ADT

Abstract Data Type-Definition

- Defined as a "class of objects whose logical behavior is defined by **a set of values and a set of operations“**
- The definition of ADT only mentions
  - **what operations are to be performed**
  - **but not how these operations will be implemented.**

Courtesy: https://www.geeksforgeeks.org/abstract-data-types/

Abstract Data Type-Definition

- A mathematical model, together with various operations defined on the model
- **An ADT is a collection of data and associated operations for manipulating that data**

Courtesy: https://www.geeksforgeeks.org/abstract-data-types/

Abstract Data Type

- It does not specify
  - how data will be organized in memory and
  - what algorithms will be used for implementing the operations.
  - Not concerned with space and time complexity
  - Not concerned with implementation details at all

Courtesy: https://www.geeksforgeeks.org/abstract-data-types/

Abstract Data Type

- May not even be possible to implement a particular ADT on a particular piece of hardware or using a particular software system

Courtesy: https://www.geeksforgeeks.org/abstract-data-types/

Abstract Data Type

Abstract-

- Because it gives an implementation-independent view.
- **The process of providing only the essentials and hiding the details is known as abstraction.**

Encapsulation-

- Think of ADT as a black box which hides the inner structure and design of the data type.
- The principle of **hiding the used data structure and to only provide a well-defined interface** is known as encapsulation.

Abstract Data Type

- ADTs support abstraction, encapsulation, and information hiding.

ADT Operations

Every Collection ADT should provide a way to:

- **Create data structure**
- **add an item**
- **remove an item**
- **find, retrieve, or access an item**

No single data structure works well for all purposes, and so it is important to know the strengths and limitations of several of them

ADT Syntax : Value Definition

Abstract typedef < *ParameterType Parameter1, ParameterType Parameter2……, ParameterType ParameterN* > ADTType

condition:

ADT Syntax : Operator definition

Abstract ReturnType OperationName (ParameterType Parameter1, ParameterType Parameter2……, ParameterType ParameterN)

Precondition:

Postcondition:

OR

Abstract ReturnType OperationName (Parameter1, Parameter2……, ParameterN)

ParameterType Parameter1, ParameterType Parameter2……, ParameterType ParameterN

Precondition:

Postcondition:

Example ADT : String

- Definition: String is a sequence of characters
- Operations:
  - StringLength
  - StringCompare
  - StringConcat
  - StringCopy

Example ADT : String

- **Value Definition**

Abstract Typedef <<Char s>>StringType

Condition: None (A string may contain n characters where n>=0)

ADT Syntax : Value Definition

Abstract typedef < *ParameterType Parameter1, ParameterType Parameter2……, ParameterType ParameterN* > ADTType

condition:

ADT Syntax : Operator definition

Abstract ReturnType OperationName (ParameterType Parameter1, ParameterType Parameter2……, ParameterType ParameterN)

Precondition:

Postcondition:

OR

Abstract ReturnType OperationName (Parameter1, Parameter2……, ParameterN)

ParameterType Parameter1, ParameterType Parameter2……, ParameterType ParameterN

Precondition:

Postcondition:

Example ADT : String Operator Definition

1. abstract Integer StringLength (StringType String)

Precondition: None (A string may contain n characters where n=>0)

Postcondition: Stringlength= NumberOfCharacters(String)

Example ADT : String Operator Definition

2. **abstract StringType** StringConcat(StringType String1, StringType String2)

Precondition: None

Postcondition: StringConcat= String1+String2 / All the characters in Strings1 immediately followed by all the characters in String2 are returned as result.

Example ADT : String Operator Definition

3. **abstract Boolean** StringCompare(StringType String1, StringType String2)

Precondition: None

Postcondition: StringCompare= True if strings are equal, StringCompare= False if they are unequal . (Function returns 1 if strings are same, otherwise zero)

Example ADT : String Operator Definition

4. **abstract StringType** StringCopy(StringType String1, StringType String2)

Precondition: None

Postcondition: StringCopy: String1= String2 / All the characters in Strings2 are copied/overwritten into String1.

