CC-213L

Data Structures and Algorithms

Laboratory 03

Abstract Data Types (ADT's)

Version: 1.0.0

Release Date: 03-10-2024

Department of Information Technology

University of the Punjab

Lahore, Pakistan

Contents:

Learning Objectives:

- C++ Pointers
- Dynamic Memory Allocation
- Understanding the Concept of an ADT
- Arrays

Resources Required:

- Desktop Computer or Laptop
- Microsoft ® Visual Studio 2022

General Instructions:

- In this Lab, you are **NOT** allowed to discuss your solution with your colleagues, even not allowed to ask how is s/he doing, this may result in negative marking. You can **ONLY** discuss with your Teaching Assistants (TAs) or Lab Instructor.
- Your TAs will be available in the Lab for your help. Alternatively, you can send your queries via email to one of the followings.

Teachers:		
Course / Lab Instructor	Prof. Dr. Syed Waqar ul Qounain	swjaffry@pucit.edu.pk
Teacher Assistant	Muhammad Tahir Mustafvi	Bcsf20m018@pucit.edu.pk

Background and Overview:

Pointers:

In C++, pointers are **variables** that hold **memory addresses**. They're declared with the data type they point to, initialized with **the address** of other variables, and can be used to access or modify the values at those addresses using the dereference operator. Null pointers indicate uninitialized or invalid addresses. Pointers are commonly used for dynamic memory allocation and can navigate arrays and data structures through pointer arithmetic. They also enable the use of function pointers for dynamic function invocation. However, proper handling is crucial to avoid null pointer errors and memory leaks. In modern C++, **smart pointers are recommended for safer memory management**, particularly in dynamic memory allocation scenarios.

```cpp
#include <iostream>
using namespace std;
int main()
{
    int x = 42;
    int* ptr = &x; // 'ptr' now points to the memory address of 'x'

    int y = *ptr; // 'y' is now 42, the value at the address pointed to by 'ptr'

    int* null_ptr = nullptr;

    //OR use NULL

    null_ptr = NULL;

}
```

Fig.01 (Pointers)

Dynamic Memory Allocation:

Dynamic memory allocation in programming refers to the process of requesting and managing memory during runtime rather than at compile time. It allows programs to allocate memory as needed, which can be particularly useful when you don't know the size of data structures or objects in advance. In languages like C++ and C, dynamic memory allocation is typically achieved using functions like **new** (in C++) and `malloc` (in C) to allocate memory on the heap and **delete** (in C++) and `free` (in C) to deallocate it when it's no longer needed. Here's an example in C++:

```cpp
#include <iostream>
using namespace std;
int main()
{
    int* ptr = new int;

    *ptr = 10;

    cout << *ptr << endl;

    delete ptr;

    int* arr = new int[10];

    delete[] arr;

    system("color 70");
    return 0;
}
```

Fig.02 (Dynamic Memory)

Explanation:

In our example code, we defined a pointer to integer and then dynamically allocated it a memory of integer. After dereferencing it we have assigned **a**n integer value to that memory. We have displayed

that value. Then we have deallocated it. Same we have done by allocating an array of integers dynamically and then assigned base address of array to pointer arr. After that we have deallocated it.

Fig.03 (Dynamic Memory)

Abstract Data Types:

An ADT is a language independent concept. As its specification can be written in any language.

Two necessary things for an ADT

a) Data

b) Operations On data

ADT on bigger level becomes class. The data becomes **data members / member variables** and

Operations become **member functions** or methods. ADT in any language is class So, one ADT can result into multiple classes based on the implementation i.e. one might use **any data structure.**

In C++, an Abstract Data Type (ADT) is a high-level description of a data structure and the operations that can be performed on it, without specifying the underlying implementation details. It abstracts away the internal workings of a data structure, allowing you to use it without needing to know how it is implemented. ADTs are a fundamental concept in computer science and are often used to create reusable, modular code.

Here are a few key points about ADTs in C++:

1. Encapsulation: ADTs encapsulate the data and operations on that data into a single unit. The user of the ADT interacts with it through a well-defined interface (public functions), and the internal details are hidden.

2. Data Abstraction: ADTs provide a way to represent complex data structures and their associated operations in a simplified and abstract manner. For example, a stack ADT may have operations like "push," "pop," and "top," abstracting the idea of a stack of elements.

3. Implementation Independence: Users of an ADT do not need to know how it is implemented internally. This allows for flexibility in changing the implementation details without affecting code that uses the ADT.

4. Separation of Concerns: ADTs promote a separation of concerns between the data structure and the algorithms that operate on it. This separation makes the code more modular and maintainable.

Key Features of an ADT

1. Conceptual Data Structures:

ADTs define data structures at a conceptual level without specifying how they are implemented. They describe what operations can be performed on the data and what the expected behavior of those operations is.

2. Separation of Interface and Implementation:

ADTs separate the interface (the set of operations that can be performed on the data) from the implementation (how those operations are carried out). This separation allows for flexibility in changing the underlying implementation without affecting the code that uses the ADT. For example, you can switch between array-based and linked-list-based implementations of a stack ADT without changing the way you use the stack

3. Reusability and Modularity:

ADTs promote code reusability and modularity. Once you've defined an ADT, you can use it in different parts of your code or in different projects without needing to reimplement the same data structure and operations. This saves development time and reduces the chance of introducing bugs.

