2023/2024(2)

EF234201 Data Structure

Lecture #6

Single Linked List Non Circular

Misbakhul Munir IRFAN SUBAKTI

司馬伊凡

Мисбакхул Мунир Ирфан Субакти

Linked List: History

- Developed in 1955-1956 by Allen Newell, Cliff Shaw and Herbert Simon at the RAND Corporation as the main data structure for the Information Processing Language (IPL) language
  - IPL was created to develop **artificial intelligence** programs, such as creating Chess Solver
- Victor Yngve at the Massachusetts Institute of Technology (MIT) also uses linked lists in **natural language processing** and machine transitions in the COMMIT programming language.

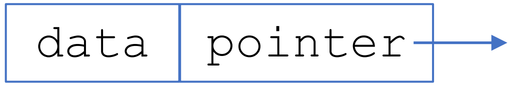
Linked List: What is It?

- Linked List is a form of data structure, containing a collection of data (nodes) that are **arranged** sequentially, **interconnected**, **dynamic** and limited
- Linked List are often called Chained List
- Linked Lists are connected to each other with the help of pointer variables
- Each data in a Linked List is called a node which occupies dynamic memory allocation and is usually in the form of a struct consisting of several fields

Array vs Linked List

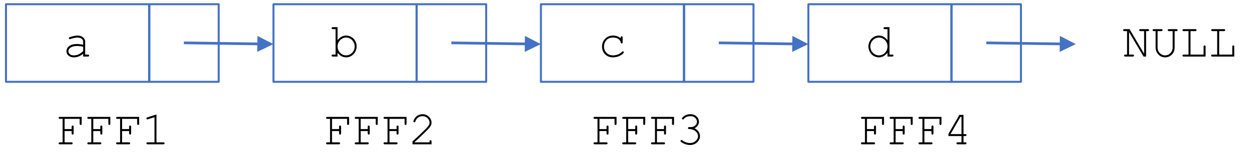
Array	Linked List
Static	Dynamic
Limited data addition/subtraction	Unlimited data addition/subtraction
Random access	Sequential access
Deleting the array is impossible	Easy for deleting the linked list

Single Linked List Non Circular (SLLNC)



Occupies a specific memory address

- Single: it means that the pointer field is only one in one direction and at the end of the node, the pointer points to `NULL`
- Linked List: it means that the nodes are connected to each other

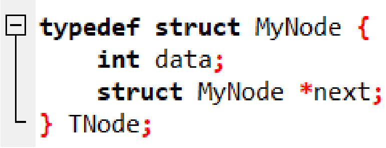


- Each node in a linked list has a field that contains a pointer to the next node, and also has a field that contains data
- The last node will point to `NULL` which will be used as a **stop condition** when reading the content of the linked list
- Keyword for `NULL` in C++ is `nullptr`

Single Linked List Non Circular: Create

- Node Declaration

```
typedef struct MyNode {
   int data;
   struct MyNode *next;
} TNode;
```



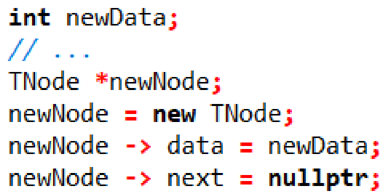
```
typedef struct MyNode {
    int data;
    struct MyNode *next;
} TNode;
```

- Explanation
  - Creating a struct called `TNode` which contains 2 fields, namely the `data` field of type integer and the `next` field which is of type pointer from `TNode`
  - After creating the struct, create a `head` variable of type pointer from `TNode` which is useful as the **head** of the linked list

Single Linked List Non Circular: Create (cont'd)

- The keyword `new` is used which means preparing a new node along with its memory allocation, then the node is filled with `data` and the `next` pointer is pointed to `NULL`
- `nullptr` is used in C++ for `NULL`

```
int newData;
// …
TNode *newNode;
newNode = new TNode;
newNode -> data = newData;
newNode -> next = nullptr;
```



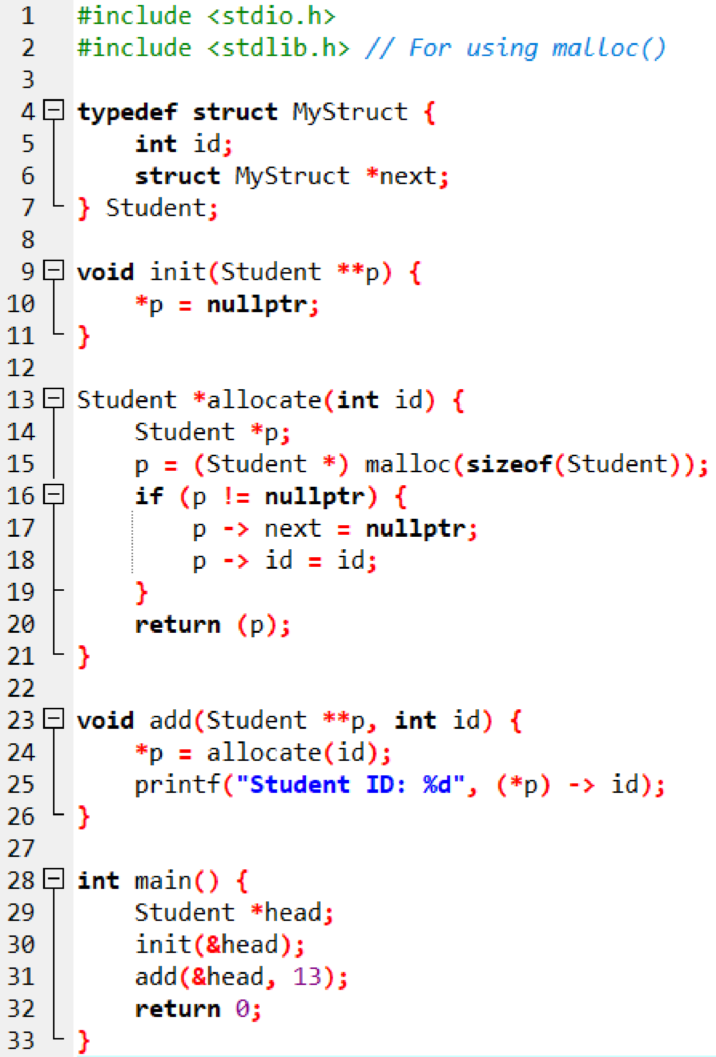
```
int newData;
// ...
TNode *newNode;
newNode = new TNode;
newNode -> data = newData;
newNode -> next = nullptr;
```

Pointer Allocation: Another Way

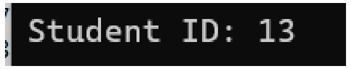
- Using manual memory allocation
- Use `stdlib.h` or `malloc.h` headers
- Using function:

```
    <pointer type> *malloc(int size);
```

Pointer Program Example



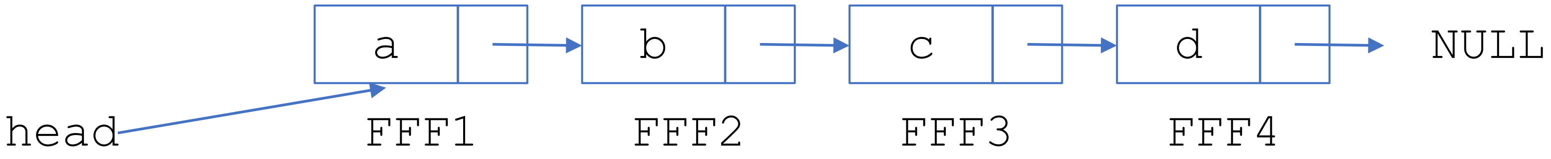
```
#include <stdio.h>
#include <stdlib.h> // For using malloc()

typedef struct MyStruct {
    int id;
    struct MyStruct *next;
} Student;

void init(Student **p) {
    *p = nullptr;
}

Student *allocate(int id) {
    Student *p;
    p = (Student *) malloc(sizeof(Student));
    if (p != nullptr) {
        p -> next = nullptr;
        p -> id = id;
    }
    return (p);
}

void add(Student **p, int id) {
    *p = allocate(id);
    printf("Student ID: %d", (*p) -> id);
}

int main() {
    Student *head;
    init(&head);
    add(&head, 13);
    return 0;
}
```



```
Student ID: 13
```

Headed SLLNC

- One pointer variable is required: **head**
- **head** will always point to the first node



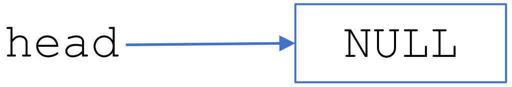
- Single Linked List Headed Pointer Declaration
  - Manipulation of linked list cannot be done directly to the destination node, but must use a pointer to the first node in the linked list (in this case is **head**)
  - The declaration is as follows

```
TNode *head;
```

Headed SLLNC: init and isEmpty

- Single Linked List Initialization Function

```
void init() {
   head = nullptr;
}
```



