



# CS50-V

## Data Structures

Data Structures essentially are forms of organisation in memory.

Abstract D-S are those that we can conceptually imagine.  
Learning these will make it easier later to understand how to implement more concrete D-S.

	Insertion	Deletion	Look up	Sort	Size	Flexibility
Arrays						
Linked List						
Hash Tables						
Tries						

## Structures

Structures provide a way to unify several variables of different types into a single, new variable type which can be assigned its own type name.

We use structures (struct) to group together elements of a variety of data types that have a logical connection.

```
struct car
{
    int year;
    char model[10];
```

Once we have defined a structure, we have effectively created a new type. We can create variables of that type using the familiar syntax. We can also access the various fields of the structure, using the dot operator (.).

# Stacks and Queues

## Queues

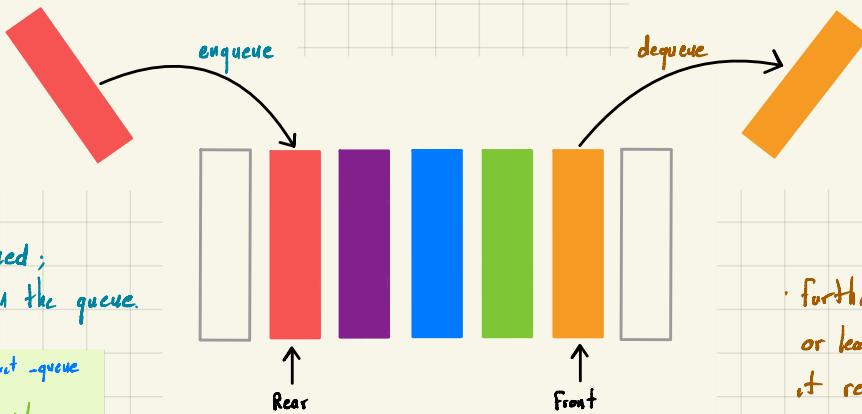
Queues are one form of abstract data structure.

This data structure is commonly implemented in two ways: as an array or as a linked list.

- An item can be enqueued; that is the item can join the queue.

```
typedef struct _queue {  
    VALUE array[Capacity];  
    int front;  
    int size;  
} queue;
```

```
typedef struct _queue {  
    VALUE val;  
    struct _queue *prev;  
    struct _queue *next;  
} queue;
```



Queues have specific properties.

Namely they are FIFO or "First In First Out".

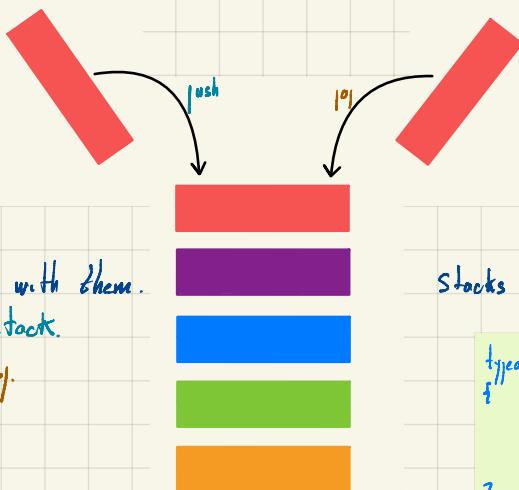
- further, an item can be dequeued or leave the queue once it reaches the front.

## Stack

```
typedef struct {  
    VALUE array[Capacity];  
    int size;  
} stack;
```

Stacks have specific actions associated with them.

- Push places something on top of a stack.
- Pop is removing something from the top.



This data structure is commonly implemented in two ways: as an array or as a linked list.

Stacks are LIFO or "Last In First Out".

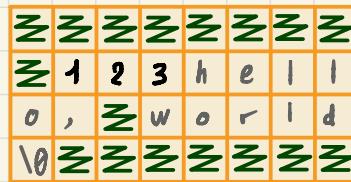
```
typedef struct _stack {  
    VALUE val;  
    struct _stack *next;  
} stack;
```

## Resizing Arrays

An array is a block of **contiguous** memory.



