Algorithmns and Data Structures

Data Structures

What (the ****) is a data structure?

A data structure special way of organizing and managing a data collection in order to work with it more efficently. Hereby, data structures can help to store, access and modify the data easily, show relations between the objects and include functions or operations that can be applied on the data.

Array (default, ⚡ appending, deleting, ✅ searching)

One of the most commenly used and also one of the most simplest data structures is the `Array`. An Array is used to store elements of the same data type. It is classified as a linear, static data structure and the objects are stored in continous memory locations. It is also the most efficient data structure for storing and accessing a sequence of objects. Why? Let's have a look on the underlying structure:

The image shows an sample Array, the objects (boxes) with their index (on the bottom) and their assigned memory location (on the top). Hereby, an Array has a fixed size and the objects are saved in memory locations next to each other. One advantage is that we can very quickly store new objects. Due to the fixed size, the computer can reserve the memory in compile-time and new objects can easily be stored on a specific index which points to a reserved memory location. It is not necessary to run through all the elements and hence inserting data runs in O(1) time. Additionally, we don't waste any memory since we don't have any unassigned memory locations in-between the objects of our Array.

Another advantage is that we can easily access any element of the Array by using an index. Behind the scenes, it takes the memory location of the first element (`Array[0]`) and retrieves the targeted memory location by adding the index. Consequently, an element can be accessed in constant time without the need to run through/search the entire Array.

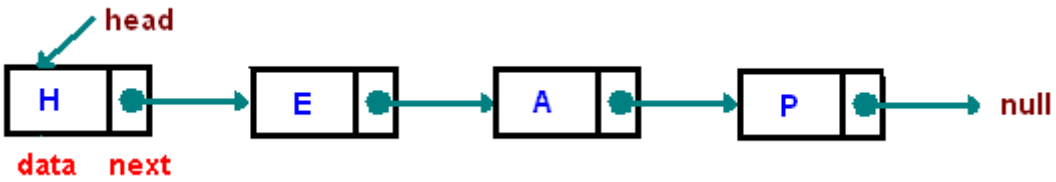
However, what happens if you want to not only store/replace an object at an existing position but rather insert a new element or delete an existing one? Due to the underlying structure of an Array, this is not easily possible, at least not with the same Array. In order to change the size of the Array, we need to create a new one with the targeted size at a different memory location and copy the elements over. Especially, if these operations are not executed on the last position of the Array, this can become relatively costly. So what can we do if we have a program with a lot of insertion and deletion operations? In this case, we can have a look on a Linked List which is optimized for these kinds of operations.

Linked List (common base for Stack and Queue, ✅ easy appending, deleting)

A Linked List is a linear, dynamic data structure where each element has a pointer referring to the next element in the list. Unlike in Arrays, the elements are not stored in contiguous memory locations and are also not indexed. Because of that, it can easily grow or shrink in size on demand and is especially very useful and needed for data

sets with an unknown or dynamic size. This is also the reason why insertion and deletion are much easier with this data structure.

But how does it work exactly? In a Linked List, each element is called a "node" and is a separate object. Starting from the first node in the list (the entry point or also called the "head"), each element/node contains next to the actual data also a reference of the next node in the list. The last node contains a reference to null.



This makes it easy to perform insert and delete operations as we only have to change the pointer in the previous node. for example, based on the image above, if we want to insert a new node with the data string `"S"` inbetween `"A"` and `"P"`, we can simply modify the pointer of the `"A"`-node to point to the `"S"`-node and let the `"S"`-node point to the `"P"`-node. Hereby, the memory for the `"S"`-node is allovated in run/execution-time. Presuming that we are already on the right node where we have to change the pointer, we can perform this operation in constant time. In an extended version of the Linked List, the Doubly Linked List, each node contains additionally a reference to the previous item, making it easier to iterate through the list and performing for example delete operations on it. But how do we get to the previous node in first place? Can we just use its index?

No, we can't use the index. Due to the fact, that the objects of a Linked List are not saved in contiguous memory locations, the elements of the List can't be efficiently indexed. Instead, we have to start from the head of the List and go from node to node until we find the node with the right data. So, while insertion and deletion are more efficent in a Linked List, it takes more time for accessing a specific element as it has to loop through the entire List in worst-case. Also something to be considered: Linked Lists require more memory space than Arrays because they have to save the pointer to the next node (in a Singly Linked List) next to the actual data. On a 32-Bit CPU this extras 4 Byte per reference.

