Topic 6 - Linear data structures

Introduction to data structures

Definition of data structure

A data structure is a **container of data** where **data is organized in a very specific way**. For example, lists are organized in a linear manner, whereas trees and heaps are organized in a hierarchical way. Finally, every data structure has a **set of specific operations or functions to access and manipulate the data** stored in it

Algorithms	Analysis of algorithms
	Recursion
	Sorting
	Hasing

Data Structures	Lists, stacks, queues, hash tables, arrays
	Trees
	Heaps
	Graphs

Linked List Introduction

Arrays and linked lists are very different in the way they access and manipulate data. The core difference between them is the way they are stored in memory.

Array

X		X	X	X	X	X			
2	3	5	7				X	X	
	X	X	X	X		X			
			X		X	X	X	X	

The system needs to search for a chunk of n available **contiguous** memory positions. As a result of arrays being allocated contiguous memory positions, moving through them using a for loop is easy

Linked list

A linked list is created **one element at a time**. **No contiguous** position
Each element in the list not only must **store the data itself,** but also **the address of the memory position of the next element** in the list

Creating a Linked list:

1) Address of the first node stored in the (Head)
2) The element is stored in the node and the address of the next element is also stored
3) The last value contains the element and a NULL value for the address

[2]→[3]→[5]→[7]⊐

X	Head 0x3	X	2 0x7	X	X	X	3 0xA	X	X
5 0xD	X	X	7 ⊐				X	X	
	X	X	X	X		X			
			X		X	X	X	X	
X	X	X	X	X	X	X	X	X	X

Linked List INSERT Operation

Head [0x9] → [3 | 0xE] → [7 | 0x16] → [2 | 0x6] → [9 | NULL]

Node: [data | next]

Steps to insert data

1) Create a new node EG: [4 | null]
2) Linking the node to the list. One of the nodes or the head must point to the new node, and the new node must point to some node of the list or to null.
   - Insert as first node:
   - New node => next = head [4 | 0x9]

- Head = New Node

Head [0xF] → [4 | 0x9] → [3 | 0xE] → [7 | 0x16] → [2 | 0x6] → [9 | NULL]

```
head: contains address of first element of list
x: number to insert
T(N) = Θ(1)

function INSERT ( head, x )
      newNode = New Node(x)
      newNode->head = head
      head = newNode
```

Other versions of insert

- Insert **node at the end** of the linked list. Last node has to point to the address of the new node
- Insert a **node in between other nodes**. The new node has to point to the next node, and the previous node has to point to the address of the new node

Linked List DELETE Operation

```
x: data to be found in the linked list          list: linked list
function DELETE ( list, x )
      // Pointers to traverse the list
      Node tmp = head;
      Node prev
      // Verification if the list is not empty
      if tmp == NULL
            print ("There is nothing to delete")
            return list
      else
            if (tmp == x )      // delete if x is in the first node
                  head = tmp->next
                  return list
            else  // traverse the list until find x
                  prev = tmp
                  tmp = tmp-> next;
                  while(tmp != NULL)
                        if( tmp->data == x )
                              prev->next = tmp->next
                              return list
                        prev = tmp
                        tmp = tmp->next
                  print ("Element not found")
end function
```

Note: Some languages ask for an explicit instruction to release that memory position, and some other languages do it automatically when an element is no longer pointed out by any other element in the program.

Linked List Summary:

- The **operation search** is embedded in the operation delete
- The **complexity of insert** depends on whether you insert the new node at the start, the end, or at any arbitrary position of the list.
- Inserting an Item Time complexity:
  - Best Case: $T(N) = \Theta(1)$ (First node)
  - Worst Case: $T(N) = \Theta(N)$ (Last node)
- Delete an item Time Complexity:
  - Best Case: $T(N) = \Theta(1)$ (First node)
  - Worst Case: $T(N) = \Theta(N)$ (Last node)
- Search an item Time Complexity
  - Best Case: $T(N) = \Theta(1)$ (First node)
  - Worst Case: $T(N) = \Theta(N)$ (Last node)
  - **Note**: Hashing is not possible with linked lists

