Complexity and Algorithm Analysis of three Linked Lists implementations.

Juan Sebastián Rueda Segura

Data Structures
Universidad Nacional de Colombia Sede Bogotá
18/11/2024

Contents

1 Introduction

To access the full code and previous implementations, you can check out this Github repository.

1.1 Goal

This document's objective is to study the implementation of the **LinkedList**, **TailedLinkedList** and **TailedDoublyLinkedList** data structures, analyzing their corresponding algorithms to study the behavior and time complexity using the Big O notation and also, to practice writing academic documents in English. The implementation is presented in $C++$ and the obtained results are graphed on *Python*.

1.2 Motivation

The three previously mentioned data structures arise from the need to create a collection of elements without knowing the number of inputs, since the previous implementation, **List**, needed an integer parameter to define a static array with a maximum size.

This is achieved by using **Nodes**, objects that contain the value to be stored and a pointer to another Node. This *links* all the nodes together and allows us to create this list.

Note that the memory locations are not necessarily contiguous. This could mean an improvement in time, since in an array, the operative system must search for free memory that satisfies the given size and is contiguous. Nevertheless, in this implementation the memory consumption is generally bigger since in an integer array each object will only take 4 bytes but in the Nodes we will need to store the key and the pointer, so it will be who times the needed space or even three times in doubly linked lists as we will see later.

2 Code Analysis

In this section we will analyze each method of the list API, inspecting the corresponding code, making pertinent comments and finally we will give the time complexity in Big O notation.

2.1 Used libraries

```
#include <iostream>
#include <chrono>
using namespace std;
```

We will be using *chrono* to measure the time needed to execute a function, and *iostream*, the input and output library.

```
import matplotlib.pyplot as plt
```

Additionally, we will use *matplotlib* to graph the results obtained from the data analysis.

2.2 Interfaces & Templates

```
//Interface(abstract class) for the List.
template <typename T> class List {
    public:
    virtual void pushFront(T val) = 0;
    virtual void pushBack(T val) = 0;
    virtual void popFront() = 0;
    virtual void popBack() = 0;
    virtual T topFront() = 0;
    virtual T topBack() = 0;
    virtual T find(T key) = 0;
    virtual void erase(T key) = 0;
    virtual bool empty() = 0;
    virtual void addBefore(T key, T newKey) = 0;
    virtual void addAfter(T key, T newKey) = 0;
    virtual void print() = 0;
};
```

$C++$ doesn't have properly defined *interfaces*, but they can be implemented through abstract classes, as the previous code shows, and then the data structures classes inherit from this class. It defines all the methods that have to be implemented and in case one or them is missing in the child class, the code won't compile and will show an error.

Templates or generics are a way to input parameters to functions or classes so that they can support multiple types of variables. This is useful in case we need to create strings or integer lists without having to change the implementation.

2.3 Nodes

2.3.1 Single Node

```
template <typename T>
class Node{
    private:
    T val;
    Node<T>* next;

    public:
    Node(T key){
        val = key;
        next = nullptr;
    }

    T getKey(){
        return val;
    }

    Node<T>* getNext(){
        return next;
    }

    void setKey(T key){
        val = key;
    }

    void setNext(Node<T>* n){
        next = n;
    }
};
```

As we discussed earlier, a node is an object that stores a key and a pointer. In C++ we can´t have an attribute of the same class, so we pass a pointer. We define the constructor method to create a Node, setting next to *nullptr*, meaning that it doesn't point to any other node. Next we define the getters and setters. Of course, this is done in constant time.

Complexity: O(1)

2.3.2 Double Node

```
template <typename T>
```

```
class DoubleNode{
    private:
    T val;
    Node<T>* next;
    Node<T>* prev;

    public:
    Node(T key){
        val = key;
        next = nullptr;
    }

    T getKey(){
        return val;
    }

    Node<T>* getNext(){
        return next;
    }

    Node<T>* getPrev(){
        return prev;
    }

    void setKey(T key){
        val = key;
    }

    void setNext(Node<T>* n){
        next = n;
    }

    void setPrev(Node<T>* p){
        prev = p;
    }
};
```

This class differs from the previous as it has another pointer, *prev* that will point to the previous node on the list. It's initialized as nullptr. This concept will be the key to understand Doubly Linked Lists.

