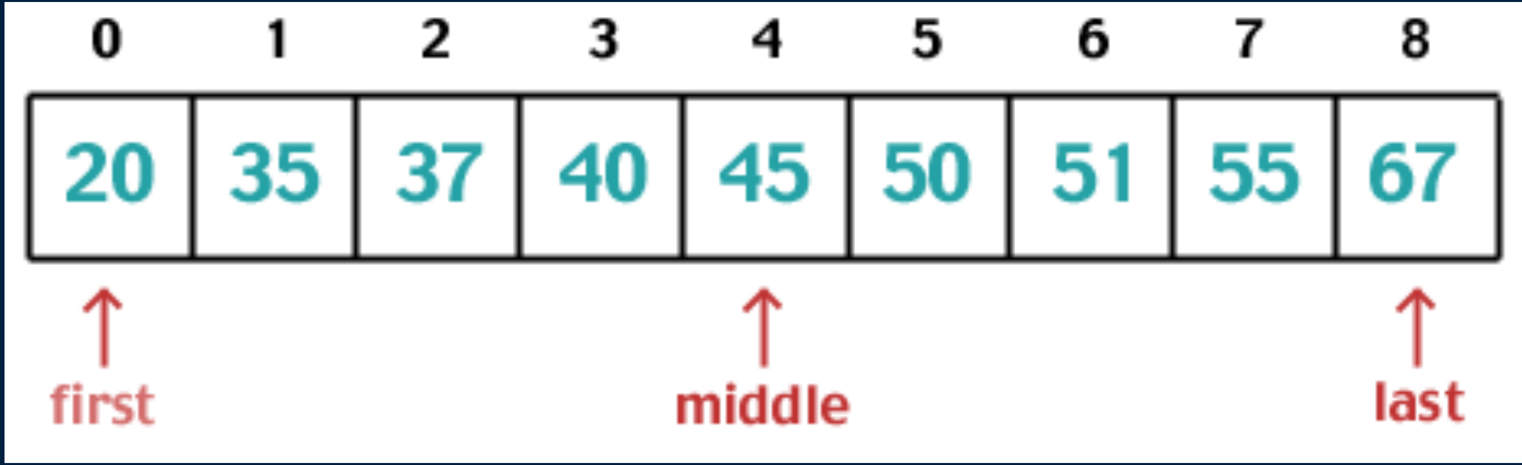
Binary Search (Space Complexity)

Compare 51 with middle

if 51 == middle then "Hurray"

if 51 < middle then between first and middle

if 51 > middle then between middle and last



Dealing with data...

How to use it ?

How to store it ?

How to process it ?

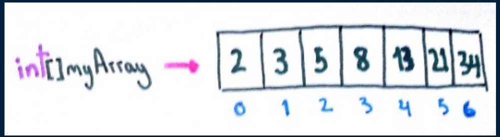
How to gain "knowledge" from it ?

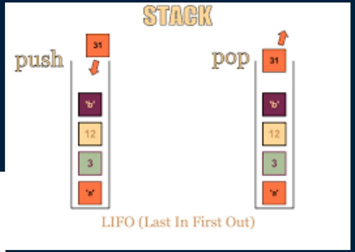
How to keep it secret?

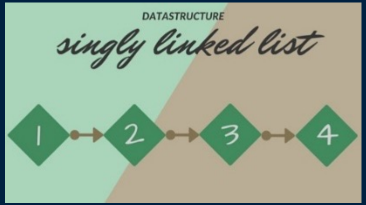
Data Structures

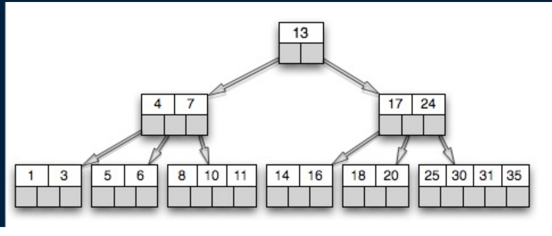
A Data Structure is:

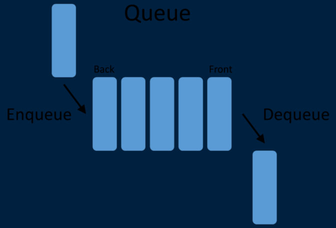
- “an organization of information, usually in computer memory, for better algorithm efficiency."