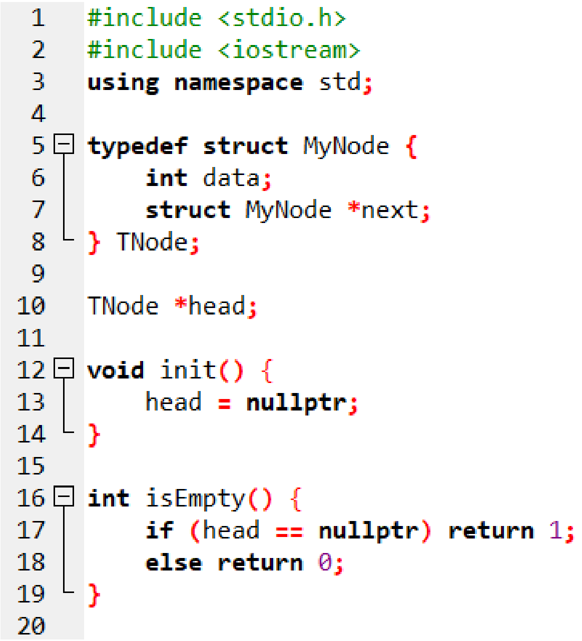
- Function to find out whether Single LinkedList is empty or not
  - If the **head** pointer does not point to a node then it is empty

```
int isEmpty() {
   if (head == nullptr) return 1;
   else return 0;
}
```

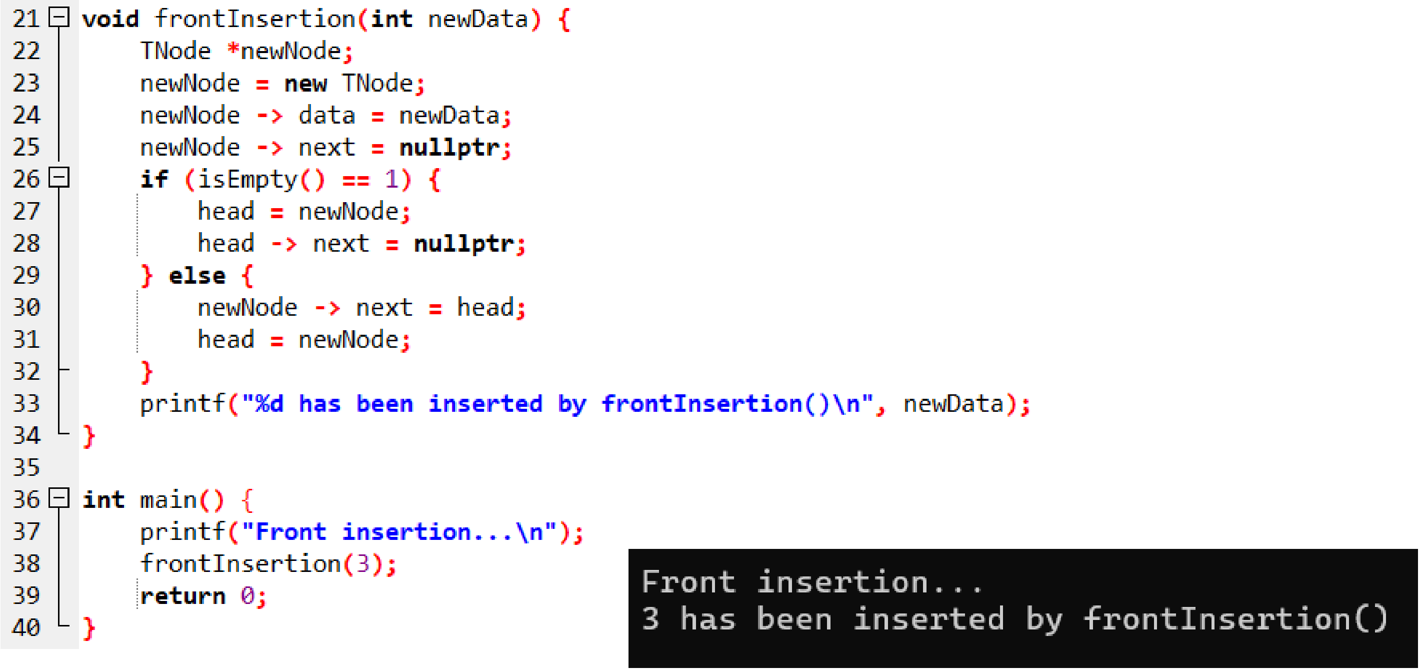
Headed SLLNC: frontInsertion

- Added data at the front
  - The addition of a new node will be linked to the **frontmost** node, but for the first time (the data is still empty), data is added in this way: the head node is shown to the new node
  - The principle is to associate a new node with the head, then the head will point to the new data so that the head will always remain the leading/top (first) data

Headed SLLNC: `frontInsertion` (cont’d)



```
#include <stdio.h>
#include <iostream>
using namespace std;

typedef struct MyNode {
    int data;
    struct MyNode *next;
} TNode;

TNode *head;

void init() {
    head = nullptr;
}

int isEmpty() {
    if (head == nullptr) return 1;
    else return 0;
}

```



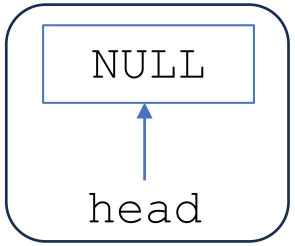
```
void frontInsertion(int newData) {
    TNode *newNode;
    newNode = new TNode;
    newNode -> data = newData;
    newNode -> next = nullptr;
    if (isEmpty() == 1) {
        head = newNode;
        head -> next = nullptr;
    } else {
        newNode -> next = head;
        head = newNode;
    }
    printf("%d has been inserted by frontInsertion()\n", newData);
}

int main() {
    printf("Front insertion...\n");
    frontInsertion(3);
    return 0;
}
```

```
Front insertion...
3 has been inserted by frontInsertion()
```

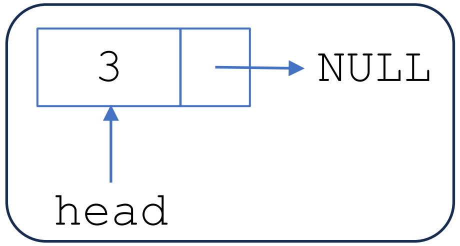
Headed SLLNC: `frontInsertion` (cont’d)

- Illustration

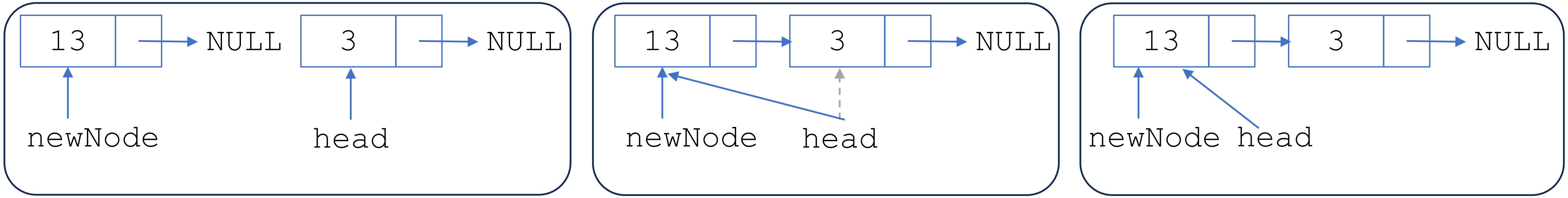
1. The list is still empty.



2. Enter the new data, e.g., 3.



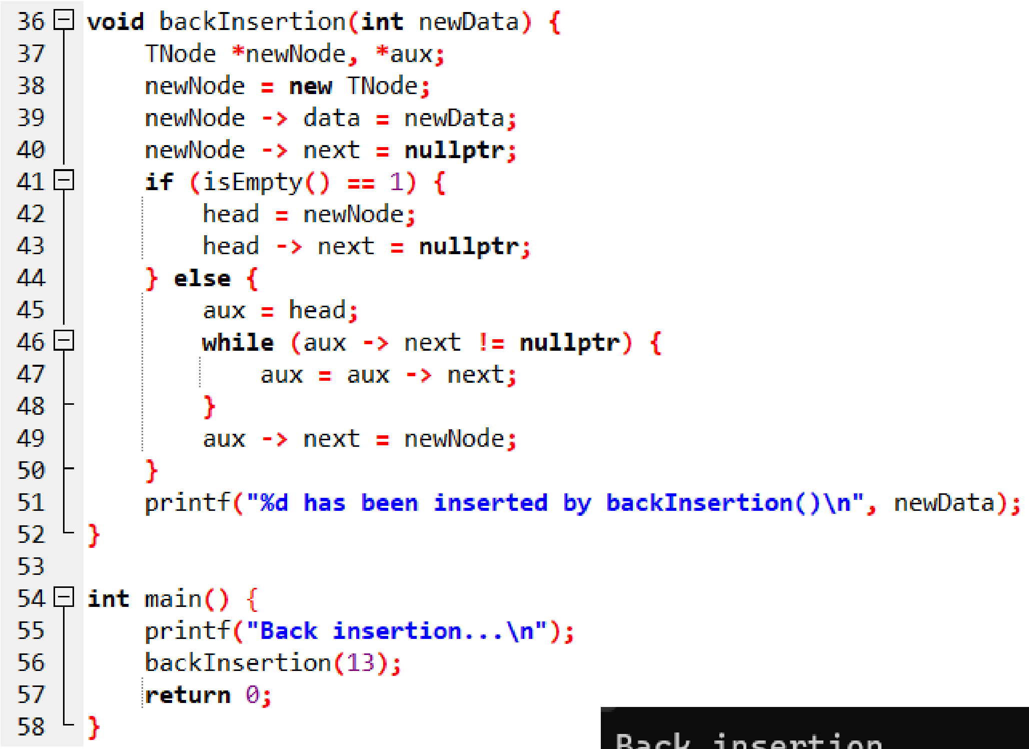
3. Enter the new data, e.g., 13. Insertion at the front.



