ADTs & Algorithms Lecture 2

Prof Muliaro Wafula

ABSTRACT DATA TYPE

- Specifies the logical properties of data type or data structure.
- Refers to the mathematical concept that governs them.
- They are not concerned with the implementation details like space and time efficiency.
- define data structures based on their behavior from the user's perspective, rather than their implementation
- ADT defines the types of operations that can be performed on the data, but not how these operations are implemented.
- ADT describes *what* data structure does, not *how* it is done.
- Programmers use ADT without needing to understand the underlying complexity.x`x x

- They are defined by 3 components called Triple =(D,F,A)
  - ✓ D=Set of domain
  - ✓ F=Set of function
  - ✓ A=Set of axioms / rules

Examples: Stack (LIFO - Last In, First Out)

A stack is an ADT where elements are added and removed in a particular order:
the last element added is the first one to be removed.

Operations:

push(x) Adds an element x to the top of the stack.

- **pop()**: Removes and returns the top element.
- **peek()**: Returns the top element without removing it.
- **isEmpty()**: Checks if the stack is empty.

NOTE: Function calls in programming languages use a stack to keep track of active subroutines.
When a function is called, it is "pushed" onto the stack, and when it completes, it is "popped" off.

Queue (FIFO - First In, First Out)

A queue is an ADT where elements are added at one end (rear) and removed from the other end (front).

Operations:

enqueue(x): Adds an element `x` to the end of the queue.

- dequeue(): Removes and returns the element at the front.
- peek(): Returns the element at the front without removing it.
- isEmpty(): Checks if the queue is empty.

- Example:

➢ Think of a line at hospital/Bank. The first person in line is the first to be attended to by the Doctor/Teller.

➢ Application: Task scheduling in operating systems often uses queues to manage processes, where the first process added to the queue is the first to be executed.

List

A list is an ADT where elements are stored sequentially and can be accessed by their index. **Operations**:

insert(index, x): Inserts an element `x` at a specific index.

- remove(index): Removes the element at a specific index.
- get(index): Returns the element at a specific index.
- size(): Returns the number of elements in the list.

Example:

- A list of registered members where you can access any member by their position (index).

Application: Lists are commonly used to store ordered data, eg like a playlist, in a music app where you can jump to a specific song by its position. Class list, inventory etc

Set

A set is an ADT that stores unique elements (no duplicates) and typically has no defined order.

Operations:

add(x): Adds element `x` to the set.

- - **remove(x)**: Removes element `x` from the set.
- - **contains(x)**: Checks if element `x` is in the set.
- - **union(S)**: Combines two sets into one.
- - **intersection(S)**: Returns elements that exist in both sets.

Example:

- ➢ A collection of unique student IDs, Car Reg Numbers etc.
- ➢ Application: Sets are used in database query operations where duplicate entries must be eliminated, or when checking memberships in groups.

Dictionary (or Map)

A dictionary (or map) is an ADT that stores key-value pairs, where each key is unique and maps to a specific value.

Operations:

put(key, value): Adds a key-value pair to the dictionary.

- get(key): Returns the value associated with the key.
- remove(key): Removes the key and its associated value.
- containsKey(key): Checks if the dictionary contains the key

Example:

- A phone book where you look up a phone number (value) based on a person’s name (key).
- Country and Capital Cities etc

The Array as an Abstract Data Type

- The Array ADT is a set of values (index, item) and a set of operations known as Array create(), Item Retrieve(), and Array Store().
- The Array ADT algorithm is given by
- Array ADT is
- **objects:** A set of pairs <index,item> where for each index there is an item.
- **functions:** for all A€Array, i€index, x€item
- Array create() - It creates a new empty array
- Item Retrieve(A,i) - It returns a value with a particular index, if the index is valid or an error if the index is invalid
- Aray Store(A,i,x) - It stores an item
- end Array

The Stack as an Abstract Data Type:

ADT stack is

- **objects:** a finite ordered list with zero or more elements
- **functions:** S € Stack, item € Element
- Stack create() := create an empty stack
- Stack push(S,item) := insert item into top of stack
- Element pop(S) :=remove and return the item at the top of the stack
- end Stack

The Queue as an Abstract Data Type

- ADT queue is
- **objects:** a finite ordered list with zero or more elements
- **functions:** Q € Queue, item € Element
- Queue Create() := create an empty stack
- Queue addq(Q,item) := insert item into queue
- Element deleteq(Q) :=remove and return the item from the queue
- end Stack

The Binary Tree as an Abstract Data Type

- ADT BinaryTree is
- **objects:** a finite set of nodes
- **functions:** for all bt, bt1, bt2 ϵ BinTree, item ϵElement
- BinTree Create() := create an empty BinaryTree
- Boolean IsEmpty := if(bt== empty binary tree) return TRUE else return FALSE
- BinTree MakeBT(bt1,item,bt2) := return a binary tree whose left subtree is bt1,
- whose right subtree is bt2, and whose root node contains the data item.
- BinTree Lchild(bt) := It return left subtree of bt
- Element Data(bt) := It return the data in the root node of bt
- BinTree Rchild(bt) := It return the right subtree of bt
- End ADT BinaryTree

