KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

REPORT

PERSONAL HOMEWORK 2

UNIVERSITY OF SCIENCE

VIETNAM NATIONAL UNIVERSITY

HO CHI MINH CITY

INFORMATION TECHNOLOGY

Lecturers:

- Châu Thành Đức
- Phan Thị Phương Uyên
- Ngô Đình Hy
- Phan Thị Phương

Student:

- Nguyễn Đức Huy -19127422

Progress:

- Finish all the task in Personal Homework 2

CONTENT

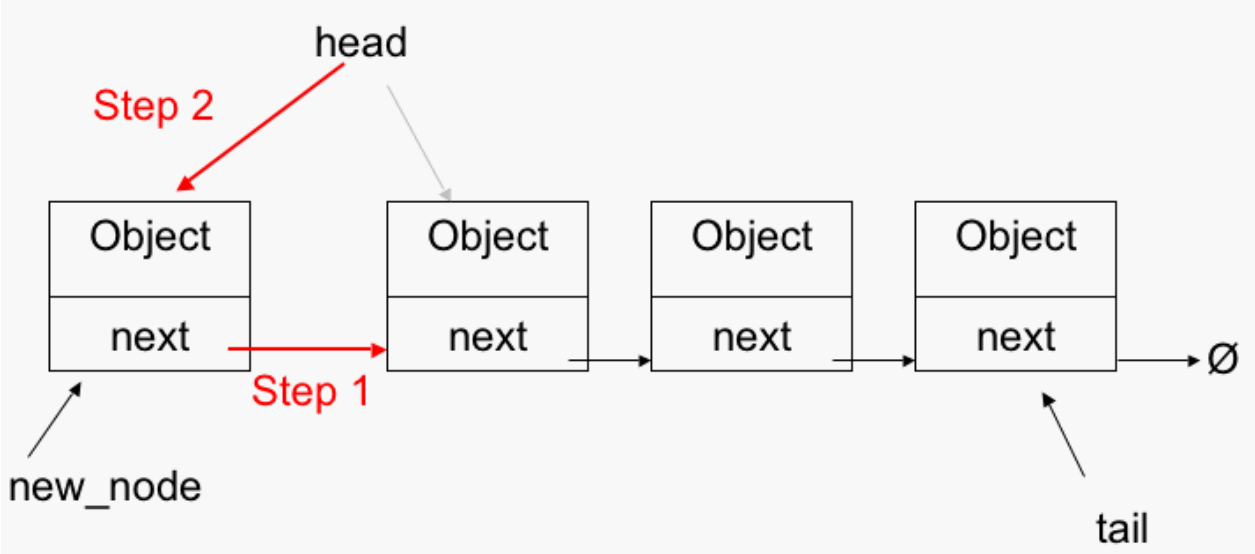
Personal Homework 2
Linked List – Stack – Queue

1 Analyze thecomplexity

1.1 Linked list

1.1.1 InsertHead

In Theory:



In this function that you want to insert new node to the head node. The first step, the node you need to insert contain the next node which is pointed to the head. And the second step, you must be update the head node after you add new node to head node, the head node now is the node which you inserted. After two step, a new node is inserted into the linked list so let consider each step. Both first step and second step it's just use few assignments to point the next node to the head and update the head so there are two instructions in this case. Therefore, the complexity of this case is O(1).

InsertHead

In Experiment:

First of all I show the basic implementation of Linked list:

```
struct List{
  NODE* p_head;
  NODE* p_tail;
};
```

```
struct NODE {
    int data;
    NODE* p_next;
};
```

For this function insertion, the linked list will insert a new node in the head of the linked list. As you can see, this function always take **5 assignments**(3 for create a new node and 2 for update the node) and **2 comparisons** (1 for check if it's NULL and 1 for check if Head is NULL) so that the complexity for insertion of the new node after a node is given is O(1). To prove that, in the below photo, I run 100 times to count how many assignments and comparison in reality when you run this function

```
bool InsertHead(List*& L, int data)
{
    // create a new node
    NODE* p = new NODE;

    if (p == NULL) {
        return false;
    }

    p->data = data;
    p->p_next = NULL;

    if (L->p_head == NULL) {
        L->p_head = L->p_tail = p;
    }
    else {
        // update next link and head node
        p->p_next = L->p_head;
        L->p_head = p;
    }
    return true;
}
```

