Índice

Índice
Implementación en listas
- Ejercicio 1
- Ejercicio 2
- Ejercicio 3
- Ejercicio 4
- Ejercicio 5
- Ejercicio 6
- Ejercicio 7
- Ejercicio 8
- Ejercicio 9

Uso de Listas
- Ejercicio 1 - Celiacos
- Ejercicio 2 - Universidades
- Ejercicio 3 - Restaurantes

Implementación en listas

Ejercicio 1

```
template <typename Tipo>
void Lista<Tipo>:: intercambiar(int posA, int posB) {
    if (posA >= 0 && posA < elementos && posB >= 0 && posB < elementos) {
        Nodo<Tipo> *nodoA = obtenerNodo(posA);
        Nodo<Tipo> *nodoB = obtenerNodo(posB);
        Tipo aux = nodoA->obtenerDato();
        nodoA->asignarDato(nodoB->obtenerDato());
        nodoB->asignarDato(aux);
    }
    else
        std::cout << "\n\tLas posiciones a intercambiar son invalidas";
}
```

Ejercicio 2

```
template <typename Tipo>
void Lista<Tipo>:: agregarLista(Lista<Tipo> *&lista) {
    for (int i = 0; i < lista->obtenerElementos(); i++) {
        this->agregarAlFinal(lista->obtenerDato(i));
    }
}
```

Ejercicio 3

```
template <typename Tipo>
Lista<Tipo>* Lista<Tipo>:: unirLista(Lista<Tipo> *&lista) {
    Lista<Tipo>* listaNueva = new Lista<Tipo>;
    listaNueva = this;
    listaNueva->agregarLista(lista);
    return listaNueva;
}
```

Ejercicio 4

```
template <typename Tipo>
void Lista<Tipo>:: invertirLista() {
    int i = 0, j = elementos - 1;
    int medio = elementos / 2;
    while (i != j && i != medio) {
        intercambiar(i, j);
        i++;
        j--;
    }
}
```

Ejercicio 5

```
template <typename Tipo>
Lista<Tipo>* Lista<Tipo>:: obtenerListaInvertida() {
    Lista<Tipo>* listaNueva = new Lista<Tipo>;
    listaNueva = this;
    listaNueva->invertirLista();
    return listaNueva;
}
```

Ejercicio 6

Aclaración: No usé el método `comparar_con()` que se menciona en el enunciado porque quería probar el código y no tenía ganas de implementar el método. En el caso de querer usarlo se puede reemplazar el bloque `if/else if/else` por un `switch()`

```
template <typename Tipo>
void Lista<Tipo>:: mergearLista(Lista<Tipo> *&lista) {

    int posOriginal = 0, posRecibida = 0;
    Tipo datoNuevo;

    while (posOriginal < elementos && posRecibida < lista->elementos) {

        datoNuevo = lista->obtenerDato(posRecibida);

        if (this->obtenerDato(posOriginal) < datoNuevo)
            posOriginal++;

        else if (this->obtenerDato(posOriginal) > datoNuevo) {
            this->agregarEnPosicion(datoNuevo, posOriginal + 1);
```

```
            posRecibida++;
        }

        else {
            posOriginal++;
            posRecibida++;
        }
    }

    while (posRecibida < lista->elementos) {
        datoNuevo = lista->obtenerDato(posRecibida);
        this->agregarAlFinal(datoNuevo);
        posRecibida++;
    }
}
```

Ejercicio 7

```
template <typename Tipo>
void Lista<Tipo>:: eliminarDato(Tipo dato) {
    int i = 0;
    bool encontrado = false;
    while (i < elementos && !encontrado) {
        if (dato == this->obtenerDato(i)){
            Nodo<Tipo> *anterior, *siguiente, *actual = obtenerNodo(i);
            if (actual == primero) {
                siguiente = obtenerNodo(i + 1);
                primero = siguiente;
            }
            else if (i == elementos - 1) {
                anterior = obtenerNodo(i - 1);
                anterior->asignarSiguiente(nullptr);
            }
            else {
                anterior = obtenerNodo(i - 1);
                siguiente = obtenerNodo(i + 1);
                anterior->asignarSiguiente(siguiente);
            }
            encontrado = true;
            delete actual;
            elementos--;
        }
        i++;
    }
}
```

Ejercicio 8

```
template <typename Tipo>
void Lista<Tipo>:: eliminarDatoMultiplesOcurrencias(Tipo dato) {
    int repeticiones = 0;
    for (int i = 0; i < elementos; ++i) {
        if (dato == obtenerDato(i))
            repeticiones++;
    }
    while (repeticiones > 0) {
        eliminarDato(dato);
        repeticiones--;
    }
}
```

Ejercicio 9

```
template <typename Tipo>
Lista<Tipo>* Lista<Tipo>:: originalMenosRecibida(Lista<Tipo> *&lista) {
    Lista<Tipo>* nueva = new Lista<Tipo>;
    bool encontrado = false;
    for (int i = 0; i < this->elementos; ++i) {
        Tipo datoA = this->obtenerDato(i), datoB;
        int j = 0;
        while (j < lista->obtenerElementos() && !encontrado) {
            datoB = lista->obtenerDato(j);
            if (datoA == datoB)
                encontrado = true;
            j++;
        }
        if (!encontrado)
            nueva->agregarAlFinal(datoA);
        encontrado = false;
    }
    return nueva;
}
```

