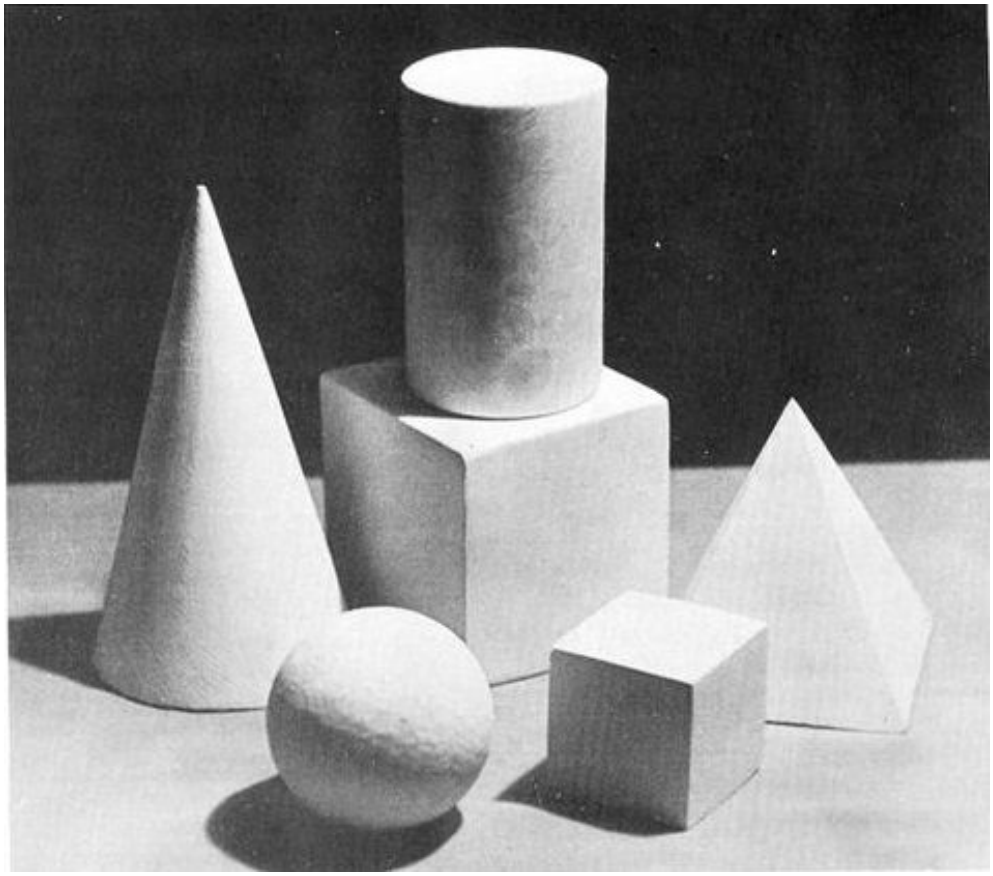
# Data Structures and Algorithms

*A brief study guide*



By

Jonathan Y. Huang

# Table of Contents

# Linked Lists

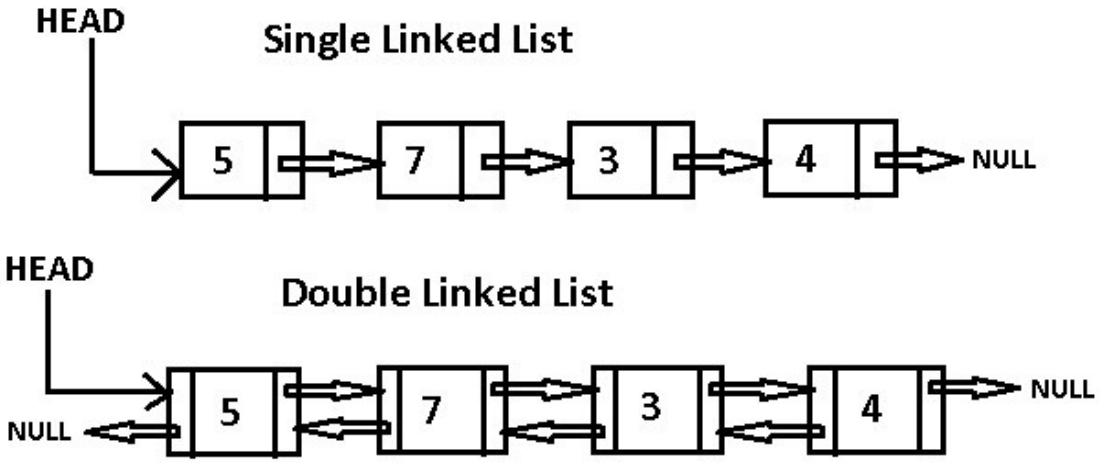
*They're like lists, but linked*

| | Insertion | Deletion | Search | Space |
|---|---|---|---|---|
| **Complexity:** | O (1)* | O (1)* | O (n) | O (n) |

* Provided you have a reference to the node/location you want to be deleted/inserted. Otherwise, you will have to traverse the list, which is O (n)

## Description:

A linked list is simply a sequence of nodes. Each node consists of a value and one or two pointers.




Source: https://medium.com/journey-of-one-thousand-apps/data-structures-in-the-real-world-508f5968545a

## Implementation:

A linked list can be implemented easily with a single node class*. Your node object should have a data attribute and a next attribute that is a node. You can add more methods for insertion and deletion depending on your use case, but they are not necessary.

Ex.

```
1 struct node
2   {
3     int data;
4     node *next;
5   };
```

## Applications:

Storing utterable data that gets appended often and other future data structures. Ex. Image viewers, storing forward/backward pages in a browser, stacks, queues, hash maps

## Know how to:

- Mutate a linked list (Append, delete, change value)
- Loop/ recurse through a linked list

## Details:

- Head: The head is the first item of a linked list, it is used to keep track of the start of a linked list