Complexity: O(1)

2.4 Linked List (tailless)

2.4.1 Constructor

```
template <typename T> class LinkedList : public List<T>{
    Node<T>* head;

    public:
    LinkedList(){
```

```
        head = nullptr;
    }

    ~LinkedList(){
        Node<T>* next = head;
        while (next->getNext() != nullptr){
            Node<T>* prev = next;
            next = next->getNext();
            delete prev;
        }
    }
```

This data structure consists of a pointer, *head*, that will point to the first Node on the list. Since C++ doesn't have garbage collection, meaning that if nothing is pointing to the memory address of an object it will be automatically deleted, we implement the $\sim LinkedList()$ method, that will do the job. It will delete all the Nodes on the list.

2.4.2 pushFront()

```
    void pushFront(T val){
        Node<T>* newNode = new Node<T>(val);
        newNode->setNext(head);
        head = newNode;
    }
```

First, we create a new Node as a pointer, and then we set it's next node to the one that the head is pointing to. Finally, we make head point to the newNode. This process is made in constant time.

Complexity: O(1)

2.4.3 pushBack()

```
    void pushBack(T val){
        Node<T>* newNode = new Node<T>(val);
        Node<T>* next = head;
        if (head == nullptr){
            head = newNode;
            return;
        }
        while (next->getNext() != nullptr){
            next = next->getNext();
        }
        next->setNext(newNode);
    }
```

First, we create a new Node as a pointer. If the list is empty, we make the head point to this first node. In other case, we have to traverse the list searching for the node that points to nullptr, meaning that it is the last one, and we set the new node next. Since this method depends on the size of the list, it requires linear time to complete.

Complexity: O(n)

2.4.4 popFront()

```
1    void popFront(){
2        if (head == nullptr){
3            cout << "Error: Lista vacia. No se puede eliminar el
     elemento." << endl;
4            return;
5        }
6        Node<T>* next = head;
7        head = next->getNext();
8        delete next;
9    }
```

First, we see if the list is empty, checking that head points to nullptr. In other case, we go to the node pointed by head and take it's next node. Then, we make head point to it, making it the first node on the list. Finally, we manually delete the node that was on the first position. This is done in constant time.

Complexity: O(1)

2.4.5 popBack()

```
1    void popBack(){
2        if (head == nullptr){
3            cout << "Error: Lista vacia. No se puede eliminar el
     elemento." << endl;
4            return;
5        }
6        else if (head->getNext() == nullptr){
7            delete head;
8            head = nullptr;
9            return;
10       }
11       Node<T>* next = head;
12       Node<T>* prev = nullptr;
13       while (next->getNext() != nullptr){
14           prev = next;
15           next = next->getNext();
16       }
17       prev->setNext(nullptr);
18       delete next;
```

```
19      }
```

First, we see if the list is empty, checking that head points to nullptr. In other case, we search for the node that points to the last one, going through all the list. When we find it we set nullptr next to it and delete the last node manually.

Complexity: O(n)

2.4.6 topFront()

```
1       T topFront(){
2           return head->getKey();
3       }
```

We return the key of the node that head is pointing to.

Complexity: O(1)

2.4.7 topBack()

```
1       T topBack(){
2           Node<T>* next = head;
3           while (next->getNext() != nullptr){
4               next = next->getNext();
5           }
6           return next->getKey();
7       }
```

We return the key of the node that the last node is pointing to, so as we previously saw, we need to traverse the list to find it.

Complexity: O(n)

2.4.8 find()

```
1       int find(T key){
2           if (head == nullptr){
3               cout << "Error: Elemento no encontrado." << endl;
4               return -1;
5           }
6           Node<T>* next = head;
7           int i = 0;
8           while (next->getNext() != nullptr){
9               if (next->getKey() == key){
10                  return i;
```

```
            }
            next = next->getNext();
            i++;
        }
        return -1;
    }
```

We first check if the list is empty and in that case we return −1. In other case, we traverse the list checking if the nodes keys is equal to the parameter key and in that case we return the "index" where it is located. Note that if an element is repeated it will only return the position where it first occurs. In case the element is not on the list, we return −1.

