Fundamentals of Algorithms & Data Structures

for MLEs

Amber & The Insight Team

Outline

- Why do you need this?
- Strategies and tips for the coding interview
- **Algorithms and Data Structures**
- Coding craftsmanship
- Resources

- Big O notation
- Sorting
- Linked Lists
- Hash tables
- Queues and stacks
- Binary search
- Binary trees
- Heaps

Data structures to know:

- Arrays
- Linked lists
- Stacks
- Queues
- Trees
- Graphs
- Heaps
- Hash sets
- Hash maps/tables

Algorithms to know:

- Breadth-first search
- Depth-first search
- Binary search
- Quicksort
- Mergesort
- A*
- Dynamic programming
- Divide and conquer

Data structures Week 5:

- Arrays
- Linked lists
- Stacks/Queues
- Trees

Data structures Week 6:

- Graphs
- Heaps
- Hash sets
- Hash maps/tables

Algorithms Week 7:

- BFS/ DFS
- Binary search
- Quicksort/ Mergesort
- Dynamic programming

Why Software Engineering Skills?

1. It is likely that your MLE interview will involve coding. This may include more “software engineering” knowledge, such as algorithms and data structures.
2. There is an industry trend where, more and more, data scientists and MLEs are expected to ship code directly to production.
3. Good coding knowledge and craftsmanship goes a long way in ensuring reproducibility, minimizing bugs and improving communication.

Strategies and tips for the coding interview

- Make sure you understand the question
  - **Ask for a use case**
  - Give an example and ask if you got the requirements right
- If really stuck, ask for a hint. This is OK if used sparingly.
- Notify the interviewer if you have seen that question before. Say if you were asked that question before in this company's interview.
- We will see an example interview later today!

Data Structures in Python

Primitive Data Structures

a. Integers

b. Float

c. Strings

d. Boolean

- building blocks for data manipulation and contain pure, simple values of a data

Non-Primitive Data Structures

a. Arrays

b. Lists

c. Tuples

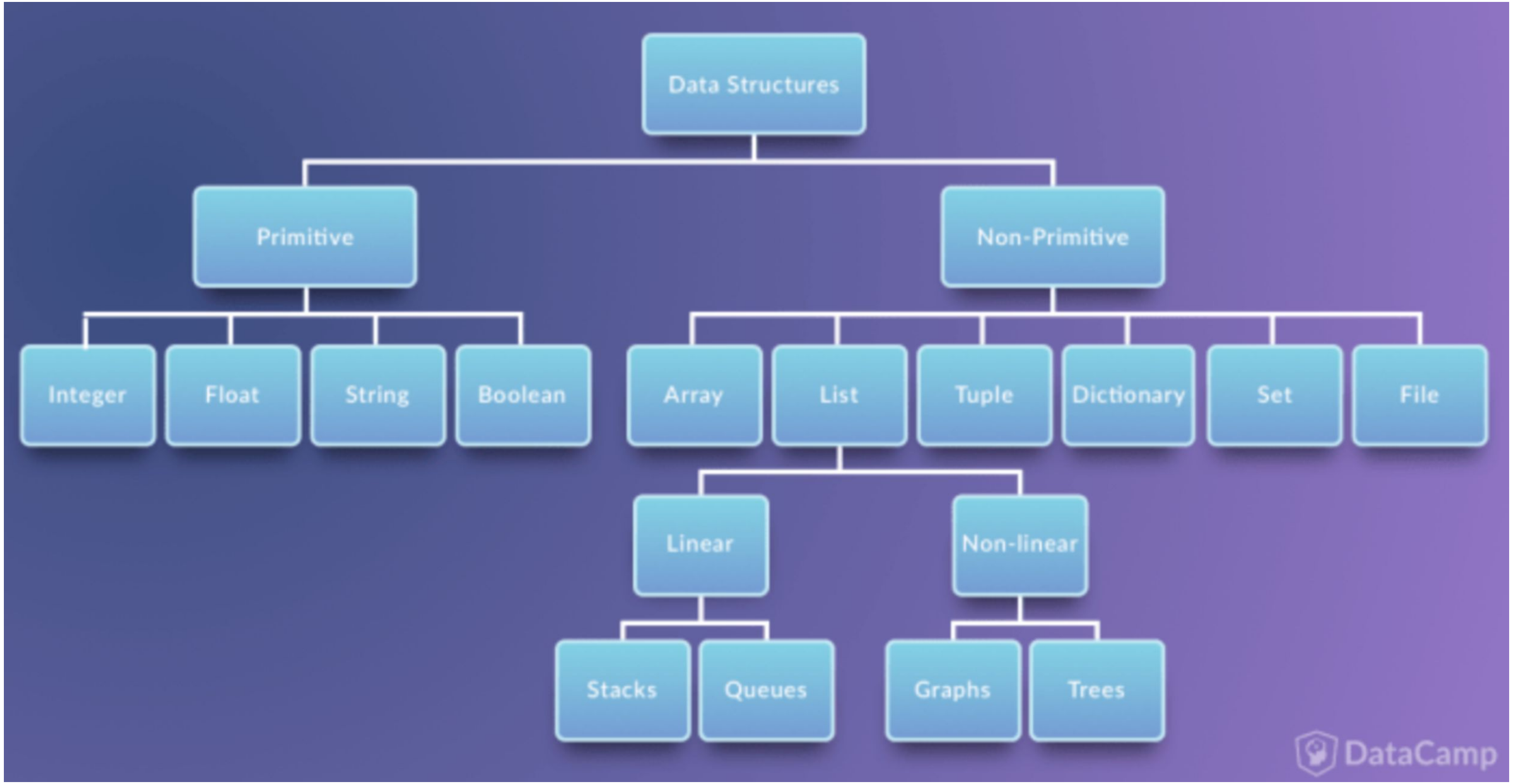
d. Dictionary

e. Sets

f. Files

- All contain the primitive data structures within more complex data structures for special purposes