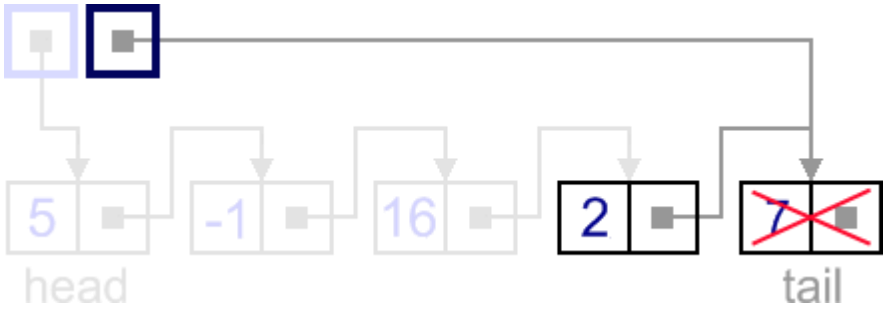
Microsoft Visual Studio Debug Console

```
0 16 5 9 15 9 13 7 19 3 8 11 18 18 5 14 11 9 0 1 7 0 17 10 6 16 7 15 2 2 16 2 5 1 17 6 14 4 12 9 3 15 4 11 10 0 9 2 1 9
1 10 3 5 15 6 15 0 9 15 11 7 15 5 9 19 2 9 13 16 13 6 8 1 7 10 7 6 1 6 2 19 19 0 2 15 10 9 5 3 15 11 18 0 2 5 17 14 19 9
99 5 15 19 5 9 1 11 5 17 17 5 17 7 18 1 6 0 5 4 15 6 15 13 11 15 5 5 17 5 6 14 17 18 16 11 13 19 10 19 6 15 4 3 5 9 11 1
3 2 5 11 18 6 18 2 10 14 6 10 15 7 14 6 15 8 14 17 2 4 11 4 11 13 9 1 9 4 7 19 5 2 16 4 3 10 5 1 18 4 9 1 16 13 4 16 13
8 15 9 2 1 10 10 6 4 10 19 12 7 14 5 11 16 13 1 17 17 14 4 14 5 3 8 1 16 10 8 12 9 13 0 13 7 3 0 15 9 4 6 18 11 5 2 18 1
0 15 1 14 5 10 11 16 5 17 9 7 2 12 7 1 11 1 3 10 15 7 18 16 17 8 11 13 13 1 13 14 13 3 12 5 3 0 19 17 3 6 0 4 3 16 8 10
5 19 13 5 9 10 1 16 0 8 13 7 0 13 10 3 0 12 14 10 19 9 7 18 8 15 11 10 1 19 15 13 0 6 10 10 17 19 8 19 9 16 16 18 11 10
6 1 13 19 0 17 13 8 7 13 16 4 999 4 16 6 12 1 8 12 2 2 17 19 15 16 9 18 19 9 9 13 16 11 8 10 12 5 14 16 14 12 16 11 12 1
1 6 0 12 9 1 17 4 10 2 2 5 4 14 13 6 11 19 11 14 15 16 0 2 18 17 17 7 17 17 16 0 2 7 13 2 19 10 18 0 5 13 11 14 16 10 11
1 6 18 5 13 18 13 3 5 18 6 9 6 14 10 17 15 15 17 17 4 4 13 1 14 0 14 2 17 16 18 12 15 6 17 19 12 15 15 19 6 9 13 10 4 1
6 2 8 17 1 3 14 6 7 17 16 13 14 1 3 11 17 4 19 4 4 8 4 4 18 19 8 13 7 10 0 16 17 5 14 12 1 9 19 13 4 14 4 17 7 18 18 19
6 17 6 1 7 7 14 12 19 15 9 9 5 14 5 15 16 11 11 6 9 15 3 9 0 4 1 15 3 14 2 2 3 6 3 2 4 6 19 5 7 16 15 15 10 19 15 16 2 0
15 11 4 17 7 5 14 3 18 15 7 19 5 4 1 9 15 14 6 5 4 11 9 3 6 19 2 999 2 0 16 1 0 13 16 7 8 19 18 10 7 13 0 10 10 13 4 11
12 10 14 5 13 13 13 9 2 2 7 14 4 6 18 8 15 11 1 0 18 19 15 0 0 16 19 0 16 3 3 17 13 16 5 13 0 9 3 0 11 19 2 1 8 1 7 4 1
3 9 2 4 10 9 5 14 12 11 6 16 14 8 4 14 0 4 19 15 6 10 13 9 4 6 3 9 11 11 19 14 10 9 9 7 1 0 8 17 17 17 19 9 4 8 11 6 8
11 14 7 10 14 12 7 3 16 17 17 18 17 19 18 19 4 6 5 5 3 17 13 15 12 7 9 11 2 10 0 14 9 14 6 6 3 18 19 6 15 14 15 0 18 4 5
4 11 16 8 16 18 1 3 1 13 6 7 5 3 14 6 11 2 0 2 6 11 1 16 10 0 14 2 15 5 1 8 19 1 15 6 18 3 11 1 12 16 5 11 4 19 10 15 1
0 8 11 9 10 17 16 6 2 18 10 4 0 19 6 5 5 7 16 17 9 19 16 4 5 16 4 16 16 13 18 11 1 16 3 13 999 9 7 8 5 12 6 12 4 10 7 3
10 18 1 10 19 3 17 2 1 2 5 1 12 3 6 2 6 6 5 1 1 12 13 19 13 10 11 11 10 19 6 5 1 14 14 5 2 18 6 19 3 3 9 2 6 15 13 15 8
19 13 8 17 16 1 0 18 13 6 8 10 1 16 19 12 13 14 10 5 0 1 2 17 14 17 14 15 10 7 8 13 6 16 4 12 14 0 15 16 1 19 9 4 5 12 1
14 10 11 11 4 5 12 0 9 1 12 5 17 2 15 2 14 16 11 5 0 17 4 5 4 14 17 13 13 15 6 11 6 3 2 11 11 11 3 8 10 5 12 0 9 0 6 7
17 16 2 0 0 0 13 17 4 4 5 12 19 15 16 6 10 14 14 11 14 5 11 2 19 10 5 14 7 15 4 13 7 12 12 12 8 2 7 8 5 18 14 11 7 5 11
13 17 19 13 5 17 6 0 15 2 16 4 16 0 1 0 0 9 6 13 10 8 0 13 4 12 2 5 17 17 0 18 5 15 8 3 12 12 11 1 14 6 11 1 12 2 16

After 100 Times:
Comparisons: 200  Assignments: 500
Average: 7
D:\Data structure and algorithm\hw2\Debug\hw2.exe (process 23572) exited with code 0.
Press any key to close this window . . .
```

1.1.2 removeTail

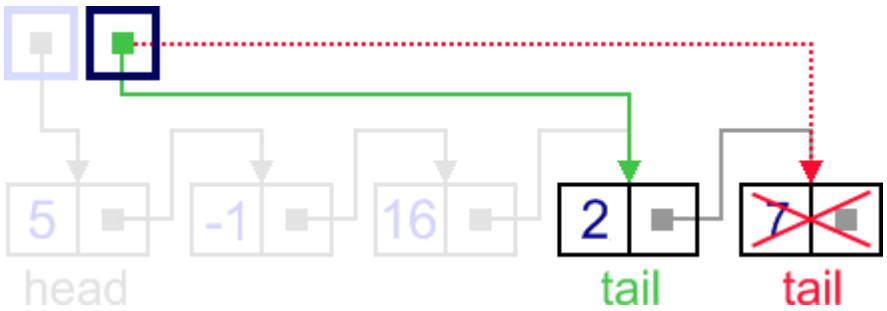
In Theory:

In this case, last node (current tail node) is removed from the list. This operation is a bit more tricky, than removing the first node, because algorithm should find a node, which is previous to the tail first.

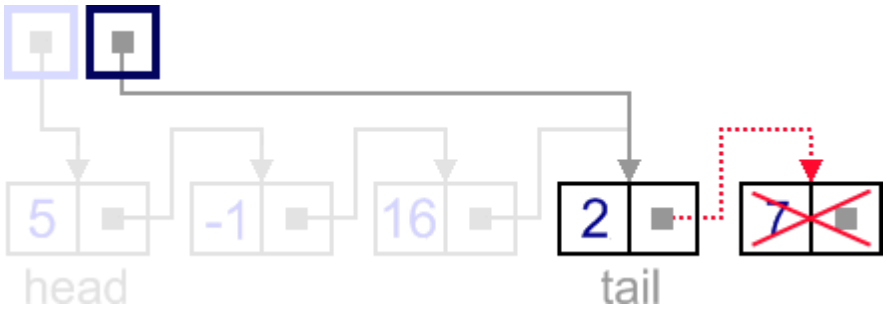


It can be done in three steps:

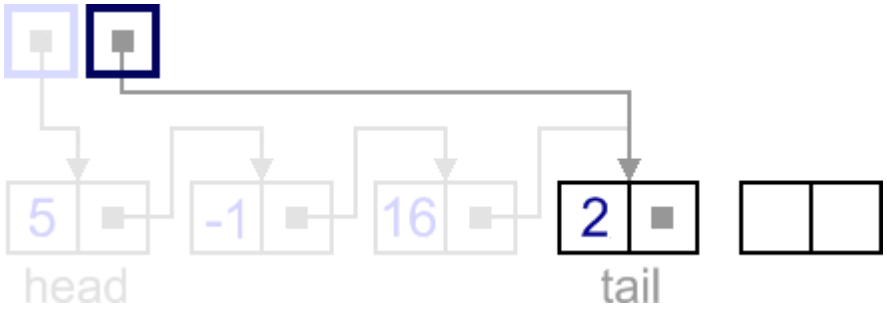
The first step: Update tail link to point to the node, before the tail. In order to find it, list should be traversed first, beginning from the head.



The second step: Set next link of the new tail to NULL



The third step: Dispose removed node



To conclude the complexity of this algorithm, let consider 3 step. The first step you need to update the tail link to point to the node previous tail node but the singly linked list is a data structure has 1 direction to traversal so that you must be traversal from the head to node which is previous before the tail node, to do this you must be traversal all the node in the linked list(if the number of node in the linked list is n, you must be traversal n – 1). And after you find the previous node before the tail node you just update this node is the tail node in the moment. In the second step, you need to point the next node in the tail node you updated to the NULL, it just take 1 assignment to do this. In the last step, you just delete the node that you separate from the linked list to avoid leak of memory.

