Facultad: Ingeniería
Escuela: Computación
Asignatura: Programación II

Tema: Tipos Abstractos de Datos (TAD's) en C++.

Objetivos Específicos

- Explicar el concepto "Tipo Abstracto de Datos" (TAD).
- Describir la sintaxis de los TAD en C++ para la aplicación de este concepto.
- Resolver problemas comunes de programación orientada a objetos por medio de los TAD.

Materiales y Equipo

- Computadora con el software DevC++.
- Guía Número 12.

Introducción Teórica

Tipos Abstractos de Datos (TAD's).

Un TAD se define como una estructura algebraica compuesta por un conjunto de objetos abstractos que modelan elementos del mundo real, y un conjunto de operaciones para su manipulación.

Un TAD es un ente cerrado y autosuficiente, que no requiere de un contexto específico para que pueda ser utilizado en un programa. Esto garantiza portabilidad y reutilización del software.

Las partes que forman un TAD son:

a) atributos (tipos de datos, identificadores, etc.)
b) funciones (rutinas) que definen las operaciones válidas para manipular los datos (atributos).

A la forma de operar y encerrar los atributos y funciones dentro un TAD se le denomina encapsulamiento.

Para implementar un TAD se siguen dos pasos:

1. Diseñar las estructuras de datos que van a representar cada objeto abstracto.
2. Desarrollar una función, por cada operación del TAD, que simule el comportamiento del objeto abstracto, sobre las estructuras de datos seleccionadas.

Las operaciones de un TAD se clasifican en 3 grupos, según su función sobre el objeto abstracto:

a) ***Constructora***: es la operación encargada de crear elementos del TAD.
b) ***Modificadora***: es la operación que puede alterar el estado de un elemento de un TAD.
c) ***Analizadora***: es una operación que no altera el estado del objeto, sino que tiene como misión consultar su estado y retornar algún tipo de información.

Entre las aplicaciones de los TAD's se encuentran el TAD lista, el TAD pila, el TAD cola, etc.

TAD's Estáticos.

La creación y mantenimiento de un TAD estático requiere de memoria no dinámica, es decir, el espacio en memoria para almacenar los datos es reservado en tiempo de compilación (Arreglos).

TAD's Dinámicos (Asignación dinámica de memoria).

La creación y mantenimiento de estructuras dinámicas de datos (TAD's dinámicos), requiere de obtener más espacio de memoria (reservar memoria) en tiempo de ejecución para almacenar datos o para almacenar el tipo de clase "***Nodo***".

Los operadores ***new*** y ***delete*** son esenciales para la asignación dinámica de memoria.
El operador "new" toma como argumento el tipo del objeto que se está asignando dinámicamente y, devuelve un apuntador hacia un objeto de ese tipo.
Por ejemplo la instrucción:

```
Nodo * siguiente_N = new Nodo(5);
```

Asigna el tamaño en bytes de Nodo y almacena en siguiente_N un apuntador hacia esta memoria. Si no hay memoria disponible, new devuelve un apuntador cero (NULL). El 5 indica que serán cinco los datos del objeto de nodo.

El operador “delete” libera la memoria que se asignó mediante new. Para liberar la memoria asignada dinámicamente mediante el new anterior, se utiliza la instrucción:

delete siguiente_N;

El TAD pila.

Una **pila** (***stack***) es una secuencia de cero o más elementos de un mismo tipo, que puede crecer y decrecer por uno de sus extremos (el tope de la pila).

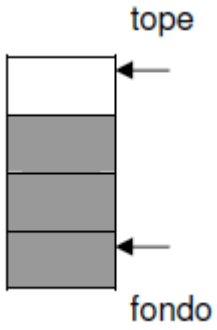

Figura 1. Una Pila ó stack.

Las pilas se denominan también estructuras LIFO (Last In First Out), porque su característica principal es que el último elemento en llegar es el primero en salir. Son muy utilizadas en programación para evaluar expresiones.

En todo momento, el único elemento visible de la estructura es el último que se colocó.

Se define el tope de la pila como el punto donde se encuentra dicho elemento, y el fondo, como el punto donde se encuentra el primer elemento incluido en la estructura.

En el TAD pila se definen operaciones constructoras para crear la pila, operaciones modificadoras para agregar y eliminar elementos y analizadoras para retornar el elemento del tope de la lista, las cuales serán analizadas más adelante.

Procedimiento

Se implementarán algunos ejemplos para tener una idea del funcionamiento de los Tipos Abstractos de Datos utilizando estructuras de datos como son las listas y las pilas.

EJEMPLO No. 1 - Implementación de pilas a través de una lista utilizando clases.