Example ADT : Rational Number

- **Definition: expressed as the quotient or fraction of two integers,**
- **Operations:**
  - **makeRational()**
  - **IsEqualRational()**
  - **MultiplyRationa()**
  - **AddRational()**

Example ADT : Rational Number

- **Value Definition**

abstract TypeDef<integer x, integer y> RATIONALType;

Condition: RATIONALType [1]!=0;

ADT Syntax : Value Definition

Abstract typedef < *ParameterType Parameter1, ParameterType Parameter2……, ParameterType ParameterN* > ADTType

condition:

ADT Syntax : Operator definition

Abstract ReturnType OperationName (ParameterType Parameter1, ParameterType Parameter2……, ParameterType ParameterN)

Precondition:

Postcondition:

OR

Abstract ReturnType OperationName (Parameter1, Parameter2……, ParameterN)

ParameterType Parameter1, ParameterType Parameter2……, ParameterType ParameterN

Precondition:

Postcondition:

Example ADT : Rational Number Operator Definition`

- abstract RATIONALType makerational<x,y>

integer x,y;

Preconditon: y!=0

postcondition :

makerational [0] =x

makerational [1] =y

- abstract RATIONALtype add<a,b>

RATIONALType a,b;

Precondition: none

postcondition :

add[0] = a[0]*b[1]+b[0]*a[1]

add[1] = a[1] * b[1]

Example ADT : Rational Number Operator Definition

abstract RATIONALType
mult<a, b>

RATIONALType a,b;

Precondition: none

Postcondition :

mult[0] = a[0]*b[0]

mult[1] = a[1]*b[1]

abstract ReturnType?
Equal<a,b>

RATIONALType a,b;

Precondition: none

Postcondition: equal=true if a[0] * b[1] = = b[0] * a[1]

ADT Syntax : Operator definition

Abstract ReturnType OperationName (ParameterType Parameter1, ParameterType Parameter2……, ParameterType ParameterN)

Precondition:

Postcondition:

OR

Abstract ReturnType OperationName (Parameter1, Parameter2……, ParameterN)

ParameterType Parameter1, ParameterType Parameter2……, ParameterType ParameterN

Precondition:

Postcondition:

