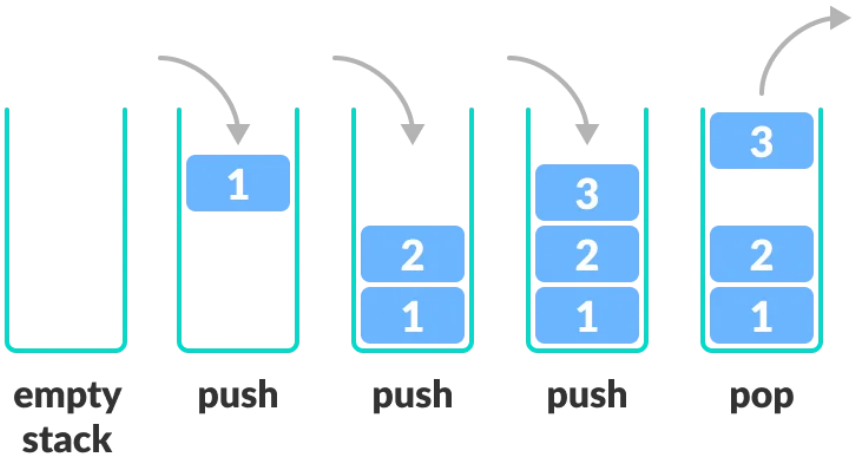
Data Structures (python3)

Stacks

Collection of objects that are inserted according to **LIFO** principles (last in is first out)

Elements can be inserted anytime, but only the most recently inserted object can be removed.



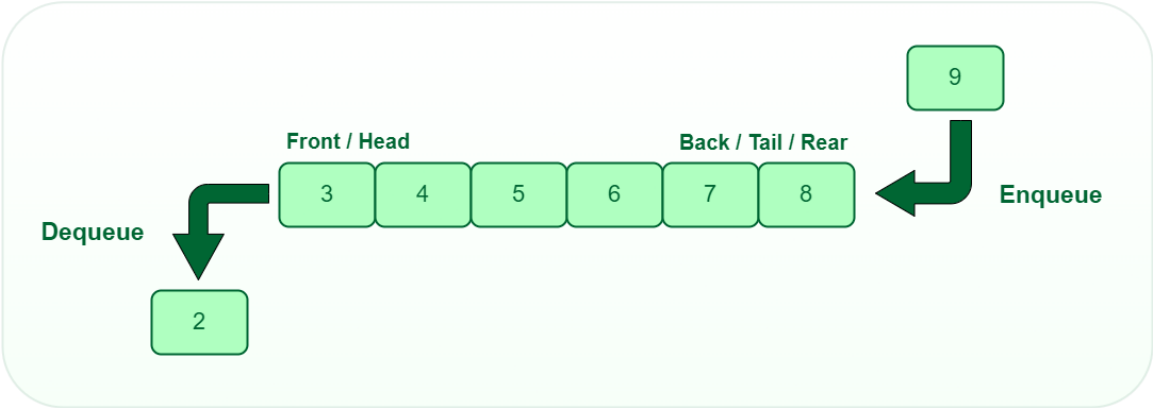

```
class Stack:
    '''Python implementation the stack'''

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

    def is_empty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def top(self):
        return self.items[len(self.items) - 1]
```

```
    def size(self):
        return len(self.items
```

Queues

Collection of objects that are inserted and removed according to the **FIFO** principle. (first in is first out).

Elements can be inserted anytime, but only the one that has been in the queue the longest can be removed.

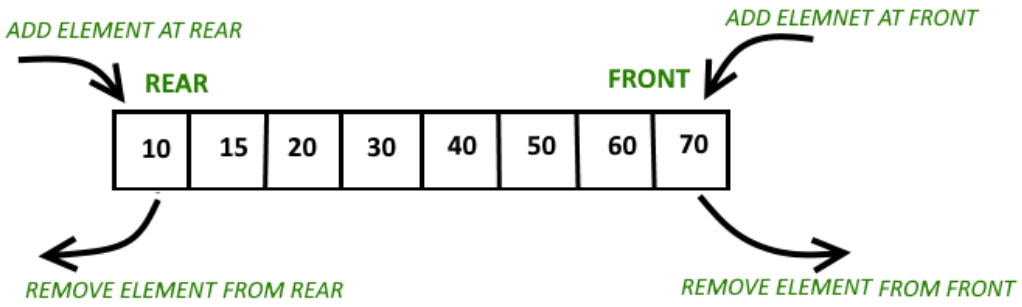



```
class Queue:

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

    def is_empty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        return self.items.pop()
```

```
    def size(self):
        return len(self.items)
```

