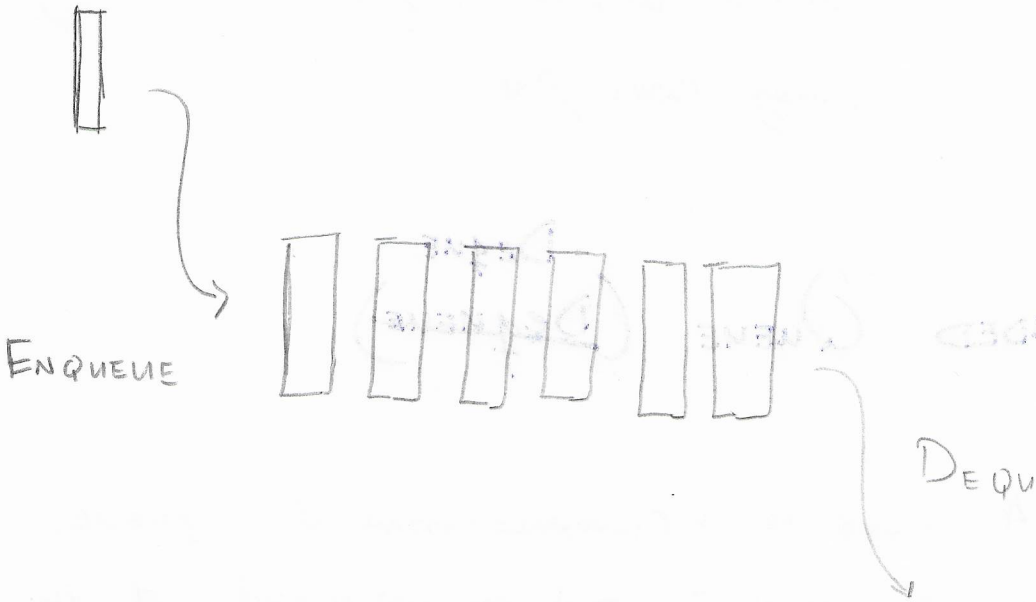
# Data Structures

1) Array
2) Stacks
3) Queue
4) Linked Lists
5) Trees
6) Graphs
7) Hash tables
8) Tries

# Queue




**Main Objective:** To keep order of a collection of items (First in First out).

**Main Operations:**

1. Dequeue the first element
2. Peek the first element
3. Enqueue: Add an element to the end of the collection

* Queues can be used as an Implementation for BFS.

**Implementing Queues:** Linked Lists are an efficient way to implement Queues (especially Doubly linked lists). However, arrays and dynamic Arrays can also be used.

doubly linked list: O(1) insert & delete @ Beginning/end

↓

Singly linked list: only O(1) insert @ start/beginning. If you keep a pointer to the last node this makes it O(1) @ the end also, and more memory efficient than doubly linked list.

# Double-ended Queue (Deque)

Objective: A deque is a generalization of a queue. It allows for insertion/deletion on both ends of the queue.

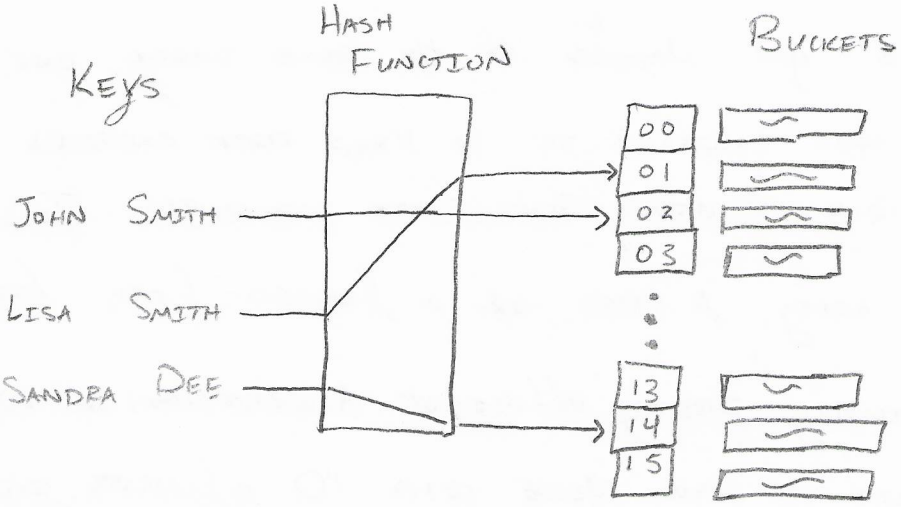
* A deque should be implemented using a dynamic array or doubly linked list.

## Operations

In general, the following operations are used for Deques (and stacks, since they can be thought of as a specific case of a double ended queue).

| | C++ | Python |
|---|---|---|
| insert @ end | push_back | append |
| insert @ front | push_front | appendLeft |
| remove @ end | pop_back | pop |
| remove @ front | pop_front | popleft |
| peek @ end | back | <obj>[-1] |
| peek @ front | front | <obj>[0] |

# Hash Table




Summary: Essentially what happens is that keys are passed into a hash function that maps them to an index of an array where the values are stored. It's best practice to have a hash function that outputs a uniform distribution of hash values, otherwise it's more likely that the hash function will map two keys to the same index — this is called a collision.