4. Abstract Behavior:

ADTs define the expected behavior of operations rather than the specific algorithm used to achieve that behavior. For instance, the push operation for a stack ADT is expected to add an element to the top of the stack, but it doesn't dictate whether this is done through an array, linked list, or any other means.

5. Real-world Analogy:

ADTs often have real-world analogies. For example, a stack ADT is analogous to a physical stack of items where you can push items onto the top and pop items from the top. This analogy helps users understand the behavior of the ADT even if they don't know the underlying implementation.

6. Standardization:

ADTs can become standardized, and there are often libraries in C++ that provide commonly used ADTs such as lists, queues, and maps. These standardized ADTs are part of the Standard Template Library (STL) in C++.

7. Custom ADTs:

You can create custom ADTs tailored to the specific needs of your application. For example, if your application requires a specialized data structure, you can define an ADT for it, encapsulating the necessary data and operations.

Activities:

Pre-Lab Activities:

List ADT: The List ADT defines a set of operations that can be performed on a list, but it does not specify how these operations are implemented. Instead, it serves as a blueprint or contract for how a list should behave.

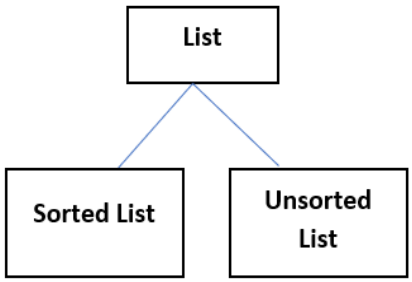

Fig.03 (List ADT)

Task 01: UnsortedList

Implement the **Unsorted List ADT** as given the prototype of the ADT. Kindly make sure the implement it as **Class Template.** The prototypes have been given in specific type **(int).**

Note: There shouldn`t any memory leak or dangling pointer in your implementation of ADT. All the corner cases should be handled properly. Try to throw Exceptions where necessary.

```
class UnsortedList
{
	int* arr;
	int maxSize, currSize;
public:
	UnsortedList();
	~UnsortedList();
	UnsortedList(const UnsortedList&);
	const UnsortedList& operator = (const UnsortedList&);
	bool insert(int val);
	bool isFull() const;
	bool isEmpty() const;
	void display() const;
	bool remove(int index);
};
```

The program should be multi files. You should make sure to write each and every method with efficient approach. There should not be extra space or time in your implementation of method being consumed. The credit of full marks only be given for efficient work and handling all types of problem instances for that algorithm.

Task 02: Polynomial ADT

You are required to implement the polynomial ADT. An array data structure is to be used for storing the terms in such a way that the coefficient of the term with exponent k will be stored in array at index k.

For example, to store the polynomial 15x^2 + 5x. We'll have to store value 15 (which is co-efficient of 15x^2) at the index 2 and 5 (coefficient of 5x) at the index 1.

To figure out whether the term with exponent n exists in the polynomial or not, one has to simply

check whether the value at index n is zero or not. If it's zero, it means that coefficient of term with degree n is zero which implies that the term does not exist.

Keeping in mind the above information, your job is to provide the implementation of the following methods.

Polynomial (int max_degree) ;

"max_degree" is not the actual degree of the polynomial. It is just the highest exponent a term can possibly have in the polynomial. It helps us to figure out the size of array to be created for storing our polynomial.

Polynomial (const Polynomial& obj) ; int get _degree() const;

This method will not return the max_degree argument that we received in the constructor, but the actual degree of the polynomial, which is the value of the highest exponent among all the terms.

Polynomial& operator = (const Polynomial& other);

void add_term(int coefficient , int degree) ;

Throw an exception if the degree of the term to be added to the polynomial is greater than size of the array we created in constructor. If the term already exists in polynomial, simply overwrite the coefficient.

void remove_term (int degree);

void set_coeffecient(int coefficient , int degree) ;

friend ostream& operator << (ostream& os, const Polynomial& poly);

Should print the polynomial on console as we usually write polynomials in mathematics such as 5x^4 + 3x^2 + 5.

double operator () (double x) const ;

Evaluates the value of polynomial for given value of x and returns the answer.

Polynomial operator + (const Polynomial& other) ;

Performs addition operation on polynomials and returns the new resultant polynomial. Write this function smartly to get the full credit :)

Polynomial operator * (const Polynomial& other) ;

~Polynomial ();

Submissions:

- For In-Lab Activity:
  - Save the files on your PC.
  - TA's will evaluate the tasks offline.
- For Pre-Lab & Post-Lab Activity:
  - Submit the .cpp file on Google Classroom and name it to your roll no.

Evaluations Metric:

- All the lab tasks will be evaluated offline by TA's
- **Division of Pre-Lab marks:** **[20 marks]**
  - Task 01: Unsorted List ADT [10 marks]
  - Task 02: Polynomial ADT [10 marks]

References and Additional Material:

- Abstract Data Types
  https://www.geeksforgeeks.org/abstract-data-types/
- Sparse Matrices
  https://www.geeksforgeeks.org/sparse-matrix-representation/