  - * If you are just using a node class, then trouble may arise if you have references to a head and then that head changes. This can be mitigated by creating a wrapper class that keeps track of the head.

### Arrays vs Linked Lists

- Similarly to arrays, access to a linked list is O(1). However unlike arrays, you may only access the current nodes data and must iterate to other values.
- You can append a linked list in O(1) time, unlike arrays which can not be appended.

### Tips

- When iterating/recursing through a linked list, the end/base case is when the current node or current node's next is equal to Null
- Sometimes it may be helpful to have two pointers in an array moving at different speeds

## Interview Problems:

- Return the Kth to last element of a singly linked list
- Given two linked lists, return the smallest node.

# Stacks and Queues

*You're gonna have to wait in line for this one*

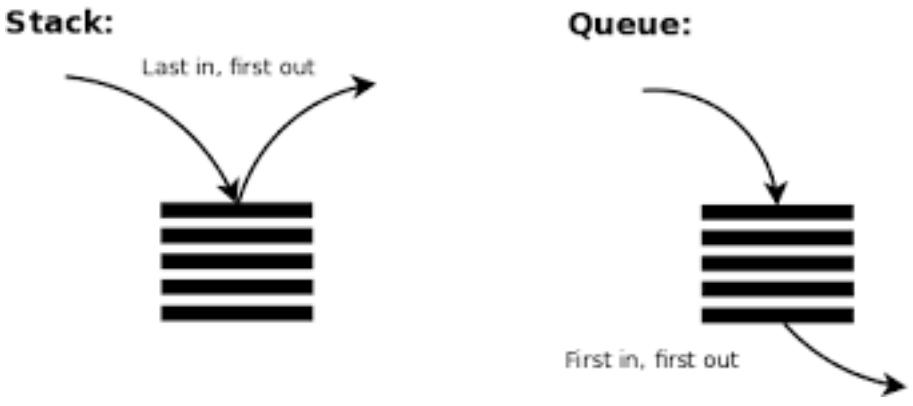
| | Insertion | Deletion | Search | Pop | Space |
|---|---|---|---|---|---|
| **Complexity:** | O (1) | O (1) | O(n) | O(1) | O (n) |

*While searching is technically O(n), because you must go through the entire structure in order to find your value, accessing the top value is O(1)

## Description:

Stacks and queues are data structures in which only the top item can be accessed. While this may seem like a limitation, it also allows constant time interaction. In fact, this "limitation" actually lends itself very well to certain algorithms, and makes stacks/queues a very powerful data structure.

