Implementación de una lista simplemente enlazada en C++

1. Introducción

En este documento se explica la implementación de una `forward_list` en C++. La lista es simplemente enlazada y cuenta con operaciones básicas y adicionales. Puede revisar la implementación completa en el repositorio de github.

2. Estructura `Node`

Cada nodo de la lista contiene un dato (`data`) y un puntero al siguiente nodo (`next`).

```
template <typename T>
struct Node {
    T data;
    Node *next;

    Node(T data) : data(data), next(nullptr) {}
    Node() {}
};
```

3. Clase `forward`

La clase administra el puntero a la cabeza de la lista y define las operaciones principales.

3.1. Constructor y Destructor

- El constructor inicializa la lista vacía.
- El destructor recorre la lista y libera la memoria de cada nodo.

3.2. Acceso a elementos

- `front()` devuelve el primer elemento y lanza un error si la lista está vacía, esta decisión fue tomada ya que se requiere que la función retorne un valor de tipo `T`.

```
T front() {
    if (!head)
        throw std::runtime_error("List is empty!");
    return head->data;
}
```

- `back()` devuelve el último elemento recorriendo la lista. Similar a la función anterior, lanza un error si la lista está vacía.

```
T back() {
    if (!head)
        throw std::runtime_error("List is empty!");
    Node<T> *curr = head;
    while (curr->next)
        curr = curr->next;
    return curr->data;
}
```

- `empty()` retorna verdadero si la lista no tiene nodos.

```
bool empty() {
    return head == nullptr;
}
```

- `size()` cuenta los nodos de la lista. Para ello utiliza un contador `c` y un puntero auxiliar `curr` que recorre toda la lista de inicio a fin.

```
int size() {
    int c = 0;
    Node<T> *curr = head;
    while (curr) {
        curr = curr->next;
        c++;
    }
    return c;
}
```

- Operador `[]` accede a un elemento por índice. Si la lista está vacía lanza un error, de manera similar, si el índice es negativo o es mayor o igual que el tamaño de la lista.

```
T operator[](int idx) {
    if (!head)
        throw std::runtime_error("List is empty!");
    else if (idx >= this->size() || idx < 0)
        throw std::out_of_range("Index out of bounds");

    int c = 0;
    Node<T> *temp = head;
    while (c < idx) {
        temp = temp->next;
        c++;
    }
    return temp->data;
}
```

3.3. Inserciones y Eliminaciones

- `push_front()` inserta al inicio. Para ello se crea un nodo temporal con el valor recibido, luego, el nodo que le sigue apuntá a la cabeza actual, y finalmente se

actualiza la cabeza para que apunte al nuevo nodo.

```
void push_front(T val) {
    Node<T> *temp = new Node<T>(val);
    temp->next = head;
    head = temp;
}
```

- `push_back()` inserta al final recorriendo hasta el último nodo. Si la lista está vacía, se crea un nodo y se le asigna directamente a la cabeza. En caso contrario, se recorre la lista hasta el último nodo y se hace que su puntero `next` apunte al nuevo nodo creado.

```
void push_back(T val) {
    if (!head) {
        head = new Node<T>(val);
        return;
    }
    Node<T> *temp = new Node<T>(val);
    auto curr = head;
    while (curr->next)
        curr = curr->next;
    curr->next = temp;
}
```

- `pop_front()` elimina el primer nodo. Si la lista está vacía lanza un error, caso contrario, usando un puntero temporal, se almacena la cabeza y luego hacemos que esta sea igual que el próximo elemento, finalmente se elimina la cabeza (almacenada en un puntero temporal).

```
void pop_front() {
    if (!head)
        throw std::runtime_error("List is empty!");
    Node<T> *temp = head;
    head = temp->next;
    delete temp;
}
```

- `pop_back()` elimina el último nodo. Para listas con más de dos elementos, se avanza hasta el penúltimo elemento y luego se elimina el último.

```
void pop_back() {
    if (!head)
        throw std::runtime_error("List is empty!");
    if (!head->next) {
        head = nullptr;
        return;
    }
    Node<T> *temp = head;
    while (temp->next->next) {
        temp = temp->next;
    }
```

```
    temp->next = nullptr;
}
```

- `clear()` elimina todos los nodos. Para lograrlo guarda la cabeza en un puntero temporal, luego avanza la cabeza al siguiente nodo y finalmente elimina la cabeza original (almacenada en el puntero temporal).

```
void clear() {
    while (head) {
        Node<T> *temp = head;
        head = head->next;
        delete temp;
    }
}
```