Therefore, the first step need you to traversal almost the node in the linked list right and the second, the third step It's take few assignments to do this so that the complexity depend on the first step and that is O(n)

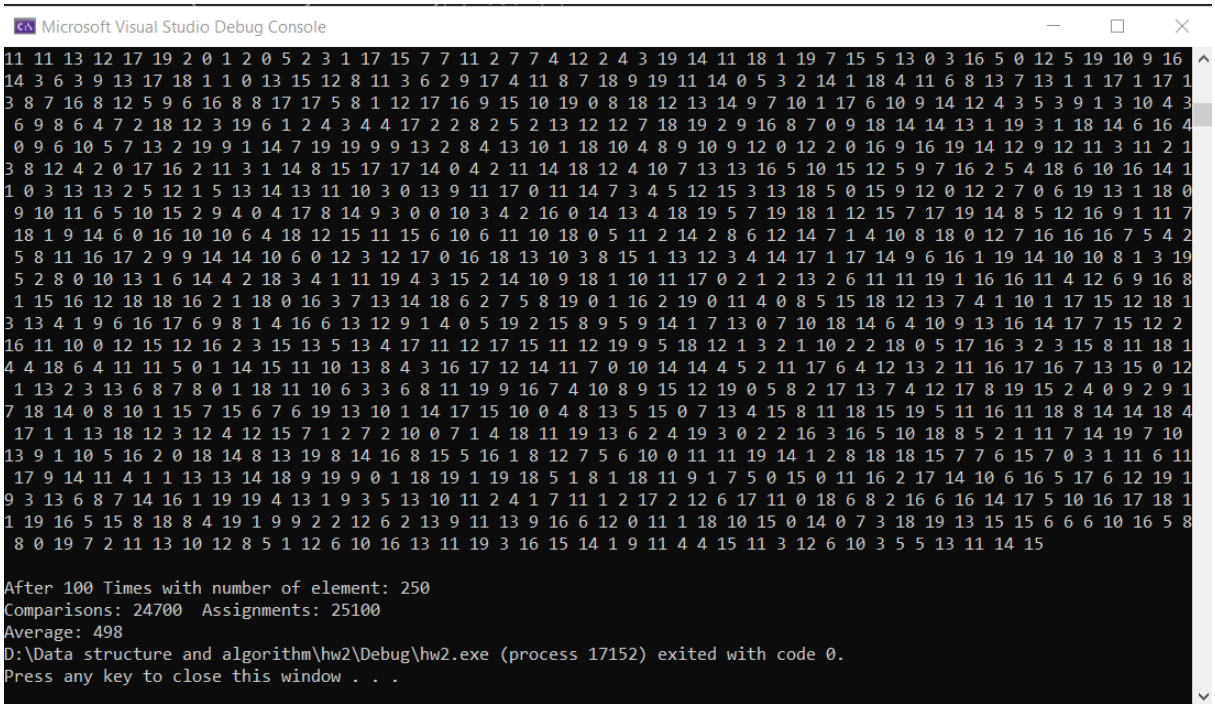
removeTail

```
void removeTail(List*& L) {

    NODE* traversal = L->p_head;

    //find the node previous tail node
    while (traversal->p_next != L->p_tail) {
        traversal = traversal->p_next;
    }

    NODE* temp = traversal->p_next;

    traversal->p_next = NULL;
    L->p_tail = traversal;

    delete temp;

}
```

In Experiment:

For this function, it take:
Assignments: 4 + (n – 2)
Comparisons: n – 1
So that the sum instructions of this function is 2n + 1. Therefore, the complexity of this function is O(n)

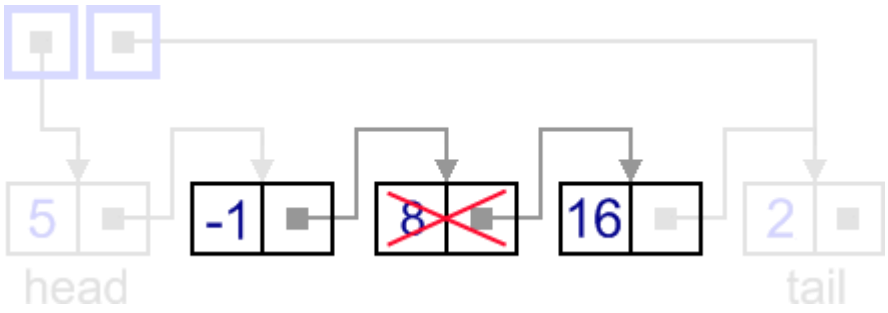
In the photo below to prove that in the reality experiment, I run this function with 100 times with 250 elements and sum of instructions are 498 (it may be lost 3 instructions from 2n+1 because I move variable count_comparison inside While loop so that maybe it checked but it did not go into the loop



1.1.3 RemoveAtPosition

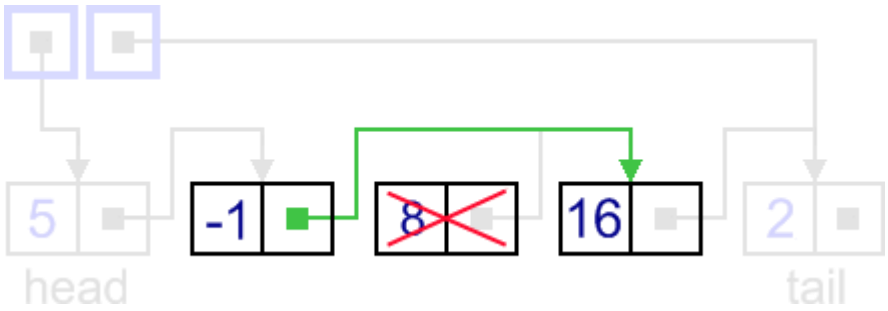
In Theory:

In general case, node to be removed is always located between two list nodes. Head and tail links are not updated in this case.

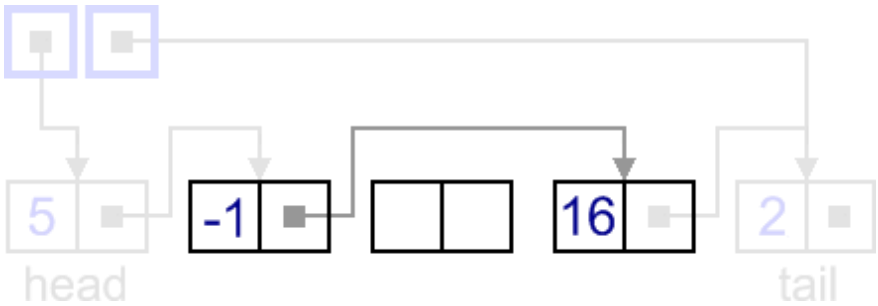


Such a removal can be done in two steps:

The first step: Update next link of the previous node, to point to the next node, relative to the removed node.



The second step: Dispose removed node.



To conclude this function (remove at this position). Let consider two steps above, the first step is you need to update the node which is previous before the node you want to delete, update this next node to the next node of this node you deleted. Because the singly linked list has one direction to traversal so that you have no idea to delete if you know position to delete. To do the first step you must traversal from the head node to point to the previous the node you want to delete. After you find the previous node you just update the pointer of the next node in this node to next node which is next from the node you delete. The second step you need to delete the node you separate link in the linked list to avoid leak of memory.

Therefore, the complexity of this function depend on the position of the node you want to delete. The worst case is if you want to delete the node before the tail or tail node because to do this you must be traversal almost the node in the linked list so that in the worst case the complexity of this function is O(n).