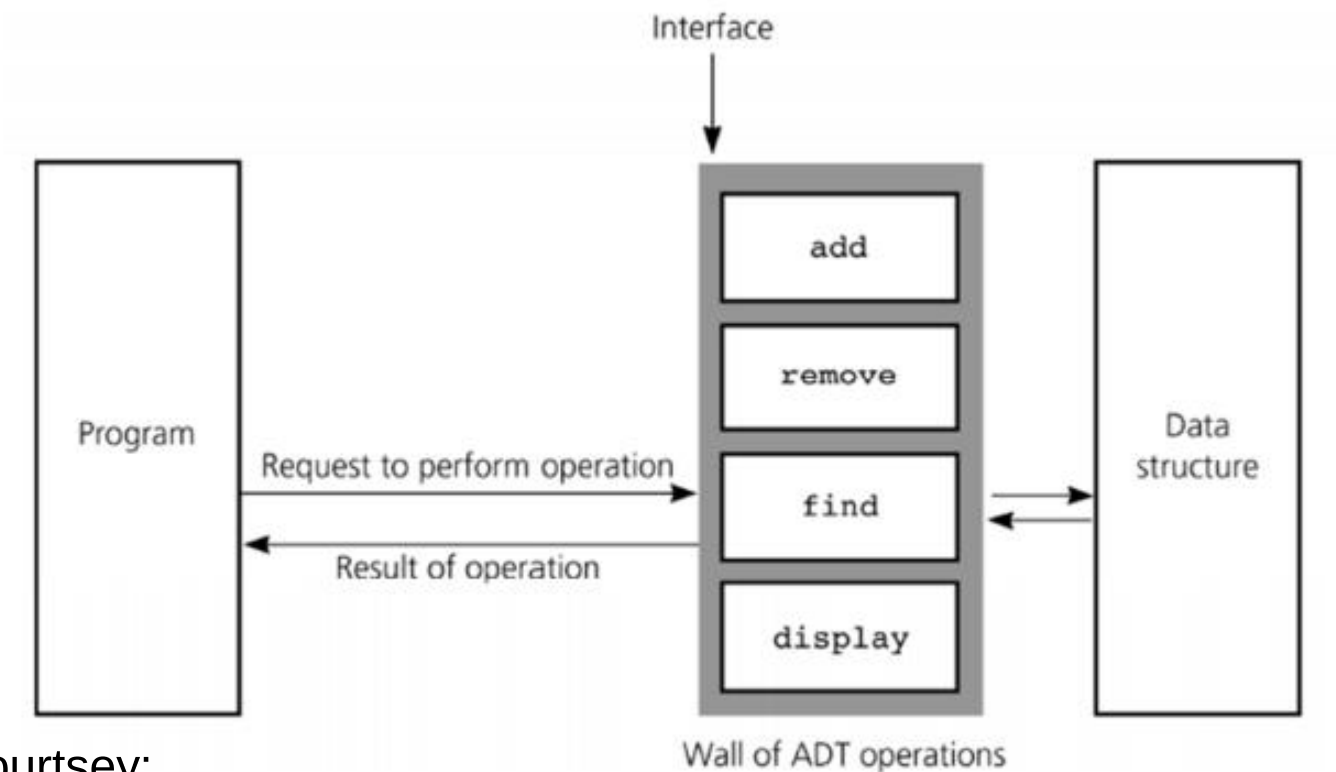
Abstract Data Types: Advantages

- Hide the unnecessary details by building walls around the data and operations
  - so that changes in either will not affect other program components that use them
- Functionalities are less likely to change
- Localize rather than globalize changes
- Help manage software complexity
- Easier software maintenance

Courtsey: https://www.comp.nus.edu.sg/~stevenha/cs1020e/lectures/L5%20-%20ADT.pdf

A wall of ADT operations

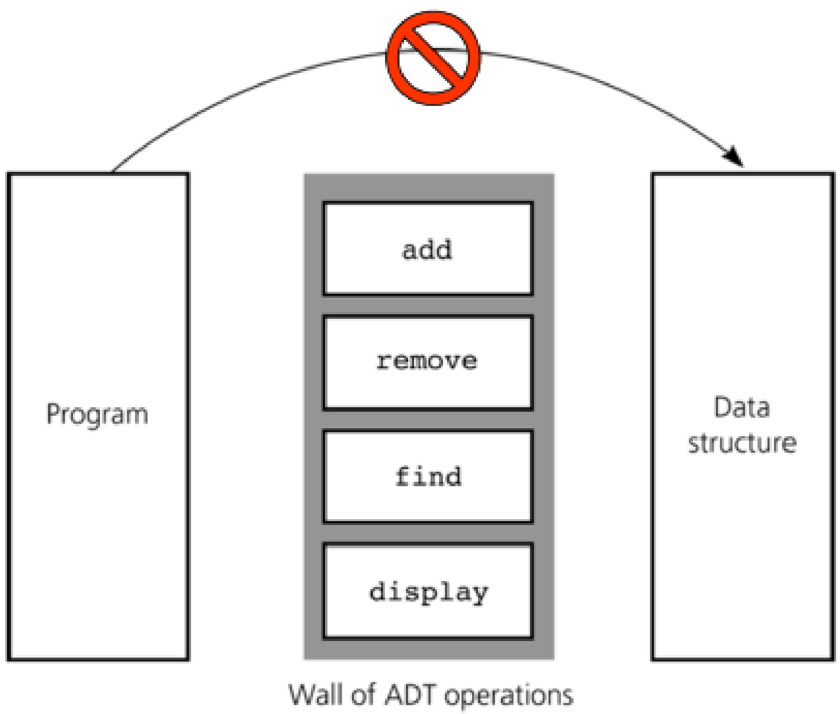
- ADT operations provides:
  - Interface to data structure
  - Secure access



Courtsey: https://www.comp.nus.edu.sg/~stevenha/cs1020e/lectures/L5%20-%20ADT.pdf