So, it depend on the your program and the connected operations which of these two data structures makes more sense for you.

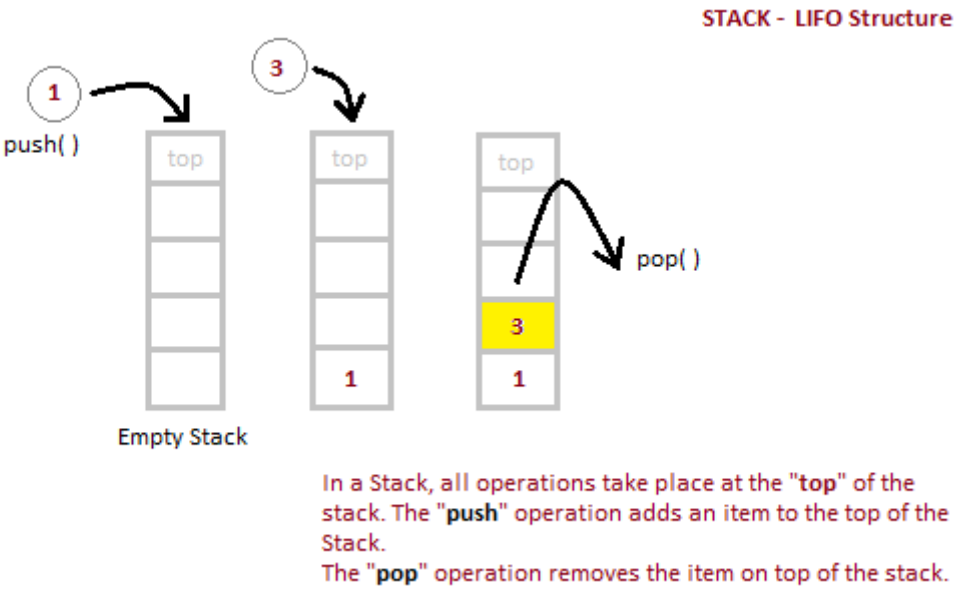
An custom implementation of a Linked List in Golang can be found here:

Stack (✅ easy appending at end)

Now, as we found a data structure for effiently inserting and deleting, we can have a look on data structures which simplify accessing for special cases.

> Stacks and Queues can have both Arrays and Linked Lists as basis for storing data. Since Stacks and Queues are dynamic data structures, in most cases it is recommended to use a Linked Lists as basis because it can grow and shrink in size easily and works most efficient therefore.

A Stack is a linear, dynamic data structure where Insertion and Deletion are only performed on one end (e.g the top) of the data sequence. If we're adding an element to the stack, it will be added to the top. If we're removing an element, it will be removed from the top. This schema is also called "Last In First Out" or short "LIFO". Meaning, the last element we have put into the Stack is the first item that gets out of the Stack. The following image visualizes this a bit better:



Due to these restrictions for inserting and deleting of objects, a Stack can be very useful for particular application cases. Do you ever wanted to represent the manual cleaning of dirty plates in code and didn't find the right data structure for it?... No? Anyways, there a lot of cases where it actually makes sense to use a stack. For example the command history on your local computer or the website history in your local browser are based on stacks to enable you reversing the last command or going back to the previous website.

Herby, the two core operations of a Stack are:

- **push(element)**
- and **pop()**

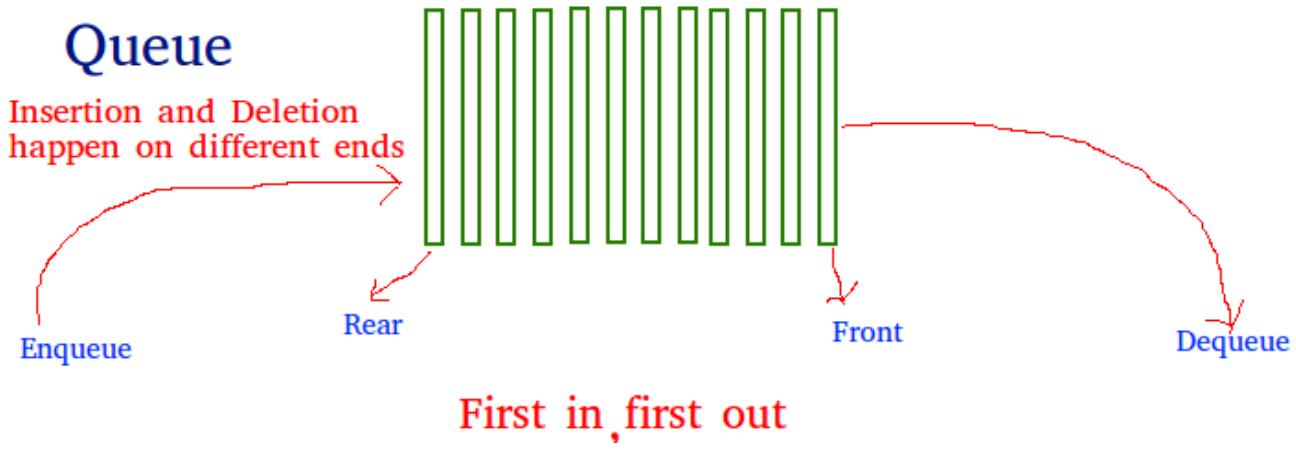
push(element) adds a new element to the top, while **pop()** retrieves the element on the top of the Stack, removes it and might returns it.

When customly implementing a Stack based on a Linked List, it makes sense to use the `head` as `top` as insertion and removal operations can be executed in constant time on the first node of a Linked List. This means that accessing of the top element in a Stack can be performed in constant time.

Queue (✅ easy appending in front)

But what is if we don't only want to insert and remove at the same end but rather on different ends of the data sequence? Well, then we can use the Queue data structure.

A Queue is a linear, dynamic data structure where insertion and deletion/removal happen on different ends. Hereby, it follows the schema "First In First Out" or short FIFO. This means that the first element that was inserted is also the first element that is retrieved. The difference to a Stack is that we remove the least recently added item first while in a Stack, we remove the most recently item first. In a simplified way, a Queue data structure can be compared to a queue in the real-world. The following graphic, helps to understand the behavior better:



The first element in a Queue is called the `Front`, the last element is called `Rear`. Likewise the Stack, a Queue is also restrictive in which way the operations are performed and supports two main operations:

- **enqueue(element)**
- and **dequeue()**

enqueue(element) adds an element to the Rear of the the Queue. **dequeue()** returns the first item in the Queue (at the Front position), removes it and might return it.

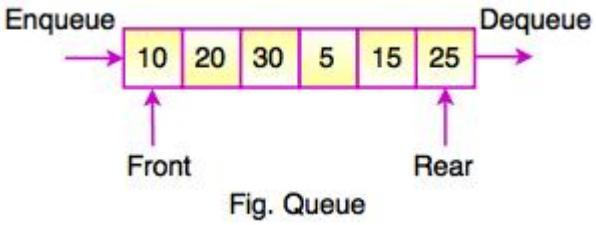


Fig. Queue

When we are using a Linked List, it depends on the implementation which direction the Queue has. However, in oposite to the Stack, either insertion or removal take linear time, as we have to loop through the list to the last element.

But wait one second... we now found ways for working with data for particular cases. But we haven't find a way yet to access a specific object anywhere in a Linked List without running through all other objects until we eventually find it. Isn't there a way to have something like an index, but still having all advantages of a dynamic data structure? Let's have a look on the Hash Table.

Hash Table (✅ inserting, deleting, fast searching)

Indexing gives us the possibility to access elements of a data sequence in constant time. A requirement for efficient indexing in data sequences is that the data is stored in contiguous memory locations and the index can be used based on the memory location of the first element. Therefore, we need to reserve a fixed size of memory in compile time and hence need a fixed size for the data sequence. But how can we apply indexing on our data sequence without a fixed size? That's the point where the Hash Table comes into play.

An Hash Table is an abstract data structure based on an Array. However, unlike in an Array, it has unique indexes which don't have to be integers. Herby, the indexing is not based on memory locations but converted using a hashing function. The hashing function maps a key of any data type to an index in the range of the underlying Array indexes. The associated value of the key is then saved at the regarding memory location of this index. If we

want to receive a value from the Hash Table, we just use the hashing function to calculate the index from the key and then get the element at the corresponding memory location to the index. This let us run storing and accessing/searching operations in constant time while at the same time using an custom index.