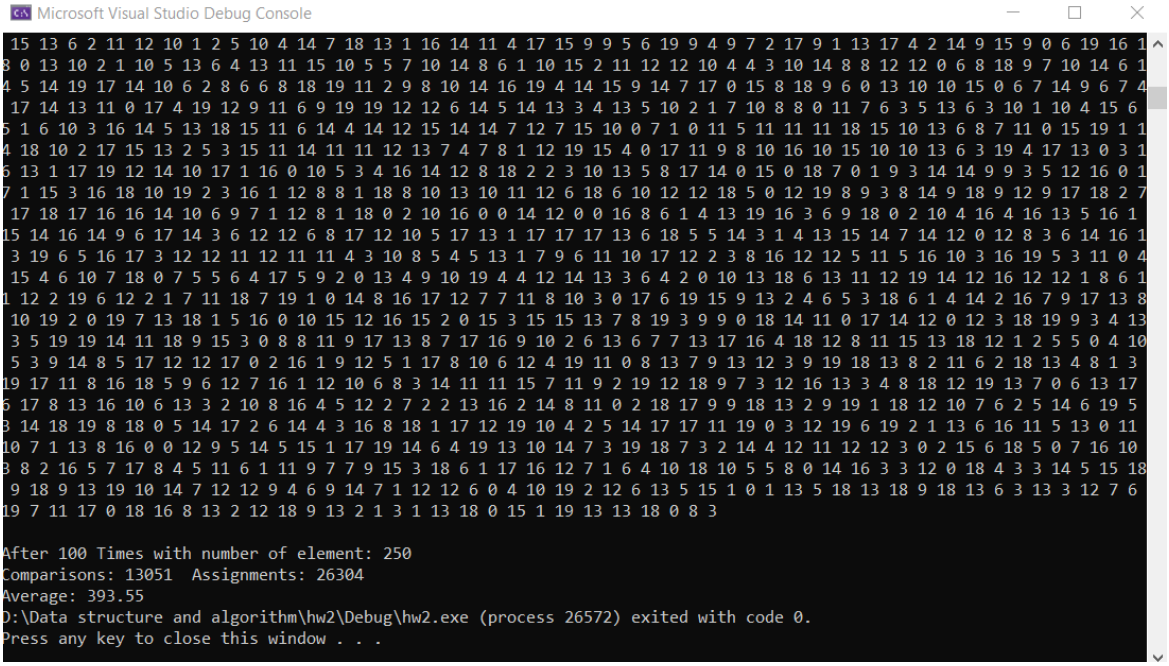
RemoveAtPosition

In Experiment:

In this function (RemoveAtPosition), the worst case in this function when we delete the last node. In this case, it take: Comparisons: 1 + (n - 1) + 1 = n + 1
Assignments: 2 + 2*(n - 1) +2+1 = 2n + 3 Sum of this it take 3n + 4 instructions to run in this case. Therefore, the complexity of this function is O(n)

The photo in the below prove that the complexity of this function is O(n). The average instructions in the photo is different with 2n+3 because I run this function to delete random position in the linked list

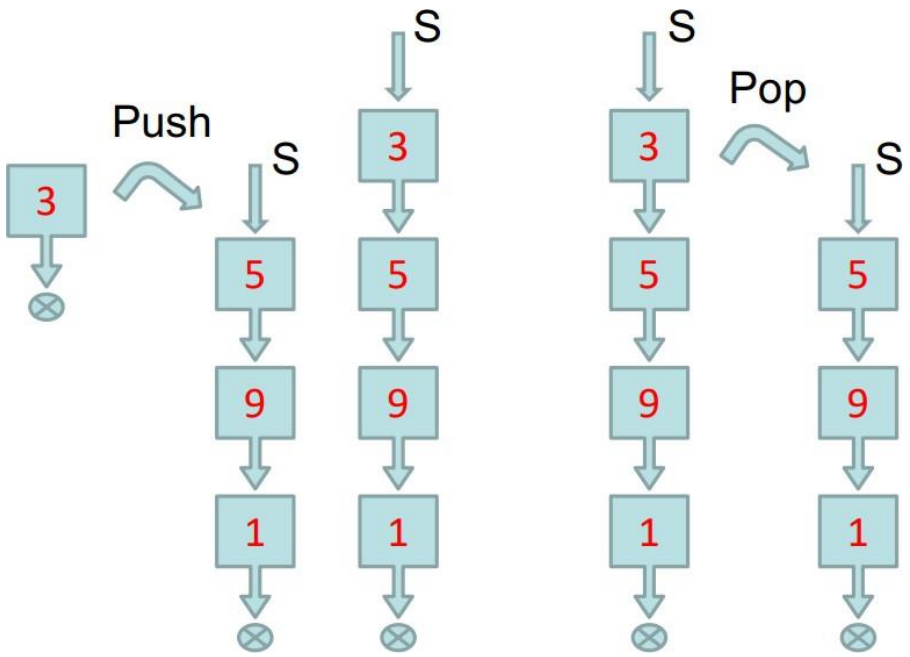
```
void RemoveAtPosition(List*& L, int pos) {

    NODE* p = L->p_head;
    int i = 1;

    if (pos == 1) {
        NODE* q = L->p_head;
        L->p_head = q->p_next;

        delete q;
        return;
    }

    while (i < pos - 1) {
        p = p->p_next;
        i++;
    }

    NODE* q = p->p_next;
    p->p_next = q->p_next;

    if (q == L->p_tail) {
        L->p_tail = p;
    }

    delete q;

}
```



Compare with array when you use array and add new element to the array. Both Linked List and Array are used to store linear data of similar type, but an array consumes contiguous memory locations allocated at compile time, i.e. at the time of declaration of array, while for a linked list, memory is assigned as and when data is added to it, which means at runtime. This is the basic and the most important difference between a linked list and an array. Because In an array, elements are stored in contiguous memory location or consecutive manner in the memory so that Insertion and Deletion operation takes more time, as the memory locations are consecutive and fixed. And when the array want to add new element or remove element, it must be shift every single element to create or reduce 1 element. While in the linked list, a new element is stored at the first free and available memory location, with only a single overhead step of storing the address of memory location in the previous node of linked list. It just few step to update the next node and update head node or tail node in the linked list. Therefore, the performance of these operations in Linked lists is fast. But the requirement of memory is less due to actual data being stored within the index in the array. As against, there is a need for more memory in Linked Lists due to storage of additional next and previous referencing elements. To conclusion, depend on what you want to optimize speed priority or save memory instead.