- Stacks: stacks implement last in first out ordering (LIFO), ie. The last item inserted will be the first one accessible. Think of a stack of dishes, the first dish in the stack gets washed last.
- Queues: queues implement a first in first out ordering (FIFO), ie the first item in is the first item accessible. Think of a line at the movies, the first person in line gets their ticket first




Source: https://medium.com/x-organization/stack-and-queue-60f365963552

## Implementation:

Stacks and queues can be implemented with a linked list. Each item is placed in a node and then appended to either the front or the back of the list depending on if it is a stack or queue. To access the stack/queue the value of the current head is returned.

Ex.

```
class stack{
    node* head;

    public:

    void push(int x){
        node* temp = new node();
        temp -> data = x;
        temp -> next = head;
        head =  temp;
    }

    int pop(){
        int popped = head -> data;
        head = head -> next;
        return popped;
    }

};
```

## Applications:

Stacks are powerful for the ability to match patterns, they are commonly used in syntax parsing and regular expressions. Queues are useful for keeping things in order, so they are used for scheduling tasks for a CPU or depth first search.

## Know how to:

- Implement a stack or queue and the following methods:
    - pop( ): removes the top item (and depending on the requirements returns it).
    - push(item): adds an item to the stack/queue.
    - peek( ): returns the top item of the stack/queue.
    - isEmpty( ): Returns true if the stack/queue is empty.

## Details:

- Stacks and queues can also be implemented with arrays. This offers more efficient memory usage over linked lists.

    - However, an array based implementation will not be able expand dynamically like a linked list based implementation would be able to.

### Tips

- Stack/queue problems can be some of the harder questions to answer, so it is very important to emphasize the LIFO/FIFO nature when generating a solution. Think, "How does a stack/queue specifically help me in this situation?", "Does this problem exhibit anything similar to LIFO/FIFO?"

## Interview Problems:

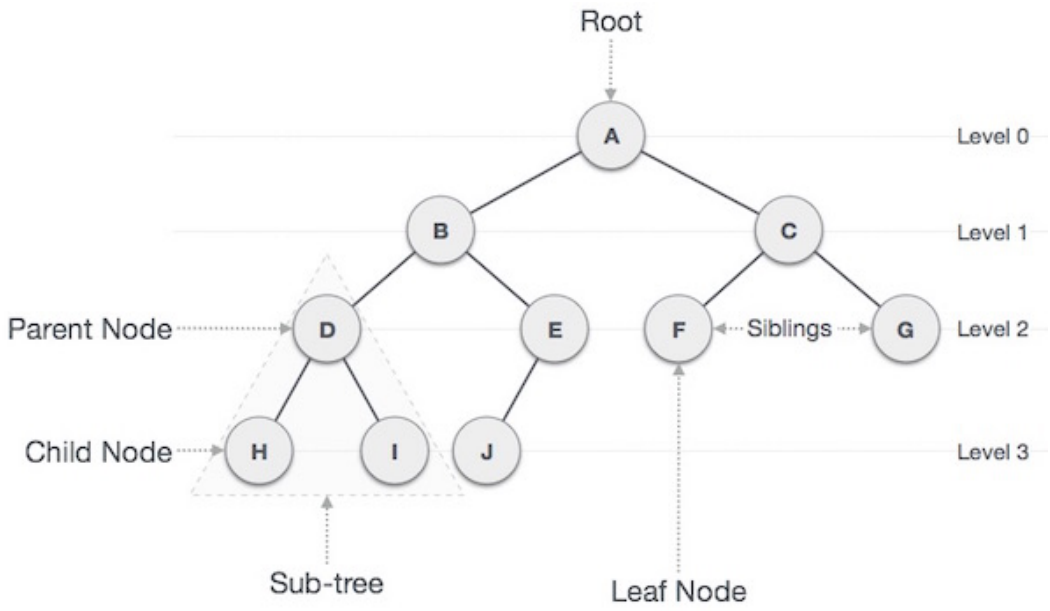
- Implement a queue with two stacks.
- Implement a stack with the function min( ), which returns the current min element of the stack in O(1) time.

