**Material de apoyo para segundo parcial de Programación de Estructuras Dinámicas**

**¿Qué son las estructuras dinámicas?**

Las estructuras de datos dinámicas son tipos de datos que permiten almacenar y organizar información de manera flexible, especialmente en entornos donde el tamaño y la cantidad de datos no se conocen de antemano o pueden cambiar durante la ejecución del programa. Estas estructuras se caracterizan por su capacidad para expandirse o contraerse dinámicamente en tiempo de ejecución, a diferencia de las estructuras de datos estáticas que tienen un tamaño fijo predefinido.

**Pilas**

**¿Qué son?**

Las pilas son una estructura de datos lineal que sigue el principio LIFO (Last In, First Out), lo que significa que el último elemento insertado es el primero en ser eliminado.

Ventajas de utilizar pilas

**Facilidad de implementación:** Las operaciones en una pila son relativamente simples: push (añadir), pop (eliminar), peek (observar el elemento superior) y isEmpty (verificar si está vacía). Esto hace que la implementación y el uso de la pila sean simples y directos.

**Eficiencia en la gestión de memoria:** Las pilas suelen ser más eficientes en la gestión de la memoria. Al seguir el principio LIFO (Last In, First Out), las operaciones se realizan en un extremo de la estructura de datos, lo que hace que la inserción y eliminación de elementos sean más rápidas.

**Aplicaciones en la vida real:** Las pilas se utilizan ampliamente en la informática, desde la administración de memoria hasta el manejo de llamadas a funciones en la pila de ejecución del programa. También son útiles para revertir operaciones, como la navegación en un navegador web o el uso de botones "deshacer" y "rehacer" en aplicaciones.

**Estructura de datos versátil:** Las pilas son una estructura de datos versátil que se utiliza en algoritmos y problemas de programación. Son la base de muchas estructuras más complejas y su comprensión es esencial para la resolución de problemas.

**Fácil anidamiento:** Las pilas permiten anidar elementos en su estructura, lo que resulta útil en situaciones donde es necesario mantener un seguimiento del estado en niveles múltiples (como en la evaluación de expresiones matemáticas).

|  |  |
| --- | --- |
| Ejemplo de definición de un nodo para una pila | Ejemplo de una estructura para la pila |
| struct Node {  int data;  Node\* next;  }; | struct Stack {  Node\* top;  }; |

**Funciones elementales de las pilas:**

|  |  |
| --- | --- |
| Tipo de función | Definición de la función |
| Inicializar la pila | Stack\* createStack() {  Stack\* stack = new Stack;  stack->top = nullptr;  return stack;  } |
| Verificar si la pila está vacía | bool isEmpty(Stack\* stack) {  return (stack->top == nullptr);  } |
| Insertar un elemento en la pila | void push(Stack\* stack, int data) {  Node\* newNode = new Node;  newNode->data = data;  newNode->next = stack->top;  stack->top = newNode;  cout << data << " ha sido agregado a la pila." << endl;  } |
| Eliminar un elemento de la pila | void pop (Stack\* stack) {  if (isEmpty(stack)) {  cout << "La pila está vacía. No se puede eliminar nada." << endl;  return;  }  Node\* temp = stack->top;  stack->top = stack->top->next;  cout << temp->data << " ha sido eliminado de la pila." << endl;  delete temp;  } |
| Elemento en la cima de la pila | int peek(Stack\* stack) {  if (isEmpty(stack)) {  cout << "La pila está vacía." << endl;  return -1;  }  return stack->top->data;  } |
| Imprimir cola | void printStack(Node\* top) {  cout << "Pila: ";  while (top != nullptr) {  cout << top->data << " ";  top = top->next;  }  cout << endl;  } |

**Colas**

**¿Qué son?**

Las colas son una estructura de datos lineal y dinámica que sigue el principio de "primero en entrar, primero en salir" (FIFO, por sus siglas en inglés: First-In, First-Out). En una cola, los elementos se agregan al final y se eliminan del principio, manteniendo el orden en el que fueron insertados.

Ventajas de utilizar colas

**Mantenimiento del orden de llegada:** Garantizan que los elementos se procesen en el orden en el que se agregaron, lo que es útil en aplicaciones como la impresión, gestión de tareas, etc.

**Aplicaciones en procesamiento asíncrono:** Se utilizan en sistemas que manejan tareas a medida que llegan, procesando una tras otra en un flujo secuencial.