1.2 Stack

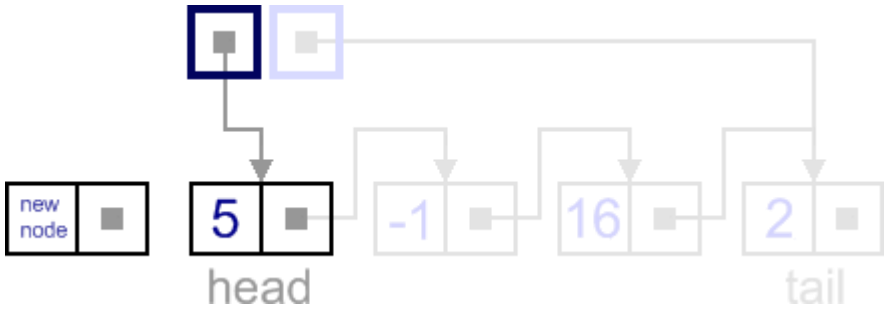
Use a data structure like this to implement Stack function: (use the linked list to implementation)



1.2.1 Push

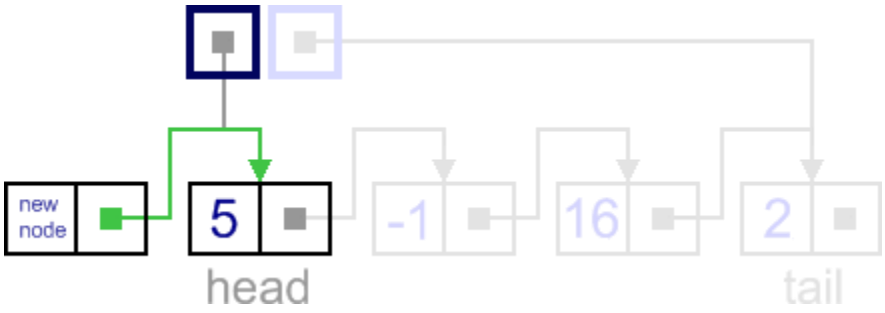
In Theory:

In this case, new node is inserted right before the current head node.

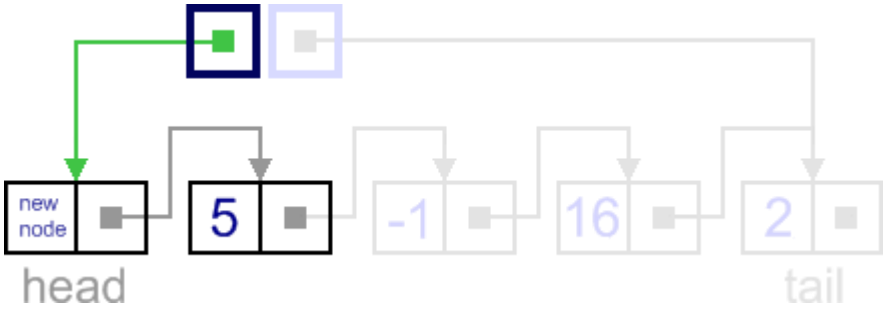


It can be done in two steps:

The first step: Update the next link of a new node, to point to the current head node.



The second step: Update head link to point to the new node.



To conclude this function, let consider 2 steps. In the first step, you need to update the next link of the new node point to the head node. Then the second step, you just update the head node is the new node that you inserted. Both of two steps use few assignments to do because you are given the head node (also the tail node). Therefore, the complexity of this function is O(1)

Push

In Experiment:

Here is some structure and some function to implement Stack

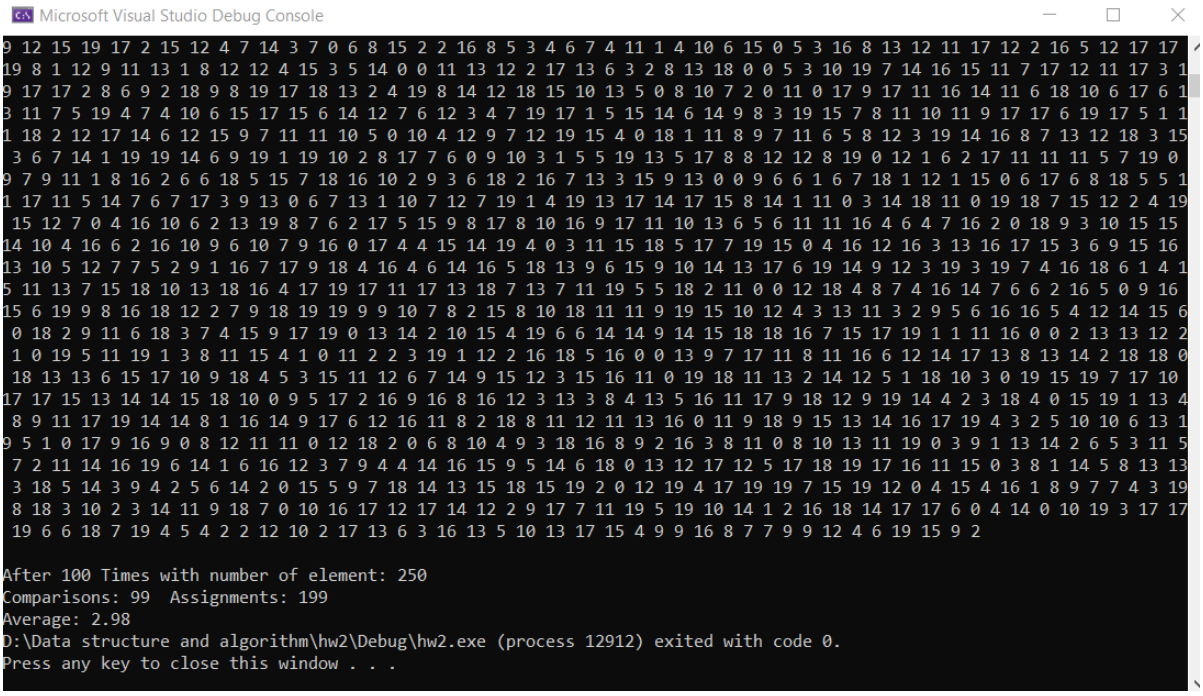
```
struct NODE {
    int data;
    NODE* p_next;
};
```

```
struct List {
    NODE* p_Head;
    NODE* p_Tail;
};
```

In this function, it just take 2 assignments to push 1 node to the stack (implementation by Linked list structure) and 1 comparison to check. Therefore, the complexity of function push is O(1) (becaue there are 3 instructions)

```
void Push(List& L, NODE* p) {

    if (L.p_head == NULL)
        L.p_head = L.p_tail = p;

    // add new node into head in the linked list
    p->p_next = L.p_head;
    L.p_head = p;

}
```

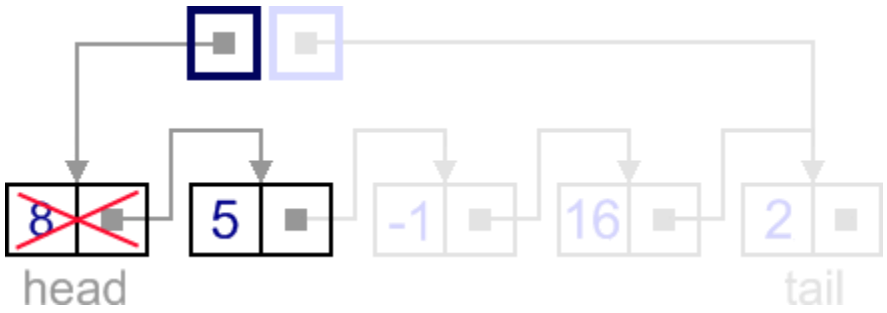
In the photo below to prove that the complexity of this function is O(1). Sum of instructions is 2.98 because I random number of element input for linked list so that maybe in many cases List happened NULL



1.2.2 Pop

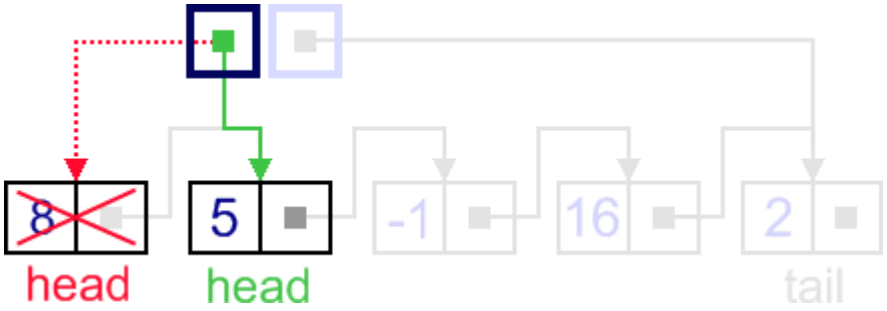
In Theory:

In this case, first node (current head node) is removed from the list.

