César Silva - A01252916

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

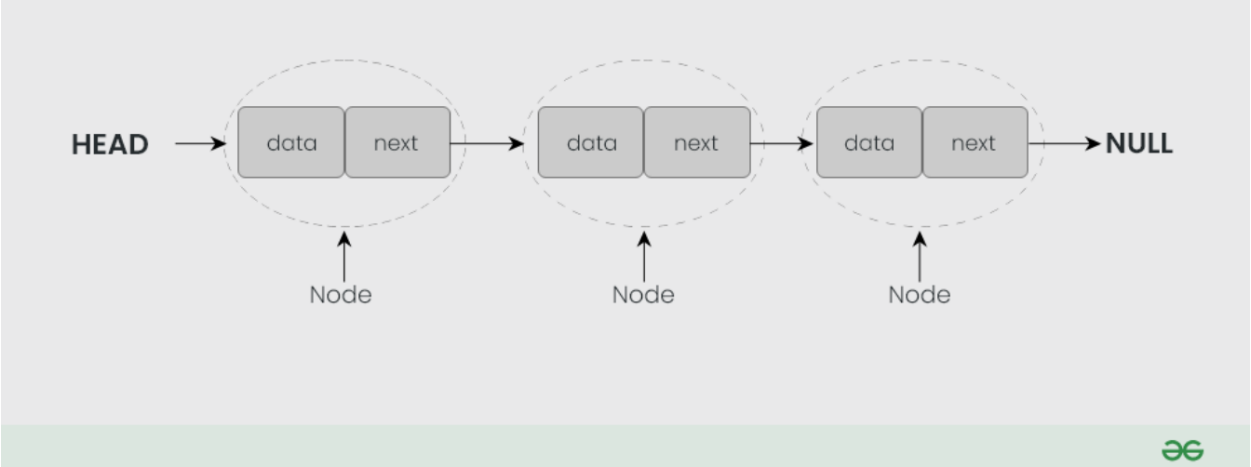
Repositorio en github

Introducción

Una lista anidada es una estructura de datos lineal en la que los elementos no se almacenan de forma contigua en la memoria. Esto implica que cada valor debe tener apuntadores hacia las ubicaciones de otros valores en la lista. En esta estructura, se utilizan nodos, y cada uno de estos nodos contiene un valor o dato, así como un puntero que señala al siguiente nodo en la secuencia. Es importante destacar que el último nodo siempre debe apuntar a NULL o nullptr, indicando que no existe otro nodo después de él.

En contraste con los arreglos estáticos, las listas enlazadas nos dan una flexibilidad y eficiencia significativas en la gestión de datos. Estas estructuras dinámicas permiten la organización secuencial de elementos mediante nodos interconectados. El hecho de que cada nodo contiene datos y una referencia al siguiente nodo en la lista les permite establecer conexiones lógicas entre elementos y abrir puertas a una amplia variedad de aplicaciones.

En este documento se explorará la implementación de esta estructura de datos mediante operaciones CRUD (Create, Read (buscar), Update, Delete) en C ++. Además, se analizarán las complejidades temporales de dichas operaciones en la lista anidada.



Nodo: Un nodo tiene dos componentes, dato y apuntador hacia el siguiente.
Cabeza.: Primer nodo.
Dato/Valor: Aquí se almacena el valor asociado con el nodo, por ejemplo 1 o “Pépé”.

Puntero: Este guarda la dirección en la memoria del siguiente nodo.
Algunas ventajas que tenemos al utilizar listas enlazadas:

- Facilidad para insertar y borrar: en comparación de arreglos, no ocupamos mover de lugar los datos, solamente necesitamos actualizar las direcciones.
- Memoria dinámica: Podemos alocar o borrar la cantidad de memoria a tiempo de corrida si se trata de insertar o borrar.
- Tamaño variable: las listas enlazadas pueden crecer o reducirse para adaptarse a la cantidad de datos a manejar, en comparación de los arreglos estos ya tienen un tamaño definido en el momento de su creación.
- Implementación: Las listas enlazadas nos pueden servir como base para poder implementar estructuras de datos más complejas como por ejemplo pilas o mapas.

