# SIMPLE_LINKED_LIST

**T front(); // Retorna el elemento al comienzo**

1. Verifica si la lista está vacía  comprobando si head es nullptr. Si está vacía, retorna un objeto por defecto del tipo T.
2. Pero si no está vacía, retorna el valor del *“head”*.

**T back(); // Retorna el elemento al final**

1. Verifica si la lista está vacía comprobando si head es nullptr. Si está vacía, retorna un objeto por defecto del tipo T.
2. Si no está vacía. Se crea dinámicamente un nuevo nodo llamado *“temp”* que apunta al inicio de la lista *“head”*.
3. Se entra al bucle para recorrer donde inicia con *“temp”* al inicio de la lista y mientras el *“temp->next”* no sea “*nullptr”*, *“temp”* avanzará, de esa forma saldrá del bucle cuando “temp” sea el último.
4. Cuando salga del bucle ya estará en el último nodo así que solo retorna el valor de ese puntero *“temp”*.

**void push_front(T); // Agrega un elemento al comienzo**

1. Recibe como parámetro un valor *“val”* de tipo T.
2. Se crea dinámicamente un nuevo nodo llamado *“new_nodo”* con el valor *“val”*. Este nodo tiene *next* = *nullptr* por defecto.
3. Cómo se ingresará al comienzo, hacemos que *“temp->next”* apunte a donde estaba apuntando *“head”*, es decir al primer nodo que ya estaba en la lista.
4. Ahora se actualiza el *“head”*, haciendo que apunte al nuevo nodo, porque él es el nuevo primero de la lista.
5. Por último, *“tamanho”* se incrementa.

**void push_back(T); // Agrega un elemento al final**

1. Recibe como parámetro un valor *“val”* de tipo T.
2. Se crea dinámicamente un nuevo nodo llamado *“new_nodo”* con el valor *“val”*. Este nodo tiene *next* = *nullptr* por defecto.
3. Verifica si la lista está vacía, comprobando si *“head”* es nullptr. Si la lista está vacía, el nuevo nodo se convierte en el primer nodo *“head”*. Se retorna inmediatamente, ya que no es necesario recorrer nada más.
4. Si la lista no está vacía, se crea un puntero de tipo nodo llamado *“temp”* que apunta al inicio de la lista es decir al *“head”.*
5. Se entra al bucle para recorrer donde inicia con *“temp”* al inicio de la lista y mientras el *“temp->next”* no sea “*nullptr”*, *“temp”* avanzará, de esa forma saldrá del bucle cuando “temp” sea el último.
6. Cuando salga del bucle ya estará en el último nodo, así que, *“temp->next”* será el *“new_nodo”*, agregándole al final de la lista.
7. Por último, *“tamanho”* se incrementa.

**void pop_front(); // Remueve el elemento al comienzo**

1. Verifica si la lista está vacía, comprobando si *“head”* es nullptr. Si la lista está vacía, no hacemos nada porque no hay nada que borrar.
2. Si la lista no está vacía, se crea un puntero de tipo nodo llamado *“temp”* que apunta al inicio de la lista es decir al *“head”.*
3. Luego movemos el *“head”* al *“temp->next”* que es el siguiente nodo, o sea, el que sigue después del primero.
4. Eliminamos el nodo “temp” que habíamos guardado, porque ya no está en la lista.
5. Por último, *“tamanho”* disminuye.

### void pop_back(); // Remueve el elemento al final

1. Verifica si la lista está vacía, comprobando si *“head”* es nullptr. Si la lista está vacía, no hacemos nada porque no hay nada que borrar.
2. Si la lista tiene sólo un nodo, es decir, *“head->next es nullptr”*. Se elimina el único nodo. Después de eliminar el nodo, *“head”* se pone a nullptr para indicar que la lista está vacía.
3. Si la lista tiene más de un nodo, se crea un puntero de tipo nodo llamado *“temp”* que apunta al inicio de la lista es decir al *“head”.*
4. Se entra al bucle para recorrer donde inicia con *“temp”* al inicio de la lista y mientras el *“temp->next->next”* no sea “*nullptr”*, *“temp”* avanzará, de esa forma saldrá del bucle cuando “temp” sea el penúltimo.
5. Cuando salga del bucle ya estará en el penúltimo nodo, así que, eliminaremos el último nodo con *“ delete temp->next->next”*
6. Después de eliminar el último nodo, el next del penúltimo nodo se pone a null es decir *“temp->next = nullptr”*, indicando que es ahora el último nodo de la lista.
7. Por último, *“tamanho”* disminuye.

### T operator[ ](int); // Retorna el elemento en la posición indicada

1. Se logra con polimorfismo ad-hoc mediante sobrecarga de operadores *“[ ]”*.
2. Recibe como parámetro un valor *“position”* de tipo entero que será la posición a retornar.
3. Se crea un puntero de tipo nodo llamado *“temp”* que apunta al inicio de la lista es decir al *“head”.*
4. Se entra al bucle para recorrer donde inicia con *“temp”* al inicio de la lista con una variable “i” iniciada en 0 y mientras “i < *position”*, *“temp”* avanzará, de esa forma saldrá del bucle cuando “temp” llegue a la posición pedida.
5. Cuando salga del bucle ya estará en el nodo de la posición pedida, devolvemos el valor de ese nodo.