# Trees

*It's the root of a lot of problems*

## Description:

A tree is a non linear data structure consisting of nodes. Each node has a parent and children (like a family tree).




Source: https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm

## Implementation:

There are very many different types of trees, each with different implementations, but this is a very basic implementation of a tree, a node with data and children.

Ex.

```
1 struct node
2   {
3     int data;
4     node children[x];
5   };
```

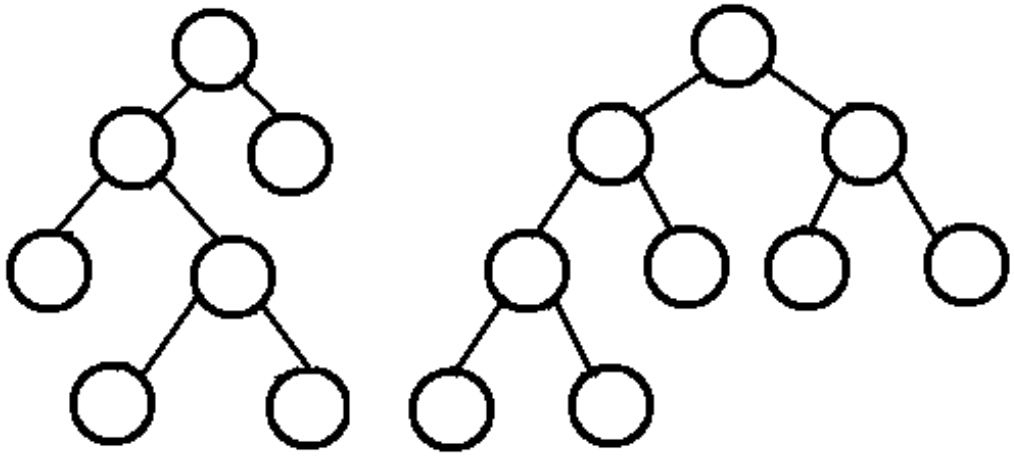
## Applications:

Trees are great for storing data that has a natural hierarchy, and are widely used. Their node based nature allows for easy modification (inserting, deleting, and moving around subtrees).

- One real world application of a tree is file storage in an operating system.

## Details:

- N-ary trees: a n-ary tree is a tree in which each child has at most n children.
    - For example, a binary tree has at most two children, and a ternary tree has at most three.
- Complete: a complete tree is a tree in which all but the last layer are full. The last layer can be partially filled, but it must be filled left to right.
- Full: a full tree is a tree in which each node has either it's max number of children or none.



**full tree** **complete tree**

- Perfect: a perfect tree is both full and complete.
    - A perfect tree will have all its leaves at the same level.