...But wait. How does this should be able to grow and shrink in size? We still have an underlying Array of fixed size, so we can't store more elements than the capacity of the Array? ...This is actually a problem. Additionally, hashing functions can create the same output for different inputs. So, it could be that the indexes calculated from "Lisa" and "Jan" are actually the same.

Collision Handling

To solve these problems, also known as "Collision", one strategy besides others is the useage of an Array of Linked Lists as underlying data structure. When we are insert an value for a specific key, an object consisting of the key and the value is added to the LinkedList at the associated index. In case we want to retrieve the value for a specific key, first, it runs the hashing function and retrieves the index. If the Linked List contains only one node, it can easily return tha value of the first node's object. If the List contains multiple nodes (because multiple keys map to the same index), it runs through the nodes and searches for the key. This enables the Hash Table to contain more elements than the capacity of the Array and still allow a quick accessing using the key (constant running time).

An easier understandable illustrtion of a Hash Table can be seen in the following graphic:

Because Hash Tables are very flexible but still can perform all run-time operations in constant time, they're widely used data structures for many different use cases.

Graph

All of the previous data structures are able to store and retrieve data but don't provide an easy solution for showing relations between the data. This is the point where a Graph can be very useful.

Applications of complex data structures

Sorting & Searching

-> from slowest to fastest, explination of running time with code example

Selection Sort (one of the most simplest sorting algorithms ⚡ two loops)

```
[SORT] array of integers in asc order {
        [LOOP] through every array elements {
        [SAVE] current index as `min` element

        [LOOP] through all following elements {
            [IF] inner loop element is smaller as `min`
                [SAVE] index of new position as `min`
        }

        [SWAP] value at the current index with
               the value of the `min` element
```

```
        }
}
```

Bubble Sort (most simple sorting algorithm ⚡ long runtime in worst case - can run n^2)

```
[SORT] array of integers in asc order {
    [REPEAT] as long array is not sorted {
        [LOOP] through every array elements {
            [IF] current element is bigger than the following
                                [SWAP] both elements
        }

        [IF] loop ran without single swap
            [STOP] function because it is sorted
        }
}
```

Insertion Sort (also one loop per element, however could run in O(n) as well if already sorted)

```
[SORT] array of integers in asc order {
    [LOOP] through every element {
        [LOOP] through every previous element in desc order {
            [IF] upper loop element is smaller than inner loop element
                [SWAP] elements
            [ELSE]
                [GO TO] next element0
        }
    }
}
```

-> devide, conquer, combine

Quick Sort (recursive strategy, one loop per partition)

```
/*
[PARTITION] takes a randonmly selected element from the*array(the pivot)
to split the array into a lower (values less than the pivot value) and
higher (values higher than the pivot value) group. The pivot will be
placed inbetween both groups.
*/

[QUICKSORT] array of integers in asc order {
    [IF] array has length of 1
        [RETURN] because it is already it is sorted

    [PARTITION] array and get pivot index {
```

```
        [ASSIGN] pivot value
        [LOOP] through each element of array {
            [COMPARE & SWAP] if element is not in right group
        }

        [SWAP] pivot to position inbetween groups
    }

    //recursive calls
    [QUICKSORT] lower group of the array
    [QUICKSORT] higher group of the array including the pivot
}
```

Merge Sort (recursive strategy, just deviding, no sorting)

```
[MERGESORT] array of integers in asc order {
        [IF] array has length of 1
        [STOP] recursive call because it is sorted

    [IF] array has length of 2
        [COMPARE & SWAP] if elements not in right order
        [STOP] recursive call because it is sorted

    [CALCULATE] midelst index

    //recursive calls
    [MERGESORT] first half of array
    [MERGESORT] second half of array

    [COMBINE] both halfs {
        [LOOP] through elements of both halfs at same time {
            [COMPARE & ASSIGN] to right index
        }
    }
}
```

-> searching common requirement / practice -> most efficent searching algorithm

Binary Search (not needed in Hash Tables)

---------- -->