### bool empty(); // Retorna si la lista está vacía o no

1. Verifica si la lista está vacía, comprobando si *“head”* es nullptr. Si la lista está vacía, se retorna *“true”.*
2. Si la lista no está vacía, se retorna *“false”*.

### int size(); // Retorna el tamaño de la lista

1. Verifica si la lista está vacía, comprobando si *“head”* es nullptr. Si la lista está vacía, se retorna *“0”.*
2. Se crea un puntero de tipo nodo llamado *“temp”* que apunta al inicio de la lista es decir al *“head”*.
3. Se crea una variable de tipo entero llamada *“size”* de valor 1.
4. Se entra al bucle para recorrer donde inicia con *“temp”* al inicio de la lista y mientras el *“temp->next”* no sea “*nullptr”*, *“temp”* avanzará y *“size”* se incrementará en *“1”*, de esa forma saldrá del bucle cuando “temp” sea el último.
5. Cuando salga del bucle ya estará en el último nodo y se habrá incrementado el *“size”* por cada nodo.
6. Se retorna el *“size”.*

### void clear(); // Elimina todos los elementos de la lista

1. Verifica si la lista está vacía, comprobando si *“head”* es nullptr. Si la lista está vacía, se retorna vacío.
2. Se crea un puntero de tipo nodo llamado *“temp”* que apunta al inicio de la lista es decir al *“head”*.
3. Se entra al bucle para recorrer donde inicia con *“temp”* al inicio de la lista y mientras el *“temp->next”* no sea “*nullptr”*, *“temp”* avanzará, actualizará el inicio haciendo que *“head = temp->next”*, limpiará la memoria haciendo *“delete temp”* y para que siga avanzando *“temp = head”*, de esa forma saldrá del bucle cuando “temp” sea el último.
4. Cuando salga del bucle ya estará en el último nodo y habrá eliminado el resto, así que solo queda eliminar ese último nodo con *“delete temp”* y qué *“head = nullptr”*.
5. Por último, *“tamanho”* pasa a ser “0”.

### void reverse(); // Revierte la lista

1. Se crea un puntero de tipo nodo llamado *“copia_head”* que apunta al inicio de la lista es decir al *“head”* y será el que vamos a recorrer.
2. Se crea un puntero de tipo nodo llamado *“reverse_head”* que apunta a *“nullptr”*, es decir está vacío y será donde guardaremos la lista en reversa.
3. Se entra al bucle para recorrer donde inicia con *“copia_head”* al inicio de la lista y mientras *“copia_head”* no sea “*nullptr”*, *“copia_head”* avanzará, de esa forma saldrá del bucle cuando *“copia_head”* sea el último.
4. . En cada iteración del bucle, se crea un puntero de tipo nodo llamado *“next”* que apunta a *“copia_head->next”*, esto para no perder la referencia al nodo siguiente y siga avanzando *“copia_head”*
   . Se crea un puntero de tipo nodo llamado *“ingresar”* que apunta al nodo actual de *“copia_head”*, ya que en cada iteración irá avanzando.
   . Lo que se busca es realizar un tipo *“push_back”*a *“reversa_head”*, para es, el nodo que ingresará apuntará al *“head”* de la lista donde estamos haciendo la reversa, es decir *“ingresar->next = reverse_head”*, y actualizamos su head de esa lista de reverse, *“reverse_head = ingresar”.*
   . Avanzo el *“copia_head”* igualando al next que ya habíamos guardado la referencia antes. *“copia_head = next”*.
5. Una vez la lista ya esté en reversa, solo igualamos *“head = reverse_head”*.

**void sort(); // Implementa un algoritmo de ordenación con listas enlazadas**

1. Tenemos la función *“merge”*, donde recibe 2 listas ordenadas con uno o más elementos. Lo que hace es comparar índices desde los menores (en caso sea ascendente) o sea, los de la izquierda.
   Mientras la primera lista tenga sus elementos menores que el primer índice de la segunda lista, o ya no tenga elementos, se ingresará los elementos de la primera lista, caso contrario se ingresará de la segunda lista y así se irá avanzando.
2. También tenemos la función *"merge_sort"*, lo que hace esta función es hacer recursividad para poder dividir entre subconjuntos partiendo a la mitad cada lista hasta llegar a 1 elemento.
3. Para la función *“sort”.* Se crea un objeto *“simple_linked_list<T>”* llamado *“copia”* que será la lista donde se guardará la lista ordenada de forma ascendente.
4. Por último, ese objeto *“copia”* se le igual a *“*this”* para que la lista principal esté ordenada.

**void print(); // Imprimir todos los elementos de la lista (función personal)**

1. Verifica si la lista está vacía, comprobando si *“head”* es nullptr. Si la lista está vacía, se retorna vacío.
2. Se crea un puntero de tipo nodo llamado *“temp”* que apunta al inicio de la lista es decir al *“head”*.
3. Por un tema estético y ordenado hago que imprima *"List: {"*, junto con el valor del primer nodo de la lista *“temp->val”*.
4. Avanza al siguiente nodo con *“temp = temp->next”*.
5. Se entra al bucle para recorrer donde inicia con *“temp”* al inicio de la lista y mientras el *“temp”* no sea “*nullptr”*, *“temp”* avanzará e imprimirá el valor de cada nodo separado por “,” por un tema estético, de esa forma saldrá del bucle cuando “temp” sea el último.