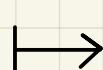
In memory, there are other values being stored by other programs, functions, and variables. Many of these may be unused garbage values that were utilized at one point but are available now for use.



If you wanted to store a 4<sup>th</sup> value in an array,

What would be needed is to allocate a new area of memory and move the old array to a new one.

old garbage value would be overwritten with the new data.



One of the drawbacks of this approach is that it's bad design; Every time we add a number, we have to copy the array item by item.

C comes with a very useful function called `realloc`. It takes two arguments:

The array you are attempting to copy.

The size the final array should be.

`int *tmp = realloc(list, 4 * sizeof(int));`

## Linked Lists

1. operator `.` dot notation allows to access variables inside a structure.

The `*` operator is used to declare a pointer or dereference a variable.

→ `operator:` goes to an address and looks inside of a structure

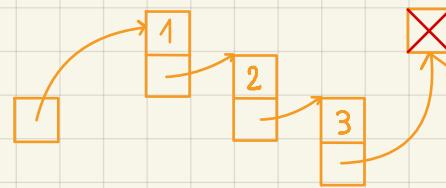
A linked list  
one of the most  
powerful data structures  
within C.

It allows to include values that are located at varying areas of memory.

Further, they allow to **dynamically grow and shrink the list** as desired.

## Singly-linked Lists

We could use more memory to keep track of where the next item is. By convention, we would keep one more slot in memory, a pointer, that keeps track of the first item in the list. These boxes are called nodes.



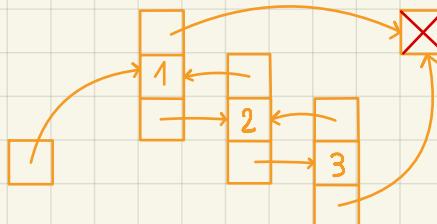
A linked list node is a special kind of struct with two members:

- Data of some data type
- A pointer to another node of the same type

```
typedef struct node  
{  
    int number;  
    struct node *next;  
}  
node;
```

## Doubly-linked Lists

A doubly-linked list allows to move forward and backward through the list, all by simply adding one extra pointer to the struct.



- Data of some data type
- A pointer to the next node
- A pointer to the previous node

```
typedef struct node  
{  
    int number;  
    struct node *next;  
    struct node *previous;  
}  
node;
```

Linked lists are not stored in a contiguous block of memory. They can grow as large as wished, provided that enough system resources exist.

The downside, however, is that more memory is required to keep track of the list instead of an array. This is because for each element, you must store not just the value of the element, but also a pointer to the next node.

Further, linked lists cannot be indexed into like is possible in an array because we need to pass through the first  $n-1$  elements to find the location of the  $n^{\text{th}}$  element. Because of this, linked lists must be linearly searched. Binary search, therefore, is not possible.

Considering the amount of time required to search this list, it is in the order of  $O(n)$ , as in the worst case the entire list must be always be searched to find an item.

## Trees

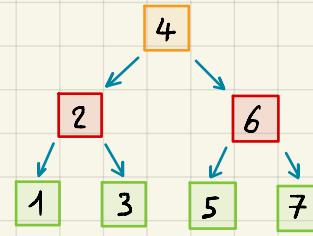
Binary Search trees are another data structure that can be used to store data more efficiently such that it can be searched and retrieved.

You can imagine a sorted sequence of numbers.

I imagine then that the center value becomes the top of a tree.

Those that are less than this value are placed to the left.

Those values that are more than this value are to the right.



Pointers can then be used to point to the correct locations of each area of memory such that each of these nodes can be connected.

A tree offers dynamism that an array does not.

It can grow and shrink as we wish.

## Dictionaries

Dictionaries are another data structure.

