CONCEPTOS FUNDAMENTALES DE PROGRAMACIÓN

L. A Morales Álvarez
7690-25-5110 Universidad Mariano Gálvez
Matemática Discreta
Lmoralesa22@miumg.edu.gt

Resumen

Este artículo explora las estructuras de los datos en la programación orientada a objetos así como las funcionalidades para tener un código más ordenado y así gestionando su eficiencia en los lenguajes de programación. Esta investigación permite ver que podemos hacer del trabajo de programación una tarea más ordenada y sencilla. Uno de los objetivos de esta investigación es poder aprender a utilizar de manera correcta la utilización de clases, objetos, metodos y herencias en el ámbito de la programación orientada a objetos, al poder dominar estos conceptos podremos entregar un software de mayor calidad y así poder usar un razonamiento más lógico y así aplicarlo en el ámbito de la resolución de problemas, cada tema investigado nos proporcionara una idea más clara sobre lo que podemos o no realizar con las definiciones y elementos básicos de la POO, estos elementos son muy útiles para programas muy grandes ya que podemos utilizar varios elementos sin necesidad de volver a una ambigüedad de clases o atributos.

Palabras claves: programación orientada a objetos, encapsulamiento, herencia, modularidad, software.

Desarrollo del tema

En el mundo de la programación orientada a objetos, las estructuras de datos tienen un papel muy importante para la organización y manipulación de información. Un ejemplo de ello son las listas, permitiendo así acceder y modificar elementos con eficiencia.
Las clases es uno de los elementos de la POO, ya que con ellos podemos definir atributos y me ‘todos. Las clases representan cosas abstractas de cosas que existen en el mundo real, como un auto, los cuales tiene como atributos la marca, el año, la capacidad y el tipo de motor, y también tiene me‘todos como movilizarse () o frenar (), por otro lado los objetos son instancias de las clases definidas, pero que poseen una existencia real dentro del programa, es importante saber la diferencia y los usos de estos terminados ya que a menudo suelen confundir o suelen ser complicados de entender para el programador.
El encapsulamiento es un principio que pertenece a la POO, permitiendo ocultar los estados internos de una clase mediante modificadores como privado, público o protegido, este método permite al usuario reducir l´ıneas innecesarias entre el sistema. La herencia por otro lado facilita la reducción de código y evitar repetir clases ya definidas entre ellas se encuentra la superclase, que es una clase que engloba un término universal y de ella derivan clases pequeñas que pueden heredar me‘todos y atributos de esta misma pudiendo as´ı evitar una ambigüedad de código.

En la estructuración de datos, las listas proporcionan ideas para secuencias de elementos relacionados, permitiendo organizar, almacenar y manipular datos de un computador y as´ı utilizarlos de una manera más eficiente entre ellos están los árboles y grafos. Los diccionarios nos ofrecen una alternativa de almacenamiento para as´ı poder acceder de una manera más rápida, estos nos permiten encapsular datos entre llaves. Las tuplas por su parte, es una colección ordenada e inmutable de elementos que pueden ser de diferentes tipos, estas son útiles para almacenar datos que no deben de cambiarse ya que son inmutables.
La modularidad mediante funciones permite la organización del código de manera lógica, permitiendo que se pueden reutilizar y as´ı facilitar el mantenimiento, este principio nos permite dividir un programa en sus partes más pequeñas e independientes donde cada función maneja una tarea en espec´ıfico, una de sus ventajas es que permite el trabajo en equipo permitiendo as´ı que cada integrante pueda desarrollar un módulo en espec´ıfico, as´ı como la reutilización en diferentes partes de un proyecto o en otros proyectos distintos, permitiendo que se reduzca la ambigüedad y duplicidad de código.

Observaciones y comentarios

Durante la investigación y desarrollo de esta investigación pudimos observar como los elementos de la POO son básicos y necesarios en el desarrollo de aplicaciones más eficientes, con base a esto es importante el recalcar la importancia de saber diferenciar y saber para que funciona cada uno de estos elementos, además de esto se observa que a la mayor´ıa de estudiantes les cuesta al principio poder entender estos términos para lo cual es necesario poder explicarlos con ejemplos de la vida real y no su pociones para una mejor comprensión.

Conclusiones

1. La OOP proporciona un estilo de programación más estético y más ordenado.
2. Elementos como la herencia y modularidad son importantes poder aplicarlos ya que facilitan y hacen más le´ıble el programa sin necesidad de tener tanta l´ınea de código.
3. La correcta aplicación de estos elementos permite que el programa sea más eficiente.
4. Los estudiantes deben de aplicar estos términos en cada proyecto de programación para as´ı poder dominar estos términos fundamentales.
5. Estos conceptos fundamentales constituyen la base para el dominio de lenguajes de programación modernos.

