Tecnológico de Monterrey

Campus Sonora Norte

Escuela de Ingeniería y Ciencia

Clase:

Programación de estructuras de datos y algoritmos fundamentales

Nombre del trabajo:

Act 2.1 - Implementación de un ADT de estructura de datos lineales

Alumno:

Andres Sandoval Ibarra - A01253138

Fecha de Entrega:

24 sept 2023

Introducción:

En el siguiente trabajo se realizará la implementación de listas enlazadas en c++. Esto con el objetivo de entender el comportamiento de esta y así poder reforzar los conocimientos vistos en clase a cerca de las estructuras de datos. El saber como crear nuevas estructuras de datos es de gran importancia, pues esto nos garantiza el ser capaces de manipular datos de forma más eficiente. Las listas enlazadas son un gran punto de partida, para poder comprender a su vez el uso de punteros. Es por esto que a continuación se presentan las funcionalidades de create, read, update y del.

Código:

“Create”:

```
    void agregarInicio(T dato) {
        Node<T>* nuevo = new Node<T>(dato);
        if (head == nullptr) {
            head = nuevo;
        } else {
            nuevo->siguiente = head;
            head = nuevo;
        }
    }

    void agregarFinal(T dato) {
        Node<T>* nuevo = new Node<T>(dato);
        if (head == nullptr) {
            head = nuevo;
        } else {
            Node<T>* temp = head;
            while (temp->siguiente != nullptr) {
                temp = temp->siguiente;
            }
            temp->siguiente = nuevo;
        }
    }

void agregaEnMedio(T dato, int posicion) {
    Node<T>* nuevo = new Node<T>(dato);
    if (posicion == 0) {
        nuevo->siguiente = head;
        head = nuevo;
        return;
    }
    Node<T>* temp = head;
```

```
    int contador = 0;
    while (temp != nullptr) {
        if (contador == posicion - 1) {
            nuevo->siguiente = temp->siguiente;
            temp->siguiente = nuevo;
            return;
        }
        temp = temp->siguiente;
        contador++;
    }
}
```

La complejidad de agregarInicio() es O(1), pues no se tiene que recorrer en ningún punto la lista enlazada. Solo estamos agregando uno al inicio.

La complejidad de agregarFinal() es de O(n), pues se necesita recorrer toda la lista enlazada para poder llegar a agregar un nuevo nodo al final de la lista. Así que el tiempo de corrida dependerá de la cantidad de nodos que existan.

La complejidad de agregarEnMedio() es de O(n), aunque esta permite agregar un dato en una posición específica de la lista, en el peor de los casos queremos agregar un nodo al final de la lista enlazada. Por lo que la complejidad de este algoritmo será el mismo del agregaFinal().

“Read”

```
    int buscar(T dato){
        Node<T>* temp = head;
        int contador = 0;
        while (temp != nullptr) {
            if (temp->dato == dato) {
                return contador;
            }
            temp = temp->siguiente;
            contador++;
        }
        return -1;
    }
```

Como el objetivo de esta función es encontrar algún dato proporcionado por el usuario en la lista enlazada, en el peor de los casos este dato se encontraría en la última posición.

Entonces, el tiempo de corrida dependerá de la cantidad de los nodos. Por ello, este algoritmo tendrá una complejidad de O(n).

“Update”

```
    void cambiar(T dato, T nuevo){
        Node<T>* temp = head;
        while (temp != nullptr) {
            if (temp->dato == dato) {
                temp->dato=nuevo;
                return;
            }
            temp = temp->siguiente;
        }
    }
```

Esta función se encarga de encontrar en donde se encuentra el dato que introduce el usuario y cambiarlo por otro que también introduce el usuario. En el peor de los casos el dato que introduzca el usuario para intercambiar se encontrará al final. Por lo que al igual que buscar(), esta función tendrá una complejidad de O(n).

“Del”

```
    void eliminar(T dato) {
        if (head == nullptr) {
            return;
        }
        if (head->dato == dato) {
            head = head->siguiente;
            return;
        }
        Node<T>* temp = head;
        while (temp->siguiente != nullptr) {
            if (temp->siguiente->dato == dato) {
                temp->siguiente = temp->siguiente->siguiente;
                return;
            }
            temp = temp->siguiente;
        }
    }
```

Como en este algoritmo se tiene que buscar el dato que el usuario introduzca para eliminarlo. En el peor de los casos este dato se encontrará al final de la lista enlazada, por

lo que la complejidad de este algoritmo es también O(n), pues la eficiencia del código empeorará de forma lineal.