Complexity: O(n)

2.4.9 erase()

```
    void erase(T key){
        if (head == nullptr){
            return;
        }
        int n = find(key);
        if (n == -1){
            cout << "Error: Elemento no encontrado." << endl;
        }
        Node<T>* next = head;
        Node<T>* prev = nullptr;
        for (int i = 0; i < n; i++){
            prev = next;
            next = next->getNext();
        }
        prev->setNext(next->getNext());
        delete next;
    }
```

We first check if the list is empty. In other case, we call the find method to get the node position and then we iterate until we are on the node that points to the one we want to erase and keep track of the previous node. Finally we make the previous node point to the one that was after and we delete the node.

Complexity: O(n)

2.4.10 isEmpty()

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

We check if the list is empty as we saw on the previous methods.

Complexity: O(1)

2.4.11 addBefore()

```
    void addBefore(T key, T newKey){
        if (head == nullptr){
            cout << "Error: Elemento no encontrado." << endl;
            return;
        }
        int n = find(key);
        if (n == -1){
            cout << "Error: Elemento no encontrado." << endl;
        }
        Node<T>* next = head;
        Node<T>* prev = nullptr;
        for (int i = 0; i < n; i++){
            prev = next;
            next = next->getNext();
        }
        Node<T>* newNode = new Node<T>(newKey);
        prev->setNext(newNode);
        newNode->setNext(next);
    }
```

We first check if the list is empty as we saw on the previous methods and call the find method. Then, we use the same process as the erase method, but we create a new node and we set the previous node to point to the new node and then this node to the one we wanted to.

Complexity: O(n)

2.4.12 addAfter()

```
    void addAfter(T key, T newKey){
        if (head == nullptr){
            cout << "Error: Elemento no encontrado." << endl;
            return;
        }
        int n = find(key);
        if (n == -1){
            cout << "Error: Elemento no encontrado." << endl;
        }
        Node<T>* next = head;
        for (int i = 0; i < n; i++){
            next = next->getNext();
        }
        Node<T>* newNode = new Node<T>(newKey);
        newNode->setNext(next->getNext());
```

```
16          next->setNext(newNode);
17      }
```

We first check if the list is empty as we saw on the previous methods and call the find method. Then, we iterate until we have the desired node and create a new node. We make the new node point to the one that points the searched node and then this node to point the new node.

Complexity: O(n)

2.4.13 print()

```
1       void print(){
2           cout << "Elementos en la lista enlazada: [";
3           Node<T>* next = head;
4           while (next != nullptr){
5               if (next->getNext() == nullptr){
6                   cout <<next->getKey();
7                   break;
8               }
9               else {
10                  cout <<next->getKey()<<" ";
11              }
12              next = next->getNext();
13          }
14          cout <<"]"<<endl;
15      }
16  };
```

We first check if the list is empty as we saw on the previous. Then, we iterate until the final node points to nullptr, printing the keys each iteration.

Complexity: O(n)

2.5 Linked List (with tail)

We will only check the methods that change with this implementation and assume the ones that aren't mentioned to be the same.

2.5.1 Constructor

```
1   template <typename T> class TailedLinkedList: public List<T>{
2       Node<T>* head;
3       Node<T>* tail;
4
```

```
5     public:
6     TailedLinkedList(){
7         head = nullptr;
8         tail = nullptr;
9     }
10
11    ~TailedLinkedList(){
12        Node<T>* next = head;
13        while (next->getNext() != nullptr){
14            Node<T>* prev = next;
15            next = next->getNext();
16            delete prev;
17        }
18    }
```

It's the same concept as the previous one, but we define the pointer *tail*, whose purpose will be to point to the last node on the list. As we will see later, this affects positively some complexities of previous methods.

2.5.2 pushFront()

```
1     void pushFront(T val){
2         Node<T>* newNode = new Node<T>(val);
3         newNode->setNext(head);
4         head = newNode;
5         if (tail == nullptr){
6             tail = head;
7         }
8     }
```

It's the same concept but in case the list is empty, we assign tail to point to it.
Complexity: O(1)

2.5.3 pushBack()

```
1     void pushBack(T val){
2         Node<T>* newNode = new Node<T>(val);
3         if (head == nullptr){
4             head = newNode;
5             tail = newNode;
6             return;
7         }
8         tail->setNext(newNode);
9         tail = newNode;
10    }
```

We check if the list is empty and in that case we make head and tail both point to the new node. In other case, we access the last node, which is the one being

pointed by tail, and then we set next the new node and make tail point to it. Note that previously this method required lineal time but this is done en constant time.