El programa siguiente utiliza una lista para manejar los elementos de la pila, a este manejo de los datos se le conoce como dinámico, las clases son muy útiles para manejarlo como un solo objetos con sus propias funciones y elementos.

```
#include<iostream>
#include<process.h>
#include<conio.h>
using namespace std;

class Clistpila
{       protected:
           struct lista      // Estructura del Nodo de una lista
           {       int dato;
                   struct lista *nextPtr;              //siguiente elemento de la lista
           };

           typedef struct lista *NODELISTA;     //tipo de dato *NODOLISTA

           struct NodoPila
           {       NODELISTA startPtr;             //tendrá la dirección del fondo de la pila
           } pila;

           typedef struct NodoPila *STACKNODE;      //Tipo Apuntador a la pila

      public:
           Clistpila( );                    // Constructor
           ~Clistpila( );                   // Destructor

           void push(int newvalue);         // Función que agrega un elemento a la pila
           int pop( );                      // Función que saca un elemento de la pila
           int PilaVacia( );                // Verifica si la pila está vacía
           void MostrarPila( );             // Muestra los elementos de la Pila

           friend void opciones(void);      // función amiga
};

                    //Funciones Miembro de la clase
Clistpila :: Clistpila( )
{          pila.startPtr = NULL;            //se inicializa el fondo de la pila.
}

int Clistpila :: PilaVacia( )
{         return((pila.startPtr == NULL)? 1:0);//note que si la pila esta vacía retorna 1, sino 0
}

void Clistpila :: push(int newvalue)            //se puede insertar en cualquier momento
{          NODELISTA nuevoNodo;                 //un nodo al tope de la pila
           nuevoNodo = new lista;               //crear el nuevo nodo
           if(nuevoNodo != NULL)                //si el espacio es disponible
```

```
        {   nuevoNodo -> dato = newvalue;          //se llena
            nuevoNodo -> nextPtr = pila.startPtr;    // tope de la pila
            pila.startPtr = nuevoNodo;         // la direccion actual que tenia el fondo de la
                                               // pila apunta al principio de cada nuevo nodo
        }
        else
        {   cout << newvalue << " No fue insertado...\n";
            cout << "No hay memoria disponble.\n";
        }
}

int Clistpila :: pop( )                    /* Elimina un nodo al tope de la pila */
{       NODELISTA temp;                    //puntero_nodo temporal
        int ValorPop;
        temp = pila.startPtr;
        ValorPop = temp -> dato;
        pila.startPtr = temp -> nextPtr;
        delete temp;                       // se saca de la pila el valor de tope
        return ValorPop;
}

void Clistpila :: MostrarPila( )           /* Imprimir Pila */
{       NODELISTA actualPtr;
        actualPtr = pila.startPtr;
        if ( actualPtr == NULL)
            cout << "** La pila esta vacia **\n\n";
        else
        {   cout << "** La pila es **\n\n";
            cout << "[tope] ";
            while( actualPtr != NULL)
            {   cout << actualPtr -> dato << " --> ";
                actualPtr = actualPtr -> nextPtr;
            }
            cout << "NULL [fondo]\n\n";
        }                                  //fin de else
}

Clistpila :: ~Clistpila( )         // Debe asegurarse liberar la memoria ocupada por la pila
{       NODELISTA freeP;   // para este caso deben liberarse: startPtr que contiene la
        if(PilaVacia( ) != 1)    // direccion de inicio de la pila (fondo) y todos los datos
        {   for ( freeP = pila.startPtr; freeP != NULL; freeP = freeP -> nextPtr)
                delete freeP;
        }
        delete pila.startPtr;
}

void opciones(void)
{       system("cls");
        cout << "Digite una opcion:\n"
             << "1. Insertar un elemento en la pila.\n"
             << "2. Borrar un elemento de la pila.\n"
             << "3. Para salir del programa.\n";
}
```

```
int main( )
{       Clistpila pil;                  // Objeto "pil" de la clase Clistpila
        int opcion = 0;
        int valor;                      //valor que será introducido en la pila

        while (opcion != 3)
        {       opciones( );
                cout << "? ";
                cin >> opcion;

                switch (opcion)
                {   case 1:
                            cout << "Digite un numero entero: ";
                            cin >> valor;
                            pil.push(valor);
                            pil.MostrarPila( );
                            system("pause");
                            break;

                    case 2:
                            if (!pil.PilaVacia( ))
                                cout << "El valor sacado de la pila fue: " << pil.pop() << "\n\n";
                                pil.MostrarPila( );
                                system("pause");
                                break;
                    case 3:
                                break;
                }
        }
        system("pause > nul");
        return 0;
}
```

Ejemplo 2. - Implementación de pilas con arreglos utilizando plantillas de clase y archivos de cabecera.