## Extensions:

- Binary search, AVL, red black, B trees
- Graphs

# Binary Search Tree

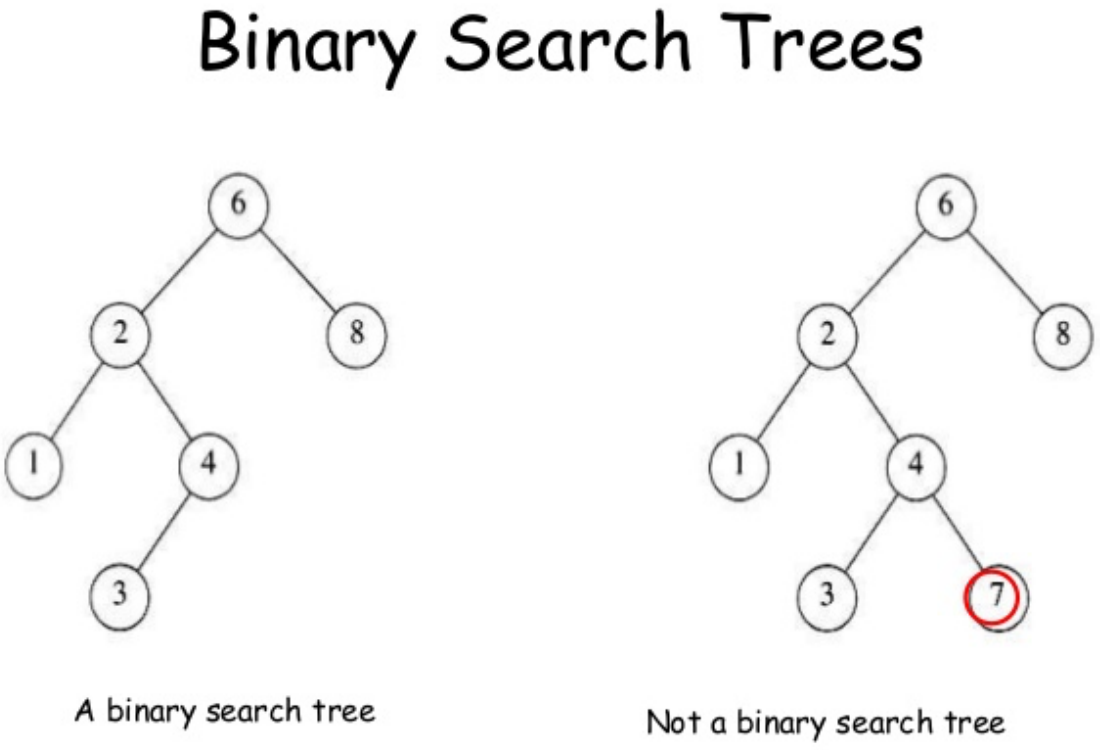
*Say bi to your old ways of searching*

| | Insertion | Deletion | Search | Space |
|---|---|---|---|---|
| **Complexity:** | O(log n)* | O(log n)* | O (log n) | O (n) |

* Assuming a roughly balanced tree; worst case can approach O(n)

## Description:

A binary search tree is a binary tree with special properties. The left child node will always be less than the parent node's value, and the right child node will always be greater than the parent node's value. In the example below, the right tree is not a binary search tree because of the node containing 7. While 7 is greater than its parent node's value, it is also greater than the root node making the left subtree invalid. The correct position for 7 would be the left child node of 8.




Source: https://www.slideshare.net/Nitians/8binry-search-tree

## Implementation:

A binary search tree is implemented the same way as any other binary tree, except with special rules for insertion and deletion. For the best understanding of binary search trees, it is recommended to implement these functions yourself at least once.

## Applications:

Binary search trees are a very powerful data structure due to its ability to provide O(log n) search time. However, it is not used in many places because there are more modern implementations that provide more consistent performance.

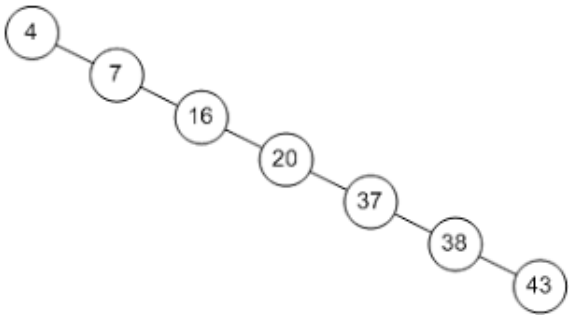
## Know how to:

- Distinguish a binary search tree.
- Search/traverse through a binary search tree.
- Insert/delete a node.

## Details:

- Binary search trees offer O(log n) search. This is due to the binary nature of the tree; because we know the direction to head (ie. Left if the item we are looking for is less than the current node, or right if the item is greater) the maximum number of nodes that have to be search is the height of the tree, which is log n.
    - However if the tree is not roughly balanced, then the height of the tree becomes n, reducing the search time to O(n).

    Ex.



- Traversal: there are three ways to traverse a binary search tree: in-order traversal, pre-order traversal, and post-order traversal.
    - in-order traversal: visits the left child, current node, then the right child.
    - pre-order traversal: visits current node, left child, and then right child.
    - post-order traversal: visits left child, right child, and then current node.

