Seminario de Estructura de Datos I

D06

“Programa 01 - TDA Lista Simple”

Abraham Magaña Hernández - 220791217
Ing. en Computación
Mstro. Noe Ortega Sanchez
11/02/2022

Seminario de Estructura de Datos I
Abraham Magaña Hernández - 220791217

TDA - Lista Simple

Para empezar nuestra TDA tiene un el tipo de dato que vamos a almacenar en los **Nodos,** este tipo de dato lo definimos como **Alumno,** al cual le agregaremos dos constructores, el default y uno que inicializa ambos atributos pasandolos como parámetros y usando el puntero **this.**

```
class Alumno{
    public:
        Alumno(){};
        Alumno(string, int);

        void setNombre(string);
        void setEdad(int);

        string getNombre();
        int getEdad();
    private:
        string nombre;
        int edad;
};
```

Al igual que tendrá sus métodos **Setters y Getters** para consultar sus atributos respetando el encapsulamiento de estos mismos.

Ahora, teniendo los tipos de datos que vamos a guardar en nuestro **Nodo**, pasemos a crear esta clase también.

```
class Nodo{
    public:
        Nodo(){};
    private:
        Nodo *siguente;
        Alumno Persona;
    friend class Lista;
};
```

```
class Lista{
    public:
        // Métodos Normale
s       void Inicializar();
        void EliminarTod ();
        bool Vacia();
        int Tamanio();
        void MostrarTodo();
        // Métodos Personalizados del Objet
o       void Insertar(Alumno);
        void InsertarFinal(Alumno);
        void InsertarPosicion(Alumno, int);
        void Eliminar(string);
        Alumno Buscar(string);
        Alumno Primero();
        Alumno Ultimo();
        Alumno Siguente(string);
        Alumno Anterior(string);
        // Helpers Personales (Validacione
s)      bool Existe(string);
    private:
        Nodo *raiz;
};
```

Esta clase es más sencilla, solo necesitamos el constructor vacío porque vamos a crear un método inicializar estando en la **Lista**, por ahora, solo necesitamos los atributos que en este caso es un puntero del mismo tipo **Nodo**, y también el dato que vamos a almacenar que es el tipo de dato **Alumno.**

Ahora teniendo esta clase hecha, necesitaremos la clase **Lista**, considerando este objeto nuestro TDA.

Los llamados métodos normales fueron métodos que no se relacionan como tal con el objeto especial a almacenar, los personalizados del objeto si tienen una relación con el objeto y los helpers personales, solamente fueron hechos para hacer validaciones al momento de llamar los demás métodos de nuestra clase Lista.

```
void Lista::Inicializar(){
    this->raiz = nullptr;
}
```

Empezando con **Inicializar**, debido a que mi constructor de Nodo no inicializa el Nodo en nullptr para su primer uso, lo hacemos con esta función haciendo uso del puntero especial de la clase Lista **this**.

```
void Lista::EliminarTod (){
    Nodo *aux;

    while(aux){
        aux = raiz;
        raiz = nullptr;
        delete aux;
    }
}
```

Ahora vamos con **EliminarTodo** yo suelo conocer este método más como anular, pero es válido llamarlo de esta manera también.

Es simple, tomamos un Nodo *aux, lo movemos por desde la raíz por todos los nodos mientras hacemos nullptr, al final eliminamos este aux y repetimos el proceso.

```
bool Lista::Vacia(){
    if(this->raiz){return false;}
    else{return true;}
}
```

Ahora el método de **Vacía**, es más simple, lo único que hacemos es comparar el nodo raíz que tiene nuestra lista como atributo, si tiene algo, debería arrojar un false, indicando que la lista no está vacía, de lo contrario un true, indicando que al menos la raíz de nuestra lista contiene algo, por lo cual, no está vacía.

```
int Lista::Tamanio(){

    Nodo *aux = raiz;
    int i = 0;

    if(Vacia()){return 0;}
    while(aux){
        i++;
        aux = aux->siguente;
    }
    return i;
}
```

Nuestro método **Tamanio** mejor expresado como Tamaño, es un método para recolectar el número de Nodos con datos almacenados en la lista, en nuestro caso, creamos un nodo aux que se vuelve una copia de la raíz o cabecera de nuestra lista, directamente si la lista está vacía, regresamos un 0 indicando que no hay ningún nodo con datos, de lo contrario, avanzamos por todos los nodos y aumentamos un contador, al final de todo, regresamos ese contador.

```
void Lista::MostrarTodo(){
    Nodo *aux;
    aux = raiz;
    int i = 1;

    std::cout << std::endl;

    while(aux){
        std::cout << "\t[" << i << "] - Alumno:   << aux->Persona.
getNombre() << ", Edad: " << aux->Persona.getEdad() << std::endl;
        i++;
        aux = aux->siguente;
    }
}
```

MostrarTodo es un método que usamos para desplegar por pantalla los objetos guardados en nuestra lista.