Doubly Linked List

- Each node has three fields, the address of the **next node** and the address of the **previous node**

Circular Linked List

- The last node of the list **points to the first node** of the list. EG: a playlist

Stacks introduction

- Objects can only be inserted at the top of the stack
- The insertion operation in a stack is called a: **push()**
- The removal operation in a stack is called a: **pop()**
- The only element you can look at is the one at the top of the stack: **peek()**
- The operation **isEmpty()**, returns true if the stack is empty, and false if it has at least one element

Stacks Implementation

- A stack is a linear data structure. So it can be implemented using other linear data structures, **an array, or a linked list**
- If arrays is used, you might run out of space or have space used for nothing

Arrays Implementation

```
T(N) = Θ(1)
T(N) = Θ(N) if the array runs out of space and the array needs to
duplicate the size.
function PUSH( x )

      // Use this if run out of space
      if ( top == size(A)-1 )
            print( "Stack overflow" )
            return

      top = top +1
      A[top] = x

T(N) = Θ(1)
function POP( )
      if (top == 0 )
            print("Empty Stack")
            return
      top = top -1

T(N) = Θ(1)
function PEEK( )
      if (top == -1 )
            print("Empty Stack")
            return
      return A[top]

T(N) = Θ(1)
function ISEMPTY( )
      if (top == -1 )
            return True
      return false
```

Linked List Implementation

```
T(N) = Θ(1)          [top|0xE]→[data|0x16]→[  |NULL]
function PUSH( x )
      newNode = new Node (data)
      newNode->next = top
      top = newNode
```

```
T(N) = Θ(1)
function POP( )
      if (top == NULL)
            print("Empty list")
            return
      top = top->next

T(N) = Θ(1)
function PEEK( )
      if (top == NULL)
            print("Empty list")
            return
      return (top->data)

T(N) = Θ(1)
function ISEMPTY( )
      if (top == NULL)
            return True
      return false
```

Queues Introduction

- Queue works in a FIFO manner, the first element in is the first element out.
- The insertion of a new element to the queue is called **enqueue()**
- The removal of an element from the queue is called **dequeue()**
- So operations **peek()** returns the value of the element at the front of the queue
- The operation **isempty()** returns the Boolean variable true if the queue is empty and false if it has at least one element.

Queues: Array-based implementation

- The array and the variables tail and front are **equal to -1** to signal an **empty queue**
- The array and the variables tail and front are **equal to 0** to signal an **queue with 1 item**

```
N: Size of the array       ;     x:data     ;     T(N) = Θ(1)
function ENQUEUE(x)
      if ((tail+1) % N == front))
            print ("Queue is full")
            return
      if (ISEMPTY())
            front = 0
            tail = 0
      else
            tail = (tail+1) % N
      A[tail] = x
```

```
N: Size of the array       ;     T(N) = Θ(1)
function DEQUEUE( )
      if (ISEMPTY())
            print("Queue is empty")
            return
      if (front == tail)
            front = -1;
            tail = -1;
      else
            front = (front+1) % N

T(N) = Θ(1)
function PEEK( )
      if (front == -1)
            print("Queue is empty")
            return
      return A[front]

T(N) = Θ(1)
function ISEMPTY( )
      if (front == -1)
            return True
      return False
```

Queues: List-based implementation

- We need to use two pointers to have T(N) = Θ(1), front and tail
- If using a single point in a linked list T(N) = Θ(N) because the algorithm require to traverse the list

```
function ENQUEUE(x)
      newNode = new Node(x)
      if (front == NULL && tail == NULL)
            front = newNode
            tail = newNode
      else
            tail->next = newNode // point previous node to last node
            tail = newNode // point tail to last node
```

```
function DEQUEUE()

      if (front == NULL && tail == NULL)
            print ("Empty Queue")
            return
      if (front == tail)
            front = NULL
            tail = NULL
      else
            front= front->next
            tail = newNode

function PEEK()
      if (front == NULL && tail == NULL)
            print("Empty Queue")
            return
      else
            return front->data
function ISEMPTY()
      if (front == NULL && tail == NULL)
            return True
      return False
```