**Implementación sencilla:** Su estructura es relativamente simple y eficiente para implementar operaciones básicas como inserción (enqueue) y eliminación (dequeue) de elementos.

|  |  |
| --- | --- |
| Ejemplo de definición de un nodo para una cola | Ejemplo de una estructura para la cola |
| struct Node {  int data;  Node\* next;  }; | struct Queue {  Node\* front;  Node\* rear;  }; |

**Funciones elementales de las colas:**

|  |  |
| --- | --- |
| Tipo de función | Definición de la función |
| Create(Crear): Crear un nuevo nodo. | Node\* createNode(int data) {  Node\* newNode = new Node;  newNode->data = data;  newNode->next = nullptr;  return newNode;  } |
| Función para inicializar(crear) la cola. | Queue\* createQueue() {  Queue\* queue = new Queue;  queue->front = queue->rear = nullptr;  return queue;  } |
| Enqueue (Agregar/Insertar): Agrega un elemento al final de la cola.  notar que en la reserva de memoria se utiliza la función de crear nodo, para evitar establecer una nueva reserva. | void enqueue(Queue\* queue, int data) {  Node\* newNode = createNode(data);  if (queue->rear == nullptr) {  queue->front = queue->rear = newNode;  return;  }  queue->rear->next = newNode;  queue->rear = newNode;  } |
| Dequeue (Eliminar/Extraer): Elimina el elemento del frente de la cola. | void dequeue(Queue\* queue) {  if (queue->front == nullptr) {  cout << "La cola está vacía." << endl;  return;  }  Node\* temp = queue->front;  queue->front = queue->front->next;  if (queue->front == nullptr) {  queue->rear = nullptr;  }  delete temp;  } |
| Front (Frente): Devuelve el elemento que está al frente de la cola, sin eliminarlo. | int front(Queue\* queue) {  if (queue->front == nullptr) {  cout << "La cola está vacía." << endl;  return -1;  }  return queue->front->data;  } |
| Empty (Vacia): Verifica si la cola está vacía. | bool isEmpty(Queue\* queue) {  return (queue->front == nullptr  && queue->rear == nullptr);  } |
| Size (Tamaño): Determina el número de elementos presentes en la cola. | int size(Queue\* queue) {  int count = 0;  Node\* current = queue->front;  while (current != nullptr) {  count++;  current = current->next;  }  return count;  } |
| Print(Imprimir): Imprimir todos los elementos de la cola. | void Print(){  Node \*temp = frenteCola;  while (temporal != nullptr)  {  cout<<"Elementos de la cola " << temporal->elemento << endl;  temporal = temporal->siguiente;  }  } |

**Nota:** Otra forma de inicializar la cola sería declarando como variables globales los punteros al frente y al final de la cola:

struct Node \*front = nullptr;

struct Node \*rear = nullptr;

De esta manera no se necesita un argumento en las funciones y puede obviarse solo la estructura creada para la cola.

**Nota:** La inicialización del nodo puede puede obviarse ya que se inicializa también al momento de añadir un elemento en la función Enqueue, sin embargo, si la creación del nodo se usa en múltiples funciones o partes del programa, sería más eficiente mantenerla como una función separada para la reutilización del código. Además, al separar la creación del nodo en su propia función, se facilita la legibilidad y el mantenimiento del código.

**Listas enlazadas**

**¿Qué son?**

Las listas enlazadas son una estructura de datos fundamental en informática y se utilizan para almacenar y organizar datos. Consisten en nodos que están enlazados entre sí mediante punteros, formando una secuencia lineal. Cada nodo contiene un dato y una referencia al siguiente nodo.

Existen varios tipos de listas enlazadas:

* **Lista enlazada simple:** Cada nodo tiene un enlace solo al siguiente nodo.
* **Lista doblemente enlazada:** Cada nodo tiene enlaces tanto al siguiente como al nodo anterior.
* **Lista circular:** La lista está organizada en un bucle, donde el último nodo apunta al primero.

Cada nodo en una lista enlazada contiene dos partes principales: el valor o dato que se está almacenando y un puntero (o referencia) al siguiente nodo. El primer nodo se conoce como el "nodo de cabeza" o "nodo principal", y el último nodo suele tener un puntero nulo o vacío para indicar el final de la lista.