Casos de prueba

```
int main ()
{
Node* head = new Node(); // Creación de nodo cabeza

head ->value = 1; // primer valor = 1
head ->next = NULL; // Apunta a null ya que no existe un nuevo elemento al cual apuntar
// CREATE
std::cout << "Creacion de elemntos, principio, en medio, final" << std::endl;
// Insertar al principio
insertFront (&head, 696); // la función recibe la dirección de la cabeza y el valor a insertar
// Insertar al final
insertEnd (&head, 420);
insertEnd (&head,3333);

// Insertar en medio
insertGivenNode(head,22);
// READ
search(head, 696);
// UPDATE
printL(head);
std::cout << "--------------------------" <<std::endl;
std::cout << "Actualizacion de valor" <<std::endl;
updateData(head, 420, 555);
printL(head);
```

```
// DELETE
std::cout << "Eliminacion de datos" <<std::endl;
std::cout << "--------------------------principio" <<std::endl;
head = head->next;
printL(head);
std::cout << "--------------------------final" <<std::endl;
deleteFromEnd(head);
printL(head);
std::cout << "--------------------------En medio" <<std::endl;
deleteFromMiddle(head);
printL(head);
}
```

En el main.cpp podemos observar los casos de prueba utilizados para cada operación CRUD.

Operaciones create:

- Inserción al principio del dato/valor 696
- Inserción al final del dato/valor 420 y 3333
- Inserción en medio del dato/valor 22

Operación read:

- búsqueda del dato/valor 696

Operación update:

- Actualización de valor 420 a 5555

Operación delete

- Eliminación del primer dato 696
- Eliminación del dato final 3333
- Eliminación del dato de en medio 1

Seleccioné estos casos de prueba específicos para poder ilustrar y analizar diversas situaciones en las que se pueden realizar operaciones de inserción y eliminación de elementos dentro de una lista anidada. Cada uno de estos casos prueba se enfoca en una ubicación particular dentro de la lista anidada donde estas operaciones pueden ocurrir, permitiendo así una comprensión más completa de cómo funcionan y cómo pueden afectar la estructura de datos.

Resultados de casos prueba

```
Creacion de datos, principio, en medio, final
Busqeda de datos
Se encontro el elemento 696 en la lista
696
22
1
420
3333
---------------------------
Actualizacion de valor
696
22
1
555
3333

Eliminacion de datos
---------------------------principio
22
1
555
3333
---------------------------final
22
1
555
---------------------------En medio
22
555
```

- Se crean los datos como lista anidada de 696, 22,1,420,3333
- Se busca el elemento 696
- Se actualiza 420 ->5555
- Se eliminan 696, 3333 y 1

Complejidades temporales

```
void insertGivenNode (Node*prev, int insertValue){
if (prev == NULL){
std::cout << "No se pude instertar en el nodo especifíco si la lista esta vacía";
return;
}
Node * newNode = new Node();
newNode->value = insertValue;

newNode->next=prev->next;
prev->next = newNode;

}
void insertFront (Node**head, int insertValue){ // tenemos un puntero a puntero ya que se
// modificara la dirección de la cabeza Si fuera Node * estrai apuntando al objeto node
// Node** apunta hacia la memoria donde el puntero head existe
Node * newNode = new Node(); // Creación de nuevo nodo a inertar
newNode->value = insertValue; // Asignación de valor a nuevo nodo
newNode->next = *head; // El nuevo nodo ahora se asigna la dirección
*head = newNode; // la dirección de la cabeza ahora esta en su nuevo lugar
}



void insertEnd (Node**head, int insertValue){
Node * newNode = new Node(); // Preparamos nodo a insertar ; es decir el nuevo nodo
newNode->value = insertValue; // El valor del nuevo nodo va a depneder del parametro insertValue que le pasemos
newNode->next = NULL; // El nuevo nodo a insertar va a apunar a NULL ya que será el ultimo
if(*head == NULL ){ // Si el puntero de la cabeza apunta a NULL signfica que la lista esta vacía
*head = newNode; // el puntero de la cabeza apuntara al nuevo nodo
return;
}
Node *lastNode = *head; // vamos a empzar la busqueda del utlimo nodo desde el principio por lo tanto *ultimo = *cabeza
while (lastNode->next != NULL){ // Mientras el apuntador del utlimo nodo no sea igual a NULL, sigue iternado
lastNode = lastNode->next;
}
lastNode->next = newNode;
```

```
}
```

Complejidad temporal CREATE

Create está compuesto de tres funciones, una para insertar al principio, final y en medio. Comencemos con la inserción en medio de la lista, donde la complejidad temporal es constante, es decir, O(1). Esto se debe a que implica operaciones como la comparación de punteros, que siempre tomarán la misma cantidad de tiempo, la creación de un nuevo nodo que no depende del tamaño de la lista, ya que solo se asigna memoria, y la asignación de valores y la actualización de punteros, que también son operaciones constantes y no dependen del tamaño de la lista.

