Chapter 1: Introduction to data structures

Type	Lecture
Reviewed	☐
Created by	
Last edited by	
Date	@December 16, 2022
Last edited time	@December 16, 2022 5:16 PM
Number	1

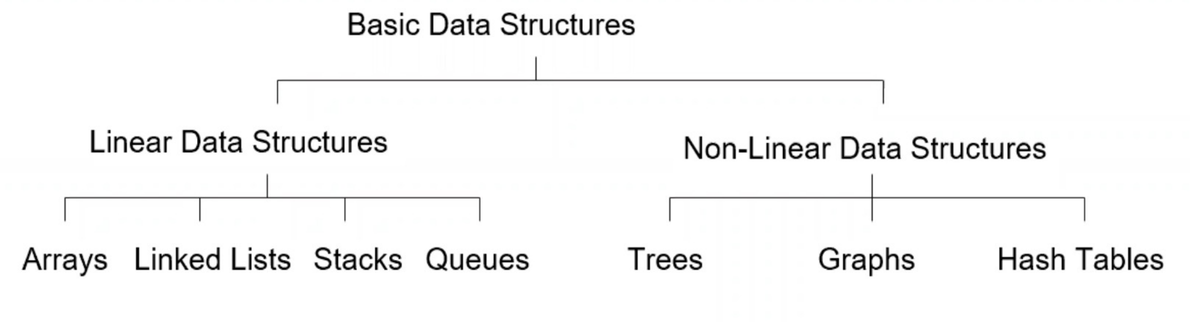
Key learnings

- ▼ Content for the lecture
  - ▼ ADTs
  - ▼ Stacks
  - ▼ Queues
  - ▼ Lists
  - ▼ Trees
  - ▼

Whats a data structure?

- Data structure is a representation of data and the operations allowed on that data.
- **A data structure is a way to store and organize data in order to facilitate the access and modifications.**

- Data Structures are the method of representing of logical relationships between individual data elements related to the solution of a given problem.



Selection of a DS?

The choice of a DS depends on

- It must be rich enough in structure to represent the relationship between data elements
- The structure should be simple enough that one can effectively process the data when necessary

Types of DS

Linear: In Linear data structure, values are arranged in a linear fashion.

- Array: Fixed-size
- Linked-list: Variable-size
- Stack: Add to top and remove from top
- Queue: Add to back and remove from front
- Priority queue: Add anywhere, remove the highest priority

Non-Linear: The data values in this structure are not arranged in order.

- Hash tables: Unordered lists which use a hash function' to insert and search
- Tree: Data is organized in branches.
- Graph: A more general branching structure, with less strict connection conditions than for a tree

Homogenous : Stores data of the same type

- Arrays

Non-Homogenous : Stores data of different types

- Classes
- Structs

Abstract Data Type(ADT)