Las listas enlazadas ofrecen diversas ventajas:

**Inserción y eliminación eficiente:** Permiten una inserción y eliminación de elementos más eficientes en comparación con estructuras de datos estáticas como los arrays.

**Flexibilidad:** Las listas enlazadas pueden cambiar de tamaño fácilmente, ya que los nodos pueden ser agregados o eliminados dinámicamente.

**Uso eficiente de la memoria:** La memoria se asigna dinámicamente para cada nodo, lo que permite una asignación de memoria más eficiente según sea necesario.

Sin embargo, también tienen algunas desventajas, como el acceso secuencial a los elementos y la necesidad de recorrer la lista para acceder a un nodo específico, lo que puede resultar menos eficiente en comparación con las estructuras de acceso aleatorio, como los arrays.

**Listas enlazadas simples**

|  |  |
| --- | --- |
| Ejemplo de definición de un nodo para una lista | Ejemplo de una estructura para la lista |
| struct Node {  int data;  Node\* next;  }; | struct LinkedList {  Node\* head;  }; |

**Funciones elementales de las listas enlazadas:**

|  |  |
| --- | --- |
| Tipo de función | Definición de la función |
| Inicializar la lista  Nota: En este caso se utiliza una función void para inicializar la lista, pero se debe recordar pasar por referencia la estructura “&”. | void initializeList(LinkedList& list) {  list.head = nullptr;  } |
| Insertar elemento al inicio | void insertAtStart(LinkedList& list, int value) {  Node\* newNode = new Node;  newNode->data = value;  newNode->next = list.head;  list.head = newNode;  } |
| Insertar elemento en medio | void insertAtPosition(LinkedList& list, int value, int position) {  Node\* newNode = new Node;  newNode->data = value;  newNode->next = nullptr;  if (position <= 0) {  cout << "La posición no es válida." << endl;  return;  }  if (position == 1) {  newNode->next = list.head;  list.head = newNode;  return;  }  Node\* current = list.head;  for (int i = 1; i < position - 1 && current != nullptr; i++) {  current = current->next;  }  if (current == nullptr) {  cout << "La posición no es válida." << endl;  return;  }  newNode->next = current->next;  current->next = newNode;  } |
| Insertar elemento al final | void insertAtEnd(LinkedList& list, int value) {  Node\* newNode = new Node;  newNode->data = value;  newNode->next = nullptr;  if (list.head == nullptr) {  list.head = newNode;  return;  }  Node\* current = list.head;  while (current->next != nullptr) {  current = current->next;  }  current->next = newNode;  } |
| Eliminar un elemento al inicio | void pop(LinkedList& list) {  if (list.head == nullptr) {  cout << "La lista está vacía." << endl;  return;  }  Node\* temp = list.head;  list.head = list.head->next;  delete temp;  } |
| Eliminar un elemento al final | void deleteAtEnd(LinkedList& list) {  if (list.head == nullptr) {  cout << "La lista está vacía." << endl;  return;  }  if (list.head->next == nullptr) {  delete list.head;  list.head = nullptr;  return;  }  Node\* temp = list.head;  while (temp->next->next != nullptr) {  temp = temp->next;  }  delete temp->next;  temp->next = nullptr;  } |
| Imprimir todos los elementos en orden | void printLinkedList(Node\* head) {  cout << "Lista Simple: ";  while (head != nullptr) {  cout << head->data << " ";  head = head->next;  }  cout << endl;  } |

**Listas doblemente enlazadas**

|  |  |
| --- | --- |
| Ejemplo de definición de un nodo para una lista doblemente enlazada | Ejemplo de una estructura para la lista doblemente enlazada |
| struct Node {  int data;  Node\* next;  Node\* prev;  }; | struct DoublyLinkedList {  Node\* head;  Node\* tail;  };  Nota: No es necesario usar dos estructuras de nodo separadas, ya que cada nodo puede contener tanto un puntero al nodo siguiente como un puntero al nodo anterior. |