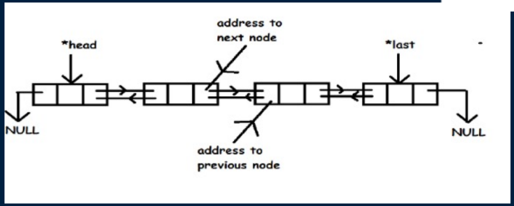












Data Structure Concepts

Data Structures are containers.

- They hold data.
- Arrays are a data structure.
- So are linked lists.

Other types of data structures:

- stack, queue, binary trees,
  other trees, hash table,
  dictionary or map, heap, priority queue, …
- en.wikipedia.org/wiki/List_of_data_structures

Different types of data structures are optimized for certain types of operations.

Core Operations

Data Structures have three core operations:

- A way to add data.
- A way to remove data.
- A way to access data, without modifying the data structure.

Details depend on the data structure.

- For instance, an *Indexed List* data structure may need to:
  - add data at position in the *List*.
  - access data by position.
  - remove data by position.

More operations are added depending on what data structure is designed to do.

- For instance, a stack data structure:
  - only add or remove data from the top of the stack only.
  - Cannot access middle data, only top record is accessible

Implementation-Dependent Data Structures

1) Arrays

- Collection of objects stored contiguously in memory.
- Accessed by index.

2) Linked Structures

- Collection of node objects.
  - Store data and reference to one or more other nodes.
- Linked Lists
  - Linear collection of nodes.
  - Single-linked List – nodes contain references to next node in list.
  - Double-Linked List – nodes contain reference to next and previous nodes in list.
- Trees
  - Hierarchical structure.
  - Nodes reference two or more “children”

Implementation-Independent Data Structures

Abstract Data Types (ADTs):

- Descriptions of how a data type will work without implementation details.
- Description can be a formal, mathematical description.
- ADT is in the logical level and data structure is in the implementation level.

Examples:

- Stack, Queue, Indexed List, Heap, Binary Search Tree, etc.

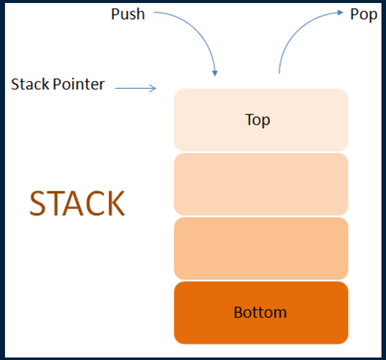
Stack and Queue ADTs

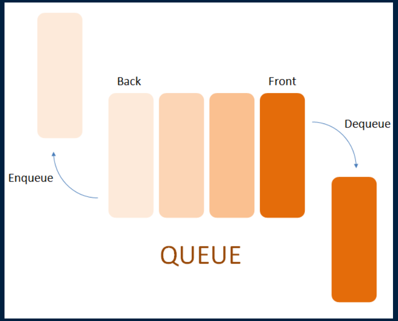
Stack ADT

- Expected behavior: Last in, First out (LIFO)
- Operations: `push`, `pop`, `peek`.

Queue ADT

- Expected behavior: First in, First out (FIFO)
- Operations: `enqueue`, `dequeue`, `front`.





List ADTs

Sorted/Ordered List

- Items in list are arranged according to "natural ordering."
- Operations: `add`, `max/min`, `find`, etc.

Indexed List

- Items are accessed by position in list.
- Operations: `get`, `remove(index)`, `add(index)`, etc.

Unsorted List

- Items are stored without an implicit ordering.
- Operations: `addToFront`, `remove(element)`, `last`, etc.

Tree ADTs

Binary Search Trees (BSTs)

- Items stored in sorted order.

Heaps (Max or Min)

- Items stored according to the "Heap Property."

AVL and Red-Black Trees

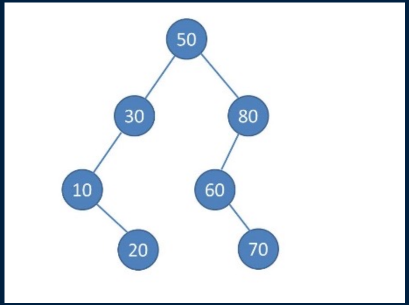
- BSTs that stay balanced.