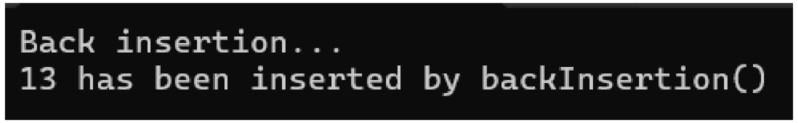
Headed SLLNC: backInsertion

- Added data at the back
  - Adding data is done at the back, but the first time, the node is directly appointed by the head
  - This addition is more difficult because we need an aux (auxiliary) pointer to find out the backward node, then after that, associate it with a new node. To find out the most recent data, loops need to be used.

Headed SLLNC: backInsertion (cont’d)



```
void backInsertion(int newData) {
    TNode *newNode, *aux;
    newNode = new TNode;
    newNode -> data = newData;
    newNode -> next = nullptr;
    if (isEmpty() == 1) {
        head = newNode;
        head -> next = nullptr;
    } else {
        aux = head;
        while (aux -> next != nullptr) {
            aux = aux -> next;
        }
        aux -> next = newNode;
    }
    printf("%d has been inserted by backInsertion()\n", newData);
}

int main() {
    printf("Back insertion...\n");
    backInsertion(13);
    return 0;
}
```

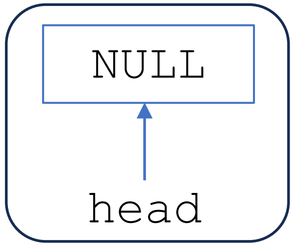


```
Back insertion...
13 has been inserted by backInsertion()
```

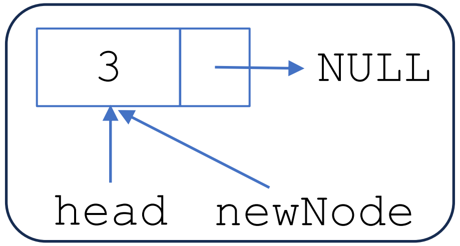
Headed SLLNC: `backInsertion` (cont'd)

- Illustration

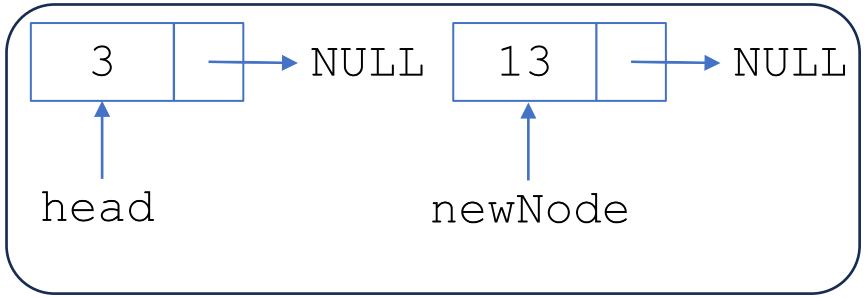
1. The list is still empty (head = NULL).

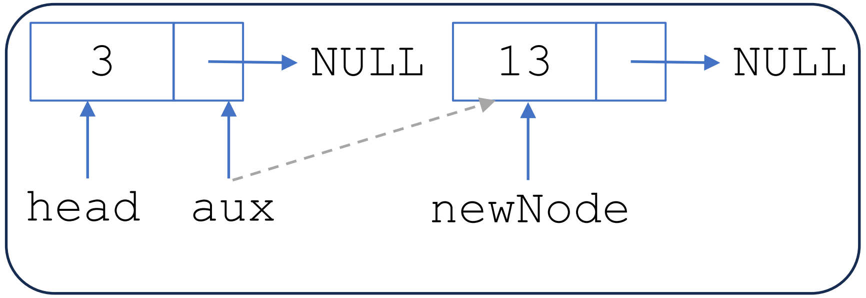


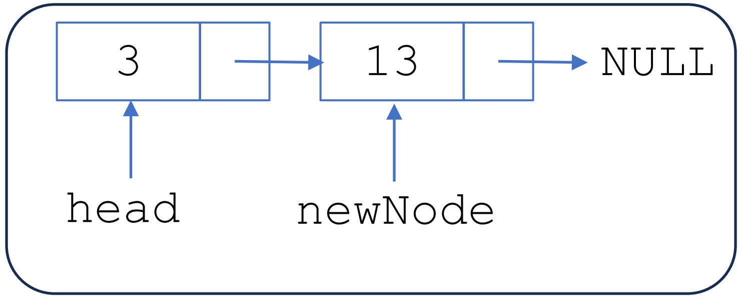
2. Enter the new data, e.g., 3.



3. Enter the new data, e.g., 13. Insertion at the back.

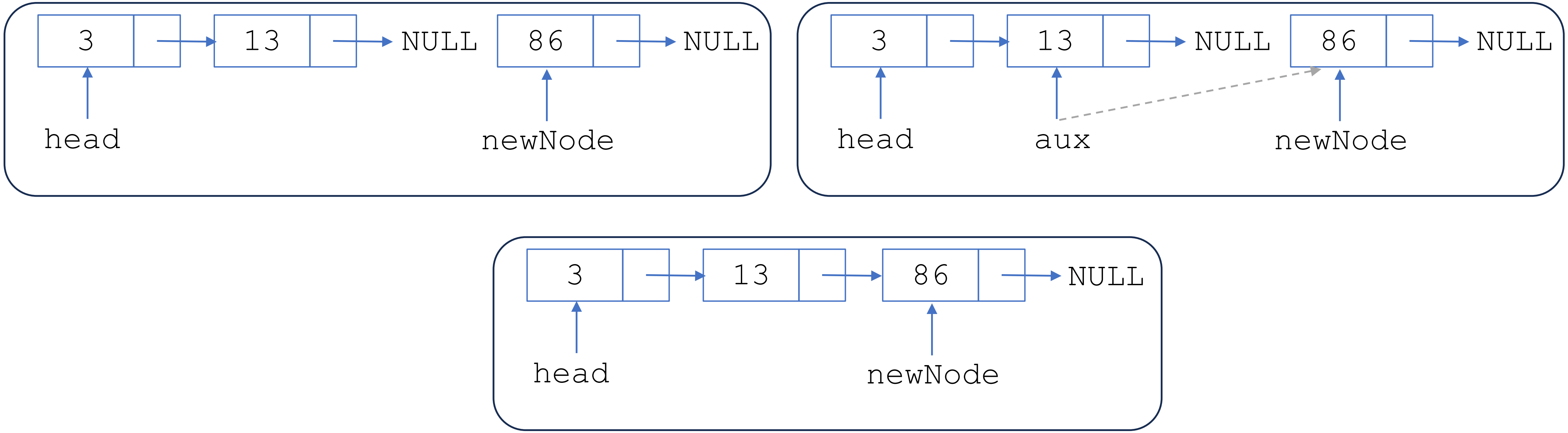






Headed SLLNC: `backInsertion` (cont’d)

4. Enter the new data, e.g., 86. Insertion at the back.



How about the insertion at the middle?

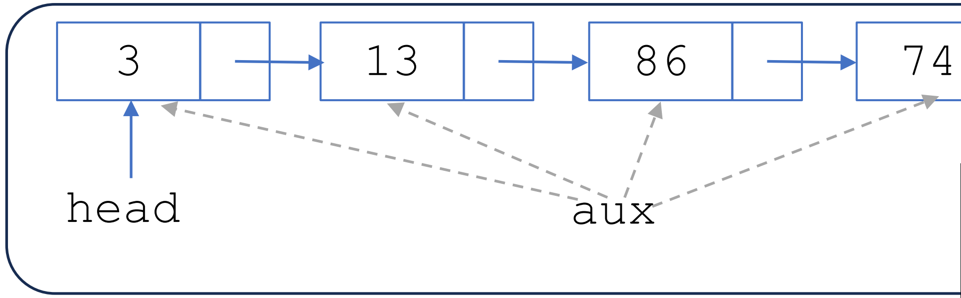
Headed SLLNC: show

```
#include <stdio.h>
#include <iostream>
using namespace std;

typedef struct MyNode {
    int data;
    struct MyNode *next;
} TNode;

TNode *head;

void init() {
    head = nullptr;
}

int isEmpty() {
    if (head == nullptr) return 1;
    else return 0;
}
```