Complexity: O(1)

2.5.4 popFront()

```
1    void popFront(){
2        if (head == nullptr){
3            cout << "Error: Lista vacia. No se puede eliminar el
     elemento." << endl;
4            return;
5        }
6        Node<T>* next = head;
7        head = next->getNext();
8        if (head == nullptr){
9            tail = nullptr;
10       }
11       delete next;
12   }
```

This is similar to the last implementation with the only thing to note is that if the list only has an element and we remove it, we set tail to nullptr, since the list is now empty.

Complexity: O(1)

2.5.5 popBack()

```
1    void popBack(){
2        if (head == nullptr){
3            cout << "Error: Lista vacia. No se puede eliminar el
     elemento." << endl;
4            return;
5        }
6        else if (head == tail){
7            head = nullptr;
8            tail = nullptr;
9            return;
10       }
11       Node<T>* next = head;
12       Node<T>* prev = nullptr;
13       while (next->getNext() != nullptr){
14           prev = next;
15           next = next->getNext();
16       }
17       prev->setNext(nullptr);
18       tail = prev;
19       delete next;
```

```
20      }
```

As before, this is similar to the last implementation with the only thing to note is that if the list only has an element and we remove it, we set tail to nullptr, since the list is now empty. We would think that having a tail would make this method constant, but the thing is that we don't have the node before the last one, so that we could make tail point to it. That's why we iterate until we find it and remove the last one. Doubly linked lists will help with this method as we will see later.

Complexity: O(n)

2.5.6 topBack()

```
1       T topBack(){
2           return tail->getKey();
3       }
```

This method now runs in constant time, as we now have the tail to access to the last element on the list.

Complexity: O(1)

2.6 Doubly Linked List (with tail)

As said earlier, we will only discuss the methods of the API that change with this implementation. Nevertheless, since the double nodes have another pointer, the methods that only change the reference of this pointer won't be included either.

2.6.1 Constructor

```
1   template <typename T> class TailedDoublyLinkedList: public List<T>{
2       DoubleNode<T>* head;
3       DoubleNode<T>* tail;
4
5       public:
6       DoublyLinkedList(){
7           head = nullptr;
8           tail = nullptr;
9       }
10
11      ~DoublyLinkedList(){
12          DoubleNode<T>* next = head;
```

```
        while (next->getNext() != nullptr){
            DoubleNode<T>* prev = next;
            next = next->getNext();
            delete prev;
        }
    }
```

Exactly the same as the single linked list, but the difference is that we are using DoubleNodes.

2.6.2 pushBack()

```
    void pushBack(T val){
        DoubleNode<T>* newNode = new DoubleNode<T>(val);
        if (head == nullptr){
            head = newNode;
            tail = newNode;
            return;
        }
        tail->setNext(newNode);
        newNode->setPrev(tail);
        tail = newNode;
    }
```

Quite similar, with the only difference that we have to set the new node previous to point to the one that was on the tail and this one's next to point to the new node. Finally make tail point to the new node.

Complexity: O(1)

2.6.3 popBack()

```
    void popBack(){
        if (head == nullptr){
            cout << "Error: Lista vacia. No se puede eliminar el
    elemento." << endl;
            return;
        }
        else if (head == tail){
            head = nullptr;
            tail = nullptr;
            return;
        }
        Node* prev = tail->getPrev();
        prev->setNext(nullptr);
        delete tail;
        tail = prev;
    }
```

This method reduces the complexity, because we only need to access the previous node of the node that the tail is pointing to. The use of Double Nodes allows this in constant time.

Complexity: O(1)

3 Time measurements

In this section we will compare the difference between the three implementations of the previously mentioned data structures and will see if the data corresponds to the code analysis.

