**Data Structures**

Computers can store and process vast amounts of data. Formal data structures enable a programmer to mentally structure large amounts of data into conceptually manageable relationships.

Sometimes we use data structures to allow us to do more: for example, to accomplish fast searching or sorting of data. Other times, we use data structures so that we can do *less*: for example, the concept of the stack is a limited form of a more general data structure. These limitations provide us with guarantees that allow us to reason about our programs more easily. Data structures also provide guarantees about algorithmic complexity — choosing an appropriate data structure for a job is crucial for writing good software.

Asymptotic Notation

A problem may have numerous algorithmic solutions. In order to choose the best algorithm for a particular task, you need to be able to judge how long a particular solution will take to run. Or, more accurately, you need to be able to judge how long two solutions will take to run, and choose the better of the two.

*Asymptotic complexity* is a way of expressing the *main component* of the cost of an algorithm.

Complexity of the algorithm can be measured in 2 ways:

1. **Time Complexity:** The number of (machine) instructions which a program executes during its running time is called its **time complexity** in computer science. This number depends primarily on the size of the program's input, that is approximately on the number of the strings to be sorted (and their length) and the algorithm used.
2. **Space Complexity:** The better the time complexity of an algorithm is, the faster the algorithm will carry out his work in practice. Apart from time complexity, its **space complexity** is also important: This is essentially the number of memory cells which an algorithm needs. A good algorithm keeps this number as small as possible, too.

There is often a **time-space-tradeoff** involved in a problem, that is, it cannot be solved with few computing time and low memory consumption. One then has to make a compromise and to exchange computing time for memory consumption or vice versa, depending on which algorithm one chooses and how one parameterizes it.

**Types of data structures**

### The Node: The first data structure we look at is the node structure. A node is simply a container for a value, plus a pointer to a "next" node (which may be null).

1. **Arrays**: Array is a collection mainly using similar data types that are stored into a common variable, forming a linear data structure. The index is usually a number used to address an element in the array.

http://upload.wikimedia.org/wikibooks/en/8/85/SimpleArray.png

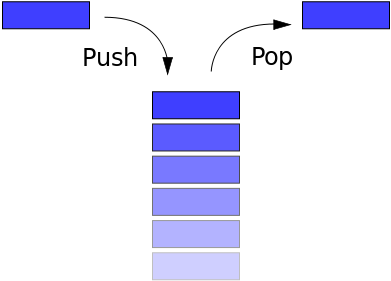
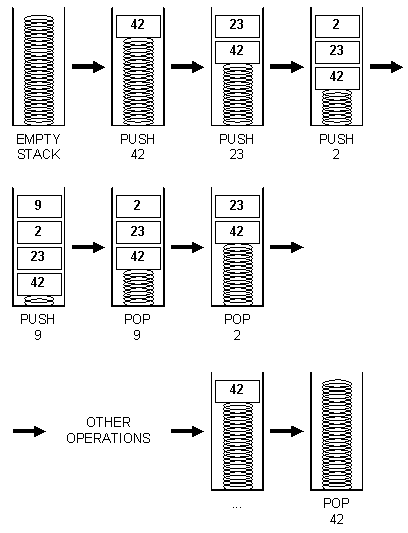
*The notebook (array) contains 12 pages (elements)*

### Stacks:

A stack is a basic data structure that can be logically thought as linear structure represented by a real physical stack or pile, a structure where insertion and deletion of items takes place at one end called top of the stack. The basic concept can be illustrated by thinking of your data set as a stack of plates or books where you can only take the top item off the stack in order to remove things from it. This structure is used all throughout programming.

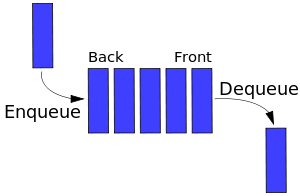
The basic implementation of a stack is also called a LIFO (Last In First Out) to demonstrate the way it accesses data, since as we will see there are various variations of stack implementations.