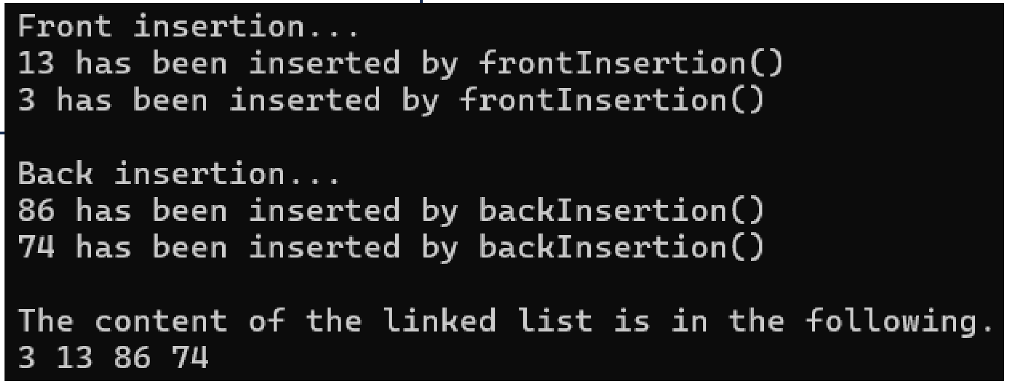
```
void frontInsertion(int newData) {
    TNode *newNode;
    newNode = new TNode;
    newNode -> data = newData;
    newNode -> next = nullptr;
    if (isEmpty() == 1) {
        head = newNode;
        head -> next = nullptr;
    } else {
        newNode -> next = head;
        head = newNode;
    }
    printf("%d has been inserted by frontInsertion()\n", newData);
}
```

```
void backInsertion(int newData) {
    TNode *newNode, *aux;
    newNode = new TNode;
    newNode -> data = newData;
    newNode -> next = nullptr;
    if (isEmpty() == 1) {
        head = newNode;
        head -> next = nullptr;
    } else {
        aux = head;
        while (aux -> next != nullptr) {
            aux = aux -> next;
        }
        aux -> next = newNode;
    }
    printf("%d has been inserted by backInsertion()\n", newData);
}
```

```
void show() {
    TNode *aux;
    aux = head;
    if (isEmpty() == 0) {
        while (aux != nullptr) {
            cout << aux -> data << " ";
            aux = aux -> next;
        }
        printf("\n");
    } else {
        printf("It's empty\n");
    }
}
```



```
int main() {
    printf("Front insertion...\n");
    frontInsertion(13);
    frontInsertion(3);

    printf("\nBack insertion...\n");
    backInsertion(86);
    backInsertion(74);

    printf("\nThe content of the linked list is in the following.\n");
    show();
    return 0;
}
```

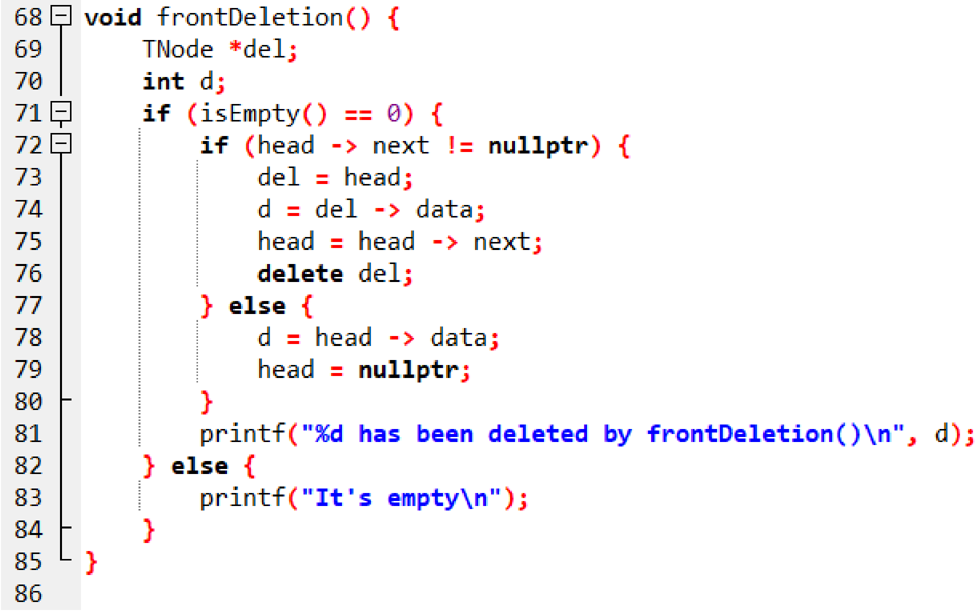


```
Front insertion...
13 has been inserted by frontInsertion()
3 has been inserted by frontInsertion()

Back insertion...
86 has been inserted by backInsertion()
74 has been inserted by backInsertion()

The content of the linked list is in the following.
3 13 86 74
```

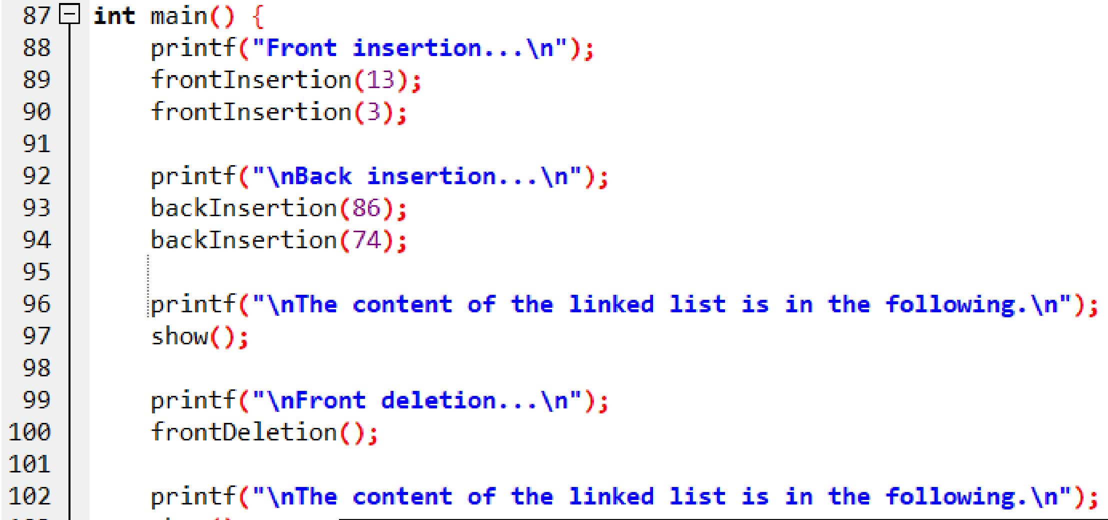
Headed SLLNC: show (continued)

- The function above is used to display all the contents of the list, where the linked list is traced one by one from the start node to the end node. This search is carried out using an aux (**auxiliary)** pointer, because in principle it is a **head** pointer which is the initial sign of the list cannot change or change the position.
- The search continues until the last node is found pointing to a NULL value. If it is not NULL, then the aux node will move to the next node and read the contents of the data using the next field so that they can be related to each other.
- If head is still NULL it means the data is still empty