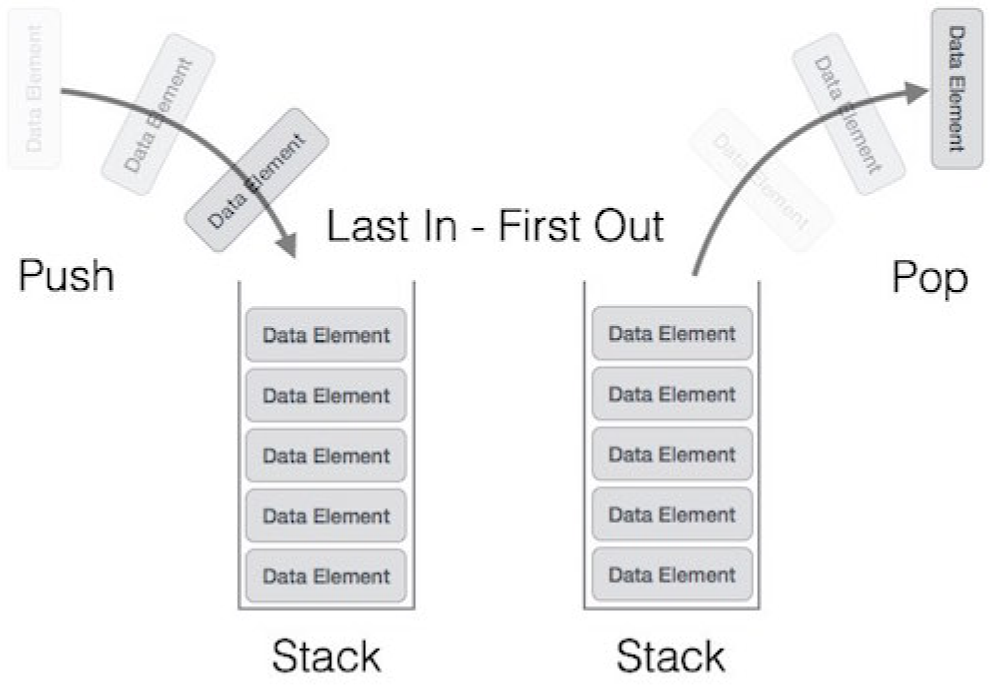
- Abstract Data Types (ADTs) stores data and allow various operations on the data to access and change it.
- **An ADT is a collection of data and associated operations for manipulating that data**
- DS is a physical implementation of an ADT
- data structures used in implementations are provided in a language (primitive or built-in) or are built from the language constructs (user-defined)
- Each operation associated with the ADT is implemented by one or more subroutines in the implementation
- ADTs support abstraction, encapsulation, and information hiding.

ADTs should provide interfaces to,

- Add data
- Remove data
- Retreive

Stacks

A stack is a linear data structure that follows a last-in, first-out (LIFO) strategy. It consists of a set of ordered items, with the addition of a new element at the top of the stack and the removal of an element from the top as well. The top of the stack is the element that was added most recently.



A stack can be implemented using an array or a linked list. In both cases, the stack data structure supports the following basic operations:

- `push`: adds an element to the top of the stack
- `pop`: removes the element at the top of the stack and returns it
- `peek`: returns the element at the top of the stack without removing it
- `is_empty`: checks if the stack is empty

```
class Stack:
    def __init__(self, max_size=10):
        self.max_size = max_size
        self.items = [None] * self.max_size
        self.num_items = 0

    def push(self, item):
        if self.num_items == self.max_size:
            raise Exception("Stack is full")
        self.items[self.num_items] = item
        self.num_items += 1

    def pop(self):
        if self.num_items == 0:
            raise Exception("Stack is empty")
        item = self.items[self.num_items - 1]
        self.items[self.num_items - 1] = None
        self.num_items -= 1
        return item

    def peek(self):
        if self.num_items == 0:
            raise Exception("Stack is empty")
        return self.items[self.num_items - 1]

    def is_empty(self):
        return self.num_items == 0
```

Stacks are often used to reverse the order of items, as they allow adding and removing items only from the top of the stack. For example, if you want to reverse the order of a list of items, you can push the items onto a stack one by one, and then pop them off the stack to get the reversed order.

Stacks are also used in many algorithms and data structures, such as depth-first search in graphs, evaluating expressions in programming languages, and implementing undo functionality in text editors.

Applications:

- Web-browser, back function
- Undo
- Reversing order of elements
- Function stack

Stack underflow:

- When there is no element in the stack, the status of stack is known as o
  stack underflow.

