En esta ocasión, repetimos el proceso llevado en la actividad 1.3 pero con un formato completamente diferente. El hecho de usar Linked Lists probaba un reto diferente y nuevo que me emocionaba mucho. Estábamos haciendo básicamente nuestro propio tipo de vector y esto era muy entretenido. Conforme avanzaba me pude dar cuenta de las diferentes dificultades y limitaciones que tenía, por lo que pasaba mucho rato escribiendo funciones que pudieran servir de “work arrounds” a estos problemas. Lo que más intimidaba era el método de ordenamiento que íbamos a utilizar. Cuando usábamos vectores, era muy fácil acceder a secciones específicas de memoria para obtener el dato y sustituir los valores entre dato y dato. Sin embargo, en Linked Lists es necesario recorrer toda la lista hasta llegar al índice que se desea utilizar, lo cual alentaba muchísimo el proceso. Fue entonces que se me ocurrió que podíamos hacer uso de los atributos “next” y “previous” de los nodos como parámetros y recorrer a partir de estos en vez de estar teniendo que recorrer desde la posición 0 hasta la posición *i* en todo momento. La lógica hacía sentido. Podría inclusive hacer nodos temporales con los que recorreremos a partir de *i* posición sin necesidad de modificar nada. Lo que terminó siendo más difícil fue la implementación del código, ya que el hecho de usar tantos apuntadores aumentaba la posibilidad de que cometiera un error y terminará teniendo segmentation faults. Y, esto fue lo que terminó pasando. Claro que, gracias a la ayuda de un compañero y el profesor, fuimos capaces de terminar de implementar esta idea en el código y tener un tiempo de ordenamiento de tan solo 1.6 segundos (lo cual es bastante bueno tomando en cuenta que a mi compañero le tomaba entre 40 y 50 segundos). El metodo de ordenamiento terminó siendo el siguiente:

PRIMERA PARTE:

```
// O(n)
// Funcion que llama al mergeSort con parametros
void LinkedList :: sort() {
    merge(0, size - 1, head, tail);
}

// O(log n)
// Funcion recursiva que realiza el proceso de 'merge'
void LinkedList :: merge(int left, int right, Node* nLeft, Node* nRight) {
    if (left < right) {
        int mid = (left + right) / 2;
        Node* nMid = recorre(nLeft, mid - left);
        merge(left, mid, nLeft, nMid);
        merge(mid + 1, right, nMid->next, nRight);
        mergeSort(left, mid, right, nLeft, nMid, nRight);
    }
}
```

Para esta ocasión utilice el método de ordenamiento “Merge Sort”. Este método hace uso de temporales y de índices para ir agarrando *n* mitades e ir ordenando. El vector no tiene problema con este método ya que puede accesar cualquier posición en cualquier momento. Sin embargo, como podemos ver en la primer iteración, nuestro primer “right” sería el último nodo de la lista, por lo que tendríamos que recorrer los 16 datos existentes. Es por esto que

pasamos como argumento el nodo “head” y “tail”, ya que estos apuntan al inicio y final de la lista. Luego tenemos que hacer un calculo del índice medio. Para obtener este si es necesario recorrer la lista, y este nodo lo obtendremos usando la funcion “recorre” que toma como parametro el nodo de la izquierda y las posiciones que se desea que recorra.

```
// O(n)
// Funcion que recorre posiciones de Nodo
Node* LinkedList :: recorre(Node* nodo, int n) {
    Node* temp = nodo;
    for(int i = 0; i < n; i++) {
        temp = temp->next;
    }
    return temp;
}
```

De esta manera, tenemos el nodo left, mid, y right, permitiéndonos llamar la función de MergeSort con estos tres parámetros.

PARTE 2

```
// O(log n)
// Funcion que agrega los datos a la lista original
void LinkedList :: mergeSort(int left, int mid, int right, Node* nLeft, Node* nMid, Node* nRight) {

    LinkedList L;
    LinkedList R;

    int n = mid - left + 1;
    int m = right - mid;

    Node* tempPos = nLeft;

    for(int i = 0; i < n; i++) {
        L.enQueue(nLeft->data);
        nLeft = nLeft->next;
    }
    for(int i = 0; i < m; i++) {
        R.enQueue(nMid->next->data);
        nMid = nMid->next;
    }
```

Aqui podemos ver entonces el inicio de nuestro mergeSort. Normalmente al entrar a los dos for, hariamos uso de un indice para accesar datos. Este indicie iria aumentando conforme aumenta *i* en ambos for. Sin embargo, esto significaria que tendriamos que recorrer la lista *i* veces cada iterarcion, alentando demasiado nuestro codigo. Es por esto que estoy pasando el nodo temporal que apunta al nodo de la izquierda, ya que podemos hacer uso de “‘->next” y “->data” para ir recorriendo la lista conforme va aumentando el *i*, reduciendo entonces la cantidad de veces que recorremos la lista a solamente una. Entonces, basta con aplicar esta logica al resto del codigo.

PARTE 3

```
while (i < n && j < m) {
    if (L.front() <= R.front()) {
        tempPos->data = L.pop();;
        i++;
    }
    else {
        tempPos->data = R.pop();
        j++;
    }
    tempPos = tempPos->next;
}

while (i < n) {
    tempPos->data =  L.pop();
    tempPos = tempPos->next;
    i++;
}

while (j < m) {
    tempPos->data = R.pop();
    tempPos = tempPos->next;
    j++;
}
```

En esta ultima seccion, haremos uso de nuestra funcion pop. Esta funcion elimina el primer elemento de un Queue y retorna el valor eliminado. De esta forma estariamos “recorriendo” la lista debido que la lista se recorre a la izquierda cada vez que se hace un pop. Esto nos permite evitar usar indices al mismo tiempo que obtenemos el valor de dicha posicion.