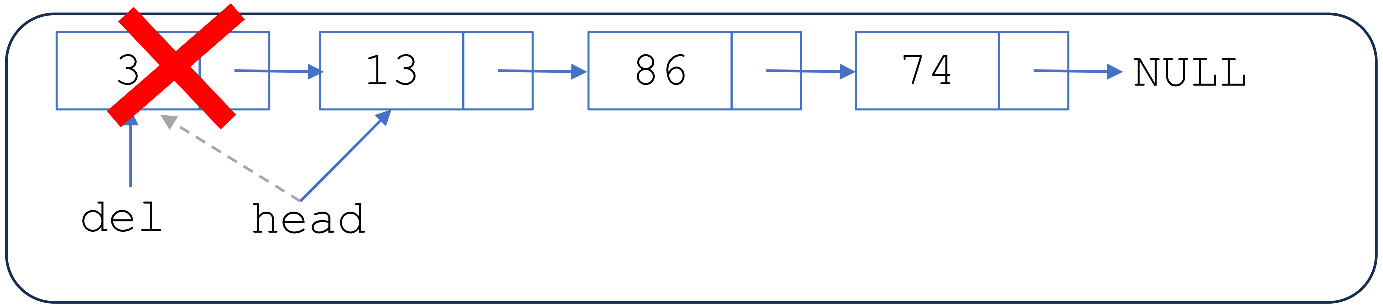
Headed SLLNC: frontDeletion

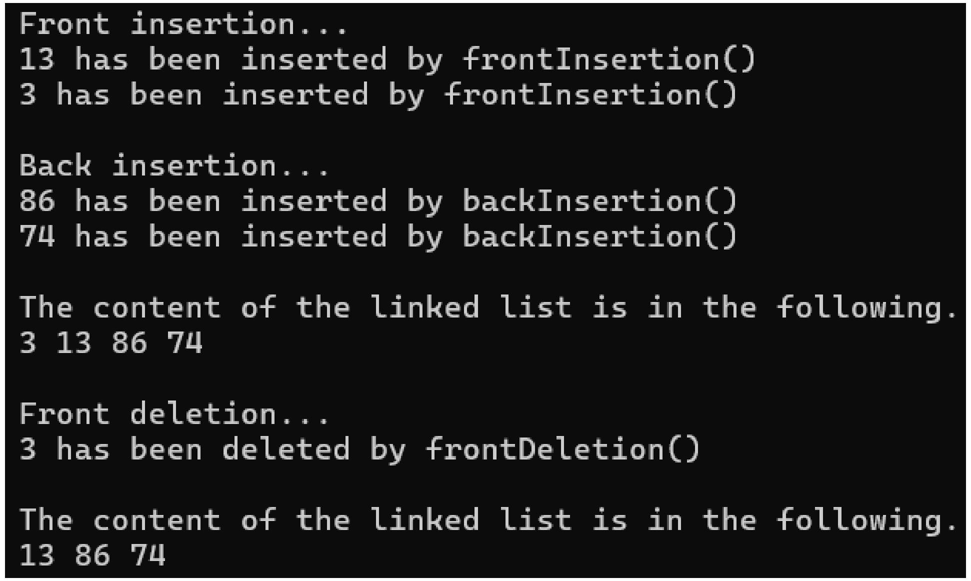


```
void frontDeletion() {
    TNode *del;
    int d;
    if (isEmpty() == 0) {
        if (head -> next != nullptr) {
            del = head;
            d = del -> data;
            head = head -> next;
            delete del;
        } else {
            d = head -> data;
            head = nullptr;
        }
        printf("%d has been deleted by frontDeletion()\n", d);
    } else {
        printf("It's empty\n");
    }
}
```



```
int main() {
    printf("Front insertion...\n");
    frontInsertion(13);
    frontInsertion(3);

    printf("\nBack insertion...\n");
    backInsertion(86);
    backInsertion(74);

    printf("\nThe content of the linked list is in the following.\n");
    show();

    printf("\nFront deletion...\n");
    frontDeletion();

    printf("\nThe content of the linked list is in the following.\n");
    show();
    return 0;
}
```



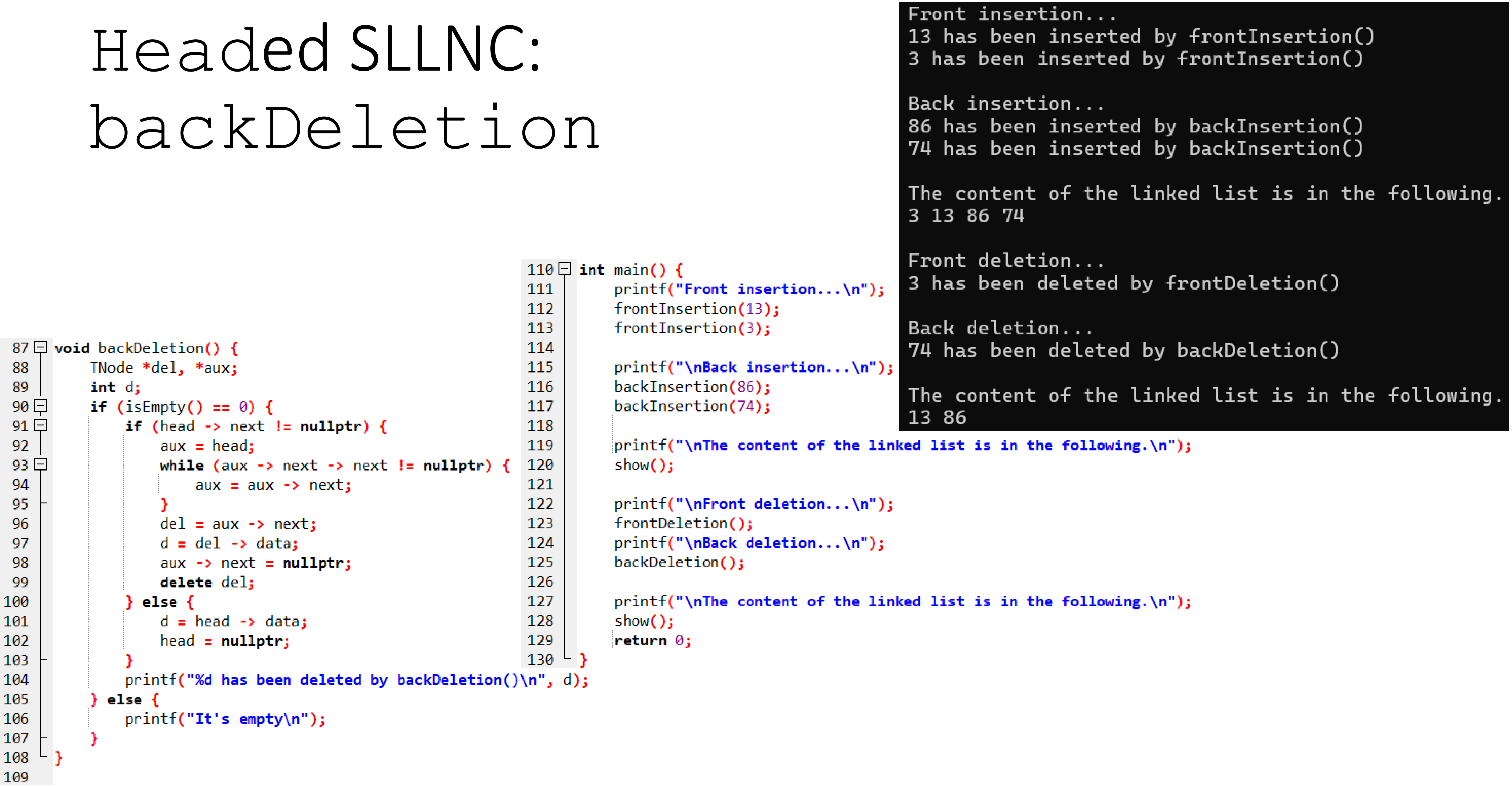


```
Front insertion...
13 has been inserted by frontInsertion()
3 has been inserted by frontInsertion()

Back insertion...
86 has been inserted by backInsertion()
74 has been inserted by backInsertion()

The content of the linked list is in the following.
3 13 86 74

Front deletion...
3 has been deleted by frontDeletion()

The content of the linked list is in the following.
13 86 74
```

Headed SLLNC: frontDeletion (cont'd)

- The function above will delete the **top** (**first**) data pointed by the **head** in the linked list
- Node deletion may not be carried out if the node is being pointed at by a pointer, so another pointer must be used to point the node to be deleted, for example a `del` pointer and then delete the `del` pointer using the **delete** command
- Before the front data is deleted, the `head` must be shown to the next node first so that the list does not break, so that the node after the old head will become the new `head` (new front data)
- If `head` is still `NULL` then it means the data is still empty

Headed SLLNC: backDeletion



```
Front insertion...
13 has been inserted by frontInsertion()
3 has been inserted by frontInsertion()

Back insertion...
86 has been inserted by backInsertion()
74 has been inserted by backInsertion()

The content of the linked list is in the following.
3 13 86 74

Front deletion...
3 has been deleted by frontDeletion()

Back deletion...
74 has been deleted by backDeletion()

The content of the linked list is in the following.
13 86
```