Creamos un nodo aux que copia a la raíz y se mueve a través de toda la lista mientras existan datos en los nodos, ahora, toma estos datos y usando los getters de nombre y edad de nuestra clase **Alumno** podemos desplegar la información mediante la terminal en secuencia hasta que topemos con un nullptr y la información dejará de desplegarse.

El método **Insertar** se encargará de insertar los datos, en este caso nuestro objeto Alumno en la Lista al principio de todo.

Creamos un nuevo nodo dinámicamente para guardar la información que queremos almacenar, si la lista está vacía, nuestra raíz se volverá el nodo que acabamos de crear, de lo contrario, el nuevo nodo que creamos en su puntero siguiente tomará la forma de la raíz y la raíz de nuestro nuevo nodo.

Esto simula un pequeño empujón para insertar el Nodo al principio de todo.

```
void Lista::Insertar(Alumno Persona){

    Nodo *aux = new Nodo;
    aux->Persona = Persona;
    aux->siguente = nullptr;

    if(Vacia()){
        raiz = aux;
    }else{
        aux->siguente = raiz;
        raiz = aux;
    }
}
```

Para el método **InsertarFinal** el proceso inicial es prácticamente igual, creamos un nodo dinámicamente con la información, si la lista está vacía, insertamos en la raíz, sino, recorremos la lista usando el puntero siguiente del aux, ¿por qué? porque donde queremos insertar es el nullptr del aux, no reemplazar ese aux, entonces, una vez llegamos a este punto, el puntero siguiente del aux tomará la referencia del nodo que acabamos de crear, de esta manera, hemos recorrido todo y insertado al final de la lista el dato que deseamos guardar.

```
void Lista::InsertarFinal(Alumno Persona){

    Nodo *tmp = new Nodo;
    tmp->Persona = Persona;
    tmp->siguente = nullptr;

    if(Vacia()){
        raiz = tmp;
    }else{
        Nodo *aux;
        aux = raiz;
        while(aux->siguente){
            aux = aux->siguente;

}       aux->siguente = tmp;
    }
}
```

```
void Lista::InsertarPosicion(Alumno Persona, int pos){

    Nodo *tmp = new Nodo;
    tmp->Persona = Persona;
    tmp->siguente = nullptr;

    if(Vacia()){
        raiz = tmp;
    }else{
        if(pos == 0){
            Insertar(Persona);
         else{
}           if(pos == Tamanio()){
            InsertarFinal(Persona);
         else{
}           Nodo *aux, *aux2;
            aux = raiz;

            for(int i = 0; i < pos; i++){
                aux2 = aux;
                aux= aux->siguente;

}

            aux2->siguente = tmp;
            tmp->siguente = aux;
            }

}   }
}
```

InsertarPosicion este método es algo más complejo, pero funciona bastante similar, primero, creamos el nodo dinámico con la información que queremos almacenar, debemos estar concientes que los casos de insertar al final y al principio pueden aplicar aquí, entonces, si la posición a insertar es la primera, solo mandamos de nuevo el dato a **Insertar** si la posición es igual al tamaño de la lista, es decir es la última, mandamos el dato a **InsertarFinal**.

Si no es ninguno de estos casos, recorremos la lista con un ciclo for hasta llegar a la posición deseada, tenemos dos nodos aux, uno para controlar el nodo anterior a donde vamos a insertar y otro que está en la posición que queremos mover, una vez ahí, hacemos un salto ¿como?, al nodo anterior le asignamos como siguiente el nuevo nodo y al nuevo nodo en su campo siguiente insertamos el nodo que antes era siguiente del anterior.