There are basically three operations that can be performed on stacks . They are 1) inserting an item into a stack (push). 2) deleting an item from the stack (pop). 3) displaying the contents of the stack(pip).

1. Queue: A queue is a basic data structure that is used throughout programming. You can think of it as a line in a grocery store. The first one in the line is the first one to be served.Just like a queue.

A queue is also called a FIFO (First In First Out) to demonstrate the way it accesses data. When you want to **enqueue** something, you simply add it to the back of the item pointed to by the tail pointer. So the previous tail is considered next compared to the item being added and the tail pointer points to the new item.

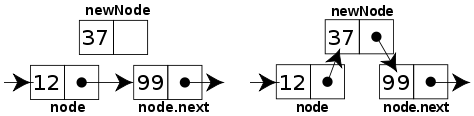


1. Linked List: Linked list is a data structure that allows sequential access to the elements. A list lays out the sequence in a row, starting at the first element (called front) and proceeding in successive order to the last element (called back). Each element in a list contains Links that identify both the next and the preceding item in the sequence. We focused on the fact that the Linked list provides efficient operations to insert and delete an element at any position in the sequence. This was very important, because we needed to understand why a Linked list is sometimes used in an application rather than an array. Programmers select a Linked list as the data structure of choice when an application wants to maintain elements by position and allow for frequent insertions node and deletions node.
2. **Non-Circular Single Linked List**

Graphical Picture:

1. 

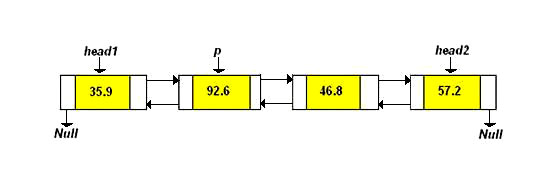
**Inserting into a Linked List:**

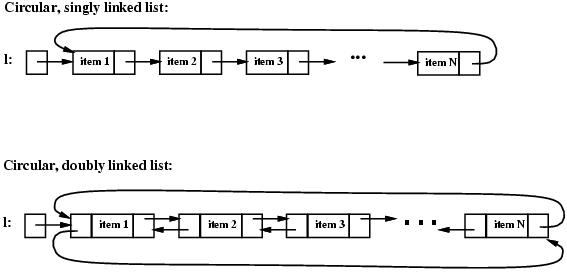


# Circular Single Linked List

Graphical Picture:



1. Doubly linked list: A linked list in which each node has 2 pointers: a forward pointer (a pointer to the next node in the list) and a backward pointer (a pointer to the node preceding the current node in the list) is called a doubly linked list.iv) 
2. Circular Doubly linked list:



# Circular Linked List Queue: A singly linked circular list is a linked list where the last node in the list points to the first node in the list.  A circular list does not contain NULL pointers.

Two Graphical Pictures:



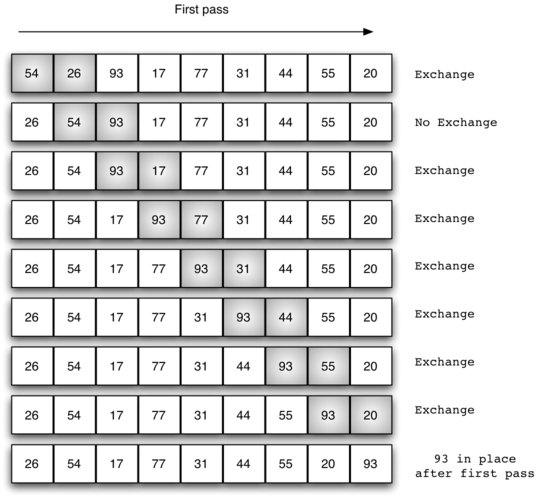


# Sorting

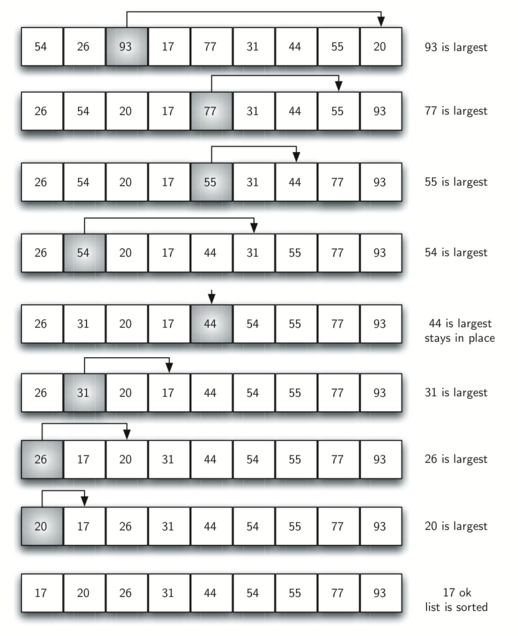
Sorting is the process of placing elements from a collection in some kind of order. For example, a list of words could be sorted alphabetically or by length. There are many, many sorting algorithms that have been developed and analyzed.

The Bubble Sort

The **bubble sort** makes multiple passes through a list. It compares adjacent items and exchanges those that are out of order. Each pass through the list places the next largest value in its proper place. In essence, each item “bubbles” up to the location where it belongs.

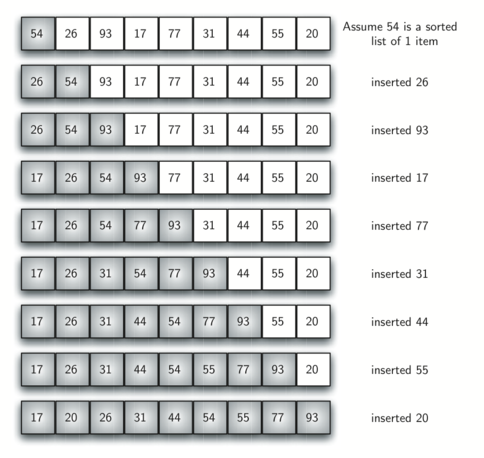


The Selection Sort

The **selection sort** improves on the bubble sort by making only one exchange for every pass through the list. In order to do this, a selection sort looks for the largest value as it makes a pass and, after completing the pass, places it in the proper location. As with a bubble sort, after the first pass, the largest item is in the correct place. After the second pass, the next largest is in place. This process continues and requires *n*−1 passes to sort n items, since the final item must be in place after the (*n*−1) st pass. 

The Insertion Sort

The **insertion sort** works in a slightly different way. It always maintains a sorted sublist in the lower positions of the list. Each new item is then “inserted” back into the previous sublist such that the sorted sublist is one item larger.

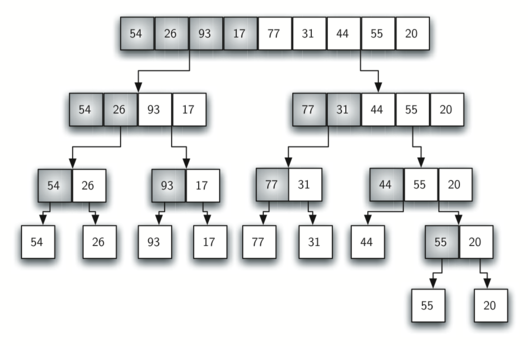


We begin by assuming that a list with one item (position 0) is already sorted. On each pass, one for each item 1 through *n*−1, the current item is checked against those in the already sorted sublist.