Example Chart



Python Specific Stuff

1. Know basic operations/facts of the language
    a. String methods (.split(), .strip(), .join(x_list), .lower() for example).
        i. See cheatsheet
    b. Set, Dict, and List operations, and time complexity of all of those operations
2. Python's Collections module is your friend
    a. Defaultdict - use when you want values to be auto-created for you when a key is missing
    b. Deque (double-ended queue)- use to make fast appends and pops on both ends of the list
3. Generators
    a. Like a LC [] that doesn’t put a whole list into memory at once. The generator () yields one item at a time and generates item only when in demand → more memory efficient than the lists.
4. PEP8
    a. Use it!

Common Ways to Fail Coding Interviews

1. Making 0 progress on a problem and not speaking your thoughts
    a. When you don’t know, don’t just do nothing
2. Using reserved words as variable names
    a. list = [x for x in range(10)] -> you can’t rescue yourself from this mistake, never make it
3. Having terrible style and/or no conventions
    a. variable_and_funciton_names should be easily read by someone new to the code
    b. ClassNames
4. Have no idea the complexity of the code you wrote
    a. You often are calling a function that has a loop built into it

Algorithms &
Data Structures

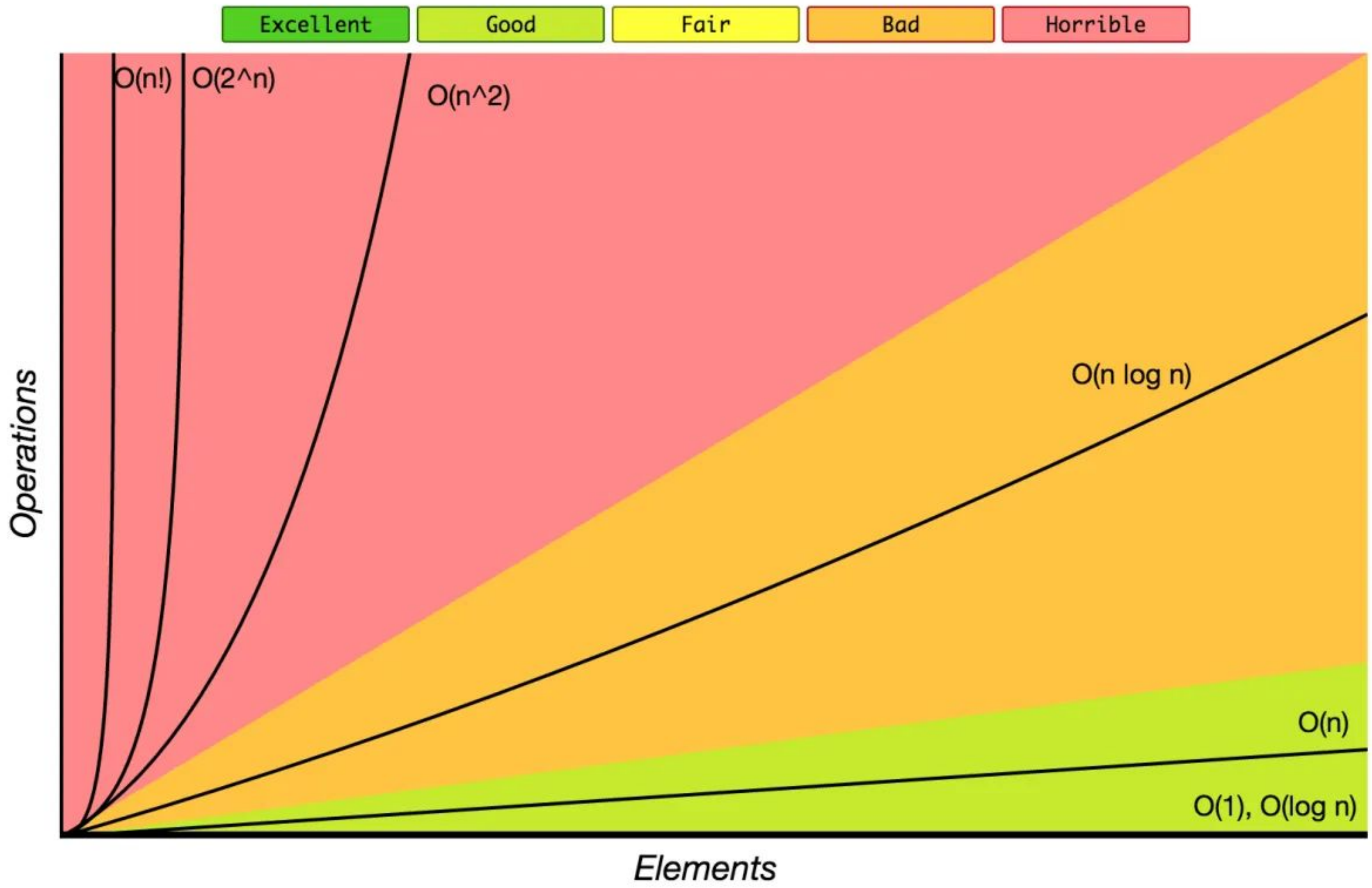
Big O Notation

$$\mathbf{O(\ f(N)\ )}$$

Common time complexities

- O(1): same actions regardless of input size
- O(log n): repeatedly cut the problem size in half
- O(n): iterate your input a set number of times
- O(n^x): X nested for loops over your input
- O(2^n): recursion with multiple calls