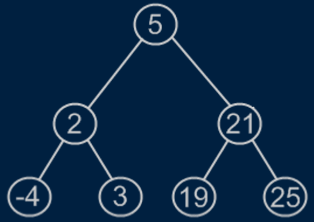
Splay Trees

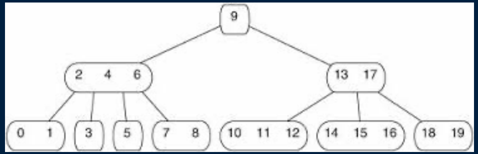
- BST with most recently items at top.

General-Trees

- Another variation of trees.
- Nodes can have more than two children.







Other ADTs

Priority Queues

- Next item removed has highest / lowest priority.

Hash Tables

- Hash function:
  - Computes an *index* into a table of *buckets* or *slots.*
  - Look in the bucket for the desired value.

Maps

- Collection of items with a key and associated values.
- Similar to hash tables.

Graphs

- Nodes with unlimited connections between other nodes.

Implementing ADTs

The operations and behaviors are already specified.

- For instance, every *Stack* has `push`, `pop` and `peek` methods.

But given an interface describing an ADT, how to implement it?

- Must decide which internal storage container to use to hold the items in the ADT.
- Usually one of the Implementation-Dependent Data Structures:
  - Arrays, Linked Lists
- But can be another ADT.
  - Using Heap to implement a Priority Queue.

Does the programming language effects the implementation of ADTs?

A) Yes

B) No

C) Can’t say

Stacks

Stack

Container of objects that are inserted and removed according to the principle of

- LIFO: Last-in-first-out

Objects can be inserted at any time, but only the most recently inserted can be removed at any time.

Operations:

- Push: enter item onto stack
- Pop: remove item from stack

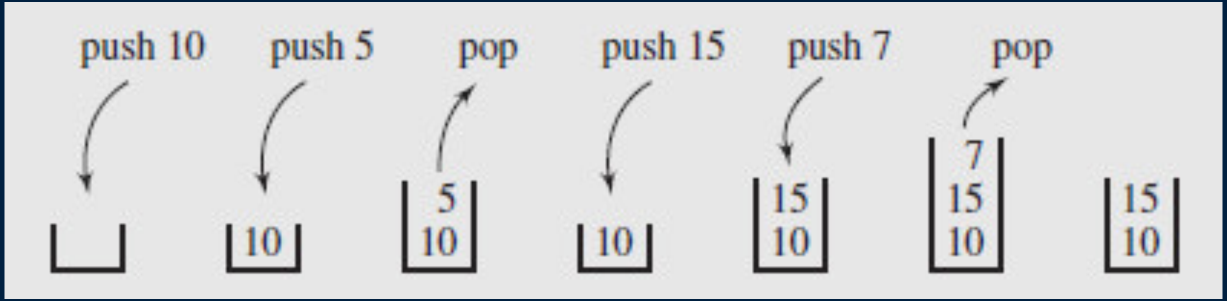
Stack Methods

push (o) – Insert an item into/onto the stack

- Input: an object. Output: none.
- If the stack has a fixed size and the stack cannot accept the push, a stack-overflow exception/error is thrown (or returned)

pop() – Returns the most recently inserted object from the stack and removes the object from the stack (an object is removed in last-in-first-out order)

- Input: none. Output: an object.
- If the stack is empty, a stack-empty exception/error is thrown (or returned)



Stack Methods

Auxiliary/Support Methods

- **size()** – Returns the number of objects in the stack
  - Input: none. Output: non-negative integer.
- **isEmpty()** (or empty()) – Returns true if there are no objects in the stack
  - Input: none. Output: true or false
- **peek()** (or top()) – Returns a reference to (alternatively, a copy of) the most recent item put into the stack
  - Input: none. Output: reference to an object (or an object if a copy)

Stack Running Times

What is the running time of each operation?