**Funciones elementales de las listas doblemente enlazadas:**

|  |  |
| --- | --- |
| Tipo de función | Definición de la función |
| Insertar elemento al inicio | void pushFront(Node\*& head, int value) {  Node\* newNode = new Node;  newNode->data = value;  newNode->prev = nullptr;  newNode->next = head;  if (head != nullptr) {  head->prev = newNode;  }  head = newNode;  } |
| Insertar elemento al final | void pushBack(Node\*& head, int value) {  Node\* newNode = new Node;  newNode->data = value;  newNode->next = nullptr;  if (head == nullptr) {  newNode->prev = nullptr;  head = newNode;  return;  }  Node\* temp = head;  while (temp->next != nullptr) {  temp = temp->next;  }  temp->next = newNode;  newNode->prev = temp;  } |
| Borrar el primer elemento | void popBack(Node\*& head) {  if (head == nullptr) {  cout << "La lista está vacía." << endl;  return;  }  if (head->next == nullptr) {  delete head;  head = nullptr;  return;  }  Node\* temp = head;  while (temp->next != nullptr) {  temp = temp->next;  }  temp->prev->next = nullptr;  delete temp;  } |
| Buscar un elemento para borrarlo | void deleteNode(Node\*& head, int key) {  Node\* temp = head;  if (head == nullptr) {  cout << "La lista está vacía." << endl;  return;  }  if (temp != nullptr && temp->data == key) {  head = temp->next;  head->prev = nullptr;  delete temp;  return;  }  while (temp != nullptr && temp->data != key) {  temp = temp->next;  }  if (temp == nullptr) {  cout << "No se encontró el nodo con el valor dado." << endl;  return;  }  if (temp->next != nullptr) {  temp->next->prev = temp->prev;  }  if (temp->prev != nullptr) {  temp->prev->next = temp->next;  }  delete temp;  } |
| Imprimir lista | void printList(Node\* head) {  cout << "Lista: ";  while (head != nullptr) {  cout << head->data << " ";  head = head->next;  }  cout << endl;  } |
| Insertar al final (utilizando la estructura con punteros hacia adelante y atrás) | void insertAtEnd(DoublyLinkedList\* list, int value) {  Node\* newNode = new Node();  newNode->data = value;  newNode->next = nullptr;  if (list->head == nullptr) {  list->head = newNode;  list->tail = newNode;  newNode->prev = nullptr;  } else {  list->tail->next = newNode;  newNode->prev = list->tail;  list->tail = newNode;  }  } |

Estas funciones están pensadas para que se inicialice el Nodo head en el main.

Recordar: Las listas enlazadas son colecciones de elementos de datos “alineados en una fila” lógicamente; pueden insertarse y eliminarse elementos en cualquier parte de una lista enlazada. Las pilas son importantes en los compiladores y sistemas operativos; pueden insertarse y eliminarse elementos sólo en un extremo de una pila. Las colas representan líneas de espera; se insertan elementos en la parte final de una cola y se eliminan elementos de su parte inicial.

**Listas circulares**

**Lista Circular Simplemente Enlazada:**

En esta estructura, cada nodo apunta al siguiente nodo y el último nodo apunta de nuevo al primero. La característica principal de este tipo de lista es que se puede recorrer indefinidamente a partir de cualquier nodo de la lista.

**Lista Circular Doblemente Enlazada:**

En una lista doblemente enlazada, cada nodo tiene un puntero al nodo anterior y al nodo siguiente, y el último nodo apunta al primero y el primer nodo apunta al último.

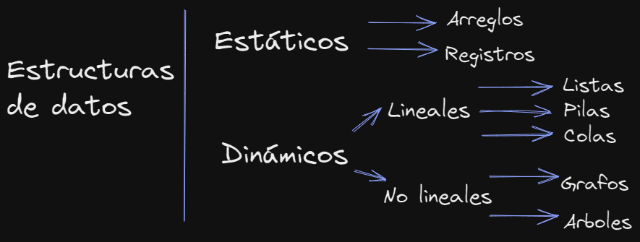
|  |  |
| --- | --- |
| Ejemplo de definición de un nodo para una lista circular | Ejemplo de una estructura para la lista circular |
| struct Node {  int data;  Node\* next;  }; | struct DoublyLinkedList {  Node\* head;  }; |