Big-O Complexity Chart



Big O Notation - Examples

- Used for time and space complexity estimates
- Drop Constants:
  - No such thing as O(2n)
  - **Correct would be O(n)**
  - **Exception is O(1)** - called “constant complexity” -- the BEST complexity
- Drop non-dominant terms:
  - We don’t care about $O(N^2 + N)$
  - This would just be written using the much larger term: $O(N^2)$

Big O Notation - Tips

- Count loops!
    - Nested loops get multiplied together (outer loop for i in N, inner loop for j in M == O(N*M)
    - Don’t forget that comprehensions are loops!
- The time complexity of both DFS and BFS traversal is O(N + M) where N is number of vertices and M is number of edges in the graph.
- If you see a problem where the something is getting halved every step (binary search) your runtime is probably $O(\log_2(N))$

Sorting

Do not worry about learning all sorts (pun intended) of algorithms here.

If you are interested, learn just how one works (merge sort - Python uses a variant of it called Timsort).

All you need to know is that the optimal time complexity is O(N log(N)), for sorting N items.

Glad that’s settled. Moving on...

You don’t realize it...

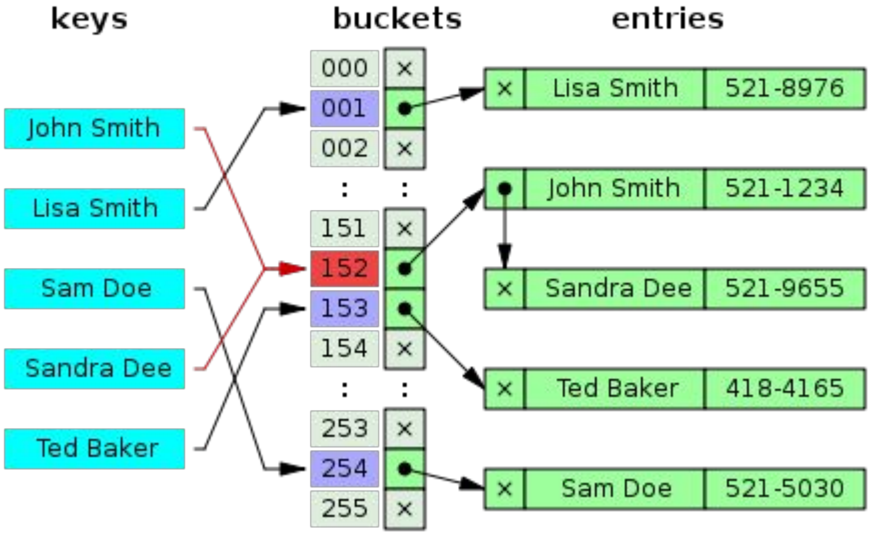
But you take for granted how easy it is to check things like:

If 1 in [0,1,2,3,4]:

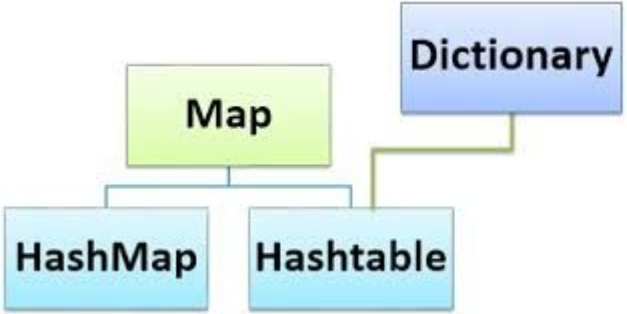
When this list is large, testing if a number is an element takes a very long time!

Hash tables

A hash table is a way to store data in a Key - Value fashion.



Dict vs Hash in Python



- In **Python**, **dictionaries** (or “dicts”, for short) are a central data structure: Dicts store an arbitrary number of objects, each identified by a unique **dictionary** key. **Dictionaries** are often also called maps, **hashmaps**, lookup tables, or associative arrays.
- Dictionary in Python is an unordered collection of data values, used to store data values like a map, which unlike other Data Types that hold only single value as an element, Dictionary holds key:value pair.
- Values in a dictionary can be of any datatype and can be duplicated
- Keys of a Dictionary must be unique and of immutable data type such as Strings, Integers and tuples, but the key-values can be repeated and be of any type.
- Note – Dictionary keys are case sensitive, same name but different cases of Key will be treated distinctly.