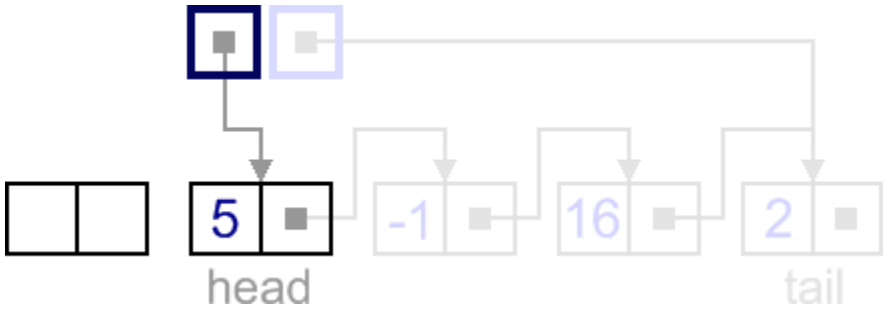


It can be done in two steps:

Update head link to point to the node, next to the head.



Dispose removed node.



To conclude this function, let consider 2 steps. In the first step you must update the head node is the next head in the list. The second step you need to dispose the node that you remove by delete to avoid leak memory.

In this function, you just use a few assignments to update and delete the head node in the list so that the complexity of this function is O(1)

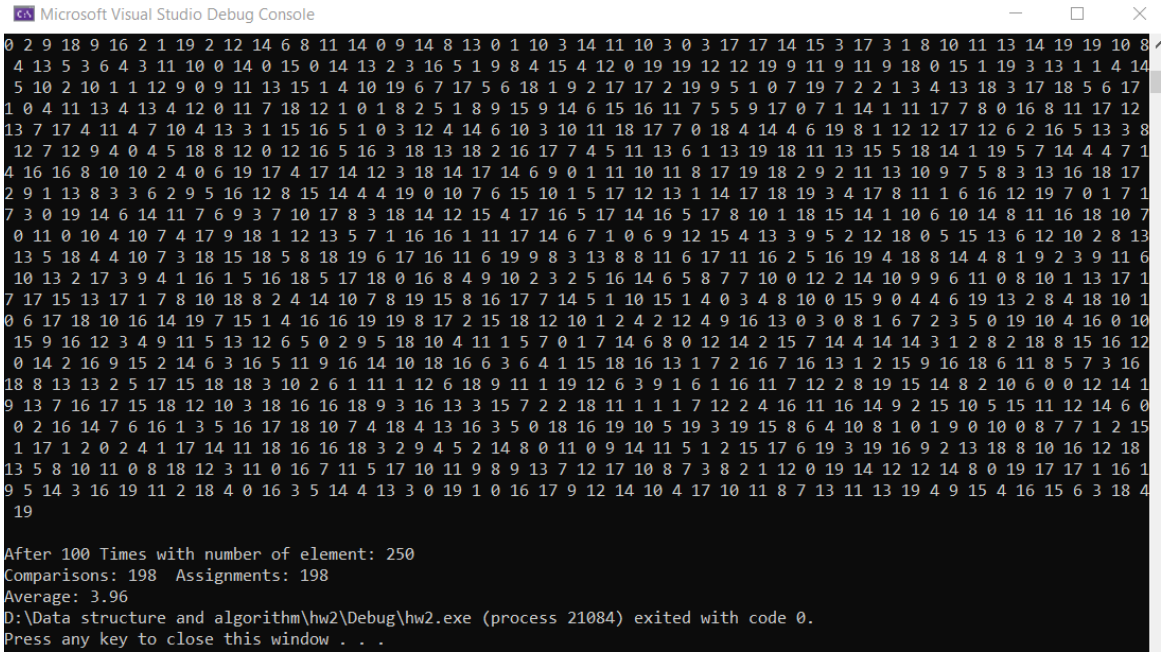
Pop

```
bool Pop(List& L) {

    if (isEmpty(L))
        return false;

    NODE* p = L.p_Head;

    L.p_Head = p->p_next;
    delete p;

}
```

In Experiment:

In this function, it just take 2 assignments and 2 comparisons (1 in the isEmpty function and 1 in the Pop function) to pop 1 node out of Stack. Therefore, the complexity of this function is O(1)

In the photo below to prove this function is O(1). Sum of instructions is 3.96 because I randomize the number of element(maybe in many cases List happened empty (it just take 2 comparisons to check instead))



1.2.3 isEmpty

In Theory:

In this function, you just need 1 comparison to check the head node is NULL, after checking if return true that is the head node is NULL and the list is empty so that there is one instruction in this function. Therefore, the complexity of this function must be O(1)

isEmpty

In Experiment:

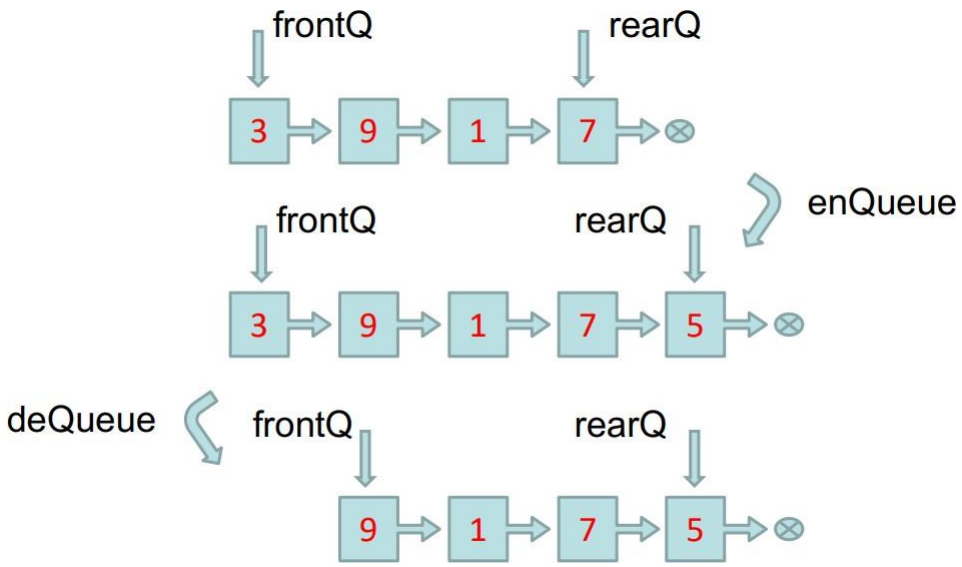
In this function, it just take 1 comparison so that it take 1 instruction to run. Therefore, the complexity of this function is O(1)

It always take 1 comparison constanly so I did not run this function to prove that

```
bool isEmpty(List& L) {
    return L.p_Head == NULL;
}
```