Deques (Double-Ended Queues)

Queue-like data structure that supports insertion and deletion at both front and back of the queue.




```
class Deque:

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

    def is_empty(self):
        return self.items == []

    def first(self):
        return self.items[0]

    def last(self):
        return self.items[-1]

    def add_first(self, item):
        self.items.insert(0, item)

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

```
    def delete_first(self):
        self.items.remove(0)

    def delete_last(self):
        self.items.pop()

    def size(self):
        return len(self.items)
```

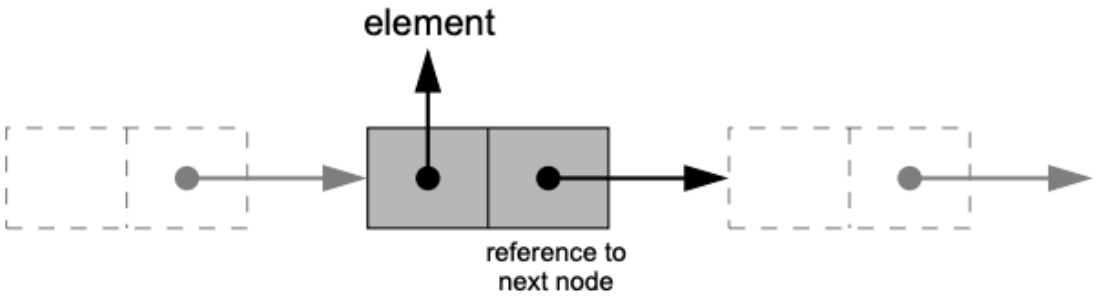
This implementation of Queues, Stacks and Deqeues relies on the adaptive pattern, i.e adapting a built-in data structure (python lists in this case) to the functionalities that we need to implement. Python lists are highly optimised, but they come with three disadvantages:

1. The length of a list might be longer than the actual number of elements it stores
2. Amortised bounds may be unacceptable in real-time systems
3. Insertion and deletion at interior positions are expensive

Linked lists provide a solution to these problems. but there's always a trade off with something else. For example, Elements of a liked list cannot be accessed at a numeric index.

Singly Linked Lists

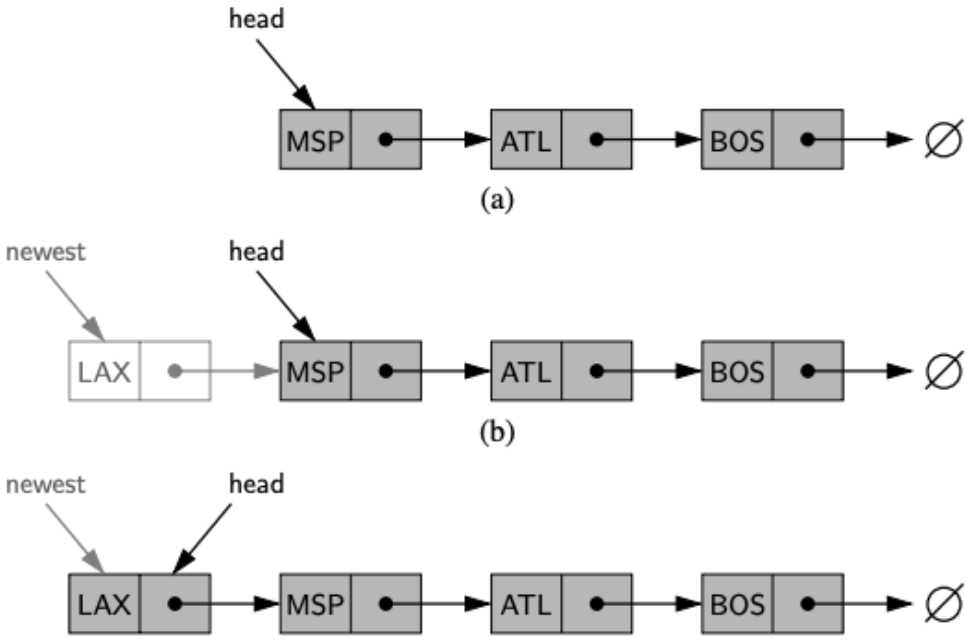
A collection of nodes that form a linear sequence. Each node stores a pointer to an object that is an element in the sequence, and a pointer to the next node of the list.



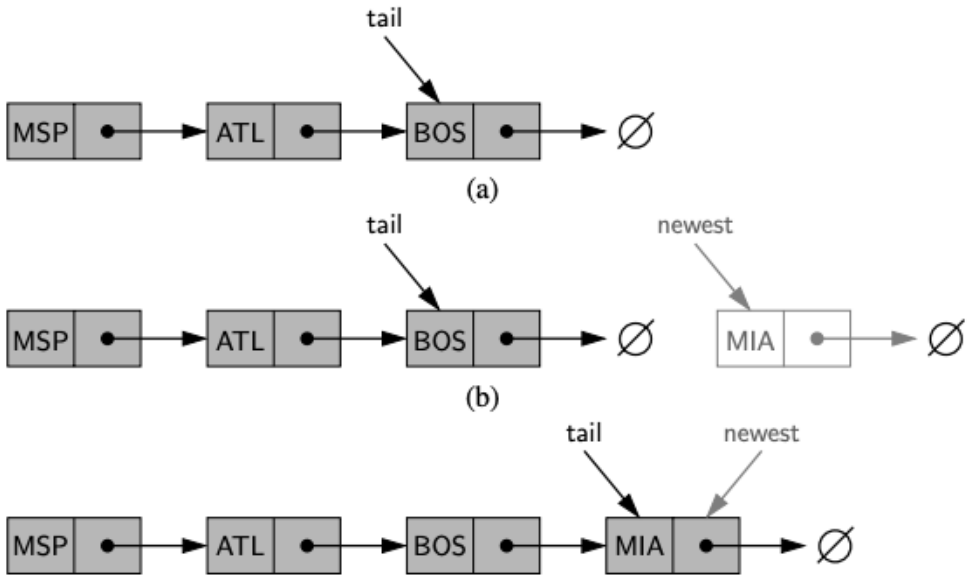
First and last elements are called the **head** and the **tail**. We can traverse the linked list by starting at the head until we get to the tail. The tail has None as its next reference.