Dictionary (C)	Hash Table (C)	Hash Table (Java)	Hashmap (Java)
Only public static members are thread safe in Dictionary. Dictionary throws an exception if we try to find a key which does not exist.	Hashtable is thread safe. Hashtable returns null if we try to find a key which does not exist.	Synchronized no more than one thread can access the Hashtable at a given moment of time	Non-Synchronized
Data retrieval is faster than Hashtable.	Data retrieval is slower than dictionary because of boxing-unboxing.	Fast	Slow
Dictionary is a generic collection. So it can store key-value pairs of specific data types.	Hashtable is a loosely typed (non-generic) collection, this means it stores key-value pairs of any data types.	Allows one null key and more than one null values	Does not allows null keys or values

Dictionary Methods

- copy()
- They copy() method returns a shallow copy of the dictionary.
- clear()
- The clear() method removes all items from the dictionary.
- pop()
- Removes and returns an element from a dictionary having the given key.
- popitem()
- Removes the arbitrary key-value pair from the dictionary and returns it as tuple.
- get()
- It is a conventional method to access a value for a key.
- dictionary_name.values()
- returns a list of all the values available in a given dictionary.
- str()
- Produces a printable string representation of a dictionary.

- update()
- Adds dictionary dict2's key-values pairs to dict
- setdefault()
- Set dict[key]=default if key is not already in dict
- keys()
- Returns list of dictionary dict's keys
- items()
- Returns a list of dict's (key, value) tuple pairs
- has_key()
- Returns true if key in dictionary dict, false otherwise
- fromkeys()
- Create a new dictionary with keys from seq and values set to value.
- type()
- Returns the type of the passed variable.
- cmp()
- Compares elements of both dict.

Hash Tables - Python How To

Dictionaries can be created in two ways:

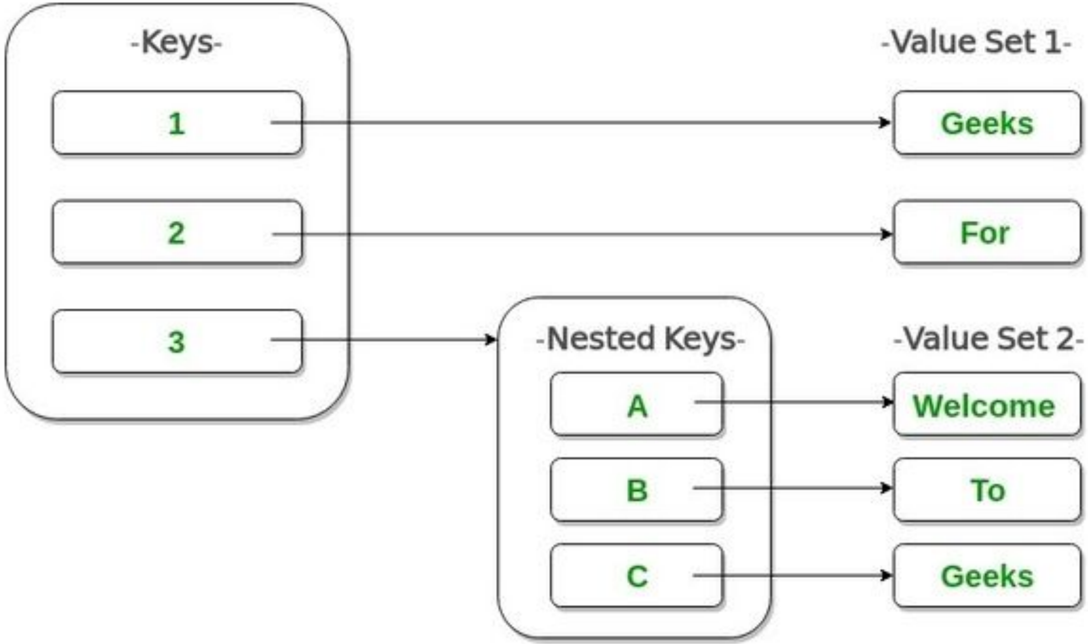
Using curly braces ({})

Using the dict() function

IT'S JUST A DICT!

```
hashTable = {}
hashTable[key1] = 'value1' #Store a value
print(hashTable[key1]) #Retrieve a value
if 'keyName' in hashTable: #Check Key existence
    #do something
```

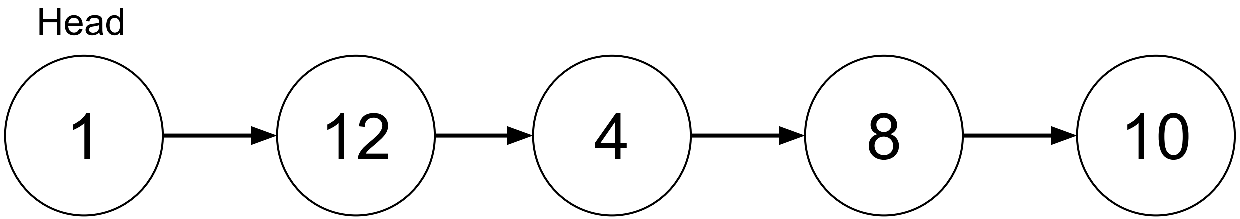
Nested Dictionary



Hash Tables - The Gist

1. Initialize array, each element is root of a linked list
2. Compute a Key’s **hash code** - different keys could have same code (collision)
3. Map hash code to index in your array - different codes could have same index (collision)
    a. Index is a function of the hash code: ex. hash_code % array_size
4. Store key and value at this index
5. Retrieval is just using key to find hash code and then index, then search linked list for value with this key
6. FAST - ***Worst Case = O(N), good collision resolution = O(1)**