Dictionaries have a key and a value.

The holy grail of time complexity is  $O(1)$   
or constant time.

Dictionaries can offer this speed of access.



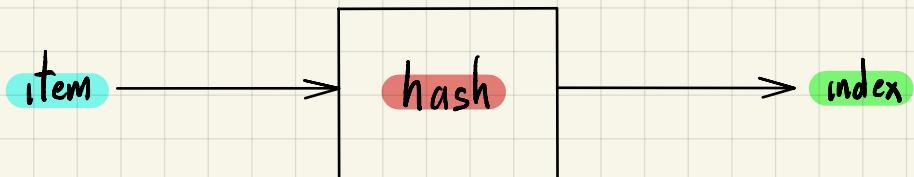
## Hashing and Hash Tables

Hashing is the idea of taking a value and being able to output a value that becomes a shortcut to it later.

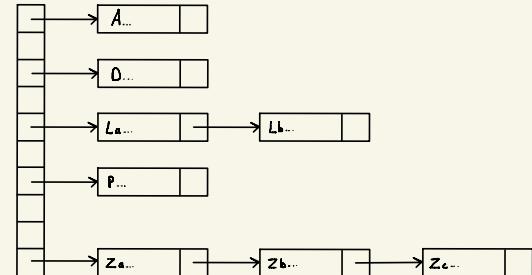
For example, hashing apple may hash as a value of 1, and berry may be hashed as 2. Therefore finding apple is as easy as asking the hash algorithm where apple is stored. While not ideal in term of design, ultimately, putting all a's in one bucket and b's in another, this concept of bucketizing hashed values illustrates how you can use this concept: a hashed value can be used to shortcut finding such a value.

A hash function is an algorithm that reduces a larger value to something small and predictable.

Generally, this function takes in an item you wish to add to your hash table, and returns an integer representing the array index in which the item should be placed.



A hash table is a fantastic combination of both array and linked lists.  
When implemented in code, a hash table is an array of pointers to nodes.



Hash tables combine the random access ability of an array with the dynamism of a linked list.

Collisions are when you add values to the hash table, and something already exists at the hashed location.

Collisions can simply be appended to the end of the list.

They can be reduced by better programming your hash table and hash algorithm.

The programmer has to make a decision about the advantages of using more memory to have a large hash table and potentially reducing search time or using less memory and potentially increasing search time.

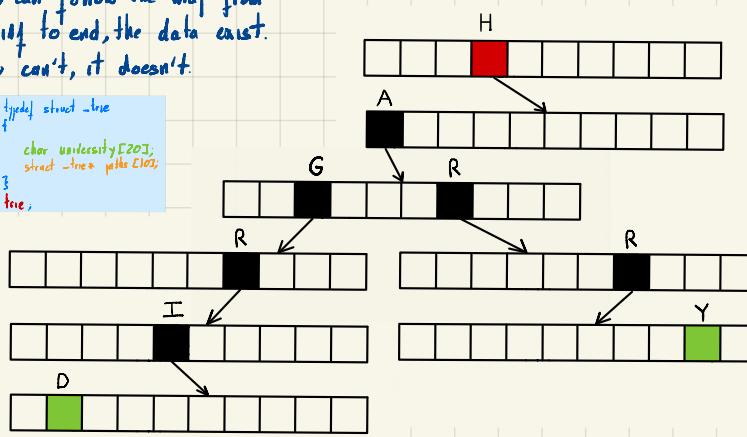
## Tries

The data stored in tries is like a road map.

- If you can follow the map from beginning to end, the data exist.
- If you can't, it doesn't.

```
typedef struct Trie
{
    char university[20];
    struct Trie *next[10];
}Trie;
```

One downside to tries is that they tend to take up a large amount of memory.



Unlike with a hash table, there are no collisions, and no two pieces of data have the same path.

To insert an element into a trie, simply build the correct path from the root to the leaf.

Tries are always searchable in constant time.