1.3 Queue

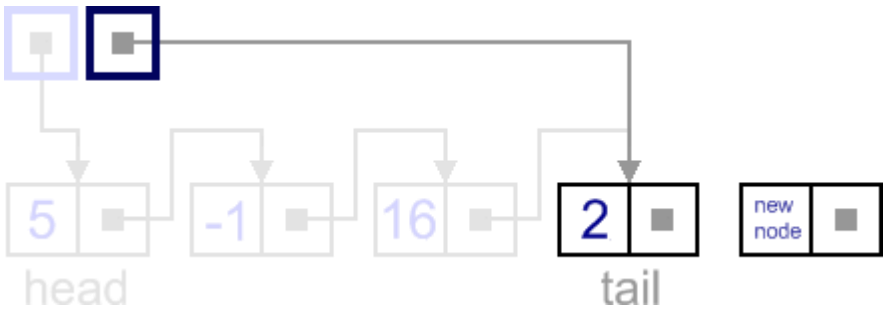
Use a data structure like this to implement Queue function:



1.3.1 Enqueue

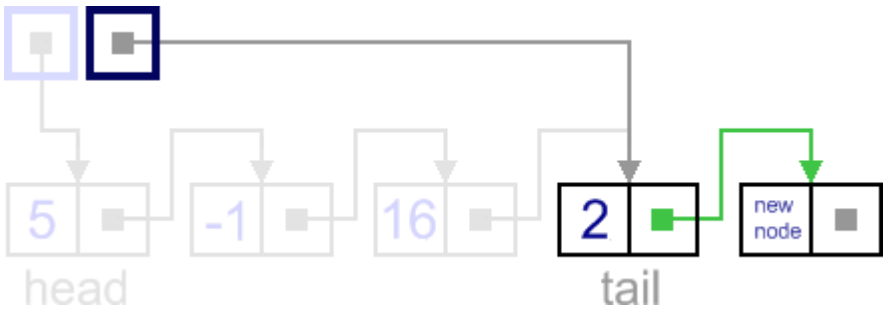
In Theory:

In this case, new node is inserted right after the current tail node.

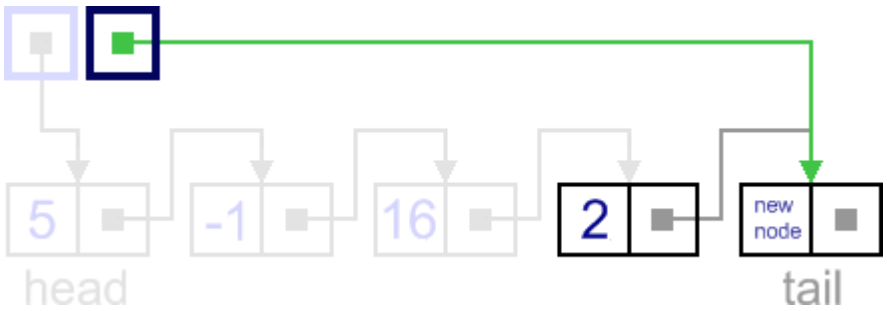


It can be done in two steps:

Update the next link of the current tail node, to point to the new node.



Update tail link to point to the new node.



To conclude this function, let consider two steps above. In the first step you need to update the next link of the tail node point to the new node. Because you are given the tail

node so that it's easy to update the next link of tail node. The second step you need to update the tail node now is the new node.

Both of two steps use few assignments to point the next link and update the tail node. Therefore, the complexity of this function is O(1).

Enqueue

In Experiment:

Here is a structure and some function to implement Queue:

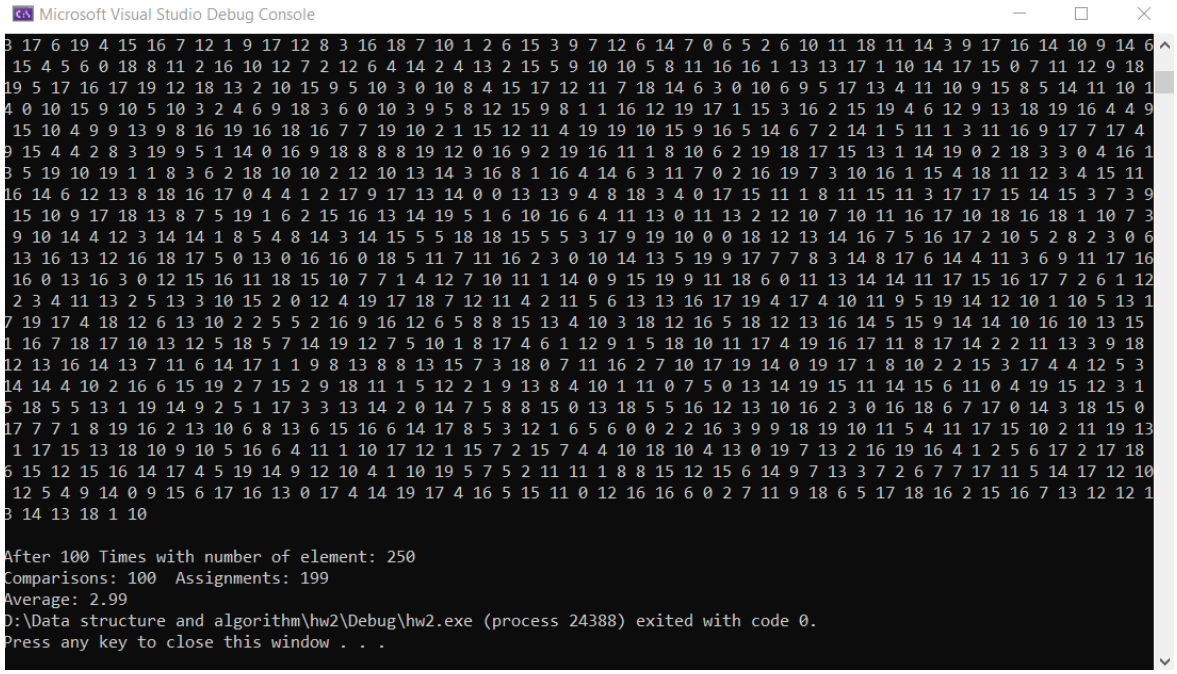
```
struct NODE {
    int data;
    NODE* p_next;
};
```

```
bool isEmpty(List& L) {

    return L.p_Head == NULL;

}
```

In this function, both of two condition do it take 1 comparison and 2 assignments to enqueue 1 node into Queue so that there are 3 instruction in worst case. Therefore, the complexity of this function is O(1)

```
void Enqueue(List& L, NODE* p) {

    if (L.p_Head == NULL)
        L.p_Head = L.p_Tail = p;
    else {
        L.p_Tail->p_next = p;
        L.p_Tail = p;
    }

}
```

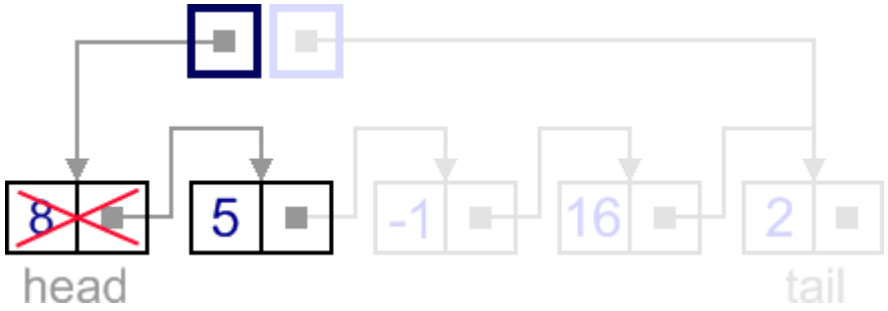
The photo below to prove the complexity is O(1)



1.3.2 Dequeue

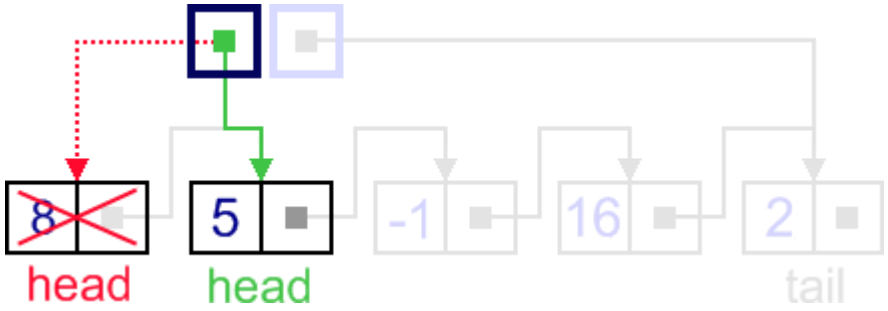
In Theory:

In this case, first node (current head node) is removed from the list.