## Interview Problems:

- Create a function to determine if a tree is a binary search tree.
- Write a function to insert a node into a binary search tree.

# Hash tables

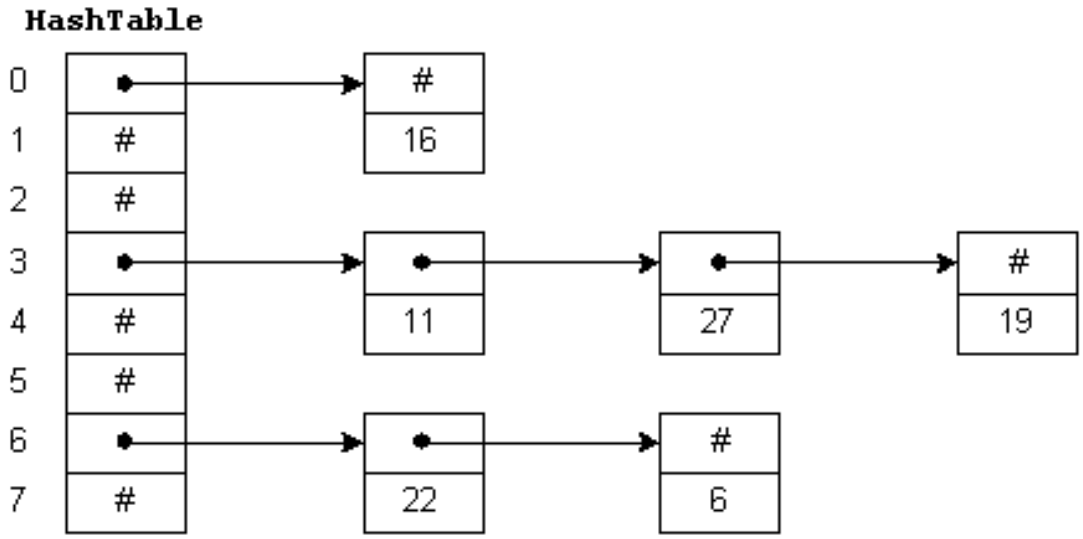
*This isn't your normal breakfast platter*

| | Insertion | Deletion | Search | Space |
|---|---|---|---|---|
| **Complexity:** | O (1) | O (1) | O(1) | O (n) |

## Description:

A data structure that maps keys to values via a hash function. Hash tables have very fast searching, but can not be iterated over. They are also known as hash maps* and dictionaries

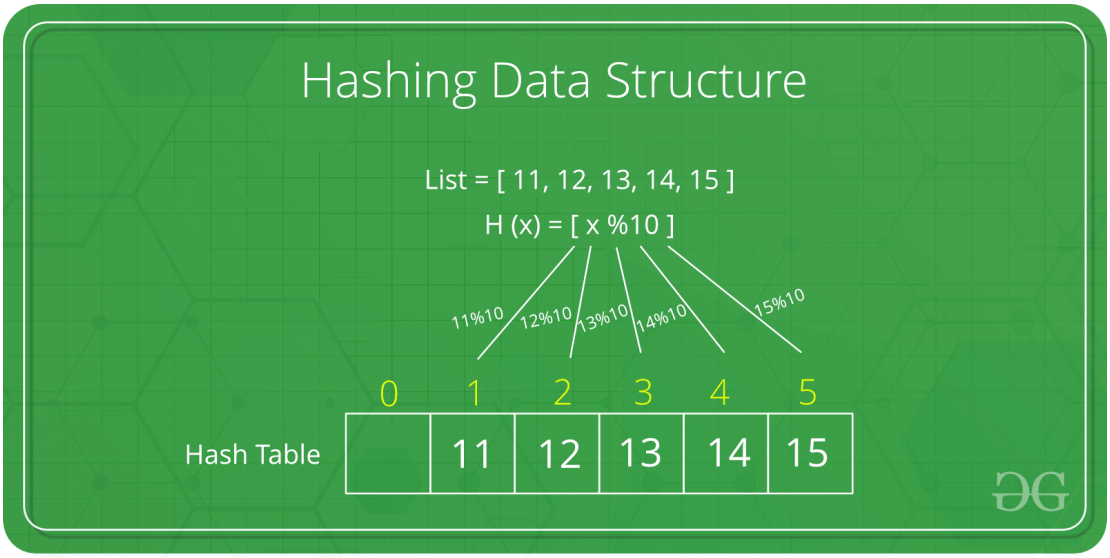
*There is actually a difference between a hash map and a hash table, but they are colloquially equivalent.