Bibliografía

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2021). *Patrones de diseño: Elementos de soft- ware orientado a objetos reutilizable*. Addison-Wesley.
Lutz, M. (2023). *Aprendiendo Python*. O'Reilly Media.
Mart´ın, R. C. (2022). *Código limpio: Manual de estilo para el desarrollo ágil*. Anaya Multimedia. Python Software Foundation. (2024). *The Python Tutorial*. htt ps://docs.python.org/3/tutorial/

Aplicación

Código Fuente

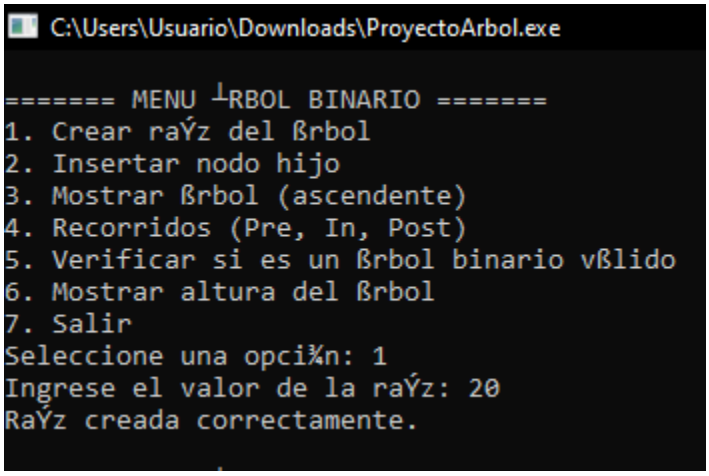
```
#include <iostream>
using namespace std;

// ---------- Estructura del Nodo ----------
struct Nodo {
   int dato;
   Nodo *izq;
   Nodo *der;
};

// ---------- Crear un nuevo nodo ----------
Nodo *crearNodoNuevo(int valor) {
   Nodo *n = new Nodo();
   n->dato = valor;
   n->izq = NULL;
   n->der = NULL;
   return n;
}

// ---------- Insertar nodo automáticamente ----------
void insertarAutomatico(Nodo *&raiz, int valor) {
   if (raiz == NULL) {
      raiz = crearNodoNuevo(valor);
      return;
   }

   if (valor == raiz->dato) {
      cout << "El valor ya existe en el árbol. No se agregará.\n";
      return;
   }

   if (valor < raiz->dato)
      insertarAutomatico(raiz->izq, valor);
   else
      insertarAutomatico(raiz->der, valor);
}

// ---------- Recorridos ----------
void preorden(Nodo *nodo) {
   if (nodo == NULL) return;
   cout << nodo->dato << " ";
   preorden(nodo->izq);
   preorden(nodo->der);
}
```

```
void inorden(Nodo *nodo) {
   if (nodo == NULL) return;
   inorden(nodo->izq);
   cout << nodo->dato << " ";
   inorden(nodo->der);
}

void postorden(Nodo *nodo) {
   if (nodo == NULL) return;
   postorden(nodo->izq);
   postorden(nodo->der);
   cout << nodo->dato << " ";
}

// ---------- Calcular la altura ----------
int altura(Nodo *raiz) {
   if (raiz == NULL)
      return 0;
   int izq = altura(raiz->izq);
   int der = altura(raiz->der);
   return (izq > der ? izq : der) + 1;
}

// ---------- Imprimir árbol de forma visual ascendente ----------
void mostrarArbol(Nodo *raiz, int nivel = 0) {
   if (raiz == NULL)
      return;

   // Primero el subárbol derecho (valores mayores)
   mostrarArbol(raiz->der, nivel + 1);

   // Imprime el nodo con sangría para representar el nivel
   for (int i = 0; i < nivel; i++)
      cout << "   ";
   cout << raiz->dato << endl;

   // Luego el subárbol izquierdo (valores menores)
   mostrarArbol(raiz->izq, nivel + 1);
}

// ---------- Verificar si el árbol es binario válido ----------
bool esBinarioValido(Nodo* raiz, int min, int max) {
   if (raiz == NULL)
      return true;
   if (raiz->dato <= min || raiz->dato >= max)
      return false;
   return esBinarioValido(raiz->izq, min, raiz->dato) &&
          esBinarioValido(raiz->der, raiz->dato, max);
```