Linked lists do not have a predetermined fixed size. We can insert an element at the head by setting it as the new head, and setting its next link to refer to the old head.

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



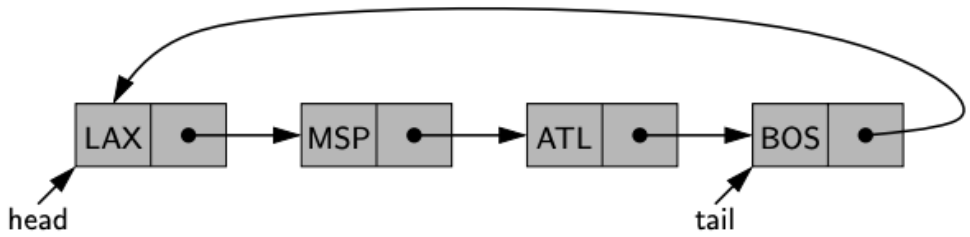
We can also easily insert elements at the end by replacing the current tail's next node (which is None) to the address of the new tail. We set the new tail's next node to None.



Removing an element from head or tail is essentially the reverse operation.

Circularly Linked Lists

Just like linked lists, but the tail's next object is the head, rather than None.

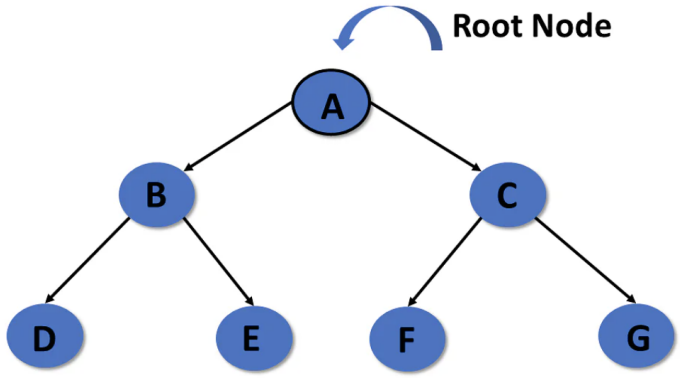


Trees

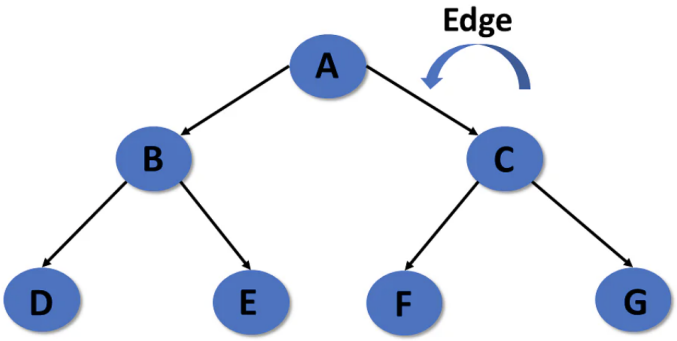
A tree is a data structure that stores elements hierarchically.

Definitions:

Root: The first node of a tree