Violating the Abstraction

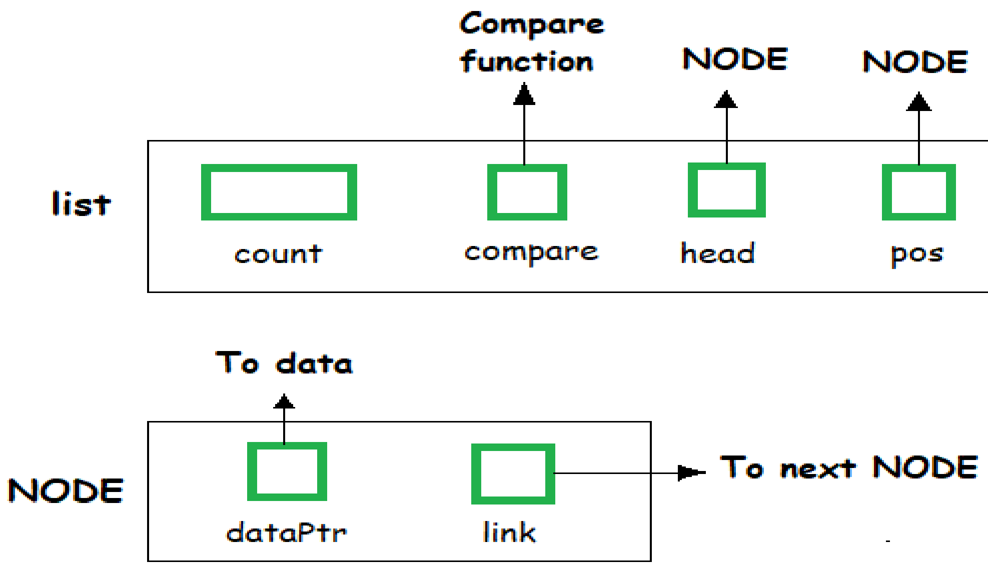
- User programs **should not**:
  - Use the underlying data structure directly
  - Depend on implementation details



Courtsey:
https://www.comp.nus.edu.sg/~stevenha/cs1020e/lectures/L5%20-%20ADT.pdf