Eliminar es uno de mis métodos favoritos, creamos un nodo aux qué es la copia de la raíz, ahora, si la raíz contiene el dato que queremos borrar, solo damos un salto a su campo siguiente, sino, avanzamos hasta encontrar el dato en el nodo siguiente del que comparamos, guardamos el nodo anterior como el que estamos, avanzamos, en ese nodo, y ahora hacemos que ese nodo anterior de un salto copiando el siguiente de su siguiente, al final borramos el que quedó en medio.

```
void Lista::Eliminar(string Nombre){
    Nodo *aux, *anterior;
    aux = raiz;

    if(Nombre == aux->Persona.getNombre()){
        raiz = raiz->siguente;
    }else{
        while(Nombre != aux->siguente->Persona
.getNombre()){
            aux = aux->siguente;

}       anterior = aux;
        aux = aux->siguente;
        anterior->siguente = aux->siguente;
    }
        delete aux;
}
```

Buscar, este método pasa por toda la lista usando un nodo aux para no modificar nada como tal, una copia, si la lista contiene datos, empieza este proceso, si encuentra en algún nodo una coincidencia con el dato que queremos buscar, que es el nombre básicamente, regresa lo que hay en ese nodo, sino, sigue avanzando.

Normalmente esta forma crearía un error de core, pero para eso veremos luego la validación de **Existe**.

```
Alumno Lista::Buscar(string Nombre){

    Nodo *aux;
    aux = raiz;

    if(!Vacia()){
        while(aux){
            if(aux->Persona.getNombre() == Nombre){
                return aux->Persona;
             else{
}               aux = aux->siguente;

}
}   }
}
```

El método de **Primero** revisa si la raíz es diferente de nullptr, indicando que al menos hay un dato almacenado, si es el caso, regresa lo que haya almacenado en la raíz, de esta manera obtenemos el primer dato en la lista.

```
Alumno Lista::Primero(){
    if(this->raiz){
        return raiz->Persona;
    }
}
```

Último, este método regresa al igual que **Primero** un tipo de dato Alumno, el proceso es similar, pero esta vez, necesitamos recorrer la lista, de nuevo usamos un nodo aux para no modificar nada y movernos libremente mientras haya datos, una vez lleguemos al final de la lista, regresamos el dato almacenado en ese nodo.

De esta manera hemos conseguido el último dato en nuestra lista.

```
Alumno Lista::Ultimo(){
    Nodo *aux;
    aux = raiz;

    while(aux->siguente){
        aux = aux->siguente;
    }

    return aux->Persona;
}
```

```
Alumno Lista::Siguente(string Nombre){
    Nodo *aux;
    aux = raiz;

    while(Nombre != aux->Persona.getNombre()){
        aux = aux->siguente;
    }
    if(Nombre == aux->Persona.getNombre()){
        if(aux->siguente){
            return aux->siguente->Persona;

}   }
}
```

El método **Siguente** lo que hace es devolver el siguente dato del que comparamos, creamos un nodo aux para recorrer la lista, mientras no topemos el nombre que queremos encontrar, seguimos avanzando, una vez llegado a esto, regresamos el siguiente del aux en su campo que guarde el dato, en este caso un Alumno.

```
Alumno Lista::Anterior(string Nombre){
    Nodo *aux;
    aux = raiz;

    while(Nombre != aux->siguente->Persona.
getNombre()){
        aux = aux->siguente;
    }
    return aux->Persona;
}
```

El método **Anterior**, crea algo similar, es el mismo proceso pero esta vez en lugar de regresar el aux en su campo siguiente lo que hay en su campo dato, bueno, regresamos lo que hay en el campo dato de nuestro aux, es como si hubieras llegado al anterior, entonces por eso podemos hacerlo.

```
bool Lista::Existe(string Nombre){
    Nodo *aux = raiz;

    if(Vacia()){
        return false;
    }else{
        while(aux){
            if(aux->Persona.getNombre() == Nombre){ return true; }
            else{ aux = aux->siguente; }

}       return false;
    }
}
```

Finalmente llegamos a un método que agregue por mi cuenta **Existe**, esto solamente se mueve por la lista comprobando si existe algún dato que coincida con estas características, así podemos mandar llamar otros métodos sin problema de que no encuentren el dato que necesitan. Este método se implementa en el menú directamente y es la ayuda más directa para lidiar con errores de core.