Stack Overflow

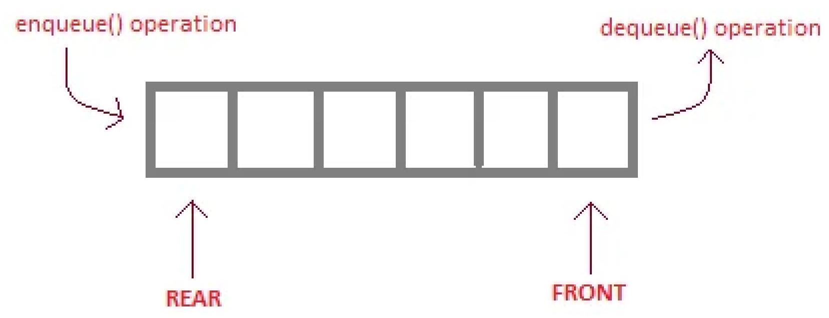
- When the stack contains equal number of elements as per its capacity and no more elements can be added, the status of stack is known as stack overflow

Stacks can be implemented in 2 ways,

- Static implementation (size fixed) : Arrays, inefficient memory usage
- Dynamic implementation (size grows) : Linked list , Efficient memory utilization through pointers

Queue

A queue is a linear data structure that follows the principle of First In First Out (FIFO). This means that the first element added to the queue will be the first one to be removed.



enqueue( ) is the operation for adding an element into Queue.

dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

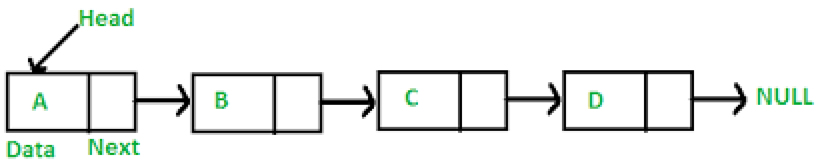
The queue data structure supports the following basic operations:

- enqueue : Enqueue refers to the process of adding an element to the back of the queue
- dequeue: Dequeue refers to the process of removing an element from the front of the queue.
- peek: Allows to view the element at the front of the queue without removing it
- is_Empty: checks if the queue is empty : return Boolean

```
class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        return self.items.pop(0)

    def peek(self):
        return self.items[0]

    def size(self):
        return len(self.items)

q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print(q.dequeue())  # prints 1
print(q.peek())  # prints 2
print(q.size())  # prints 2
```

Linked List

A linked list is a linear data structure that consists of a sequence of nodes, where each node stores a value and a reference (pointer) to the next node in the list. The first node in the list is called the head, and the last node is called the tail.



A linked list has several advantages over an array-based data structure, such as the ability to easily insert and delete elements without the need to shift other elements around. However, it has the disadvantage of being less efficient when it comes to accessing elements by their index, as this requires traversing through the list from the head node until the desired index is reached.

There are two main types of linked lists: singly-linked lists and doubly-linked lists. In a singly-linked list, each node only stores a reference to the next node, while in a doubly-linked list, each node stores a reference to both the next and previous nodes.

```
class Node:
    def __init__(self, val):
        self.val = val
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def isEmpty(self):
        return self.head == None

    def append(self, val):
        new_node = Node(val)
        if self.isEmpty():
            self.head = new_node
        else:
            curr = self.head
            while curr.next != None:
                curr = curr.next
            curr.next = new_node

    def prepend(self, val):
        new_node = Node(val)
        new_node.next = self.head
        self.head = new_node

    def delete(self, val):
        if self.isEmpty():
            return
        if self.head.val == val:
            self.head = self.head.next
        else:
            curr = self.head
            while curr.next != None and curr.next.val != val:
                curr = curr.next
            if curr.next != None:
                curr.next = curr.next.next

    def size(self):
        count = 0
        curr = self.head
        while curr != None:
            count += 1
            curr = curr.next
        return count
```