Linked List

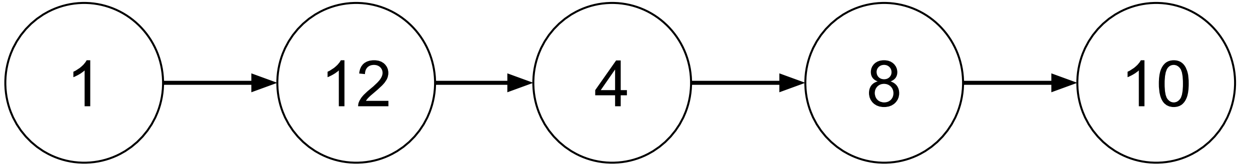


- Slow to access arbitrary element
- Fast to add and remove from the front
- Always access the data structure via the head node

Linked List - Python How To

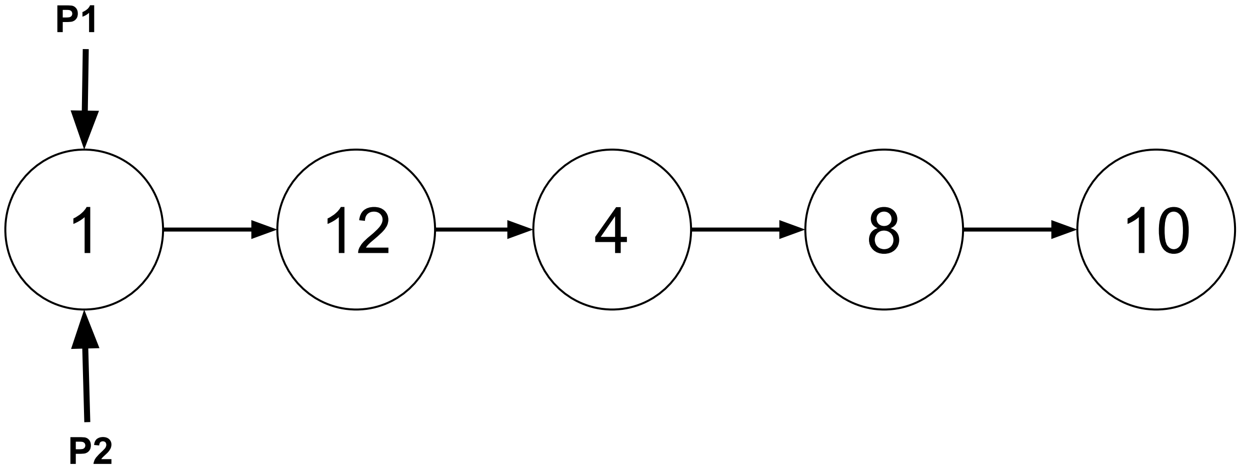
```python
class ListNode(object):
    def __init__(self, x):
        self.val = x
        self.next = None
```

Linked List - Runner Technique



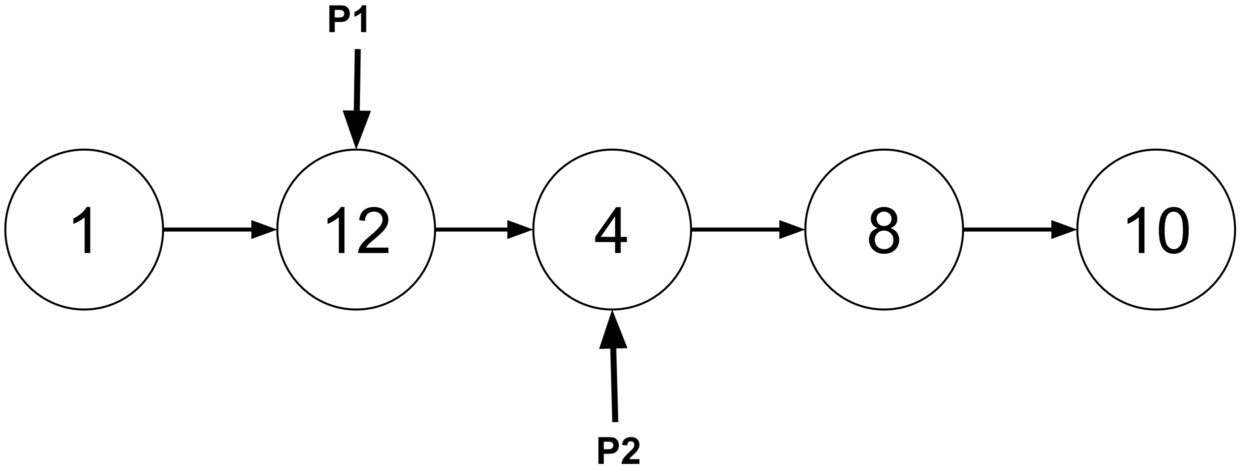
- Keep multiple pointers to a linked list with different offsets
- Example: find the middle element in a linked list

