**Dynamic data structures** are designed to change @ run time. Size can change.
**Static Data structure** has fixed memory size, created at compile time.

# Linked List:

PROS :
-can increase/decrease the number of nodes (dynamic allocation), as needed.
-can use multiple data types as elements
-one step above than arrays.
CONS :
-consumes more memory than arrays
-need to clear memory once we done with it.
-handling pointers is a bit tricky compared to arrays.. (but its needed for real time usage)

## Queues:
PROS: able to handle multiple data types and are flexible and fast. Can be of potentially infinite length.
CONS: a new element can only be inserted when *all* of the elements are deleted from the queue.

## Stacks:
Size of **Static** stack is constant throughout execution. **Dynamic stack** grows/shrinks as push/pop.

## Recursion:
Function that calls itself. Usually also uses a Stack.
-Base-case: if(n ==1) || if(n ==0) else{Continue Recursion}
-Call Stack: Stack that holds the total Function Calls
-Stack Frames: Each Function call = 1 Stack Frame.
-Stack Overflow Error: Too many calls, probably from Infinite/Excessive Recursion

**Exhaustive Search (Brute-Force)**
-Test every single Node(Possibility). Used to Enumerate every single variable possible.
-Queues would have to run through the whole queue to 'Backpedal' from a bad solution.
-Stacks use less memory and can pop() bad results and continue on.

## Linear Search:
Sequentially checks list until it finds result.
-Best Search method for Randomly ordered list.

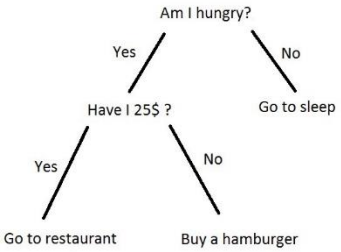
**Selection** Sort-[2 Lists(Sorted/Un)] Linear Minimum # hunter & Swaps with value+1. Then search value+2
**Merge** Sort- (Divide/Conquer) Splits by 2's & Merges Sorted Lists.
**Insertion** Sort- "Bubble-Down" to Sorted List. Compares 2 Elements, until Needs to swap.

**compareTo()**: used to **sort** string. By: Lexical order for String, Numeric order for ints, etc. ...
**compareTo()**: does a sequential comparison
Must code Custom compareTo()'s for Object for what value you want to compare.for what value you want to compare.

**Comparable<T>** interface- must have a method compareTo(**T**) to compare Objects in any custom way.

**Tree Terminology**: Root, Parent/Child, Sibling, Leaf
**Tree Terminology**: Root, Parent/Child, Sibling, Leaf
**Traversal** (Pre-order) = Node, Left, Right
**Traversal** (In-order) = Left, Node, Right
**Traversal** (Post-order)= Left, Right, Node
**Traversal** (Level-order) = Searches Each Tree Level's Sibling, then next sibling..

**Expression Tree**= Represents Algebraic Statement.




**Decision Tree**: Answer =That Question. Else this||that
Iterable<T> => current = current.getNext();
**Iterable** is an object that you can get an **Iterator** from.

Traverse Hash Tables: Linear Search + maybe hashing value for key.
NOT a technique for Avoiding Hash Collisions
a) make the hash function appear random
b) increasing hash table size
c) use uniform hashing
d) add elements with collided keys in a linked list
Method for resolving **collisions: Hash** table is taken @ Hash Key? Move sequentially until the first empty slot.

*Provide a way to compare Virus objects by mortality rates. Declare your Virus class in Java?

class Virus implements Comparable<Virus>

**INTERFACES**- **extends** = Inherit from one class. **implements** - Can use functions of multiple classes.

| |
|---|
| **SELECTION** – $O(n^2)$ |
| **INSERT** – $O(n^2)$ |
| **BUBBLE** – $O(n^2)$ |
| **MERGE** – O(n log n) |
| **PATIENCE** – O(n log n) |