## Dealing With Collisions

→ Even w/ perfect uniform random distribution, there is still likely to be a collision — this is the birthday problem.

↓

## 1) Separate Chaining

If two keys get mapped to the same index, one way to handle this collision is to keep both entries in the same bucket using some data structure. It's possible to use an array, a tree, or a linked-list, for example. This is actually pretty efficient according to Wikipedia. If you start to have more than 10 elements per bucket, you should look into a better hash function (if it's non-uniform), or a more efficient data structure inside the buckets, like a balanced search tree.

## 2) Open Addressing

Upon a collision, the program can just search linearly for the next available bucket. This is fine for small amounts of data. Note that this doesn't fix the collision per-se. That is, the keys are still mapped to the same bucket and when searching for an entry, the same linear search is performed until the key matches the key in the bucket.

## 3) Variations & Hybrids

Other ways are by employing some Hybrid of separate chaining and open addressing. It's also possible to use several hash functions for the same table.

## Resizing

Note that if all buckets fill up or more/less space is needed, then actions can be taken to resize the table.

## Pros and Cons

1) Searching, insert, and delete are <u>fast</u> compared to other table data structures. This is especially true when

   (a) We have a lot of items

   (b) We initialize the hash table w/ all the key, values at the beginning. (This helps prevent collisions).

When we only have a few items an array or something else could be more efficient.

### Cons

1) Since key, value pairs are stored (nearly) randomly in memory, it is not efficient to find the "next" key-value pair.

2) Hash tables have poor locality of reference.

* Hash tables have <u>amortized</u> $O(1)$ search, delete, and insert.
  ie. Virtually all the time it will have constant time operations, but hash collisions and resizing can occur which will degrade search to $O(n)$ in worst case.

# Stacks

★ Think of stacks like a stack of plates. The first plate that gets put in the stack will be at the bottom and thus be the last to come out. (LIFO; Last In First Out).

## Typical operations

1) push – add an element to the top of the stack
2) pop – delete the top element off the stack
3) peek – gives the top element in the stack, or maybe a pointer / index value of the top element
4) Cases need to be handled for when pop and peek are used on an empty stack and push on a stack that is full

## Implementations

→ Stacks can be implemented using arrays, dynamic arrays, or linked lists.

→ Dynamic Arrays are very efficient for stacks since it can grow and shrink as needed, and insert/delete has amoritize O(1) time.

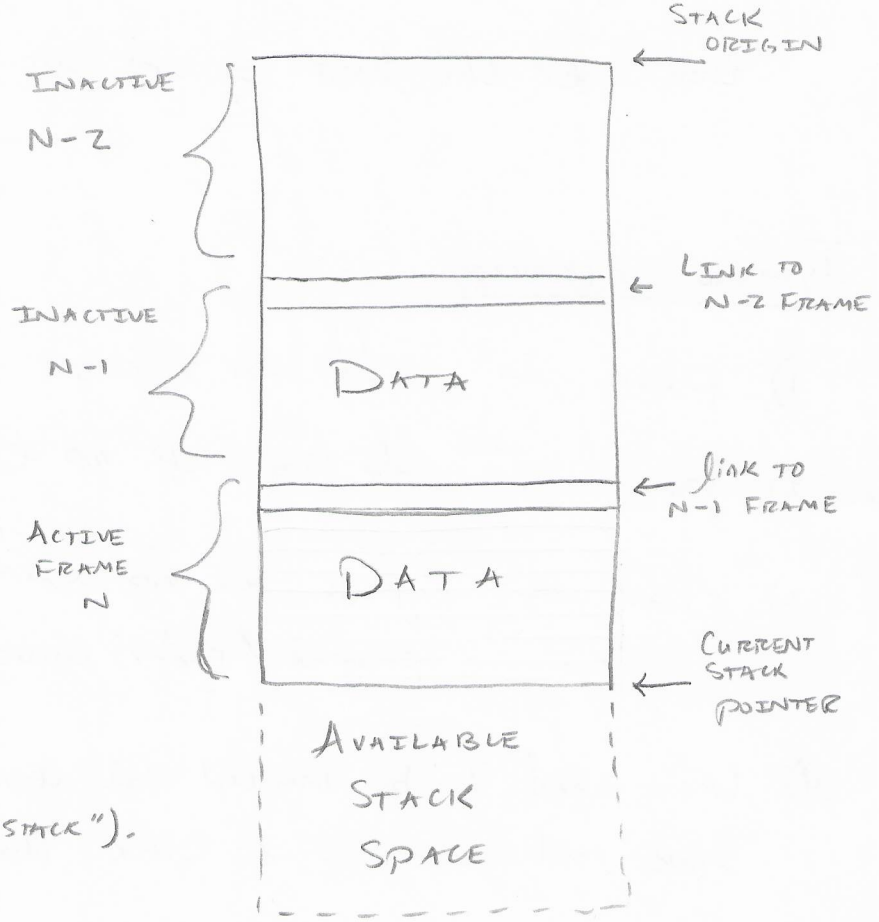
# Stacks in Hardware

→ Stacks are typically an area of computer memory w/ a fixed origin and a variable size.