(This is an example of chaining)
Source: http://www.eecs.umich.edu/courses/eecs380/ALG/niemann/s_has.htm

## Implementation:

Hash tables are implemented with an array and a hash function. The hash function determines the index of the array to store data. However, sometimes keys will have the same hash value and collide with each other, this leads to decreased efficiency. The two main ways to handle collisions are chaining and open addressing.




Source: https://www.geeksforgeeks.org/hashing-data-structure/

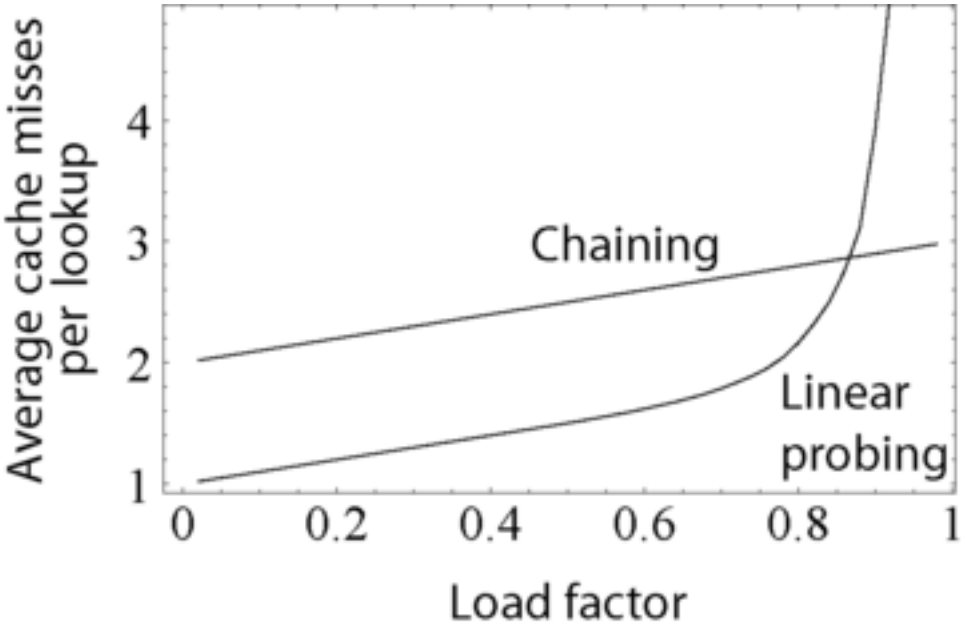
## Applications:

Hash tables are a very powerful data structure for both interview questions and real life applications. They are vital for any application that needs to link two values; any application in you which loop through an object to search for an item every time you need it you could use a hash table. For example, if you wanted to count every occurrence of a word of a book, you could loop the book once and add each word to a hash table and incrementing its value (if it has already been added).

## Details:

- Collisions: a collision is when two or more keys have the same result from the hash function
- Load factor: the ratio of stored keys vs capacity
- Open addressing: One method for handling collisions where conflicting items are placed in the next available index
  - Open addressing is very fast when the load factor is small, but as the load factor becomes larger it becomes very slow.
- Chaining: Another method of handling collisions in which conflicting items are placed in a linked list at the original index, this leads to a potential worst case of O(n)
  - Chaining's speed is affected linearly, so while it may be slower at first it becomes much more efficient when the load factor is large.

- A binary search tree can also be used if there are too many collisions, this could reduce the worst case to O(log n)




## Know:

- The strengths of hash tables
- How to deal with collisions
- How to insert an item

## Interview Problems:

- Return true if there are any duplicates characters in a string
- Return the length of the longest unique substring