Se creará primero el archivo Tpila.h

```
// TPila.H
// Plantilla simple de clase pila

#ifndef TPila_H
#define TPila_H

template <class T>
class Pila
{       private:
          int size;                     // # de elementos en la pila
          int tope;
          T *stackPtr;                  // puntero a la pila
```

```
        public:
          Pila(int = 10);                    //longitud por defecto 10

          ~Pila( )                           //Destructor
          {    delete [ ] stackPtr;
          }

          int push(const T&);                // pone un elemento en la pila
          int pop(T&);                       // saca un elemento de la pila
          int PilaVacia( ) const
          {         return tope == -1;
          }

          int PilaLLena( ) const
          {         return tope == size - 1;         //Llena = 1
          }
};

                              // Plantillas de funciones
template <class T> Pila<T> :: Pila(int s)
{         size = s;
          tope = -1;                         // Pila vacia
          stackPtr = new T[size];
}

template <class T> int Pila<T> :: push(const T &valor)
{         if (!PilaLLena( ))
          {   stackPtr[++tope] = valor;
              return 1;                                 // push correcto
          }
          return 0;                                     // push no correcto
}

template <class T> int Pila<T> :: pop(T &ValorPop)
{         if (!PilaLLena( ))
          {     ValorPop = stackPtr[tope--];
                return 1;
          }
          return 0;
}

#endif
```

Se definen las funciones básicas de una pila utilizando arreglos, el cual está limitado para 10 elementos.

Ahora se realizará el archivo donde podremos trabajar con la pila Tad2.cpp

```
#include <iostream>
#include "Tpila.h"
using namespace std;

int main( )
{       Pila<float> PilaFloat(5);       // Se crea un objeto tipo pila float con 5 elementos
        float f = 0;
        cout << "--- Pila de tipo Float --- " << endl;
        cout << "Introduciendo elementos a la pila " << endl;

        while(PilaFloat.push(f))                // Si es 1 fue exitoso
        {     cout << f << " ";
              f +=1.1;
        }

        cout << endl << "La pila esta llena. No se puede meter otro elemento: " << f
             << endl;
        cout << "Sacando elementos de la pila..." << endl;

        while ( PilaFloat.pop(f))
           cout << f << " " << endl;

        cout << "\nLa pila esta vacia.\n";
        Pila<int> PilaInt;              // Se crea un objeto tipo pila Entero con n
        int i=1;

        cout << "--- Pila de tipo Int --- " << endl;
        cout << "Introduciendo elementos a la pila " << endl;

        while (PilaInt.push(i))         // retorna 1 si fue exitoso
        {      cout << i << " ";
               i++;
        }
        cout << "La pila esta llena. No se puede meter: " << i << endl;
        cout << "Sacando elementos de la pila..." << endl;

        while ( PilaInt.pop(i))
              cout << i << " ";
        cout << "\nLa pila esta vacia \n";
        system("pause > nul");
        return 0;
}
```

Análisis de Resultados

Ejercicio 1:

Modifique el ejemplo 1, para agregar las siguientes opciones al menú:

a) Hacer una función que muestre el número de elementos que tiene la pila en cualquier momento.
b) Implementar una función muestre los elementos de la pila en cualquier momento.

Ejercicio 2:

Modifique el ejemplo 2 para crear un TAD pila que acepte datos:

- De tipo carácter.
- De tipo double.
- De tipo int

Implementar la solución a través de un menú que me permita realizar las operaciones básicas con la pila (sin importar el tipo de dato de la pila).

Investigación Complementaria

Ejercicio:

Utilizar un TAD Pila que permita determinar si una palabra o frase es un palíndromo.

Nota: Una palabra es un palíndromo si la lectura en ambos sentidos produce el mismo resultado

Guía 12: Tipos Abstractos de Datos (TAD's) en C++.

Hoja de cotejo: 12

Alumno: Máquina No:

Docente: GL: Fecha:

EVALUACIÓN					
	%	1-4	5-7	8-10	Nota
CONOCIMIENTO	Del 20 al 30%	Conocimiento deficiente de los fundamentos teóricos	Conocimiento y explicación incompleta de los fundamentos teóricos	Conocimiento completo y explicación clara de los fundamentos teóricos	
APLICACIÓN DEL CONOCIMIENTO	Del 40% al 60%				
ACTITUD					
	Del 15% al 30%	No tiene actitud proactiva.	Actitud propositiva y con propuestas no aplicables al contenido de la guía.	Tiene actitud proactiva y sus propuestas son concretas.	
TOTAL	100%				