Programación de estructuras de datos y

algoritmos fundamentales (Gpo 14)

Act 2.2 - Verificación de las funcionalidades de una estructura de datos lineal

Alumno:

Alejandro Daniel González Carrillo A01570396

Profesores:

Luis Humberto González Guerra

Monterrey, Nuevo León. 29 de septiembre de 2020.

```cpp
// A01570396 Alejandro Daniel González Carrillo
// Act 2.1 - Implementación de un ADT de estructura de datos lineales

#include <iostream>
#include "LinkedList.h"

using namespace std;

int main() {
    LinkedList<string> lista, lista2;


    // Casos de Prueba - Metodo Create
    cout << "Caso CREATE" << endl;
    lista.create("papas",19); // Caso donde la posicion es un valor por encima del rango de la linkedlist.
    lista.create("Alex",-6); // Caso donde la posicion es un valor por debajo del rango de la linkedlist.
    lista.create("come",1); // Caso donde la posicion es 1.
    lista.create("papas",2); // Caso donde la posicion es entre el valor de rangos de la linkedlist.
    lista.create("con", 5); // Caso extra.
    lista.create("chile", 6); // Caso extra.
    lista.print();
```

Se generaron tres listas encadenadas de nodos llamadas lista, lista2 y lista3. Una se fue llenando durante el proceso, la otra era una lista vacía y la ultima fue para un caso especial de Delete.

Caso Create:

Lo que hace el caso create es generar nodos fuera y dentro del rango de la lista encadenada. En este caso yo genere la frase “Alex come papas papas con chile”. Genere un duplicado de palabras, genere palabras fuera del rango de valores por ejemplo: papas en la posicion 19 y lo que haces es agregarlas hasta atrás de la lista. Asi mismo, para la palabra Alex se coloco en la posicion -6 y se agrega en la parte de enfrente como primera. Igual se probo para cualquier rango dentro del tamaño de la lista.

```
Caso CREATE
La LinkedList contiene:
Alex
come
papas
papas
con
chile
```

```
// Casos de Prueba - Metodo Read
cout << "Caso READ" << endl;
cout << lista.read("dulces") << endl; // Caso donde no exista una palabra.
cout << lista.read("Alex") << endl; // Caso donde sea cualquier palabra dentro de la estructura.
cout << lista.read("") << endl; // Caso donde se introduce un string vacio.
cout << lista2.read("Alex") << endl; // Caso donde se recibe una lista vacia, debe regresar -1;
cout << lista.read("papas") << endl; // Caso donde hay una palabra repetida se muestra la posicion de la primera palabra encontrada.
```

Caso Read:

La idea del caso read es que te pueda leer y buscar la palabra que tu le digas dentro de la lista. Mi primera prueba fue con la palabra dulces la cual no existe dentro de la lista por lo cual me regreso -1. También se probó para los strings que si existieran dentro de la lista, al hacer lista.read("Alex") regresa el valor de posición en este caso como es el primero de la lista regresa 0. Para el caso de un string vacío nos regresa -1 ya que no existe un nodo que tenga un string vacío.

No solo se probó con una lista llena, sino que también se probo con una lista encadenada vacía. Al intentar buscar "Alex" dentro de la lista vacía nos retorno -1 debido a que esta no existe. Y finalmente se hizo una búsqueda normal de un valor existente dentro de una lista llena y nos retornó la posición siendo en este caso 2 sin embargo, al ser una palabra duplicada nomas nos regrese la primera posición en donde el programa la encuentra.

```
Caso READ
-1
0
-1
-1
2
```

```
// Casos de Prueba - Metodo Update
cout << "Caso UPDATE" << endl;
lista.update("Alex", "Daniel"); // Caso en donde es intercambio de palabras.
lista.update("papas", ""); // Caso en donde se intercambia por un string vacio.
lista.print();
```

Caso Update:

Para el caso de update, lo único que realiza es un intercambio de valor entre el string que se esta buscando y con el cual se busca reemplazar. En este problema la idea fue cambiar "Alex" por "Daniel". Así también se hizo intercambio de una palabra existente y duplicada por un string vacío y lo que regreso fueron dos nodos vacíos y no crasheo. Sin embargo, los nodos seguían ahí.

```
Caso UPDATE
La LinkedList contiene:
Daniel
come


con
chile
```

```
// Casos de Prueba - Metodo Del
cout << "Caso DEL" << endl;
lista.del(""); // Caso donde se eliminan los string que esten vacios.
lista.del("Daniel"); // Caso donde se busca eliminar al primer valor.
lista2.del("Alex"); // Caso donde se busca eliminar datos de una lista vacia.
lista.print();
```

Caso Del:

Finalmente, para el caso de delete (del) lo que hace es eliminar el nodo dependiendo de si contiene la información que el usuario busca borrar. Para los casos lo que hice fue eliminar los nodos vacíos como se observa en la siguiente imagen todos los nodos que no tuvieran información iban a ser eliminados. Así como también elimine la palabra “Daniel”. De la misma manera, probé eliminando un elemento de la lista2 que está vacía y no crasheo en este caso regreso un false.

```
Caso DEL
La LinkedList contiene:
come
con
chile
```

Caso Especial (DEL):

Suponiendo que tenemos una lista3 con puros datos iguales el método Del debería eliminar toda la lista y dejarla vacía.

```
// Caso de Prueba para delete donde una lista esta llena del mismo valor.
lista3.create("Alex",0);
lista3.create("Alex",0);
lista3.create("Alex",0);
lista3.create("Alex",0);
```

```
La LinkedList contiene:
Alex
Alex
Alex
Alex
```

```
lista3.print();
lista3.del("Alex");
lista3.print();
```

```
La LinkedList contiene:
```