Edge: The connection between two nodes. A Tree with n nodes will have n - 1 edges



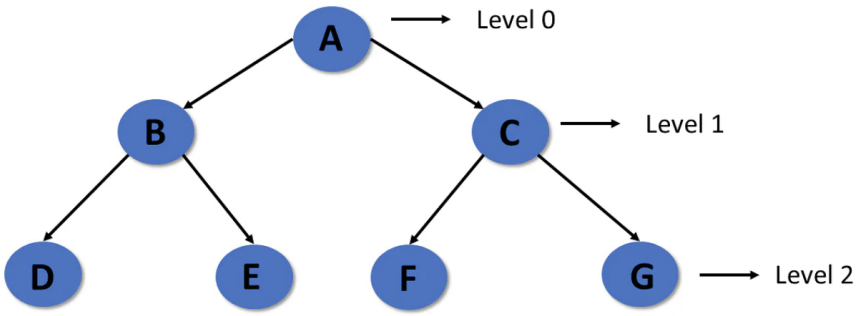
Siblings: Two nodes are siblings if they have the same parent

Leaf: (Or external nodes) A node with no children

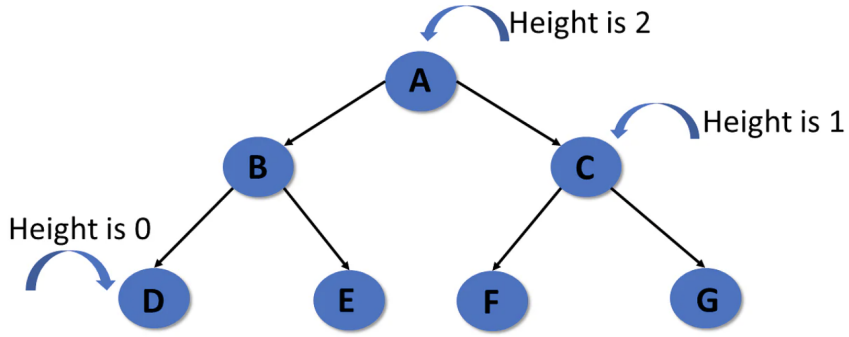
Internal node: A node with children

Degree: The degree of a node is the number of its children

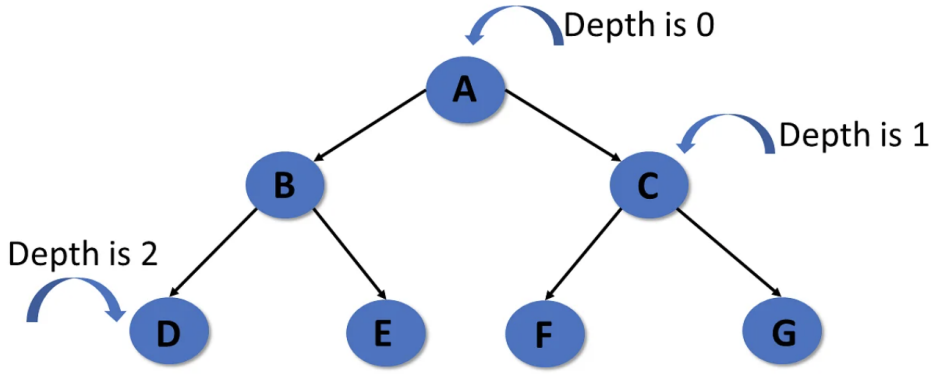
Level: The root node is at level 0, its direct children at level 1 and so on



Height: The number of edges from some leaf node to some other node:

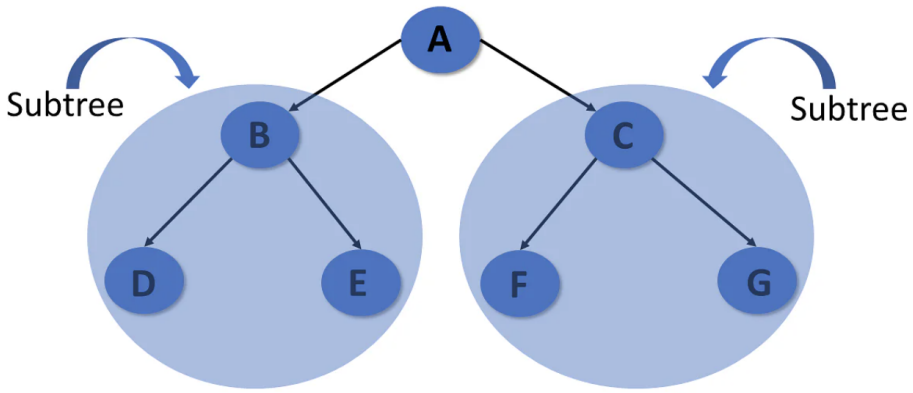


Depth: The number of edges between root and a node

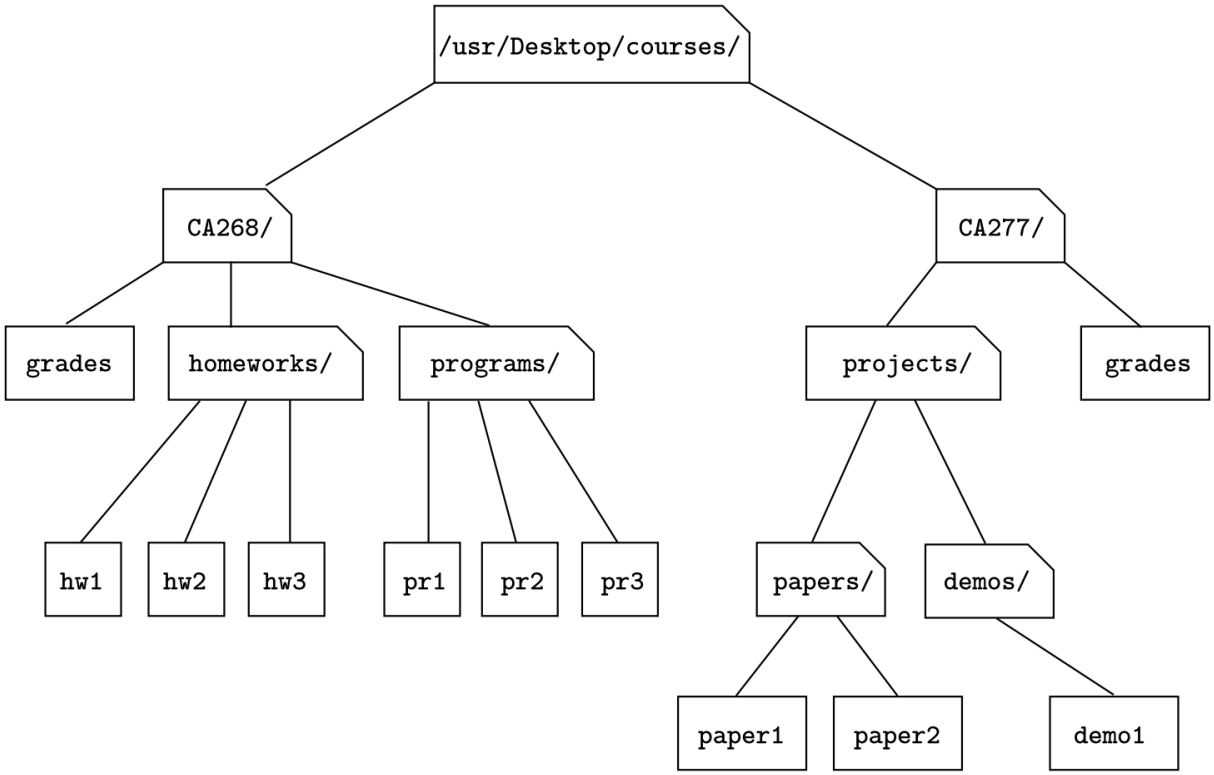


Path: The sequence of nodes and edges between two nodes

Subtree: Each child of a node makes a subtree



An example of a tree is UNIX's filesystem:



Computing depth and height