Casos de prueba:

- Para los primeros 3 casos de prueba se ponen a prueba todas las funciones, solo que cambia el tipo de dato que recibe para comprobar que esta estructura de dato se puede aplicar a cualquier tipo. Se que este código podría ser mucho más corto, pero lo hice para el primer caso y solo lo cambié los tipos de valores que iba a recibir.

```
    // Caso de prueba 1: Números enteros
    Lista<int> lista1;

    lista1.agregarInicio(1);
    lista1.agregarInicio(2);
    lista1.agregarFinal(3);
    lista1.agregarFinal(4);
    lista1.agregaEnMedio(5, 2);

    cout << "Lista de enteros: "<<endl;
    lista1.imprimir();

    int posicion1 = lista1.buscar(3);
    if (posicion1 != -1) {
            cout << "El elemento 3 se encuentra en la posición " <<
posicion1 << endl;
    } else {
        cout << "El elemento 3 no se encuentra en la lista." << endl;
    }

    lista1.cambiar(2, 6);
     cout << "Lista de enteros después de cambiar el elemento 2 por 6:
"<<endl;
    lista1.imprimir();

    lista1.eliminar(5);
      cout << "Lista de enteros después de eliminar el elemento 5:
"<<endl;
    lista1.imprimir();
    cout<<"--------------------------"<<endl;

    // Caso de prueba 2: Cadenas de texto
```

```
    Lista<string> lista2;

    lista2.agregarInicio("Manzana");
    lista2.agregarInicio("Banana");
    lista2.agregarFinal("Cereza");
    lista2.agregarFinal("Damasco");
    lista2.agregaEnMedio("Fresa", 2);

    cout << "Lista de cadenas de texto: "<<endl;
    lista2.imprimir();

    int posicion2 = lista2.buscar("Cereza");
    if (posicion2 != -1) {
          cout << "La cadena 'Cereza' se encuentra en la posición " <<
posicion2 << endl;
    } else {
          cout << "La cadena 'Cereza' no se encuentra en la lista." <<
endl;
    }

    lista2.cambiar("Banana", "Kiwi");
     cout << "Lista de cadenas de texto después de cambiar 'Banana' por
'Kiwi': "<<endl;
    lista2.imprimir();

    lista2.eliminar("Damasco");
     cout << "Lista de cadenas de texto después de eliminar 'Damasco':
"<<endl;
    lista2.imprimir();
    cout<<"--------------------------"<<endl;

    // Caso de prueba 3: Números decimales
    Lista<double> lista3;

    lista3.agregarInicio(1.5);
    lista3.agregarInicio(2.3);
    lista3.agregarFinal(3.7);
    lista3.agregarFinal(4.2);
    lista3.agregaEnMedio(5.1, 2);

    cout << "Lista de números decimales: " << endl;
    lista3.imprimir();
```

```
    int posicion = lista3.buscar(3.7);
    if (posicion != -1) {
           cout << "El elemento 3.7 se encuentra en la posición " <<
posicion << endl;
    } else {
        cout << "El elemento 3.7 no se encuentra en la lista." << endl;
    }

    lista3.cambiar(2.3, 6.0);
     cout << "Lista después de cambiar el elemento 2.3 por 6.0: " <<
endl;
    lista3.imprimir();

    lista3.eliminar(5.1);
    cout << "Lista después de eliminar el elemento 5.1: " << endl;
    lista3.imprimir();
    cout<<"--------------------------"<<endl;
```

Resultados:

Caso 1:

Lista de enteros:

2

1

5

3

4

El elemento 3 se encuentra en la posición 3

Lista de enteros después de cambiar el elemento 2 por 6:

6

1

5

3

4

Lista de enteros después de eliminar el elemento 5:

6

1

3

4

Caso 2:
Lista de cadenas de texto:
Banana
Manzana
Fresa
Cereza
Damasco
La cadena 'Cereza' se encuentra en la posición 3
Lista de cadenas de texto después de cambiar 'Banana' por 'Kiwi':
Kiwi
Manzana
Fresa
Cereza
Damasco
Lista de cadenas de texto después de eliminar 'Damasco':
Kiwi
Manzana
Fresa
Cereza

Caso 3:
Lista de números decimales:
2.3
1.5
5.1
3.7
4.2
El elemento 3.7 se encuentra en la posición 3
Lista después de cambiar el elemento 2.3 por 6.0:
6
1.5
5.1
3.7
4.2
Lista después de eliminar el elemento 5.1:
6

1.5

3.7

4.2

- Para el último caso de prueba se manda a buscar un número que no este en la lista para comprobar esta funcionalidad.

```
    // Caso de prueba 4: Buscar un dato que no esta en la lista
    Lista<int> lista4;

    lista4.agregarInicio(1);
    lista4.agregarInicio(2);
    lista4.agregarFinal(3);

    cout << "Lista de enteros: "<<endl;
    lista4.imprimir();

    posicion = lista4.buscar(4);
    if (posicion != -1) {
            cout << "El elemento 4 se encuentra en la posición " <<
posicion << endl;
    } else {
        cout << "El elemento 4 no se encuentra en la lista." << endl;
    }

    return 0;
```

Resultados:

Caso 4:

Lista de enteros:

2

1

3

Dato no encontrado.

El elemento 4 no se encuentra en la lista.