Funciones elementales de las listas circulares:

|  |  |
| --- | --- |
| Tipo de función | Definición de la función |
| Insertar al inicio | void insertAtBeginning(CircularLinkedList\* list, int value) {  Node\* newNode = new Node();  newNode->data = value;  if (list->head == nullptr) {  list->head = newNode;  newNode->next = newNode; // Único nodo apunta a sí mismo  } else {  Node\* last = list->head;  while (last->next != list->head) {  last = last->next;  }  newNode->next = list->head;  last->next = newNode;  list->head = newNode;  }  } |
| Insertar al final de la lista | void insertAtEnd(CircularLinkedList\* list, int value) {  Node\* newNode = new Node();  newNode->data = value;  if (list->head == nullptr) {  list->head = newNode;  newNode->next = newNode; // Único nodo apunta a sí mismo  } else {  Node\* last = list->head;  while (last->next != list->head) {  last = last->next;  }  last->next = newNode;  newNode->next = list->head;  }  } |
| Eliminar un valor dado | void deleteNode(CircularLinkedList\* list, int value) {  if (list->head == nullptr) {  return;  }  Node\* currentNode = list->head;  Node\* prevNode = nullptr;  do {  if (currentNode->data == value) {  if (prevNode != nullptr) {  prevNode->next = currentNode->next;  } else {  Node\* last = list->head;  while (last->next != list->head) {  last = last->next;  }  list->head = currentNode->next;  last->next = list->head;  }  delete currentNode;  return;  }  prevNode = currentNode;  currentNode = currentNode->next;  } while (currentNode != list->head);  } |
| Imprimir los datos  Nota: Para imprimir una lista circular simplemente enlazada, es esencial asegurarse de no entrar en un bucle infinito. Dado que no hay un "final" como en una lista lineal, es crucial definir el criterio de parada. Una forma es recorrer la lista mientras se verifica si se ha llegado nuevamente al nodo inicial. | void printCircularList(CircularLinkedList\* list) {  if (list->head == nullptr) {  cout << "La lista está vacía." << endl;  return;  }  Node\* currentNode = list->head;  do {  cout << currentNode->data << " -> ";  currentNode = currentNode->next;  } while (currentNode != list->head);  cout << "..." << endl; // Indica que la lista es circular  } |

**Estructuras de datos no lineales**

**Árboles**

Los árboles son estructuras de datos no lineales que se componen de nodos. Cada nodo contiene un valor y apunta a varios otros nodos llamados "hijos". Los nodos de un árbol se organizan jerárquicamente, comenzando desde un nodo en el parte superior llamado "raíz", seguido de nodos secundarios llamados "nodos hijos". Estos a su vez pueden tener sus propios nodos hijos, y así sucesivamente.

Las principales características de un árbol son:

* **Raíz:** El nodo superior del árbol desde el que se ramifican todos los demás nodos.
* **Nodo interno/Nodo rama:** Cualquier nodo que no es la raíz y que tenga al menos un nodo hijo.
* **Hoja/Nodo terminal:** Un nodo que no tiene nodos hijos.
* **Padre e hijos:** Los nodos se relacionan entre sí; un nodo se llama "padre" de sus nodos hijos directos.
* **Bosque:** colección de árboles.
* **Ramas:** conjunto finito de líneas dirigidas que conectan los nodos.
* **Altura del árbol:** La longitud del camino más largo desde la raíz hasta una hoja.
* **Profundidad de un nodo:** La longitud del camino desde la raíz hasta ese nodo.
* **Nodo descendiente:** Cualquier nodo accesible a través de ramas descendientes de un nodo.
* **Árbol binario:** Un árbol en el que cada nodo puede tener como máximo dos hijos: un hijo izquierdo y un hijo derecho.

Recordar: Solo un nodo de árbol puede ser raíz. Un nodo simple es un subárbol.

Los árboles se utilizan en una amplia gama de aplicaciones, como bases de datos, algoritmos de búsqueda (por ejemplo, árboles binarios de búsqueda), redes de computadoras, compiladores y mucho más, debido a su capacidad para organizar y estructurar datos de manera eficiente y jerárquica.

**Cuestionario:**

**¿Cuál es una característica principal de las estructuras de datos dinámicas?**

Las estructuras de datos dinámicas pueden cambiar de tamaño durante la ejecución del programa, a diferencia de las estructuras estáticas cuyo tamaño está predefinido en tiempo de compilación.

**¿Qué es una lista enlazada?**

Una lista enlazada es una estructura de datos compuesta por nodos donde cada nodo contiene un valor y un puntero (o referencia) al siguiente nodo en la secuencia.

**¿Qué tipos de listas enlazadas existen comúnmente?**

Las listas enlazadas más comunes son: simplemente enlazadas, doblemente enlazadas y circulares.

**¿Qué estructura de datos permite la inserción y eliminación eficiente al inicio y al final de la** **secuencia?**

La lista doblemente enlazada permite la inserción y eliminación eficiente tanto al inicio como al final de la secuencia.