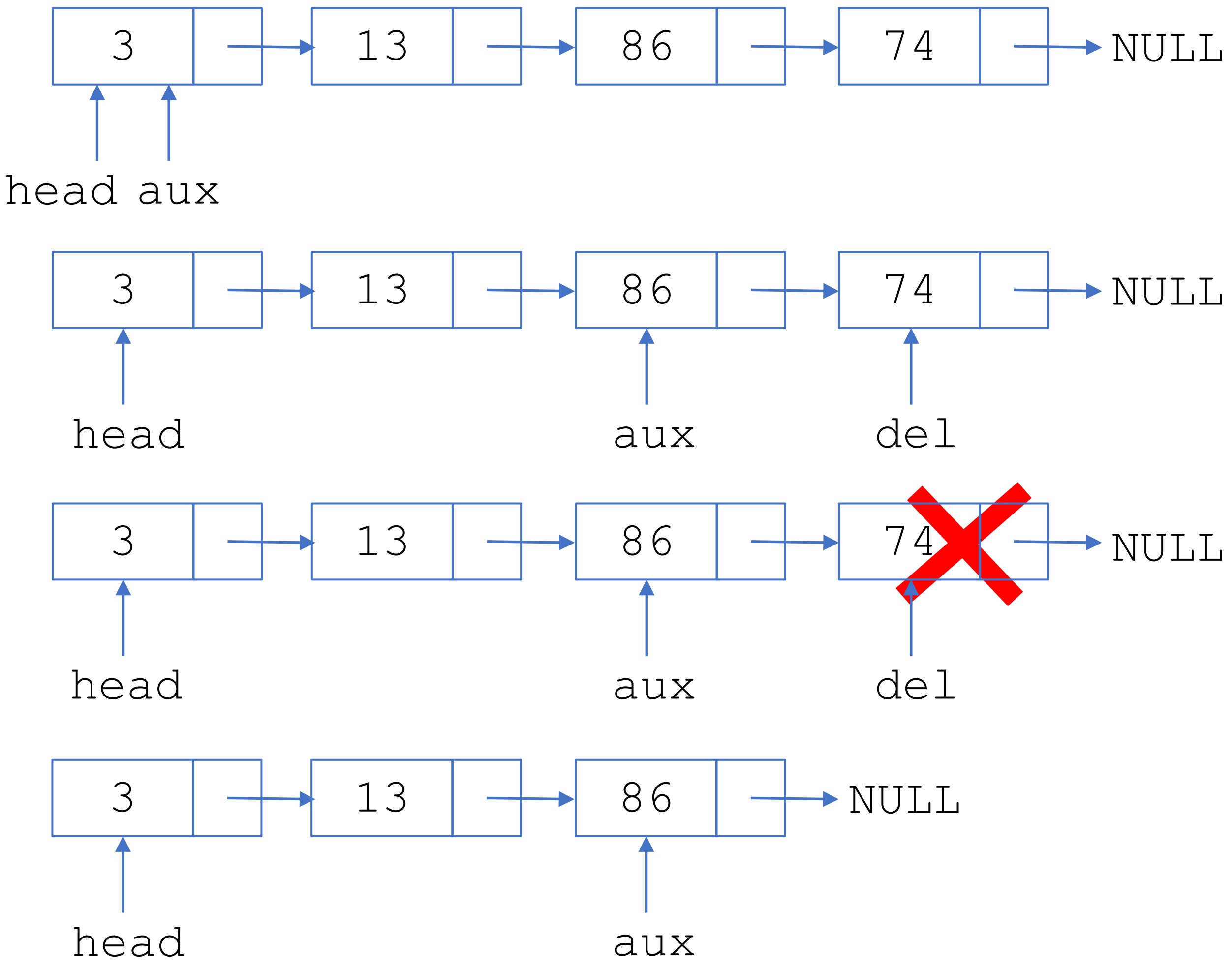
```
void backDeletion() {
    TNode *del, *aux;
    int d;
    if (isEmpty() == 0) {
        if (head -> next != nullptr) {
            aux = head;
            while (aux -> next -> next != nullptr) {
                aux = aux -> next;
            }
            del = aux -> next;
            d = del -> data;
            aux -> next = nullptr;
            delete del;
        } else {
            d = head -> data;
            head = nullptr;
        }
        printf("%d has been deleted by backDeletion()\n", d);
    } else {
        printf("It's empty\n");
    }
}

```

```
int main() {
    printf("Front insertion...\n");
    frontInsertion(13);
    frontInsertion(3);

    printf("\nBack insertion...\n");
    backInsertion(86);
    backInsertion(74);

    printf("\nThe content of the linked list is in the following.\n");
    show();

    printf("\nFront deletion...\n");
    frontDeletion();
    printf("\nBack deletion...\n");
    backDeletion();

    printf("\nThe content of the linked list is in the following.\n");
    show();
    return 0;
}
```

`Headed` SLLNC: `backDeletion` (cont’d)

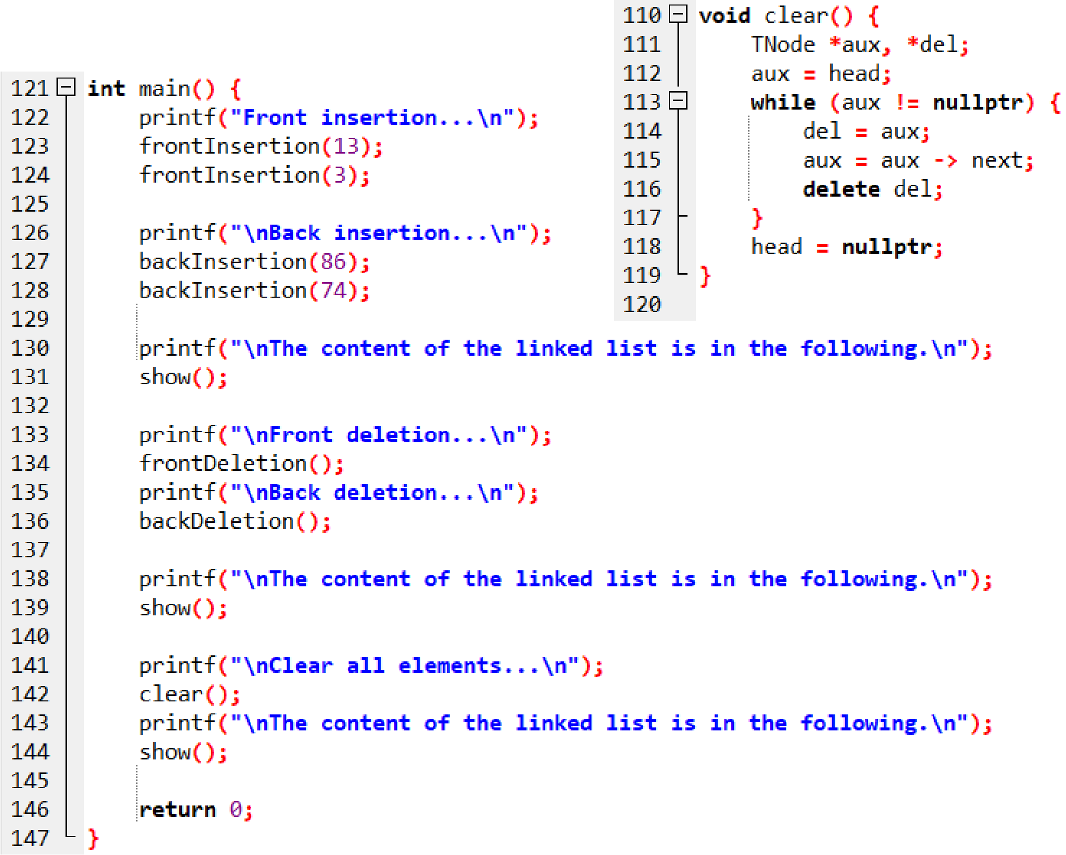
- Requires `aux` and `del` pointers
- The `del` pointer is used to point to the node to be deleted, and the `aux` pointer is used to point to the node before the deleted node which will then become the **last** node.
- `aux` pointer will be used to point to the `NULL` value
- `aux` pointer will always move until it is before the node to be deleted, then the `del` pointer is placed after the `aux` pointer. Further, the `del` pointer will be deleted, the `aux` pointer will point to `NULL`

Headed SLLNC: backDeletion (cont’d)



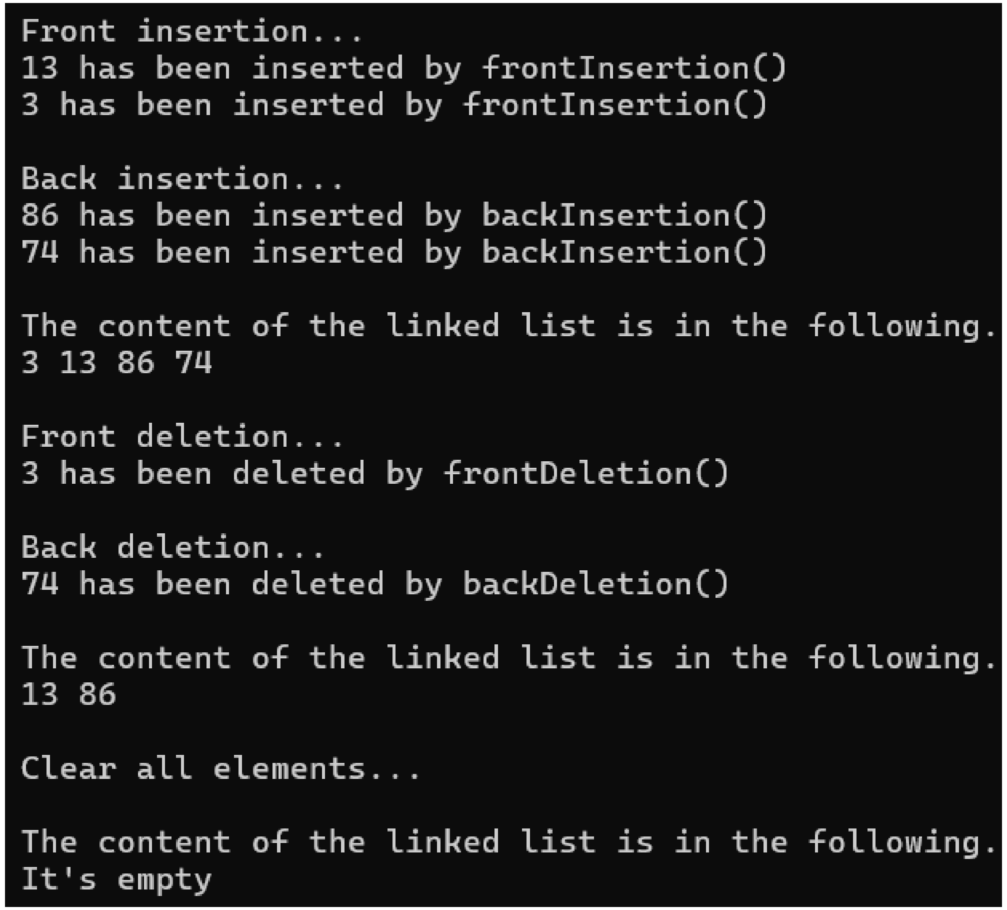
Headed SLLNC: `clear`

- Function to delete all Linked List elements



```
void clear() {
    TNode *aux, *del;
    aux = head;
    while (aux != nullptr) {
        del = aux;
        aux = aux -> next;
        delete del;
    }
    head = nullptr;
}
```