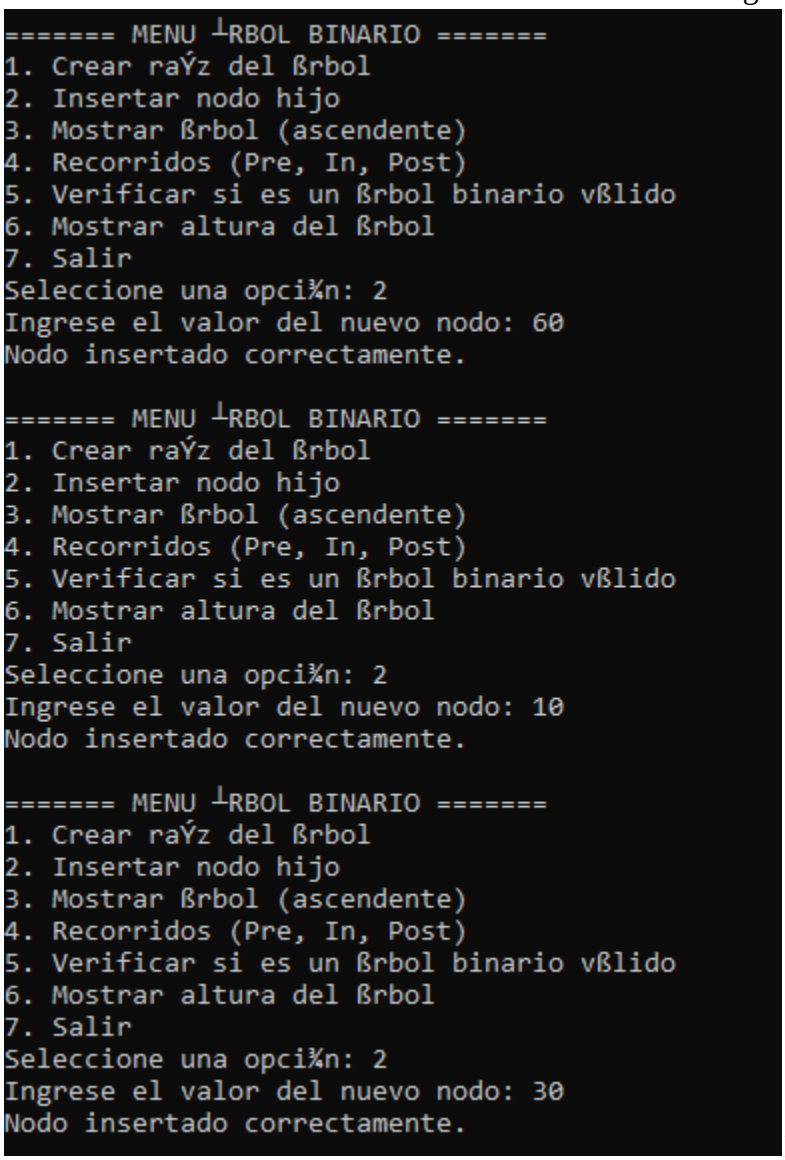
```
}

// ---------- Menú principal ----------
int main() {
    Nodo *raiz = NULL;
    int opcion, valor;

    do {
        cout << "\n======= MENU ÁRBOL BINARIO =======\n";
        cout << "1. Crear raíz del árbol\n";
        cout << "2. Insertar nodo hijo\n";
        cout << "3. Mostrar árbol (ascendente)\n";
        cout << "4. Recorridos (Pre, In, Post)\n";
        cout << "5. Verificar si es un árbol binario válido\n";
        cout << "6. Mostrar altura del árbol\n";
        cout << "7. Salir\n";
        cout << "Seleccione una opción: ";
        cin >> opcion;

        switch (opcion) {
            case 1:
                if (raiz != NULL)
                    cout << "La raíz ya fue creada.\n";
                else {
                    cout << "Ingrese el valor de la raíz: ";
                    cin >> valor;
                    raiz = crearNodoNuevo(valor);
                    cout << "Raíz creada correctamente.\n";
                }
                break;

            case 2:
                if (raiz == NULL)
                    cout << "Debe crear la raíz primero.\n";
                else {
                    cout << "Ingrese el valor del nuevo nodo: ";
                    cin >> valor;
                    insertarAutomatico(raiz, valor);
                    cout << "Nodo insertado correctamente.\n";
                }
                break;

            case 3:
                if (raiz == NULL)
                    cout << "El árbol está vacío.\n";
                else {
                    cout << "\nEstructura del árbol (ascendente):\n";
                    mostrarArbol(raiz);
                }
```

```
            break;

        case 4:
            if (raiz == NULL)
                cout << "El árbol está vacío.\n";
            else {
                cout << "\nPreorden: ";
                preorden(raiz);
                cout << "\nInorden: ";
                inorden(raiz);
                cout << "\nPostorden: ";
                postorden(raiz);
                cout << endl;
            }
            break;

        case 5:
            if (raiz == NULL)
                cout << "El árbol está vacío.\n";
            else {
                bool valido = esBinarioValido(raiz, -999999, 999999);
                if (valido)
                    cout << "? El árbol es binario de búsqueda válido.\n";
                else
                    cout << "? El árbol NO cumple las propiedades de un ABB.\n";
            }
            break;

        case 6:
            cout << "La altura del árbol es: " << altura(raiz) << endl;
            break;

        case 7:
            cout << "Saliendo del programa...\n";
            break;

        default:
            cout << "Opción inválida, intente de nuevo.\n";
      }

   } while (opcion != 7);

   return 0;
}
```