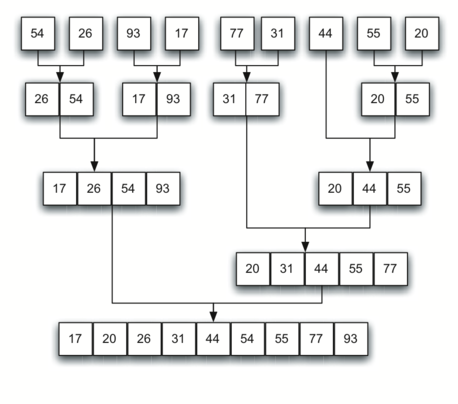
Divide and conquer techniques:

1. The Merge Sort:

Merge sort is a recursive algorithm that continually splits a list in half. If the list is empty or has one item, it is sorted by definition. Once the two halves are sorted, the fundamental operation, called a **merge**, is performed. Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted, new list.

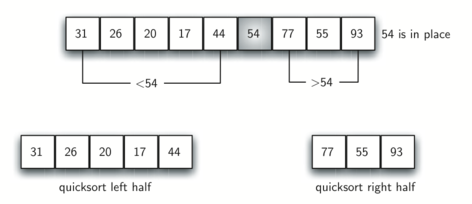
Splitting the List in a Merge Sort 

 Lists as They Are Merged Together



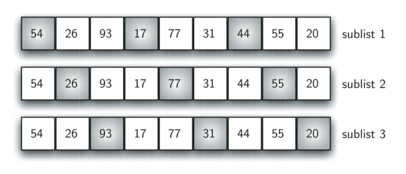
The Quick Sort

The **quick sort** uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage. A quick sort first selects a value, which is called the **pivot value**. Although there are many different ways to choose the pivot value, we will simply use the first item in the list. The role of the pivot value is to assist with splitting the list. The actual position where the pivot value belongs in the final sorted list, commonly called the **split point**, will be used to divide the list for subsequent calls to the quick sort.

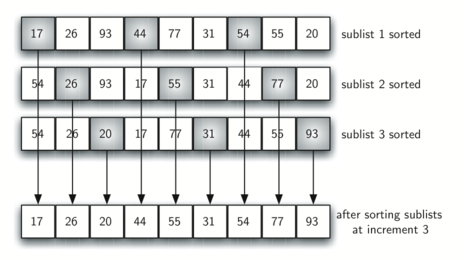


The Shell Sort

The **shell sort**, sometimes called the “diminishing increment sort,” improves on the insertion sort by breaking the original list into a number of smaller sublists, each of which is sorted using an insertion sort. The unique way that these sublists are chosen is the key to the shell sort. Instead of breaking the list into sublists of contiguous items, the shell sort uses an increment i, sometimes called the **gap**, to create a sublist by choosing all items that are i items apart.



A Shell Sort with Increments of Three



A Shell Sort after Sorting Each Sublist

Trees

A **tree** is a non-empty set one element of which is designated the root of the tree while the remaining elements are partitioned into non-empty sets each of which is a subtree of the root.

Tree nodes have many useful properties. The **depth** of a node is the length of the path (or the number of edges) from the root to that node. The **height** of a node is the longest path from that node to its leaves. The height of a tree is the height of the root. A **leaf node** has no children -- its only path is up to its parent.

### Traversal

Many problems require we visit the nodes of a tree in a systematic way: tasks such as counting how many nodes exist or finding the maximum element. Three different methods are possible for binary trees: *preorder*, *postorder*, and *in-order*, which all do the same three things: recursively traverse both the left and right subtrees and visit the current node. The difference is when the algorithm visits the current node:

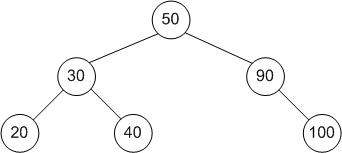
**preorder**: Current node, left subtree, right subtree (DLR)

**postorder**: Left subtree, right subtree, current node (LRD)

**in-order**: Left subtree, current node, right subtree (LDR)

**levelorder**: Level by level, from left to right, starting from the root node.

### Examples of Tree Traversals



**preorder:** 50, 30, 20, 40, 90, 100

**inorder:** 20, 30, 40, 50, 90, 100

**postorder:** 20, 40, 30, 100, 90, 50

**levelorder:** 50, 30, 90, 20, 40, 100