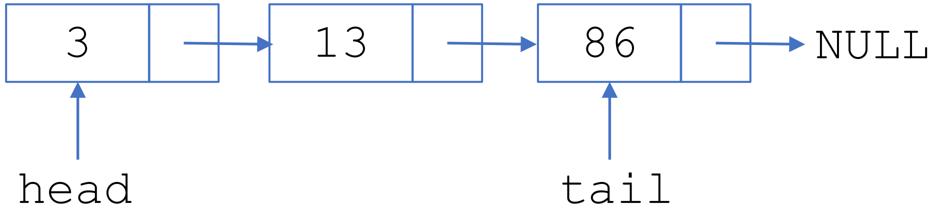
```
int main() {
    printf("Front insertion...\n");
    frontInsertion(13);
    frontInsertion(3);

    printf("\nBack insertion...\n");
    backInsertion(86);
    backInsertion(74);

    printf("\nThe content of the linked list is in the following.\n");
    show();

    printf("\nFront deletion...\n");
    frontDeletion();
    printf("\nBack deletion...\n");
    backDeletion();

    printf("\nThe content of the linked list is in the following.\n");
    show();

    printf("\nClear all elements...\n");
    clear();
    printf("\nThe content of the linked list is in the following.\n");
    show();

    return 0;
}
```



```
Front insertion...
13 has been inserted by frontInsertion()
3 has been inserted by frontInsertion()

Back insertion...
86 has been inserted by backInsertion()
74 has been inserted by backInsertion()

The content of the linked list is in the following.
3 13 86 74

Front deletion...
3 has been deleted by frontDeletion()

Back deletion...
74 has been deleted by backDeletion()

The content of the linked list is in the following.
13 86

Clear all elements...

The content of the linked list is in the following.
It's empty
```

SLLNC with `head` & `tail`

- Two pointer variables are required: `head` and `tail`
- `head` will always point to the **first node**, while `tail` will always point to the **last node**



SLLNC with head & tail: init and isEmpty

- LinkedList Initialization

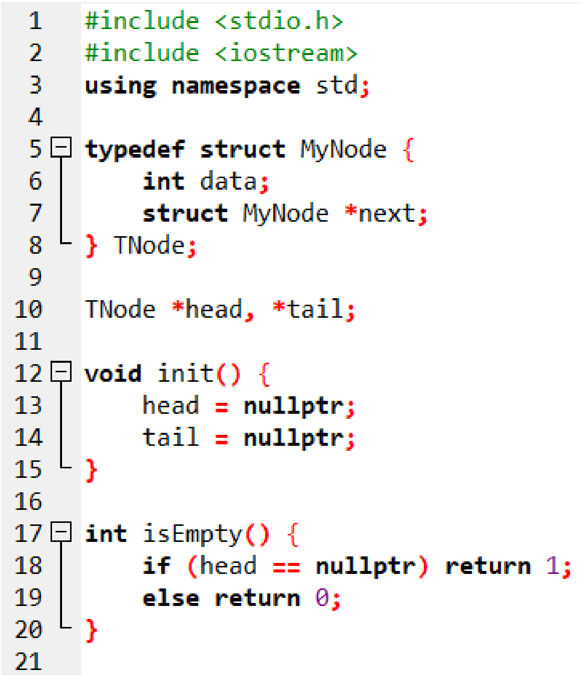
```
TNode *head, *tail;
```

- LinkedList Initialization Function

```
void init() {
    head = nullptr;
    tail = nullptr;
}
```

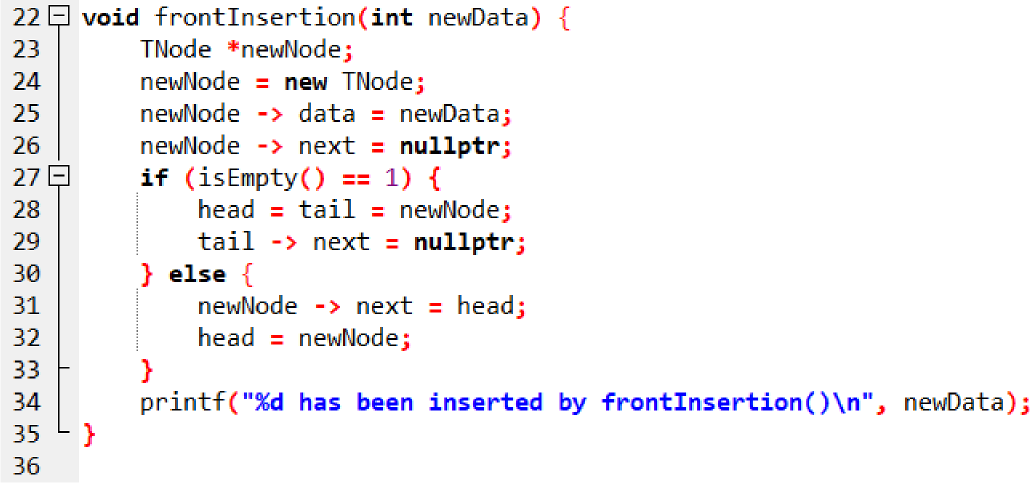
- Function to find out whether the Linked List is empty or not

```
int isEmpty() {
    if (tail == nullptr) return 1;
    else return 0;
}
```

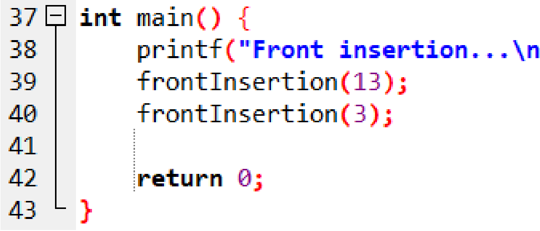


```
#include <stdio.h>
#include <iostream>
using namespace std;

typedef struct MyNode {
    int data;
    struct MyNode *next;
} TNode;

TNode *head, *tail;

void init() {
    head = nullptr;
    tail = nullptr;
}

int isEmpty() {
    if (head == nullptr) return 1;
    else return 0;
}

```

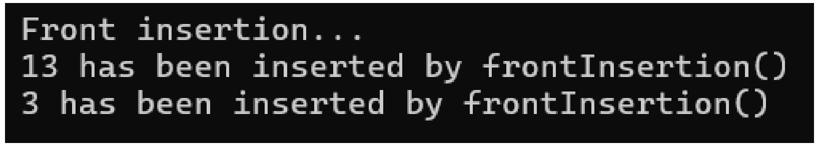
SLLNC with head & tail: frontInsertion



```
void frontInsertion(int newData) {
    TNode *newNode;
    newNode = new TNode;
    newNode -> data = newData;
    newNode -> next = nullptr;
    if (isEmpty() == 1) {
        head = tail = newNode;
        tail -> next = nullptr;
    } else {
        newNode -> next = head;
        head = newNode;
    }
    printf("%d has been inserted by frontInsertion()\n", newData);
}

```



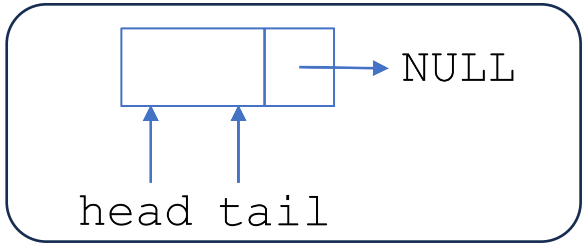
```
int main() {
    printf("Front insertion...\n");
    frontInsertion(13);
    frontInsertion(3);

    return 0;
}
```



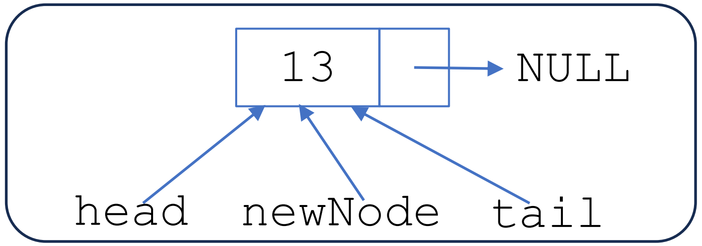
```
Front insertion...
13 has been inserted by frontInsertion()
3 has been inserted by frontInsertion()
```

SLLNC with head & tail: frontInsertion (continued)

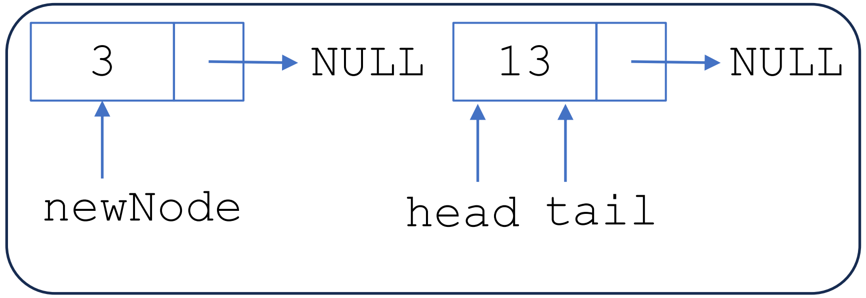
1. The list is still empty (head = tail = NULL).