Push

Pop

isEmpty()

Stack Implementation

In Python, the list can be used as a stack:

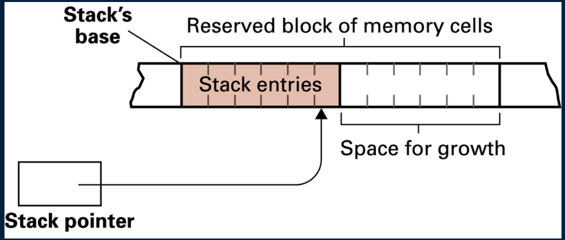
- `list.append(x)`
- `list.pop()`

Let’s try to implement our own stack as an exercise in understanding what it takes to implement a data structure

Stack Implementation in Python: EXERCISE!

```
class Stack(list):
    def push(self, item):
                                            # add item to end of list
    def pop(self):
        if not self:                        # (not self) return true if stack (list) is empty
        else:
                                             # last element of list
                                             # delete last element

    def top(self):
        if not self:
        else:                                # return last element if the list
    def isEmpty(self):
```



Brookshear Figure 8.10

Which feature on your web-browser may be using stack?

A) Printing

B) Bookmarks

C) New Window

D) Back Feature

What is this good for ?

- Page-visited history in a Web browser

What is this good for ?

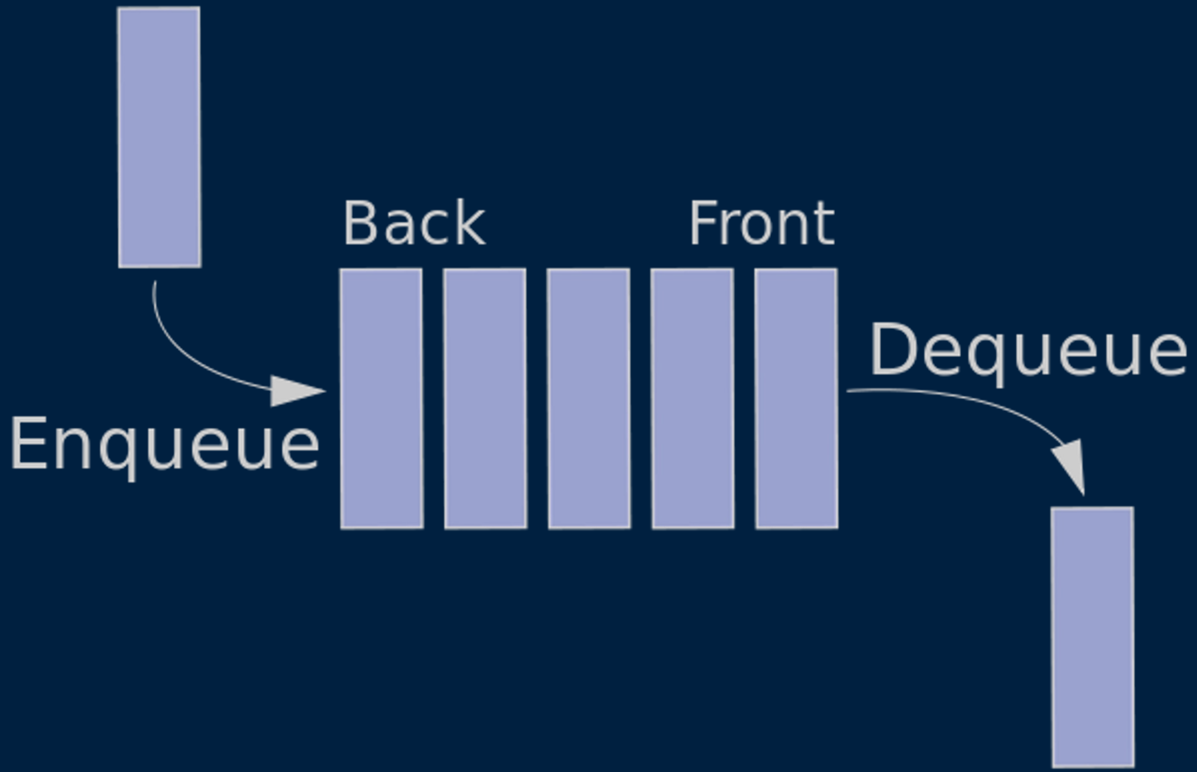
- Page-visited history in a Web browser
- Undo sequence in a text editor

What is this good for ?