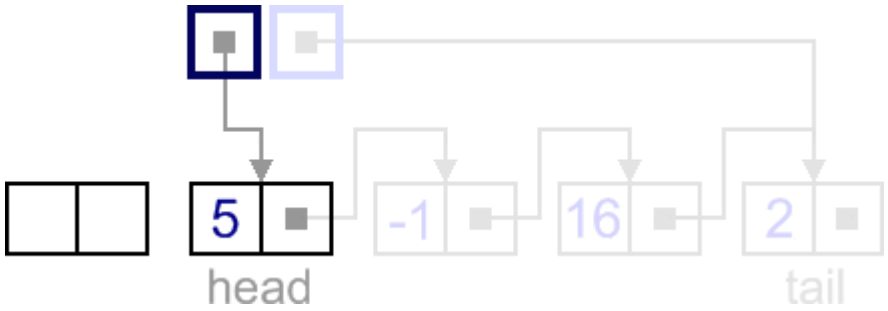


It can be done in two steps:

Update head link to point to the node, next to the head.



Dispose removed node.



To conclude this function, let consider 2 steps. In the first step you must update the head node is the next head in the list. The second step you need to dispose the node that you remove by delete to avoid leak memory.

In this function, you just use a few assignments to update and delete the head node in the list so that the complexity of this function is O(1)

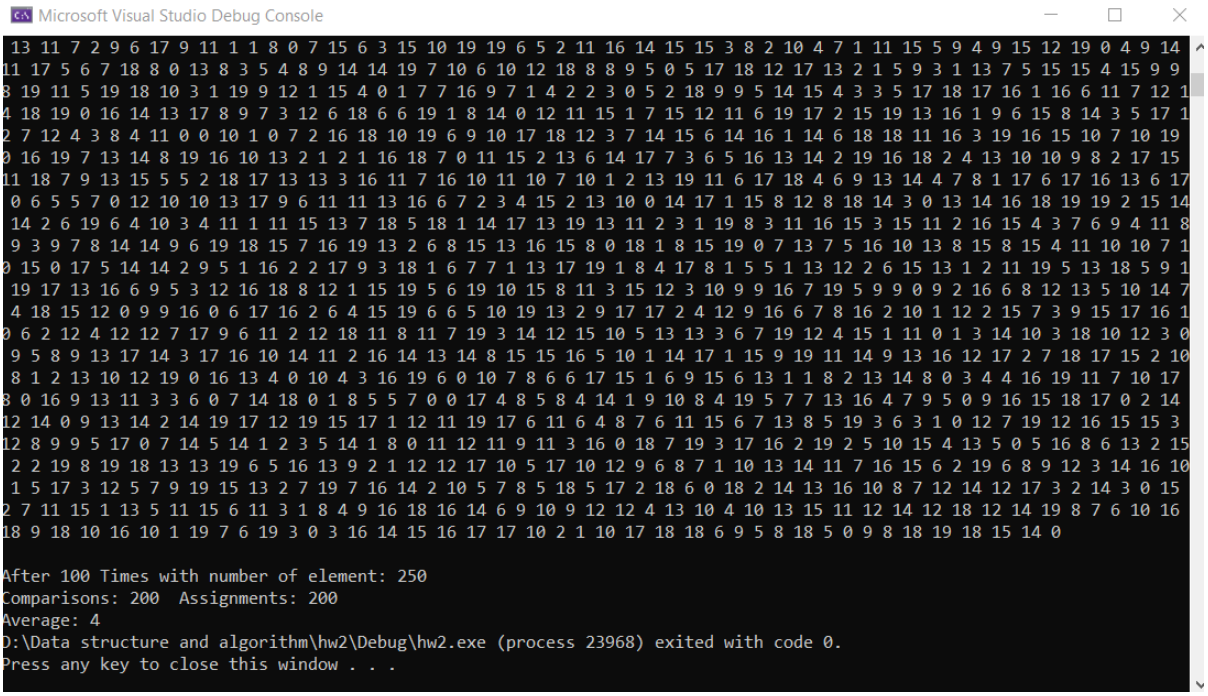
Dequeue

In Experiment:

In this function, it take 2 assignments and 2 comparison(1 in the function isEmpty and 1 in the dequeue). The complexity of this function is O(1)

The photo below to prove the complexity of this function is O(1) in the experiment

```
bool Dequeue(List& L) {

    if (isEmpty(L))
        return false;

    NODE* p = L.p_head;
    L.p_head = L.p_head->p_next;

    delete p;

    return true;
}
```



1.3.3 GetFront

In Theory:

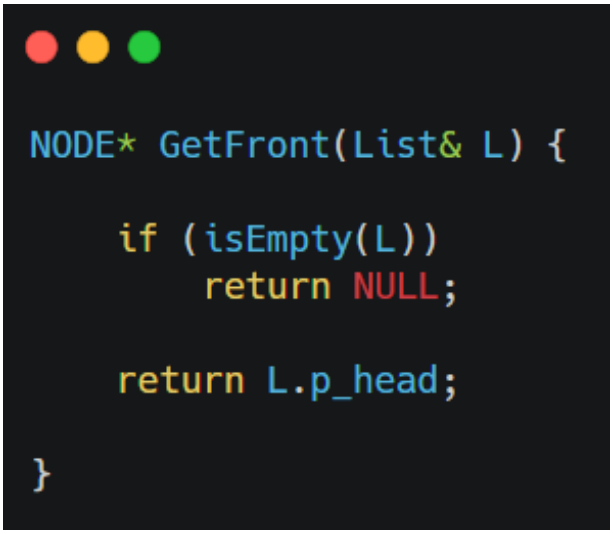
In this function, it's very simple to get front the node in queue because queue is implemented by linked list so that you just get the head node of the linked list. To conclude this function, because you are given the head node in the linked list so that you just check the linked list is empty. In this function, you just use few comparison to check the linked list. Therefore, the complexity of this function is O(1)

GetFront

In Experiment:

In this function, it take 2 comparisons(1 in isEmpty function and 1 in front function). Therefore, the complexity of this function is O(1)

There is unnecessary to run this function to verify complexity because it's very clearly to conclude the complexity of this function



```
NODE* GetFront(List& L) {

    if (isEmpty(L))
        return NULL;

    return L.p_head;

}
```

2 Reference

[1]. Linked list - Wikipedia

[2]. Stack - Wikipedia

[3]. Queue - Wikipedia

[4]. Data Structures and Algorithms Made Easy: Data Structures and Algorithmic Puzzles 5th Edition by Narasimha Karumanchi