جامعة المستقبل
AL MUSTAQBAL UNIVERSITY

كلية العلوم

قســـم الانـظـمـة الـطبية الـذكـيـة

Intelligent Medical Systems Department

Lecture: (2)

Array - stack

Subject: Data structure

Class: second

Lecturer: Prof. Mehdi Ebady Manaa

Programmer:- Fatima Hussein Jawad

Study Year: 2024-2025

1. **The Array (Matray):** The Array is a data structure that contains a set of similar items in terms of type (all items are of the same type such as correct numbers or texts). These items are stored in consecutive locations from memory and any item can be accessed using an Index (index).

The Properties of Array:

1. Any item can be accessed using the item index.
2. The index starts from scratch.
3. The size of the array is static and can not be changed after creating.

```
# Creating an array-like structure using a list :

my_array = [1, 2, 3, 4, 5]

# Accessing elements of the array :
my_array = [1, 2, 3, 4, 5]
print(my_array[0])  # Output: 1 (Python uses zero-based indexing)
print(my_array[2])  # Output: 3

# Modifying elements of the array :
my_array = [1, 12, 3, 4, 5]
my_array[1] = 10
print(my_array)  # Output: [1, 10, 3, 4, 5]

# Iterating through the array :
my_array = [1, 10, 3, 4, 5]
for num in my_array:
    print(num)

# Output:
# 1
# 10
# 3
# 4
# 5
```

# Finding the length of the array :

```
my_array = [1, 10, 3, 4, 5]
array_length = len(my_array)
print("Length of the array:", array_length)

  # Output: Length of the array: 5
```

#print the elements of an array :

```
for num in my_array:
    print(num)

# output :
1
10
3
4
5
```

#sorted an array :

```
my_array = [1, 10, 3, 4, 5]
sorted_array=sorted(my_array)
print (sorted_array)
print()
print()
Output: [1, 3, 4, 5, 10]
```

#print out an array in reverse order

```
my_array = [1, 10, 3, 4, 5]
print('print an an array in reverse order')
reversed_array=sorted_array[1-::]
print(reversed_array)

output :
print an an array in reverse order
[5, 4, 3, 10, 1]
```

# Update the value of an element in the array:

```
arr = [1, 10, 3, 4, 5]
arr[1] = 25
print("After update:", arr)
```

output : After update: [1, 25, 3, 4, 5]

# Print all elements using a loop:

```
arr = [1, 10, 3, 4, 5]
for i in range(len(arr)):
print(f"The element {i} in the array is: {arr[i]}")
```

output :

```
The element 0 in the array is: 1
The element 1 in the array is: 10
The element 2 in the array is: 3
The element 3 in the array is: 4
The element 4 in the array is: 5
```

2. Stack:

A stack is a data structure that follows the LIFO (**Last In, First Out**) principle, meaning that the element added last is the one removed first. A stack is just like a pile of books: the book placed on top is the one removed first.

Characteristics of a Stack:

Only elements can be inserted and removed from the top of the stack.

Two main operations:

1. **Push**: Add an element to the top of the stack.

2. **Pop**: Remove an element from the top of the stack.

represent an array like a stack

# Initialize an empty list to represent the stack :

```
stack = []
```

# Push elements onto the stack :

```
stack = []
stack.append(1)
stack.append(2)
stack.append(3)
```

# Print the stack after pushing elements :

```
stack = []
stack.append(1)
stack.append(2)
stack.append(3)
print("Stack after pushing elements:", stack)
```

output :

```
Stack after pushing elements: [1, 2, 3]
```

# Pop elements from the stack :

```
stack = []
popped_element = stack.pop()
print("Popped element:", popped_element)
```

# Output:

```
Popped element: 3
```

# Print the stack after popping elements:

```
stack = [1, 2, 3]
popped_element = stack.pop()
print("Popped element:", popped_element)
```

print("Stack after popping elements:", stack)

# Output:
Popped element: 3
Stack after popping elements: [1, 2]

- ❖ In data structures like Stack, there are several operations that can be applied besides the basic ones (push, pop, peek). Operations like isSize() can be useful to check the size of the Stack.

Here are some additional operations that can be used with Stack:

1. Finding the size of the stack (Size):
You can find out the number of elements in the stack using len().

```
stack = [1, 2, 3]
size = len(stack)
print(size)
# Output: 3
```

2. Reordering the stack:
If you need to rearrange the stack, you can use the reverse() function to reverse the order of the elements.

```
stack.reverse() # Reverse the stack
```

3. Clearing the stack (Clear):
You can clear all the elements from the stack using the clear() function.

stack.clear() **# Clear all elements from the stack 4. Checking the existence of a specific element (Contains): You can check the existence of a specific element in the stack using in.**

```
stack = [1, 2, 3]
stack.clear()
print(stack)
```

```
# Output: []
```

4. Checking if the stack is empty (isEmpty):

```
#You can check if the list is empty using the condition:
stack = []
len(stack) == 0
if not stack:
        print("Stack is empty")
# Output: Stack is empty
```

5. Peeking the top of the stack (Peek):

Accessing the element at the top of the stack without removing it is done by accessing the last element in the list:

```
stack = [1,2,3,4,5,6,7,8,9,10]
top_element = stack[-1] # Peek at the top element without removing it
print(top_element)
# Output:10
```

Integrated example:

```
# Push elements onto the stack:
stack = []
stack.append(5)
stack.append(10)
stack.append(15)
# Peek at the top element:
print("Top element is:", stack[-1])
# Pop the top element :
popped_element = stack.pop()
print("Popped element:", popped_element)
# Check if stack is empty :
if not stack:
        print("Stack is empty")
```

```
else:
        print("Stack has", len(stack), "elements")
# Clear the stack
stack.clear()
print("Stack after clearing:", stack)
# Output:
Top element is: 15
Popped element: 15
Stack has 2 elements
Stack after clearing: []
```

Comparison between Array and Stack:

Array:

- ✓ Suitable for storing sequential data and random access.
- ✓ Not suitable if you need to add or remove from the beginning or middle of the list continuously.

Stack:

- ✓ Suitable for operations that require access to the last element added first (LIFO).
- ✓ Used in applications that need to keep track of the context of operations, such as executing nested functions or parsing expressions.