Uso de Listas

Ejercicio 1 - Celiacos

```
Lista<Alimento*>* BuscadorDeComidas:: comidasParaCeliacos (Lista<Alimento*>
*&comidas, Lista<string> *&ingredientesPermitidos, Lista<string>
*&ingredientesNoPermitidos, unsigned int caloriaMaxima) {

    Lista<Alimento*>* nuevasComidas = new Lista<Alimento*>;

    for (int i = 0; i < comidas->obtenerElementos(); ++i) {
        int j = 0;
        bool permitido = false;
        while (j < ingredientesPermitidos->obtenerElementos() && !permitido) {
            std::string ingrediente = ingredientesPermitidos->obtenerDato(j);
            permitido = existeIngrediente(comidas->obtenerDato(i), ingrediente);
            j++;
        }
```

```
        int k = 0;
        bool noPermitido = true;
        while (k < ingredientesNoPermitidos->obtenerElementos() && noPermitido)
{
            std::string ingrediente = ingredientesNoPermitidos->obtenerDato(k);
            noPermitido = !existeIngrediente(comidas->obtenerDato(i),
ingrediente);
            k++;
        }

        unsigned int calorias = comidas->obtenerDato(i)->obtenerCalorias();
        if (permitido && noPermitido && calorias < caloriaMaxima) {
            string nombre = comidas->obtenerDato(i)->obtenerNombre();
            Lista<string>* ingredientes = comidas->obtenerDato(i)-
>obtenerIngredientes();
            Alimento* alimento = new Alimento(nombre, calorias, ingredientes);
            nuevasComidas->agregarAlFinal(alimento);
        }
    }
    return nuevasComidas;
}

bool BuscadorDeComidas:: existeIngrediente(Alimento* comida, string
&ingrediente) {
    bool existe = false;
    int i = 0;
    Lista<string>* ingredientes = comida->obtenerIngredientes();
    while (i < ingredientes->obtenerElementos() && !existe) {
        if (ingredientes->obtenerDato(i) == ingrediente)
            existe = true;
        i++;
    }
    return existe;
}
```

Ejercicio 2 - Universidades

```
Lista<Universidad*>* BuscadorUniversidades:: recomendarUniversidades
(Lista<Universidad*> *&universidades,

 Lista<string> *&vocaciones, int rankingMinimo) {
    Lista<Universidad*>* nuevasComidas = new Lista<Universidad*>;

    for (int i = 0; i < universidades->obtenerElementos(); ++i) {
        Universidad* universidad = universidades->obtenerDato(i);
        Lista<string>* carreras = universidad->obtenerCarreras();
        unsigned int ranking = universidades->obtenerDato(i)->obtenerRanking();

        int j = 0;
        bool existe = false;
        while (j < vocaciones->obtenerElementos() && !existe) {
            string vocacion = vocaciones->obtenerDato(j);
            existe = existeVocacion(universidad, vocacion);
            j++;
        }
```

```
        if (existe && ranking > rankingMinimo) {
            string nombre = universidades->obtenerDato(i)->obtenerNombre();
            Universidad* universidad = new Universidad(nombre, ranking,
carreras);
            nuevasComidas->agregarAlFinal(universidad);
        }
    }
    return nuevasComidas;
}

bool BuscadorUniversidades:: existeVocacion(Universidad* universidad, string
&vocacion) {
    bool existe = false;
    int i = 0;
    Lista<string>* carreras = universidad->obtenerCarreras();
    while (i < carreras->obtenerElementos() && !existe) {
        if (carreras->obtenerDato(i) == vocacion)
            existe = true;
        i++;
    }
    return existe;
}
```

Ejercicio 3 - Restaurantes

```
Lista<Restaurante*>*
BuscadorRestaurantes::recomendarRestaurantes(Lista<Restaurante *> *restaurantes,
                                                    Lista<string>
*platosDeseados, int precioMaximo) {
    Lista<Restaurante*>* nuevasComidas = new Lista<Restaurante*>;

    for (int i = 0; i < restaurantes->obtenerElementos(); ++i) {
        Restaurante* restaurante = restaurantes->obtenerDato(i);
        Lista<string>* platos = restaurante->obtenerPlatos();
        int precioPromedio = restaurantes->obtenerDato(i)-
>obtenerPrecioPromedio();

        int j = 0, cantidadPlatos = 0;
        while (j < platosDeseados->obtenerElementos() && cantidadPlatos <
PLATOS_MINIMOS) {
            string vocacion = platosDeseados->obtenerDato(j);
            if (existePlato(restaurante, vocacion))
                cantidadPlatos++;
            j++;
        }

        if (cantidadPlatos >= PLATOS_MINIMOS && precioPromedio < precioMaximo) {
            string nombre = restaurantes->obtenerDato(i)->obtenerNombre();
            Restaurante* restaurante = new Restaurante(nombre, precioPromedio,
platos);
            nuevasComidas->agregarAlFinal(restaurante);
        }
    }
    return nuevasComidas;
}
```

```
bool BuscadorRestaurantes::existePlato(Restaurante *restaurante, string &plato)
{
    bool existe = false;
    int i = 0;
    Lista<string>* platos = restaurante->obtenerPlatos();
    while (i < platos->obtenerElementos() && !existe) {
        if (platos->obtenerDato(i) == plato)
            existe = true;
        i++;
    }
    return existe;
}
```