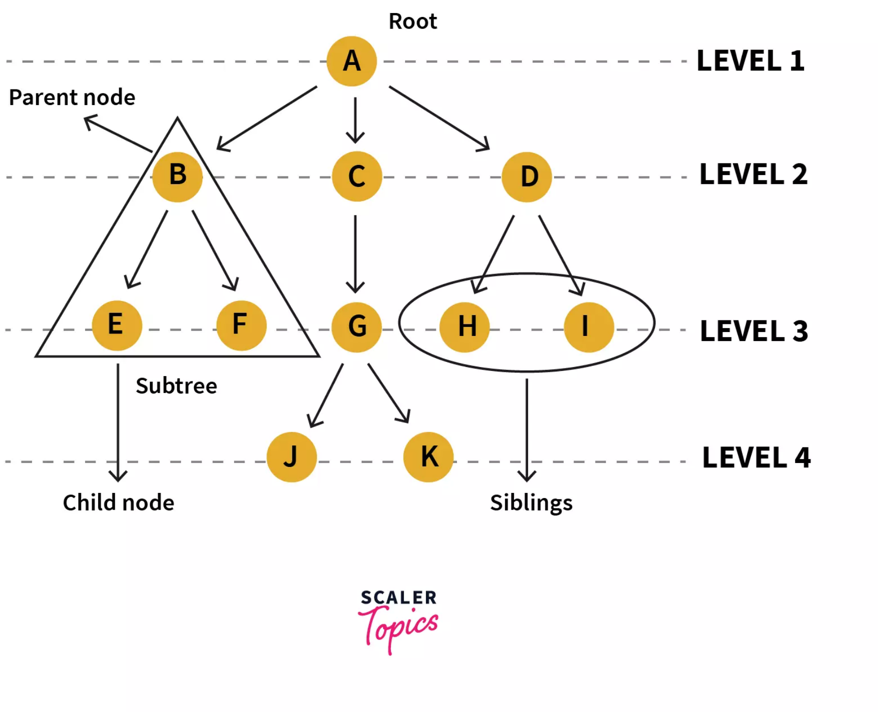
```
ll = LinkedList()
ll.append(1)
ll.append(2)
ll.append(3)
ll.prepend(0)
ll.delete(2)
print(ll.size())  # prints 3
```

It's important to note that both arrays and linked lists have their own set of advantages and disadvantages, and the choice of which one to use depends on the specific needs of the application. In general, arrays are best for cases where the size is known in advance and the elements need to be accessed quickly, while linked lists are better for cases where the size is not known in advance and the elements need to be inserted or deleted frequently.

Data Structure	Advantages	Disadvantages
Array	- Fast access to elements at known indices - Fast insertion and deletion of elements at the end of the array - Fixed size (if implemented as a static array)	- Slow insertion and deletion of elements at arbitrary indices - Size must be fixed at creation time or resizing may be necessary (if implemented as a dynamic array) - Wastes space if the array is not completely full
Linked List	- Fast insertion and deletion of elements at arbitrary indices - Dynamic size (no need to specify size at creation time) - Can save space if the list is not completely full	- Slow access to elements at known indices - Requires more memory (each element requires an extra pointer to the next element)

Tree data structure

A tree is a data structure that represents a hierarchical structure, with a set of nodes that are connected by edges. Each node in a tree can have one or more children, but only a single parent (except for the root node, which has no parent).



Here are the steps to understand a tree data structure:

1. The root node: This is the topmost node in the tree, and it has no parent. All other nodes in the tree are descendants of the root node.
2. The children of a node: Each node in a tree can have one or more children. These children are the nodes that are directly connected to the parent node via an edge.
3. The parent of a node: Each node in a tree (except for the root node) has a single parent. The parent is the node that is directly above the current node in the tree hierarchy.
4. The leaf nodes: Leaf nodes are nodes that have no children. They are the nodes at the bottom of the tree.
5. The level of a node: The level of a node is the distance from the root node to that node. For example, the root node is at level 0, its children are at level 1, and their children are at level 2.
6. The height of a tree: The height of a tree is the maximum level of any node in the tree.

7. The depth of a node: The depth of a node is the distance from that node to the deepest leaf node.
8. The siblings of a node: Siblings are nodes that have the same parent. For example, if a node has two children, those two children are siblings.
9. The subtree of a node: The subtree of a node is the set of nodes that are descendants of that node, including the node itself.

Trees are used in many applications, such as file systems, database indexing, and computer science algorithms. They offer an efficient way to store and organize data, and can be easily traversed to find specific pieces of information.