```
def depth(self, p):
""" Return the number of levels separating Position p from the root . """
    if self.is_root(p):
        return 0
    else:
        return 1 + self.depth(self.parent(p))
```

```
def height(self, p):
""" Return the height of the subtree rooted at Position p."""
    if self.is_leaf(p):
        return 0
    else:
        return 1 + max(self.height(c) for c in self.children(p))
```

Both these algorithms performs O(n).

Binary trees

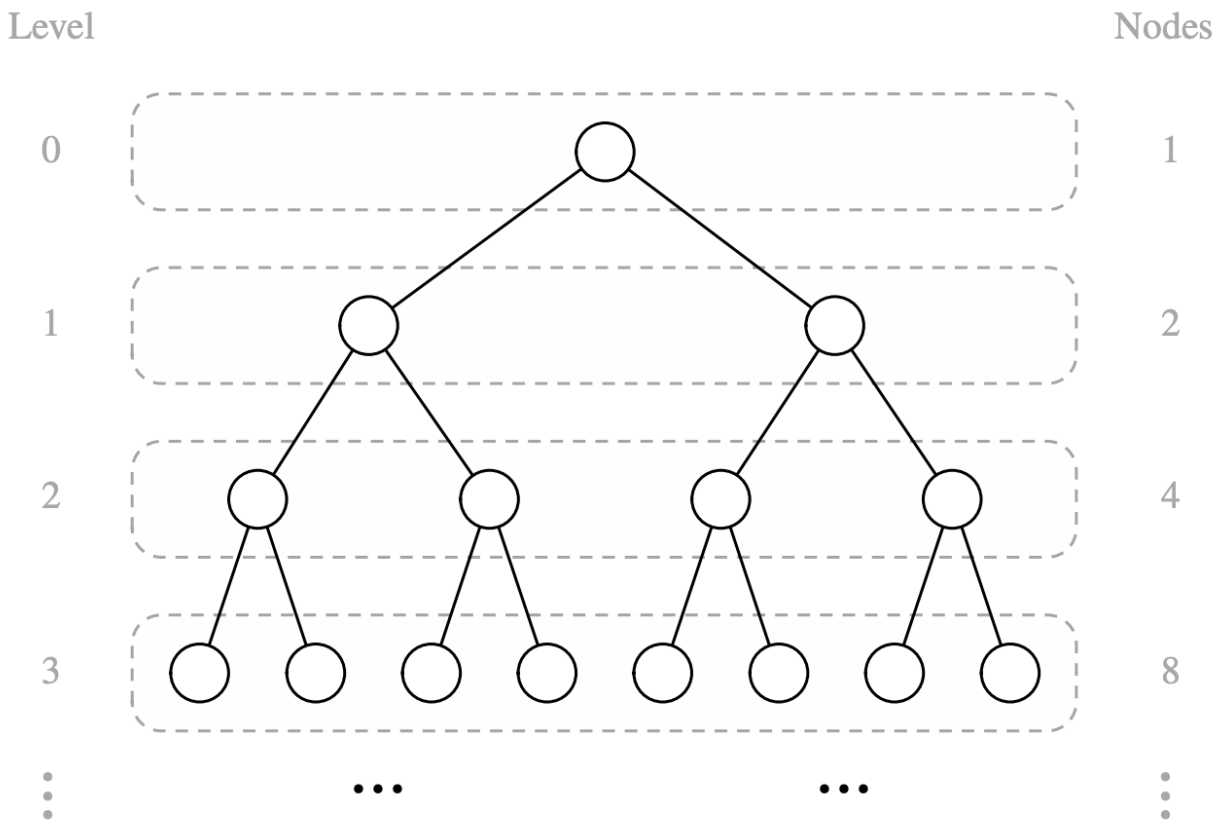
A binary tree is an ordered (meaning that the order of nodes matter) tree with the following properties:

1. Every node has at most two children
2. Every child node is labelled as either left or right child
3. A left child precedes the right child in the order of children of a node

A binary tree is proper if every node has either zero or two children.

Properties of binary trees

Level n of a binary three has at most 2^n nodes:



**Note to self: A probability tree is basically a proper binary tree*

A binary tree with n nodes will have at least $log2(n+1)$ levels

ˆ base 2

Tree traversals

Preordered traversal: Root → Left → Right

Postorder traversal: Left → Right → Root

Inorder traversal: Left → Root → Right

Breadth-first traversal: Visit all nodes at one level from left to right. Repeat for all levels.

info: https://www.freecodecamp.org/news/binary-search-tree-traversal-inorder-preorder-post-order-for-bst/#:~:text=For Inorder%2C you traverse from,subtree then to the root.

Maps

A data structure in which unique keys are mapped to associated values (Basically python's dictionaries).

Hash tables

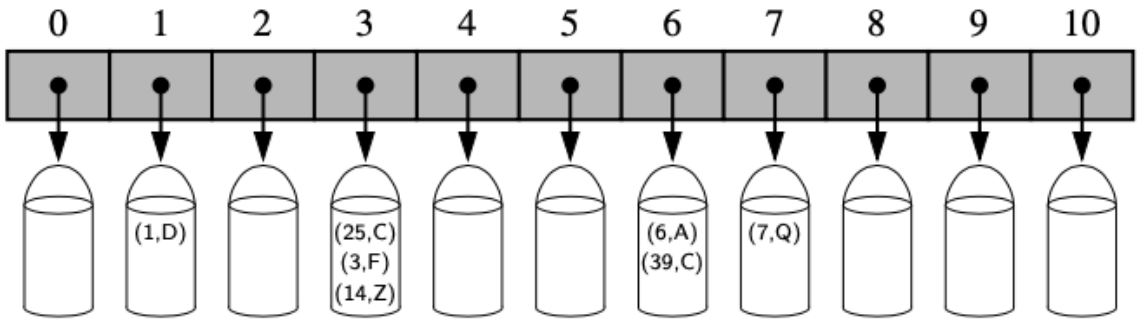
Hash tables are used to implement maps.

A hash table is an unordered collection of key-value pairs. Each key is unique.

Hash tables offer a combination of efficient lookup, insert and delete operations.

The keys are generated by using a **hash function.**

Hash tables are implemented by using a **bucket array**. Each bucket may contain a collection of items that are sent to a specific index by the hash function.



Hash functions

The goal of a hash function is to map each key k to an integer in the range [0, N - 1], where N is the capacity of the bucket array. In case of two or more keys with the same hash value, then two different items will be mapped in the same bucket (aka a **collision** has occurred).

Hash functions are made of two components:

1. A **hash code** that maps a key k to an integer
2. A **compression function** that maps the hash code to an integer with range [0, N -1]

Hash codes

There are three types of hash codes:

1. **Bit representation:** Uses an integer representation of the bits of the data