Linked List - Runner Technique



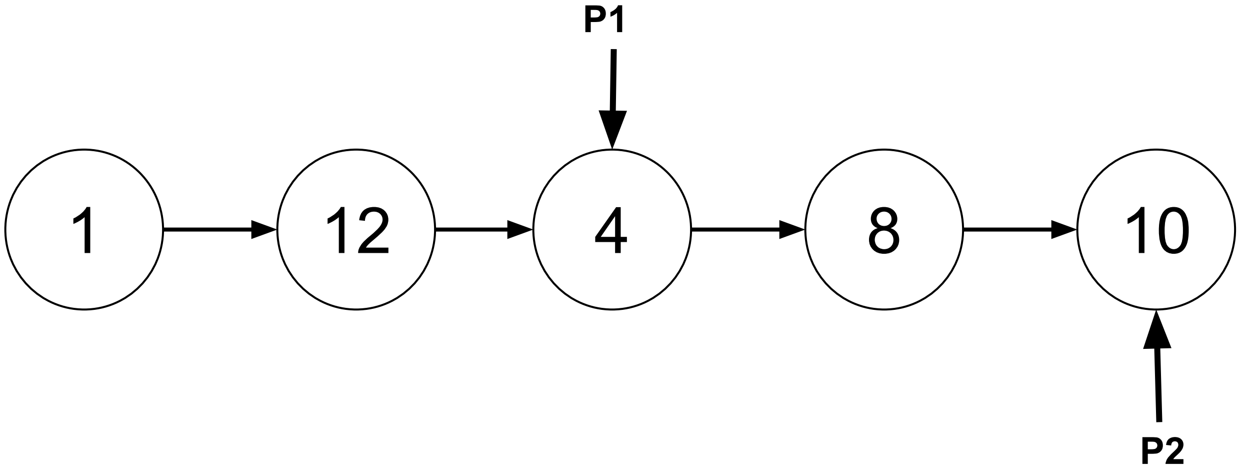
- Keep multiple pointers to a linked list with different offsets
- Example: find the middle element in a linked list

Linked List - Runner Technique



- Keep multiple pointers to a linked list with different offsets
- Example: find the middle element in a linked list

Linked List - Runner Technique



- Keep multiple pointers to a linked list with different offsets
- Example: find the middle element in a linked list

Stacks

Stack: Last in, first out (LIFO)

A stack of dishes is a stack. You can (well, should) only remove the one on top.

Example uses:

- Matching tags or parenthesis in text
- Reversing sequences
- “Undo” mechanism
- DFS Tree Traversal

Stacks - Operations

- Push: Add an element to the stack.
  - S.push(e): Add element e to the top of stack S.
- Pop: Remove an element from the stack and return it.
  - S.pop(): Remove and return the top element from the stack S; an error occurs if the stack is empty.
- Peek: Return the top element of the stack (don’t remove).
  - S.top( )
- IsEmpty: Return true/false
  - S.is empty( )
  - len(S)

Stacks - Python How To

USE A LIST!

```
stack = []

stack.append(3) #push O(1)

stack.pop() #pop O(1)

stack[ len(stack) -1 ] #peek O(1)

len(stack) == 0 #isEmpty
```

Queue

Queue: First in, first out (FIFO)

Similar to a line at a cash register.

Example uses:

- Resource allocation (scheduling)
- BFS Tree Traversal

Queues

Queue: First in, first out (FIFO).

The cash register line at the grocery store is a queue.

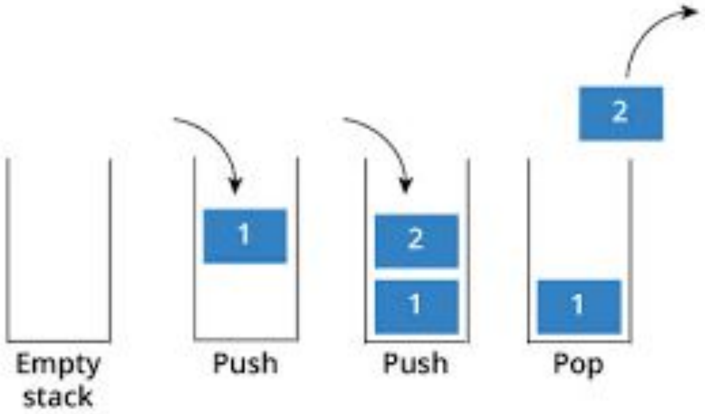
Example uses:

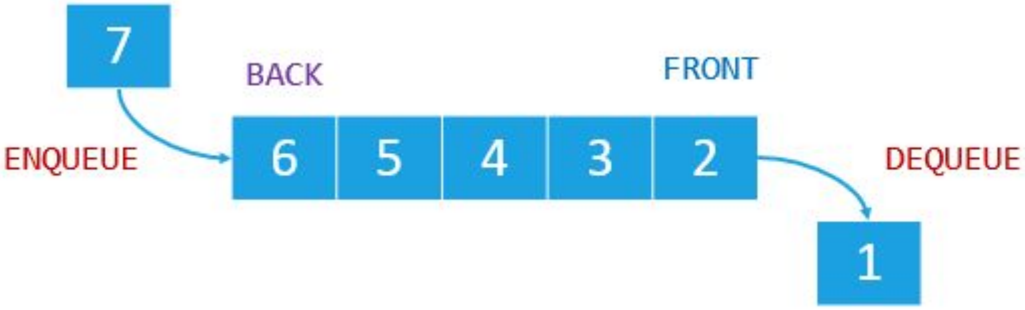
- Anything that requires waiting in line! (e.g. resource allocation)
- BFS Tree Traversal

Queues - operations

- Q.enqueue(e): Add element e to the back of queue Q.
- Q.dequeue(): Remove and return the first element from queue Q; an error occurs if the queue is empty.