**¿Cómo se accede a un elemento en una lista enlazada?**

En una lista enlazada, se accede a un elemento recorriendo la lista desde el principio (o el final) siguiendo los punteros entre nodos.

**¿Cuál es el nodo inicial de una lista enlazada?**

El primer nodo de una lista enlazada se llama nodo inicial o cabeza (head).

**¿Qué es una lista doblemente enlazada?**

Es una lista enlazada en la que cada nodo tiene un puntero tanto al siguiente nodo como al nodo anterior en la secuencia.

**¿Qué es una lista circular?**

Una lista circular es una lista enlazada en la que el último nodo apunta de nuevo al primer nodo, formando un círculo.

**¿Por qué se usan listas enlazadas en lugar de arreglos?**

Las listas enlazadas son útiles cuando se requiere una estructura de datos que pueda cambiar de tamaño dinámicamente, a diferencia de los arreglos, cuyo tamaño es fijo.

**¿Cuál es la complejidad temporal para insertar un elemento al inicio de una lista enlazada?**

La complejidad temporal para insertar un elemento al inicio de una lista enlazada es O(1).

**¿Qué es la operación "push" en una pila?**

"Push" es la operación que inserta un elemento en la parte superior (o final) de una pila.

**¿Cuál es la estructura de datos que sigue el principio LIFO (Last In, First Out)?**

La estructura de datos que sigue el principio LIFO es una pila.

**¿Qué operación se realiza en una pila para eliminar un elemento de la parte superior?**

La operación para eliminar un elemento de la parte superior de la pila se llama "pop".

**¿Qué es una cola?**

Una cola es una estructura de datos que sigue el principio FIFO (First In, First Out), similar a una fila de personas.

**¿Qué operación se realiza en una cola para agregar un elemento al final?**

La operación para agregar un elemento al final de una cola se llama "enqueue" o "push".

**¿Qué operación se realiza en una cola para eliminar un elemento del inicio?**

La operación para eliminar un elemento del inicio de una cola se llama "dequeue" o "pop".

**¿Qué es una lista circular doblemente enlazada?**

Es una lista enlazada en la que el último nodo apunta de nuevo al primer nodo, y cada nodo tiene un puntero al siguiente y al anterior.

**¿Cuál es la principal diferencia entre una pila y una cola?**

La principal diferencia es que una pila sigue el principio LIFO (Last In, First Out), mientras que una cola sigue el principio FIFO (First In, First Out).

**¿Cuál es la ventaja de una lista doblemente enlazada sobre una lista simplemente enlazada?**

La lista doblemente enlazada permite el acceso bidireccional a los nodos, lo que facilita la eliminación desde el final de la lista.

**¿Qué es un nodo en una estructura de datos?**

Un nodo es la unidad básica de una estructura de datos enlazada. Contiene datos y al menos un puntero a otro nodo.

**Ejemplo de listas**

#include <iostream>

#include <windows.h>

using namespace std;

// structs

struct Equipo

{

int goles;

string mejorJugador;

string division;

};

struct Nodo

{

string nombre;

struct Nodo \*siguiente;

struct Nodo \*anterior;

struct Equipo e;

};

// prototipo de funciones

void menu();

Nodo \*crearNodo(string d);

void agregarEquipo(string d, string player, string division, int goles);

void buscarEquipo(string name);

void mostrarJugadores();

void eliminar(string name);

// variables globales

int opcionMenu;

string nombreEquipo;

string mejorJugador;

int golesMarcados;

string division;

struct Nodo \*lista = nullptr;

struct Nodo \*ultimoNodo = nullptr;

int main()

{

SetConsoleOutputCP(CP\_UTF8);

menu();

return 0;

}

void menu()

{

do

{

cout << "\nBienvenido al registro de equipos de la UCA" << endl;

cout << "\nIngrese una opcion para continuar: " << endl;

cout << "1- Agregar equipo." << endl;

cout << "2- Buscar equipo." << endl;

cout << "3- Mostrar mejores jugadores de cada club." << endl;

cout << "4- Eliminar un equipo" << endl;

cout << "5- Salir" << endl;

cin >> opcionMenu;

switch (opcionMenu)