Capturas del código ejecutado



Ingreso de raíz



Ingreso de nodos para formar el árbol

```
======= MENU ┴RBOL BINARIO =======
1. Crear raÝz del ßrbol
2. Insertar nodo hijo
3. Mostrar ßrbol (ascendente)
4. Recorridos (Pre, In, Post)
5. Verificar si es un ßrbol binario vßlido
6. Mostrar altura del ßrbol
7. Salir
Seleccione una opci¾n: 3

Estructura del ßrbol (ascendente):
   60
      30
20
   10
```

Muestra del árbol creado

```
======= MENU ┴RBOL BINARIO =======
1. Crear raÝz del ßrbol
2. Insertar nodo hijo
3. Mostrar ßrbol (ascendente)
4. Recorridos (Pre, In, Post)
5. Verificar si es un ßrbol binario vßlido
6. Mostrar altura del ßrbol
7. Salir
Seleccione una opci¾n: 4

Preorden: 20 10 60 30
Inorden: 10 20 30 60
Postorden: 10 30 60 20
```

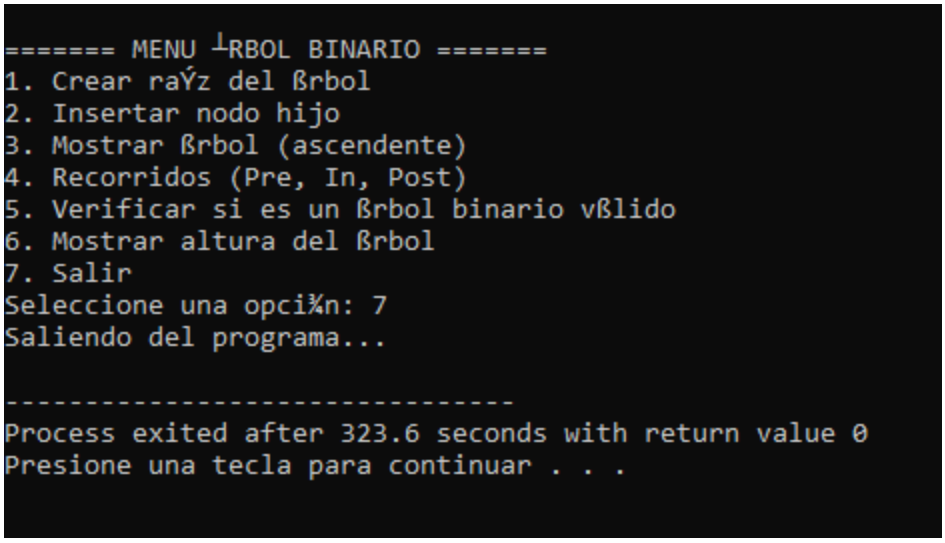
Árbol ordenado

```
======= MENU ┴RBOL BINARIO =======
1. Crear raÝz del ßrbol
2. Insertar nodo hijo
3. Mostrar ßrbol (ascendente)
4. Recorridos (Pre, In, Post)
5. Verificar si es un ßrbol binario vßlido
6. Mostrar altura del ßrbol
7. Salir
Seleccione una opci¾n: 5
? El ßrbol es binario de b·squeda vßlido.
```

Verificación de árbol binario

```
======= MENU ┴RBOL BINARIO =======
1. Crear raÝz del ßrbol
2. Insertar nodo hijo
3. Mostrar ßrbol (ascendente)
4. Recorridos (Pre, In, Post)
5. Verificar si es un ßrbol binario vßlido
6. Mostrar altura del ßrbol
7. Salir
Seleccione una opci¾n: 6
La altura del ßrbol es: 3
```

Altura del árbol



Salida del programa.