3.1 Code Analysis

```
int main(){
    LinkedList<int>* lista = new LinkedList<int>();
    int sizes[5] = {100, 1000, 10000, 50000, 100000};
    double time_per_size[5];
    double time_per_size_pop[5];

    for (int i = 0; i < 5; i++){
        auto start = chrono::high_resolution_clock::now();
        for (int j = 0; j < sizes[i]; j++){
            lista->pushFront(j);
        }
        auto end = chrono::high_resolution_clock::now();
        auto duration = chrono::duration_cast<chrono::microseconds
    >((end - start));
        time_per_size[i] = (double)duration.count() / sizes[i];
        //cout << "Time for size " << sizes[i] << ": " <<
    time_per_size[i] << " microseconds" << endl;

        start = chrono::high_resolution_clock::now();
        for (int j = 0; j < sizes[i]; j++){
            lista->popFront();
        }
        end = chrono::high_resolution_clock::now();
        duration = chrono::duration_cast<chrono::microseconds>((end
     - start));
        time_per_size_pop[i] = (double)duration.count()/ sizes[i];
        cout << "Time for pop size " << sizes[i] << ": " <<
    time_per_size_pop[i] << " microseconds" << endl;
    }
    cout << "Time per size: [" << time_per_size[0] << ", " <<
    time_per_size[1] << ", " << time_per_size[2] << ", " <<
    time_per_size[3] << ", " << time_per_size[4] <<"]" << endl;
    cout << "Time per size pop: [" << time_per_size_pop[0] << ", "
    << time_per_size_pop[1] << ", " << time_per_size_pop[2] << ", "
     << time_per_size_pop[3] << ", " << time_per_size_pop[4]<< "]"
    << endl;
}
```

This code tests the average insertion time per operation, which is calculated by $\frac{end_{time} - start_{time}}{size}$. Initializes an array with different sizes and a List. This will change during the tests. Performs an insertion and an elimination method, and stores the average time on an array.

```
int main(){
import matplotlib.pyplot as plt

if __name__ == "__main__":
    lista_size: List[int] = [100, 1000, 10000, 50000, 100000]
    times: List[float] = [0, 5.999, 50.0036, 255.95, 527.045]
    plt.plot(lista_size, times)
    plt.title("LinkedList popBack")
    plt.xlabel("Size")
    plt.ylabel("Time")
    plt.show()
```

Once we have the results, we set the *times* list to the values obtained by the previous code and then we graph using the matplotlib library, in Python.