{

case 1:

cout << "\nIngrese el nombre del equipo: " << endl;

cin >> nombreEquipo;

cout << "Ingrese el nombre del mejor jugador: " << endl;

cin >> mejorJugador;

cout << "Ingrese la división " << endl;

cin >> division;

cout << "Ingrese los goles marcados " << endl;

cin >> golesMarcados;

agregarEquipo(nombreEquipo, mejorJugador, division, golesMarcados);

break;

case 2:

cout << "\nIngrese el nombre del equipo que desea buscar: " << endl;

cin >> nombreEquipo;

buscarEquipo(nombreEquipo);

break;

case 3:

mostrarJugadores();

break;

case 4:

cout << "\nIngrese el nombre del equipo que desea borrar" << endl;

cin >> nombreEquipo;

eliminar(nombreEquipo);

break;

case 5:

break;

default:

cout << "\nIngrese una opción válida" << endl;

break;

}

} while (opcionMenu != 5);

}

Nodo \*crearNodo(string d)

{

Nodo \*nuevoNodo = new Nodo();

nuevoNodo->nombre = d;

nuevoNodo->siguiente = nullptr;

nuevoNodo->anterior = nullptr;

return nuevoNodo;

}

void agregarEquipo(string d, string player, string division, int goles)

{

Nodo \*nuevoNodo = crearNodo(d);

nuevoNodo->e.mejorJugador = player;

nuevoNodo->e.division = division;

nuevoNodo->e.goles = goles;

if (lista == nullptr)

{

lista = nuevoNodo;

ultimoNodo=nuevoNodo;

}

else

{

nuevoNodo->siguiente = nuevoNodo;

nuevoNodo->anterior = ultimoNodo;

ultimoNodo=nuevoNodo;

}

cout << "Equipo agregado exitosamente " << endl;

}

void buscarEquipo(string name)

{

Nodo \*temporal = lista;

bool found = false;

while (temporal != nullptr)

{

if (temporal->nombre == name)

{

found = true;

cout << "\nEquipo: " << name << endl;

cout << "Mejor jugador: " << temporal->e.mejorJugador << endl;

cout << "Goles marcados: " << temporal->e.goles << endl;

cout << "División: " << temporal->e.division << endl;

break;

}

temporal = temporal->siguiente;

}

if (!found)

{

cout << "No se encontró el equipo buscado" << endl;

}

}

void mostrarJugadores()

{

cout << "\n Jugadores: " << endl;

Nodo \*temporal = lista;

while (temporal != nullptr)

{

cout << "\nEquipo: " << temporal->nombre << endl;

cout << "Mejor jugador: " << temporal->e.mejorJugador << endl;

temporal = temporal->siguiente;

}

cout << endl;

}

void eliminar(string name)

{

Nodo \*aux = lista;

while (aux != nullptr && aux->nombre != name)

{

aux = aux->siguiente;

}

if (aux == nullptr)

{

cout << "No se encontró el elemento" << endl;

}

else

{

if (aux->anterior != nullptr)

{

aux->anterior->siguiente = aux->siguiente;

}

else

{

lista = aux->siguiente;

}

if (aux->siguiente != nullptr)

{

aux->siguiente->anterior = aux->anterior;

}

delete aux;

cout << "Elemento eliminado" << endl;

}

}

void eliminar2(Nodo \*nodoAEliminar) {

if (nodoAEliminar == nullptr) {

cout << "El nodo a eliminar es nulo." << endl;

return;

}

if (nodoAEliminar->anterior != nullptr) {

nodoAEliminar->anterior->siguiente = nodoAEliminar->siguiente;

} else {

lista = nodoAEliminar->siguiente;

}

if (nodoAEliminar->siguiente != nullptr) {

nodoAEliminar->siguiente->anterior = nodoAEliminar->anterior;

}

delete nodoAEliminar;

cout << "Elemento eliminado" << endl;

}

void mostrarEquiposPorGoles(int cantidadGoles) {

Nodo \*temporal = lista;

bool encontrado = false;

while (temporal != nullptr) {

if (temporal->e.goles == cantidadGoles) {

encontrado = true;

cout << "\nEquipo: " << temporal->nombre << endl;

cout << "Mejor jugador: " << temporal->e.mejorJugador << endl;

cout << "Goles marcados: " << temporal->e.goles << endl;

cout << "División: " << temporal->e.division << endl;

}

temporal = temporal->siguiente;

}

if (!encontrado) {

cout << "No se encontraron equipos que hayan marcado " << cantidadGoles << " goles." << endl;

}

}