- Page-visited history in a Web browser
- Undo sequence in a text editor
- Saving local variables when one function calls another, and this one calls another

In general, stacks are useful for *backtracking* and *recursion*

Queues


Back
Front
Enqueue
Dequeue

Queue

Container of objects that are inserted and removed according to the principle of

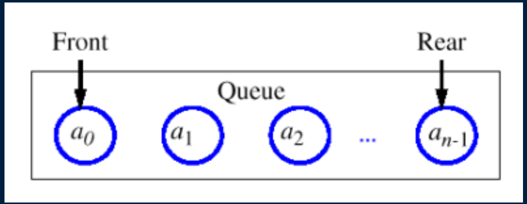
- FIFO: First-in-first-out

Objects can be inserted at any time, but only the least recently inserted can be removed at any time.

Operations:

- Enqueue: put item onto queue
- Dequeue: remove item from queue

A close example is like a lineup at your favorite coffee shop in the morning, where the service is provided to customers with FIFO order.



Why Queues?

Queues are used extensively in

- Computer networking
  - For keeping storing and sending network packets
- Operating systems
  - For scheduling processes to receive resources
- Playlists for your mp3 player
- …

Queue Methods

enqueue(item) – Insert the item into the queue.

- If the queue has a fixed capacity, an exception/error will be returned if attempting to enqueue an item into a filled queue.
- Input: item. Output: none.

dequeue() – Returns a reference to and removes the item that was least recently put into the queue (first-in-first-out)

- If the queue is empty, an exception/error will be returned if attempting to dequeue.
- Input: none. Output: item

Queue Methods

size() (or getSize()) – Returns the number of items in the queue.

- Input: none. Output: integer.

isEmpty() – Checks if the queue has no items in it. Returns true if the queue is empty.

- Input: none. Output: true or false.

front() – Returns a reference to the “first” item in the queue (the least recent item). If the queue is empty, an exception/error will be returned if attempting to dequeue.

- Input: none. Output: item.

Queue Running Times

What is the running time of each operation?

enqueue

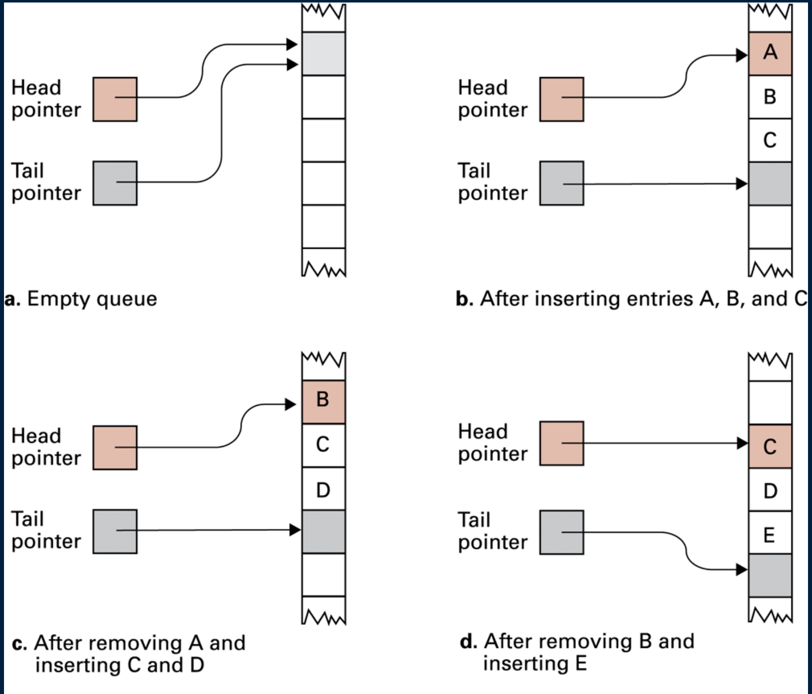
dequeue

isEmpty()

Queue Implementation

In Python, the list can be used as a queue:

- `list.append(x)`
- `list.pop(0)`



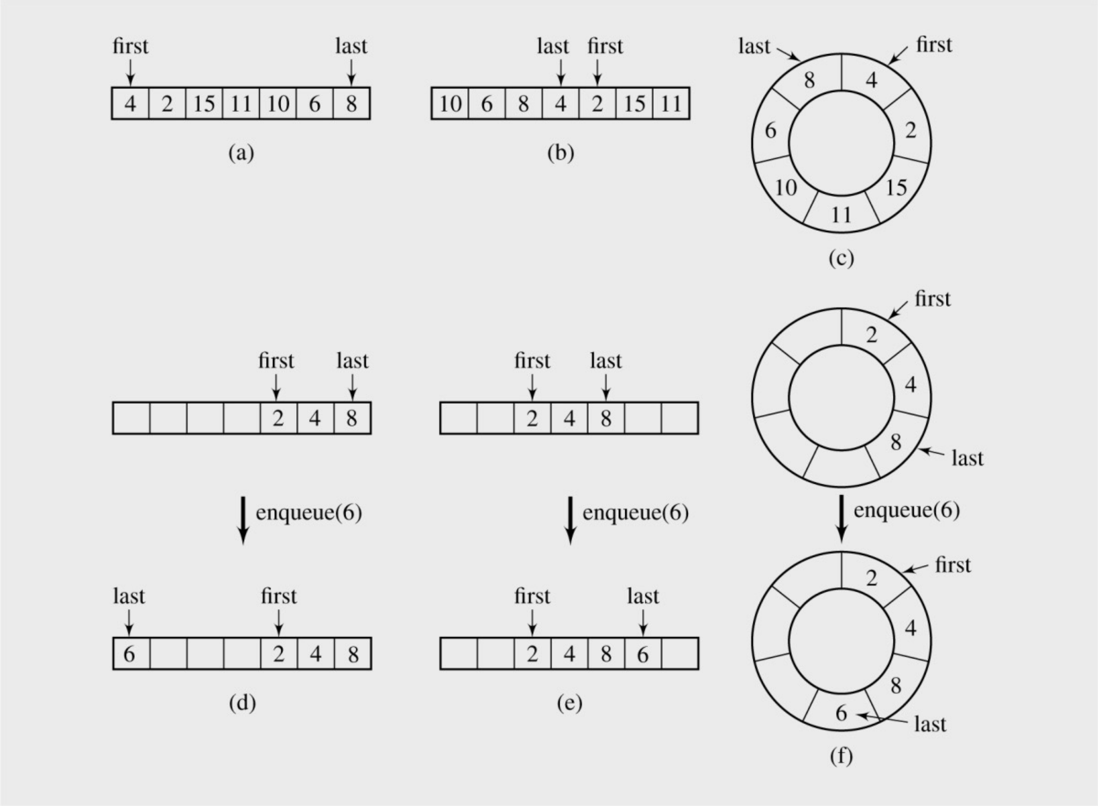
Brookshear Figure 8.13

Queue Implementation

```
class Queue:
    def __init__(self):
        self.items = []
    def enqueue(self,val):
        self.items.insert(0,val)
    def dequeue(self):
        if self.is_empty():
            return None
        else:
            return self.items.pop()
    def size(self):
        return len(self.items)
    def is_empty(self):
        return self.size() == 0
```

Circular Queue

First and last are two indexes to show the front and the back of the queue

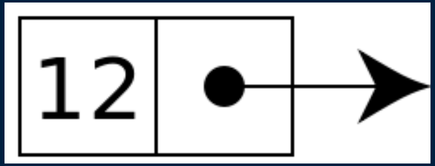


Linked Lists

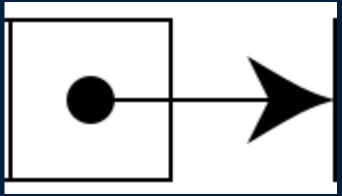
Linked Lists

Head

- A linked list consists of **nodes**.



- Each node consists of a **value** and a **pointer** to another node.



The starting node of a linked list is referred to as the **head(er)**.
Essentially, linked list is a chain of values connected with pointers.

Why use linked lists?

Linked list is often compared to arrays. Whereas an array is a fixed size of sequence, a linked list can have its elements to be dynamically allocated.

Advantages

- A linked list saves memory
- Linked list nodes can live anywhere in the memory

Disadvantages

- Linear look up time

Stop! It is not useful to implement linked lists in Python as they are a low level data structure and relevant in programming languages as C or C++

Python does not have linked lists in its standard library.

Take home messages…

- We learnt what are data structures
- We looked at stacks, queues, and linked lists

Next Class: Sorting Algorithms, Hash Tables

THE UNIVERSITY OF BRITISH COLUMBIA