3.2 Obtained results & graphs

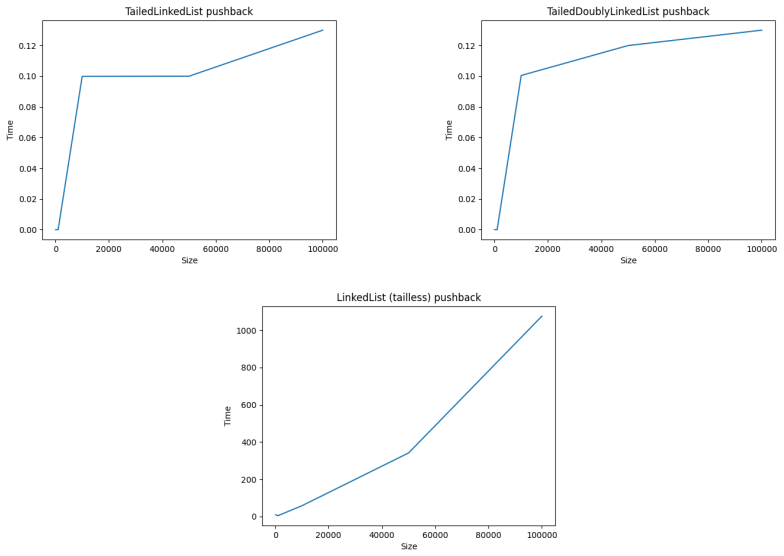

Figure 1: pushBack results

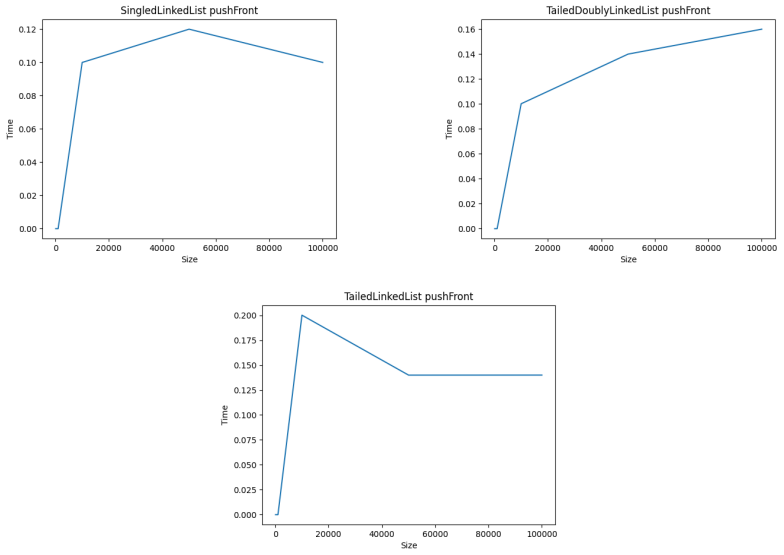

Figure 2: pushFront results

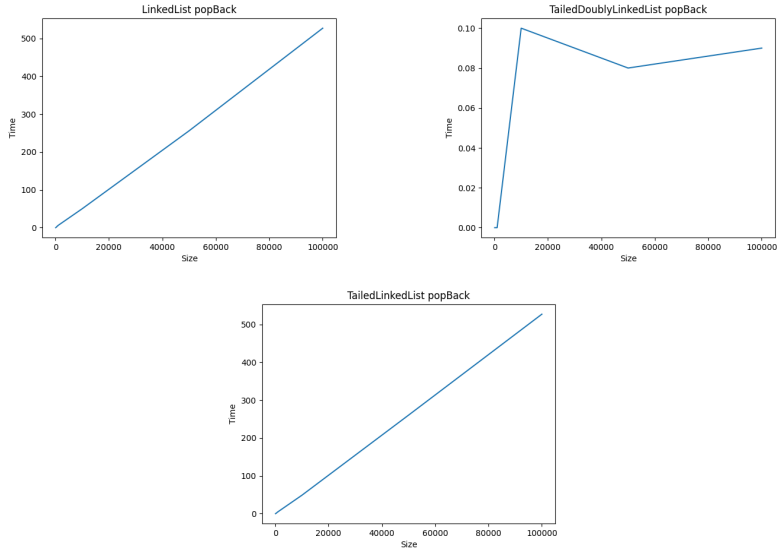

Figure 3: popBack results

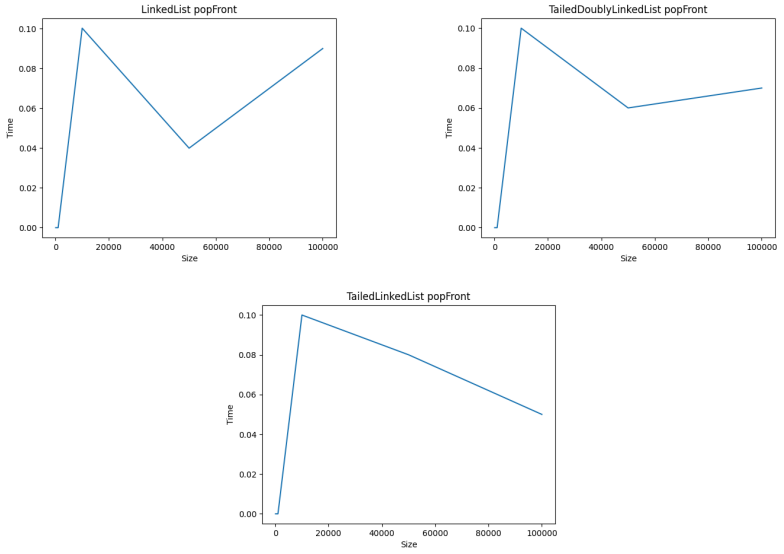

Figure 4: popFront results

As the code analysis showed, the pushBack when a tail is implemented is performed in constant time, but without it, it becomes lineal. The pushFront is always constant, since the head makes it possible. The popBack method

requires lineal time in the Single Node implementations and constant in the doubly linked list. popFront is always constant. These four methods are the most important and the ones that change from one implementation to another.

4 Conclusions

This document summarized all the implementations of the linked list data structures, analyzing the code, time complexity and obtained results given by testing the code and plotting the time vs size graphics. One thing to note is the faster results given by implementing these data structures in C++, because the array based list implementation, that was made in Python, produced longer waiting times, since it's an interpreted language. This is why, in general, most of the data structures are implemented in languages like C or C++.

The methods addBefore and addAfter could be executed in constant time, in doubly linked lists, if the find method returned the Node containing the value, but to me passing a Node as a parameter didn't make sense. If a program needed to do these operations it must be changed though.

Finally, one last thing that could be implemented and would give more results and conclusions is a way to measure the memory consumption, to see the differences between using Nodes, DoubleNodes and using an array list implementation and check if it really affects the needed amount of memory.