The Graph as an Abstract Data Type

- **ADT** Graph is
- **objects:** a set of vertices & edges
- **functions:** for all graph ϵ Graph, v, v1 and v2 ϵ Vertices
- Graph Create() := create an empty Graph
- Boolean IsEmpty() := if(graph== empty graph) return TRUE else return FALSE
- Graph InsertVertex(graph,v) := return a graph with v inserted.
- Graph InsertEdge(graph,v1,v2) := return a graph with a new edge (v1, v2) is inserted.
- Graph DeleteVertex(graph,v) := return a graph in which vertex v is Removed
- Graph DeleteEdge(graph,v1,v2) := return a graph in which the edge (v1,v2) is removed
- End ADT Graph

Properties of an Algorithm

- The properties of an algorithm is given by
- **Input:**
- The algorithm must have input values from a specified set
- **Output:**
- The algorithm must produce the output values from a specified set of input values. The output values are the solution to a problem.
- Finiteness:
- For any input the algorithm must terminate after a finite number of steps.
- **Definiteness:**
- All the steps of the algorithm must be precisely defined.
- **Effectiveness:**
- It must be possible to perform each step of the algorithm correctly and in a finite amount of time.
- ➢ Each step should be well defined. It can be divided into 3 types

1. Sequence:

- In an algorithm if all steps are shown then it is known as sequence.
- For example an algorithm for adding two values.

```
Ex: Step 1 : start
      Step 2 : read a,b
      Step 3 : r=a+b
      Step 4 : print r
      Step 5 : stop
```

2. Selection

- ❖ Here we use if condition and the condition is checked only one time.
- ❖ If a condition is satisfied then next statement is executed otherwise else statement is executed
- ❖ For example an algorithm to check whether the given number is even or odd.

```
Ex: Step 1 : start
    Step 2 : read n
    Step 3 : if(n%2==0) goto step 4
             if not goto Step 6
    Step 4 : print "Even no" goto Step 7
    Step 5 : else
    Step 6 : print "Odd no"
    Step 7 : stop
```

3. Iteration

- ❖ Here we use while, do-while and for & the condition is checked number of times.
- ❖ i.e.The statements in an iteration block are executed no. of times based on some condition.
- ❖ For example an algorithm to print 1 to n numbers using while loop.

```
Ex:  Step 1 : start
     Step 2 : read n
     Step 3 : initialize i=1
     Step 4 :while(i<=10) goto Step-5
                 otherwise goto Step 7
     Step 5 : print  i
     Step 6 : compute i++ goto Step 4
     Step 7 : stop
```

- ➢ When an algorithm gets coded in a specified programming language such as C, C++, or Java, it becomes a program that can be executed on a computer.
- ➢ Multiple algorithms can exist to solve the same problem or complete the same task.
- ➢ The appropriate algorithm can be determined based on an number of factors:

1. How long the algorithm takes to run
2. What resources are required to execute the algorithm
3. How much space or memory is required
4. How exact is the solution provided by the algorithm

Time and Space complexity

Algorithm: Step by step process of solving a problem is called an algorithm.The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process. There are two main complexity measures of the efficiency of an algorithm

- **(i)Time complexity:**
- ➢ Time complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.
- ➢ Time means the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed or some other natural unit related to the amount of real time the algorithm will take

- **Space complexity:**
- ➢ Space complexity is a function describing the amount of memory an algorithm takes in terms of the amount of input to the algorithm. We often speak of extra memory needed, not counting the memory needed to store the input itself. We can use bytes, but it's easier to use say number of integers used, number of fixed-sized structures etc.
- ➢ In the end the function will be independent of the actual number of bytes needed to represent the unit. Space complexity is sometimes ignored because the space used is minimal.
- ➢ The complexity of an algorithm is studied with respect to the following 3 cases

- **Worst Case Analysis:**
- ➢ In the worst case analysis, we calculate upper bound on running time of an algorithm.
- ➢ We must know the case that causes maximum number of operations to be executed.
- ➢ For Linear Search the worst case happens when the element to be searched is not present in the array.

- **(b)Average Case Analysis:**
- ➢ In the average case analysis, we calculate average on running time of an algorithm. We must know the average number of operations to be executed.
- ➢ In the linear search problem, the average case occurs when x is present at average of its location.

- **(c)Best Case Analysis:**
- ➢ In the best case analysis, we calculate lower bound on running time of an algorithm.
- ➢ We must know the case that causes minimum number of operations to be executed
- ➢ In the linear search problem, the best case occurs when x is present at the first location