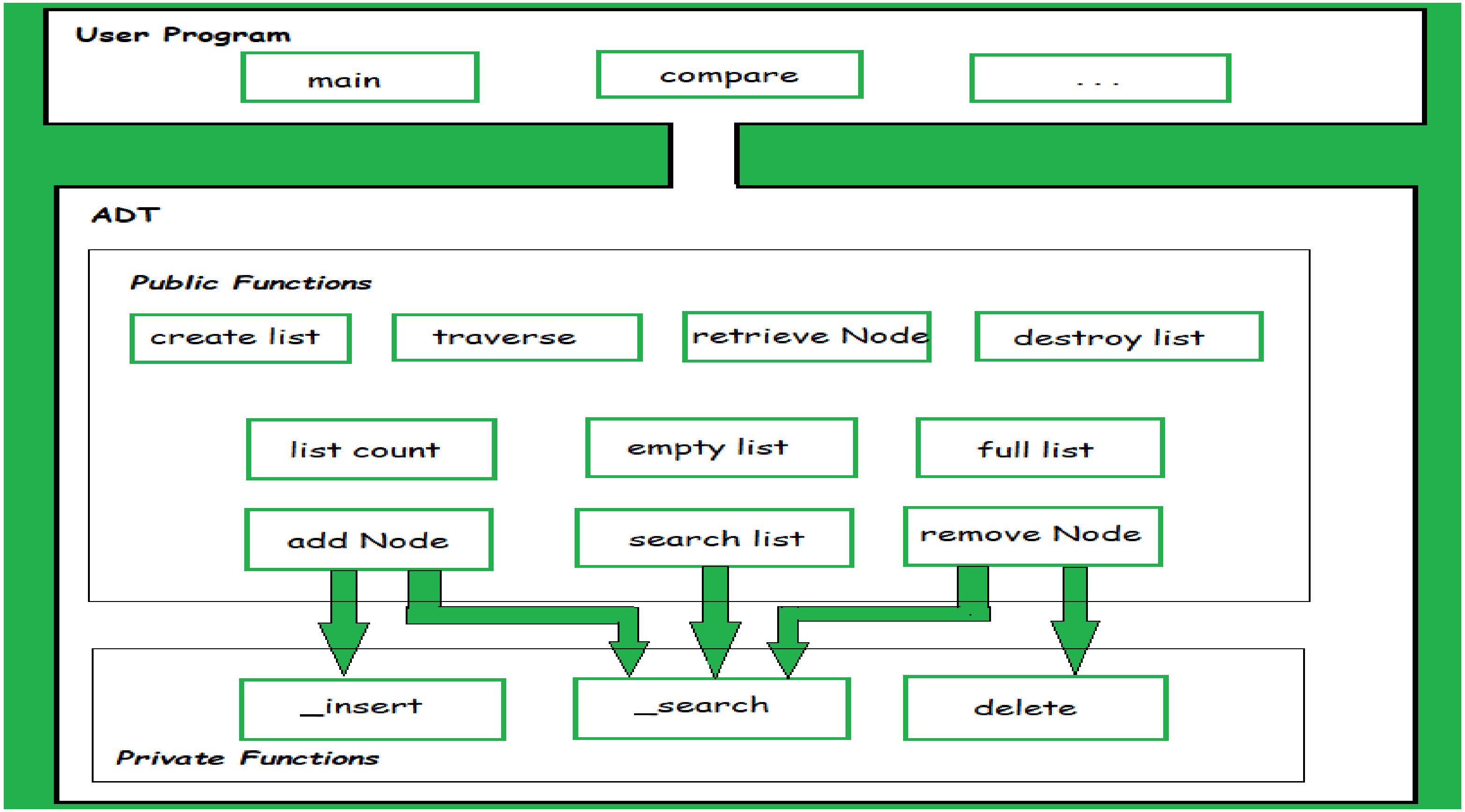
List ADT

- The data is generally stored in key sequence in a list which has a head structure consisting of *count*, *pointers* and *address of compare function* needed to compare the data in the list.



List ADT

- List ADT Functions

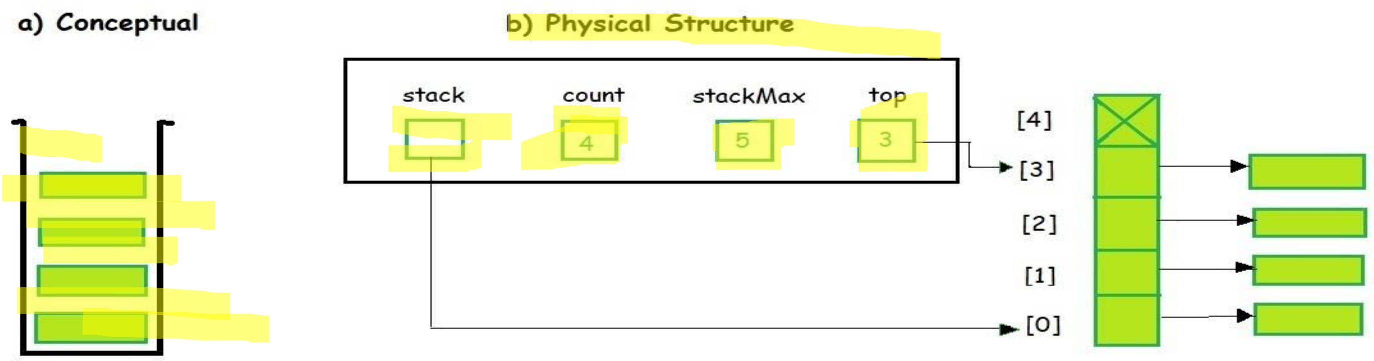


List ADT

- List ADT Functions
- get() – Return an element from the list at any given position.
- insert() – Insert an element at any position of the list.
- remove() – Remove the first occurrence of any element from a non-empty list.
- removeAt() – Remove the element at a specified location from a non-empty list.
- replace() – Replace an element at any position by another element.
- size() – Return the number of elements in the list.
- isEmpty() – Return true if the list is empty, otherwise return false.
- isFull() – Return true if the list is full, otherwise return false.

Stack ADT

- Instead of data being stored in each node, the pointer to data is stored.
- The program allocates memory for the *data* and *address* is passed to the stack ADT.



Stack ADT

- The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.
- The stack head structure also contains a pointer to *top* and *count* of number of entries currently in stack.