3.4. Algoritmos útiles

- `getMiddle()` utiliza el método de punteros rápido-lento para encontrar la mitad. Si la lista está vacía, retorna `nullptr`. La condición procura que el puntero rápido no sea nulo o el último.

```
Node<T> *getMiddle(Node<T> *node){
    if (!node)
        return node;
    Node<T> *p1 = node, *p2 = node->next;
    while (p2 && p2->next) {
        p1 = p1->next;
        p2 = p2->next->next;
    }
    return p1;
}
```

- `merge()` combina dos listas ordenadas en una sola lista ordenada. Para lograrlo se utiliza un nodo temporal que sirve como inicio de la nueva lista. Se recorre simultáneamente cada lista, comparando los valores de los nodos actuales y agregando a la nueva lista el nodo con menor valor. Una vez que se termina alguna de las listas, se enlazan los nodos restantes de la otra lista al final. Finalmente, se retorna la cabeza de la lista combinada.

```
Node<T> *merge(Node<T> *list1, Node<T> *list2) {
    if (!list1) return list2;
    if (!list2) return list1;

    Node<T> list;
    Node<T> *temp = &list;
    while (list1 && list2) {
        if (list1->data < list2->data){
            temp->next = list1;
            list1 = list1->next;
        }
        else{
```

```
            temp->next = list2;
            list2 = list2->next;
        }
        temp = temp->next;
    }
    temp->next = (list1) ? list1 : list2;
    return list.next;
}
```

- `mergeSort()` implementa el algoritmo de ordenamiento por mezcla. Recibe la cabeza de una lista y realiza lo siguiente: si la lista tiene cero o un nodo, la retorna tal cual. En caso contrario, se obtiene el nodo medio de la lista y se divide en dos listas: la izquierda (desde la cabeza hasta el medio) y la derecha (desde el siguiente del medio hasta el final). Luego, se aplica recursivamente `mergeSort()` sobre cada mitad y finalmente se combinan ambas listas ordenadas usando `merge()`, retornando la cabeza de la lista completamente ordenada.

```
Node<T> *mergeSort(Node<T> *node) {
    if (!node || !node->next)
        return node;

    Node<T> *middle = getMiddle(node);
    Node<T> *right = middle->next;
    middle->next = nullptr;
    Node<T> *left = node;

    Node<T> *leftSorted = mergeSort(left);
    Node<T> *rightSorted = mergeSort(right);
    return merge(leftSorted, rightSorted);
}
```

- `sort()` ordena la lista completa llamando a `mergeSort()`.

3.5. Reversión

Existen dos implementaciones:

- `reverse1()` recibe la cabeza de una lista y verifica si la lista está vacía o tiene un solo nodo; en ese caso la retorna tal cual. Si hay más de un nodo, invierte recursivamente la lista desde el siguiente nodo y luego hace que el siguiente del nodo actual apunte a este nodo, rompiendo el enlace original. Finalmente retorna la nueva cabeza de la lista invertida.

```
Node<T> *reverse1(Node<T> *node){
        if (!node || !node->next)
            return node;
        Node<T> *reverse = reverse1(node->next);
        node->next->next = node;
        node->next = nullptr;
        return reverse;
}
```

- `reverseEXTRA()` intenta invertir la lista dividiéndola en mitades (similar a merge sort). Esta es mi implementación del algoritmo en alto nivel que propuse en clase, la complejidad es mayor que la que contempla un $O(n \log n)$, porque se itera sobre la lista en cada llamada recursiva, básicamente $O(n \log n) + O(n)$ que es prácticamente $O(n \log n)$ pero requiere más iteraciones. Considero que una manera de optimizar el algoritmo sería guardando la cola en la estructura `Node` o similar; sin embargo, este no se vuelve más eficaz que la manera clásica de invertir una lista simplemente enlazada.
  La idea principal es descomponer la lista en sublistas más pequeñas, invertir cada sublista y luego recombinarlas para obtener la lista completamente invertida.

```
Node<T> *reverseEXTRA(Node<T> *node) {
    if (!node || !node->next)
        return node;
    Node<T> *middle = getMiddle(node);
    Node<T> *right = middle->next;
    middle->next = nullptr;
    Node<T> *left = node;

    Node<T> *leftInverted = reverseEXTRA(left);
    Node<T> *rightInverted = reverseEXTRA(right);

    auto tail = rightInverted;
    while(tail->next != nullptr)
        tail = tail->next;
    tail->next = leftInverted;
    return rightInverted;
}
```

- `void reverse()` invierte la lista llamando a `reverse1()`.
- `void reverseExtra()` invierte la lista llamando a `reverseExtra()`.

3.6. Utilidades

- `print()` recorre la lista imprimiendo cada elemento.

```
void print() {
    Node<T> *p1 = head;
    while (p1) {
        std::cout << p1->data << " ";
        p1 = p1->next;
    }
    delete p1;
}
```