2. **Polynomial hash codes:** Generates a hash code by evaluating a polynomial that uses a the values of the key as coefficients and powers from 0 to len(key) of a prime number.

   (info here https://www.geeksforgeeks.org/string-hashing-using-polynomial-rolling-hash-function/)
3. **Cyclic shift hash codes:** Shifts the 5 leftmost bits representation of a key to the right

Compression functions

Goal: mapping the integer generated by the hash code to an integer in the range [0, N - 1] with the smallest possible amount of collisions.

There are two types of compression functions:

1. **Division method:** Maps an integer i to $i \ mod \ N$, where N is the size of the bucker array. Choosing N to be a prime number helps spreading the keys more evenly
2. **Mad method:** A more sophisticated compression function which helps eliminate repeated patterns. The method maps integer i to:

$$[ai + b]mod(p)]mod(N)$$

- N = size of the bucket array
- p = a prime number > N
- a and b = random integers from the interval [0, p - 1], a > 0

Ideally, the probability of collision of a compressing function would be of 1/N. This method gets close to that.

Efficiency of hash tables

Operation	List	Hash table
get item	O(n)	O(1)
set item	O(n)	O(1)
del item	O(n)	O(1)
len	O(1)	O(1)
iteration	O(n)	O(n)

Sorted maps

A sorted map has all the same functionalities of a normal map, plus:

- **M.find_min()** - Return the (key, value) pair with minimum key
- **M.find_max()** - Return the (key, value) pair with maximum key
- **M.find_lt(k)** - Return the (key, value) pair with the greatest key < than k
- **M.find_le(k)** - Return the (key, value) pair with the greatest key ≤ than k
- **M.find_gt(k)** - Return the (key, value) pair with the smallest key > to k
- **M.find_ge(k)** - Return the (key, value) pair with the smallest key ≥ to k
- **M.find_range(start, stop)** - Iterate all (key, value) pairs with start ≤ key < stop.
- **iter(M)** - Iterate all keys of the map according to its order
- **reversed(M)** - Iterate all keys of the map in reverse order

---

Sorted search tables

Simplest implementation of a sorted map.

Stores the map’s items in an array so that they are in increasing order of their keys (assuming that they keys have a naturally defined order).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

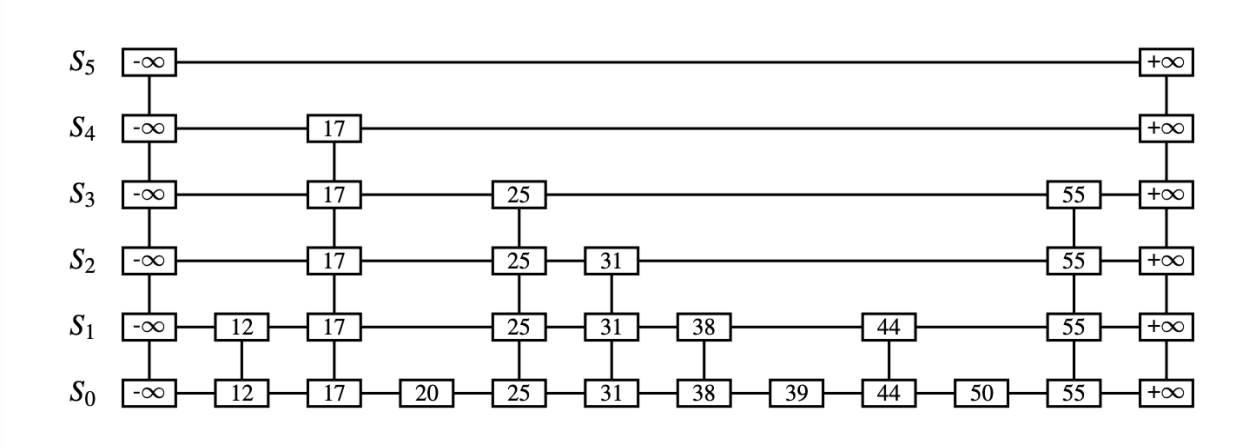
Skip lists

A data structure used to implement sorted maps.

A skip list S for a map M consist of a series of lists {S0, S1, … Sh}. Each list Si stores a subset of the items of M sorted by increasing keys, plus two sentinel items denoted -∞ and +∞, where -∞ is is smaller than every possible key that can be inserted in M and +∞ is larger than every possible key that can be inserted in M. In addition:

- List S0 contains every item of the map M (plus the sentinels -∞ and +∞)
- For i ∈ {1, 2, … , h}, Si contains a randomly generated subset of the items in list Si - 1 (plus the sentinels -∞ and +∞)
- List Sh contains only the sentinels (-∞ and +∞)

Thus, h is the height of the skip list.



The skip list is implemented such that for each item n in list Si, the probability of n being in Si+1 is 1/2. Therefore, on average, the size of Si+1 will be half of the size of Si.

Note that:

- List Si will have on average n/s^i items
- The height h of S will be on average log n

Traversing a skip list

We view a skip list as a two-dimensional collection of positions arranged horizontally into **levels** and vertically into **towers**. Each level is a list Si and each tower contains positions storing the same item across consecutive lists.

For a given position p into a list Si:

- **next(p)** - Returns the position following p on the same level
- **prev(p)** - Returns the position preceding p on the same level
- **below(p)** - Returns the position below p in the same tower
- **above(p)** - Returns the position above p in the same tower

If the above operations return None then the position requested does not exist

Searching a skip list

Suppose we search for a key k. We start by setting a position p to the topmost left position of the skip list S. So the start position is the position of Sh storing -∞. Then, we perform the following steps:

1. if S.below(p) is None, the search terminates. Otherwise, we drop down to the next level by setting = S.below(p)
2. Starting at position p, we move p forward until it is at the rightmost position on the present level such that key(p) ≤ k (forward step).
3. Return to step 1

Note that even if k is found high up in a tower, the search continues until we reach S0.

More info here: https://www.youtube.com/watch?v=hqHwQUdTgLM&ab_channel=TylerProgramming

Insertion in a skip list

To insert a (k, v) pair:

1. Search for the position p with the largest key ≤ to k.
2. If key(p) = k, the associated value is overwritten with v. Otherwise, we insert (k, v) immediately after position p within S0, and create a new tower. The height of the tower is decided by repeating coin-flips until we get a tail.
3. We then link together all the references to the new item (k, v) created in this process.

Removal in a skip list

To delete a (k, v) pair:

1. Search for the position p with key = k
2. Remove p and all positions above it in the tower

if k is not found, raise a keyError

Efficiency of skip lists

Search	O(log n)
Insertion	O(log n)
Deletion	O(log n)