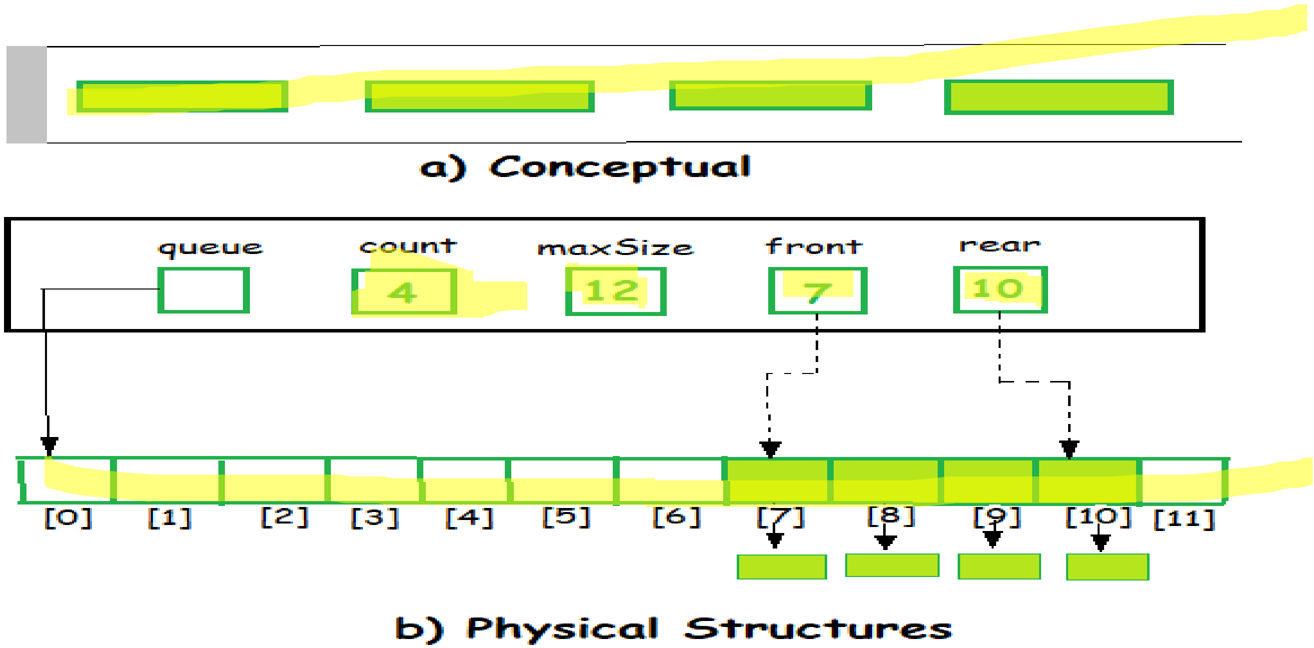
Stack ADT

- A Stack contains elements of the same type arranged in sequential order.
- All operations take place at a single end that is top of the stack and following operations can be performed:

- **push() – Insert an element at one end of the stack called top.**
- **pop() – Remove and return the element at the top of the stack, if it is not empty.**
- **peek() – Return the element at the top of the stack without removing it, if the stack is not empty.**
- **size() – Return the number of elements in the stack.**
- **isEmpty() – Return true if the stack is empty, otherwise return false.**
- **isFull() – Return true if the stack is full, otherwise return false.**

Queue ADT

- The queue abstract data type (ADT) follows the basic design of the stack abstract data type.



- Each node contains a void pointer to the *data* and the *link pointer* to the next element in the queue. The program's responsibility is to allocate memory for storing the data.

Queue ADT

- A Queue contains elements of the same type arranged in sequential order.
- Operations take place at both ends, insertion is done at the end and deletion is done at the front.

Queue ADT

- Following operations can be performed:

- **enqueue() – Insert an element at the end of the queue.**
- **dequeue() – Remove and return the first element of the queue, if the queue is not empty.**
- **peek() – Return the element of the queue without removing it, if the queue is not empty.**
- **size() – Return the number of elements in the queue.**
- **isEmpty() – Return true if the queue is empty, otherwise return false.**
- **isFull() – Return true if the queue is full, otherwise return false.**