- Q.first(): Return the element at front of the queue (don’t remove).
- Q.is empty( ): Return true/false
- len(Q)


2
1
2
1
1
Empty stack
Push
Push
Pop

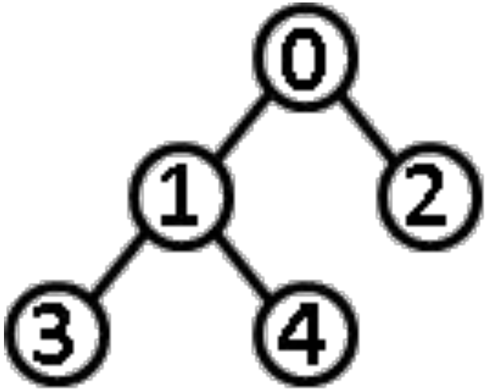

7
BACK
FRONT
ENQUEUE
6
5
4
3
2
DEQUEUE
1

Queue - Python How To

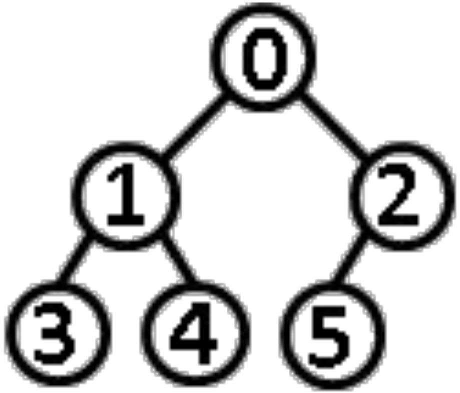
```
import collections

queue = collections.deque()

queue.appendleft(1) #enqueue

queue.pop() #dequeue
```

Binary Tree - Properties

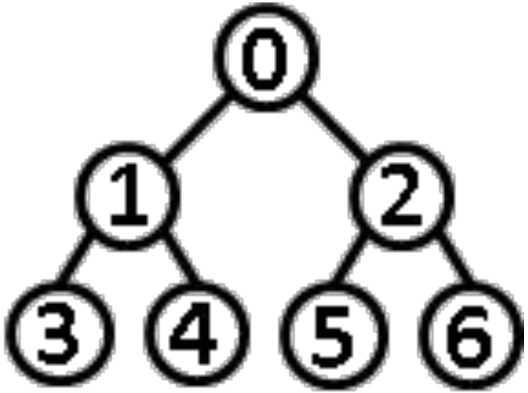
- Complete
  - Every level fully filled except perhaps the last level.
  - Last level is filled from left to right
- Full
  - Every node has either zero or two children
- Perfect
  - Both complete and full and last level is filled



full binary tree



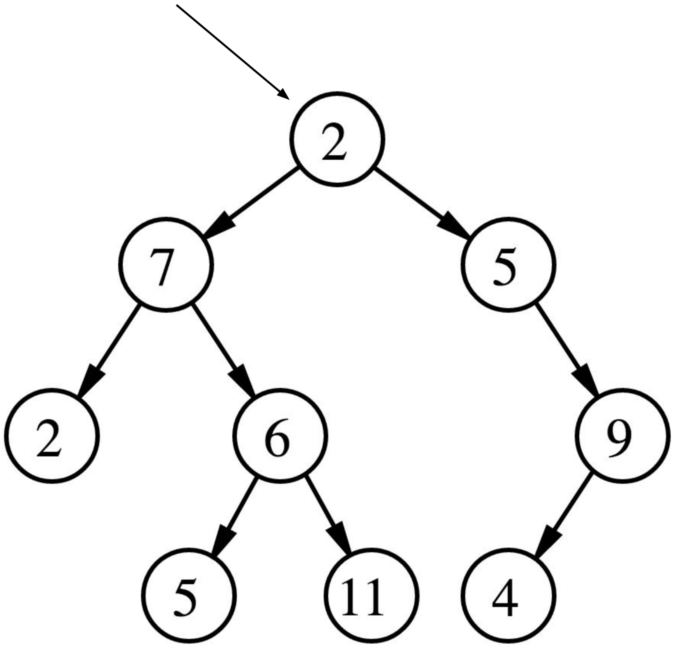
complete binary tree



perfect binary tree

Binary trees

A tree where each node has up to two child nodes, one left, one right. Similar to linked lists it's accessed through the **root node**.

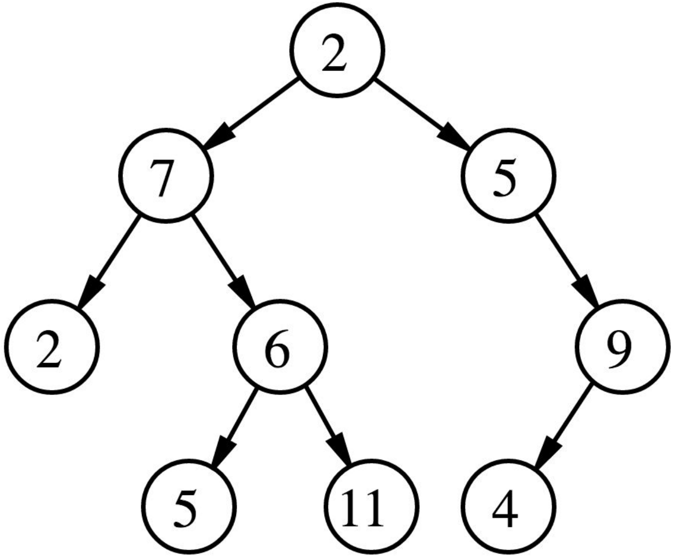


Binary trees

A tree where each node has a left and a right node.