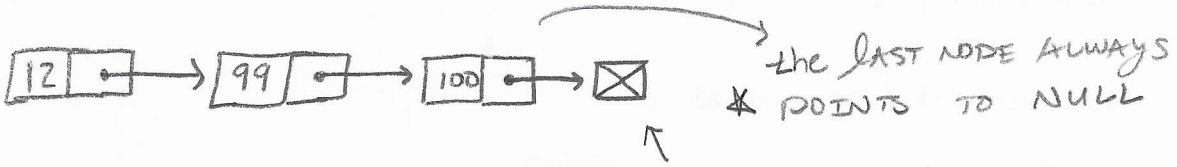
→ If push() is called on a full stack, we get a stack overflow

→ if pop() is called on an empty stack, we get a stack underflow.

→ Stacks are used a lot for compile-time memory management, and for subroutines (think of the "call stack").



# Linked List




Linked list node has:

1) A value
2) the Memory Address of the next node

Differences between arrays & linked lists:

1) Arrays are initialized w/ fixed amount of memory, where as a linked list just points to a node.

2) It is easier to insert/delete in a linked list than in an array. Linked lists are better for Dynamic Manipulation.

3) Arrays are way faster to search, linked lists are much slower for searching.

4) Linked lists use more storage, arrays are more efficient

# Complexity

Indexing: $O(n)$

Insertion/Deletion:

→ At beginning: $O(1)$

→ At end: $O(1)$ if last element is known, else $O(n)$

→ In the middle: $O(n+1)$

Space: $O(n)$

## Dynamic Array vs. Linked List:

1) Although both can handle dynamic manipulation, linked lists are faster. Dynamic arrays still have to shift elements around to insert or delete.

2) Dynamic arrays offer $O(1)$ random access, so searching is way faster.

* Essentially this:
  - Arrays - best for searching
  - Linked Lists - best for insertion/deletion

DESTRUCTORS: ESSENTIALLY A CLASS METHOD EXPLAINING HOW TO DECONSTRUCT A CLASS WHEN IT IS DESTROYED IN MEMORY

INHERITANCE:

```
CLASS BASE {
...
}

CLASS DERIVED : PUBLIC BASE {
...
}
```

POLYMORPHISM: CONSIDER A BASE CLASS & A DERIVED CLASS WHERE THE DERIVED CLASS OVERRIDES A FUNCTION OF THE BASE CLASS.

```
BASE b
b.PRINT() ⇒ CALLS BASE FUNC

DERIVED d
d.PRINT() ⇒ CALLS DERIVED FUNC

BASE * bp = new DERIVED();
bp.PRINT() ⇒ BASE.PRINT()

BASE & br = new DERIVED();
br.PRINT() -> BASE.PRINT();
```

Polymorphism: References & pointers use functions based off the Type they were defined as instead of the INSTANCE they're pointing to.

Virtualism: Tells C++ compiler to use the function of the class at the bottom of the hierarchy.

## Dynamic Memory

Stack: how memory is usually created & destroyed

Heap: this is where dynamic memory is usually placed using the keyword "new"

"new" is requesting to allocate memory on the Heap.

"delete": deletes the VALUE. The variable pointing to the value still exists on the stack.

* Never return a pointer or reference to a local variable
* Only return a pointer to indicate position

① LINUX OS, PYTHON / C++, good IDE w/ debugging, INTELLISENSE

② (a) SORT FIRST then COMPARE
(b) BIT SET

③ like: EASY, SIMPLE, lots of LIBRARIES
DISlike: ABSTRACTS A LOT AWAY

④ A CONCISE AND FAST WAY TO CREATE LISTS

⑤ WHEN NEW MEMORY IS ALLOCATED dynamically BUT NEVER DEALLOCATED

⑥ POINTER holds the MEMORY ADDRESS
REFERENCE points to the SAME ADDRESS — IT's like SOMEONE W/ TWO NAMES

⑦ Ref to Ref : No
POINT TO POINT : yes

⑧ (a) RUNNING A SANDBOXED VERSION of the code
(b) log FILES
(c) STRATEGIC BREAK POINTS
(d) EVALUATING the CALL STACK

① As a software dev, what is your ideal work environment?
eg. which OSs, which IDEs, etc.

② We give you an array of 100 elements (ie. #'s 0 through 99), but some #'s are missing. How do you find the missing #'s?

③ What do you like & dislike about python?

④ What is list comprehension in python?

⑤ What is a Memory Leak?

⑥ Difference between a pointer and a Reference?

⑦ Can you make a Reference to a Reference? A pointer to a pointer?

⑧ How do you debug a program?