En cuanto a la inserción al principio de la lista, también presenta una complejidad temporal de O(1). Similarmente, involucra la creación de un nuevo nodo, que es independiente del tamaño de la lista, la asignación de valores al nuevo nodo y la actualización de punteros, ya que simplemente se actualiza el nodo cabeza al insertar un elemento al principio.

Finalmente, la inserción al final de la lista tiene una complejidad temporal de O(n), donde "n" representa la cantidad de elementos presentes en la lista. Esto se debe a que se utiliza un ciclo while para determinar cuál es el nodo final o el que apunta a NULL, y luego se inserta el nuevo nodo.

Complejidad temporal READ

```
bool search (Node * head, int target){
Node* current = head;
while(current != NULL){
if (current->value == target){
std::cout << "Se encontro el elemento " << target << " en la lista" << std::endl;
return true;
}
current = current->next;
}
std::cout << "No se encontro el elemento" <<target << "en la lista" << std::endl;
return false;
}
```

En esta función de búsqueda de elementos en una lista enlazada, la complejidad temporal es de O(n), donde "n" representa la cantidad de datos en la lista. Esto se debe a que la función utiliza un ciclo while para recorrer la lista en busca del elemento deseado. En el peor de los casos, se

debe iterar a través de toda la lista, especialmente cuando el elemento buscado se encuentra en el último nodo.

Complejidad temporal UPDATE

```
void updateData (Node * head, int target, int updateValue){
Node* current = head;
while (current != NULL){
if (current->value == target){
current->value = updateValue;
}
current = current->next;
}

}
```

En esta función, también se presenta una complejidad temporal de O(n), similar a la función de búsqueda. Esto se debe a que, al igual que en la búsqueda, primero debemos localizar el elemento que deseamos actualizar, lo que implica la utilización de un ciclo while para asegurarnos de que no hemos llegado al final de la lista. La diferencia radica en el momento en que se encuentra el valor a actualizar; en ese punto, simplemente se actualiza el valor del puntero con el nuevo valor deseado.

Complejidad temporal DELETE

```
void deleteFromMiddle(Node* prev){
if (prev == nullptr || prev->next == nullptr){
return;
}
Node* ndelete = prev->next;
prev->next = ndelete->next;
delete ndelete;

}
```

```
void deleteFromEnd(Node *head){
Node * temp = head;
while (temp->next->next != NULL){
temp = temp->next;
}
temp->next = NULL;
}
```

Las funciones de delete se dividen en 3, borrar al principio, al final y en medio.

La complejidad temporal de la función para eliminar elementos al principio de la lista es O(1). Esto se debe a que la operación implica simplemente la actualización del puntero de la cabeza para que apunte al siguiente nodo, es decir, head = head->next. Esta operación no depende de la cantidad de datos en la lista y siempre tomará la misma cantidad de tiempo, lo que la hace constante y eficiente, independientemente del tamaño de la lista.

La complejidad temporal de la función deleteFromMiddle es O(n) en el peor caso debido a la necesidad de encontrar el nodo anterior al nodo que se eliminará. Para hacerlo, la función debe recorrer la lista desde el inicio hasta el nodo justo antes del que se va a eliminar. En situaciones en las que el nodo a eliminar se encuentra en el medio de la lista, esto implica recorrer aproximadamente la mitad de la lista, lo que resulta en una complejidad de O(n).

Finalmente, en la función de eliminación al final de la lista, su complejidad temporal es O(n), donde "n" representa el número de nodos en la lista. Esta complejidad viene debido al ciclo while que recorre la lista para localizar el penúltimo nodo, una operación necesaria para eliminar el último nodo.

Conclusiones

Para concluir las listas enlazadas son estructuras de datos versátiles que ofrecen ventajas significativas en la gestión de datos, como la facilidad para insertar y borrar elementos, la capacidad de ajustar dinámicamente la memoria, y su utilidad como base para estructuras de datos más complejas. Es esencial comprender las complejidades temporales asociadas a las operaciones CRUD en listas enlazadas. Estas complejidades varían desde operaciones constantes y eficientes, como la inserción al principio, hasta operaciones lineales que requieren recorrer la lista, como la inserción al final y la búsqueda. Comprender estas complejidades nos ayudan a diseñar algoritmos eficientes y tomar decisiones informadas al elegir una estructura de datos en función de los requisitos de una aplicación específica.