There are many useful applications:

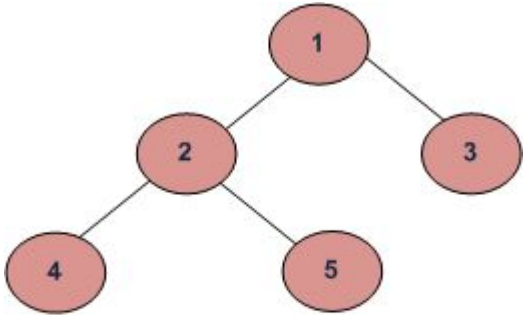
- Binary search trees
- Heaps
- Binary Tries
- Binary space partitioning
  - K-D Trees (what are they used for?)



Binary Tree Traversal

Depth First Search (DFS):

(a) Inorder (Left, Root, Right) : 4 2 5 1 3
(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1
You can use a stack for this.



Breadth First Search (BFS):

Level Order Traversal : 1 2 3 4 5
You can use a queue for this.

Binary Trees - Python How To

```python
class TreeNode(object):
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None
```

BFS - Exercise

Print the levels of a binary tree

e.g.

1

2,3

4,5

http://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/

Binary Tree - Variations

- Binary Search Tree (BST)
  - All left descendents are smaller than the current node
  - All right descendents are larger than the current node
  - Quickly search for an elements existence. O(log n)
- Heaps: **min-heap** or max-heap
  - Complete binary tree
  - Each node is smaller than its children
  - Therefore, the root is the smallest element
  - Quickly find/remove the smallest/largest element. O(1)

Heaps

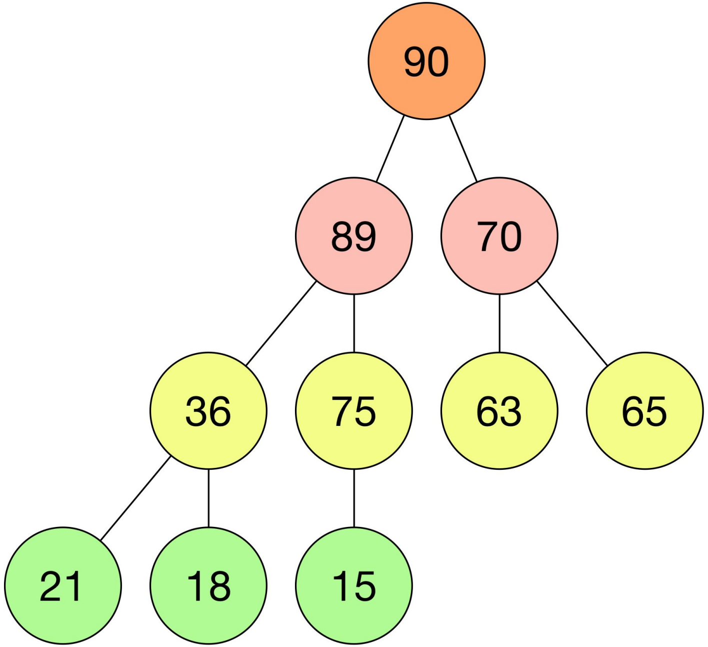
A special type of binary tree - it’s a complete binary tree that implements a “heap property”. The heap is filled left-to-right.

The “heap property” is that, for each node, all children have a larger (smaller) value than its own. There is no left-right requirement, only per-level.

After inserting a new value at the bottom level (again, left to right), the heap property is preserved by “percolating” (i.e. switching with its parent) this new node until the heap property is preserved. It’s essentially just checking if the new node has a larger value than the parent - if so, switch places with it.

Very useful for implementing a top N, bottom N, priority queue; also any time you need quick access to largest or smallest item.

Heaps



90	89	70	36	75	63	65	21	18	15
0	1	2	3	4	5	6	7	8	9

Given location *i*

Its left child:	$2i + 1$
Its right child:	$2i + 2$
Its parent:	$(i-1)/2$

Heaps - Exercise

Write an efficient algorithm for finding the top K largest values in a list with N entries.

(Hint: use a heap)

Resources

Resources

- Effective Python
- Cracking the Coding Interview
- Problem Solving With Algorithms and Data Structures in Python
  - http://interactivepython.org/runestone/static/pythonds/index.html#
- Donne Martin's Interactive Algorithms and Data Structures Coding Challenge
  - A large set of Jupyter notebooks.
  - Git repo.
- Bill Howe's Data Science Coursera course
  - Covers many topics, including MapReduce
  - Coursera.
- Geeks for Geeks website (link)
- Datacamp

Other topics not covered here

- Recursion and dynamic programming
- Graphs

Questions?