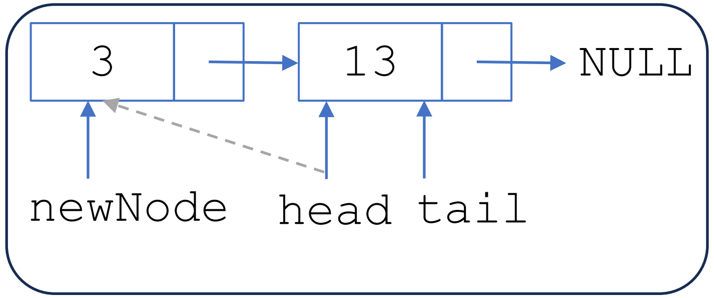


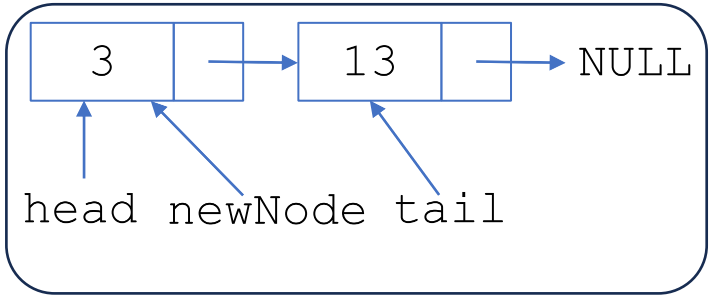
2. Enter the new data, e.g., 13.



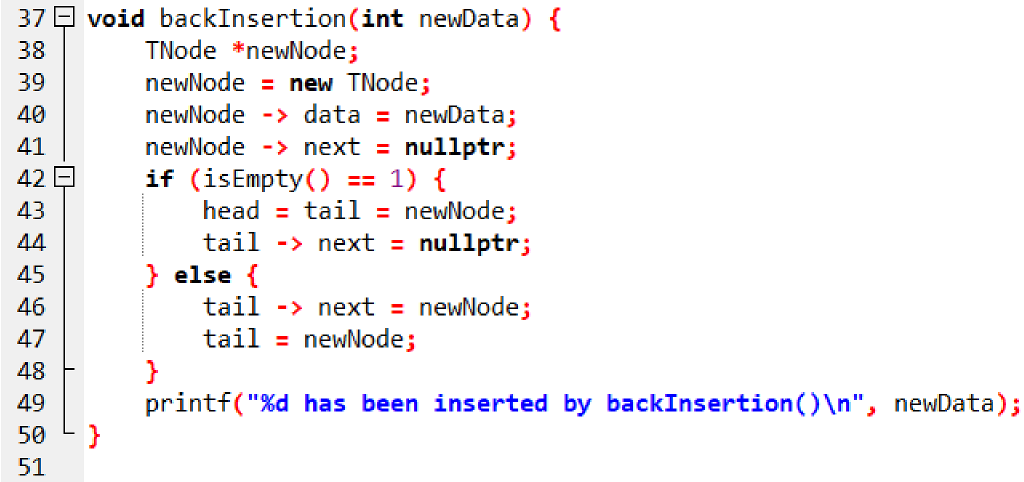
3. Enter the new data, e.g., 3. Insertion at the front.



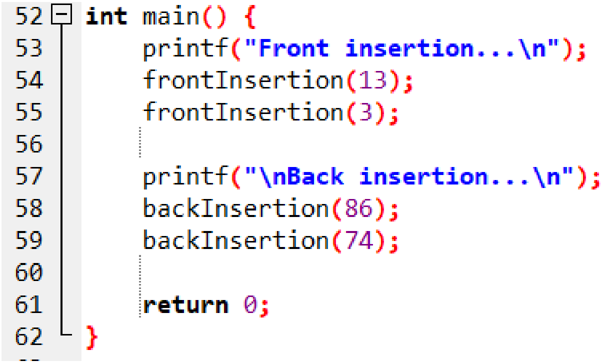




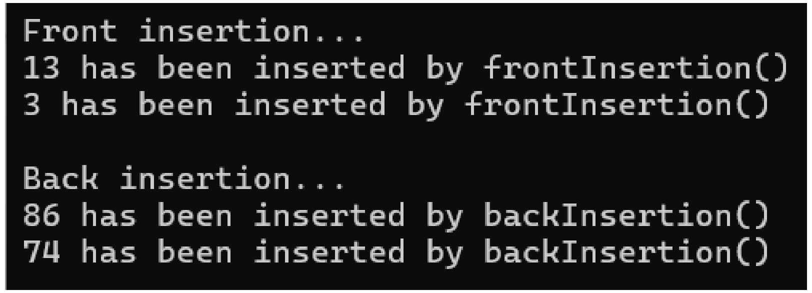
SLLNC with head & tail: backInsertion



```
void backInsertion(int newData) {
    TNode *newNode;
    newNode = new TNode;
    newNode -> data = newData;
    newNode -> next = nullptr;
    if (isEmpty() == 1) {
        head = tail = newNode;
        tail -> next = nullptr;
    } else {
        tail -> next = newNode;
        tail = newNode;
    }
    printf("%d has been inserted by backInsertion()\n", newData);
}

```



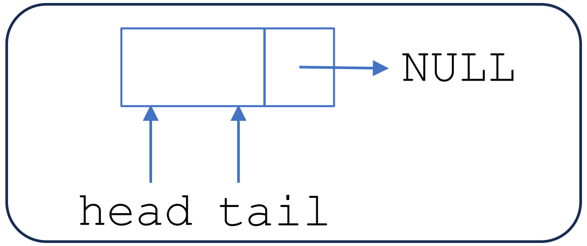
```
int main() {
    printf("Front insertion...\n");
    frontInsertion(13);
    frontInsertion(3);

    printf("\nBack insertion...\n");
    backInsertion(86);
    backInsertion(74);

    return 0;
}
```



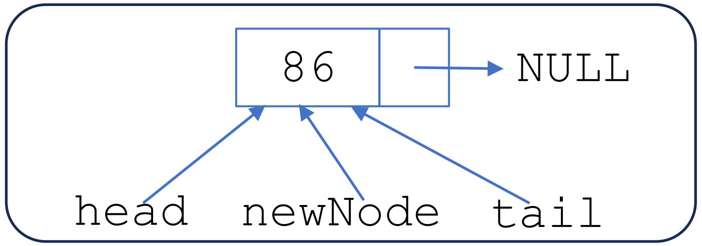
```
Front insertion...
13 has been inserted by frontInsertion()
3 has been inserted by frontInsertion()

Back insertion...
86 has been inserted by backInsertion()
74 has been inserted by backInsertion()
```

SLLNC with head & tail: backInsertion (continued)

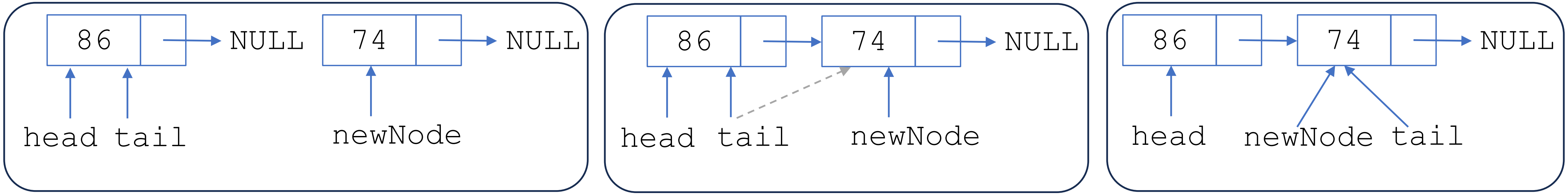
1. The list is still empty (head = tail = NULL).



2. Enter the new data, e.g., 86.

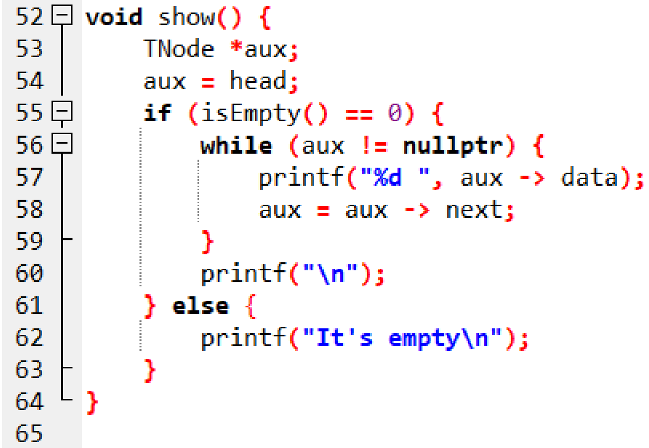


3. Enter the new data, e.g., 74. Insertion at the back.

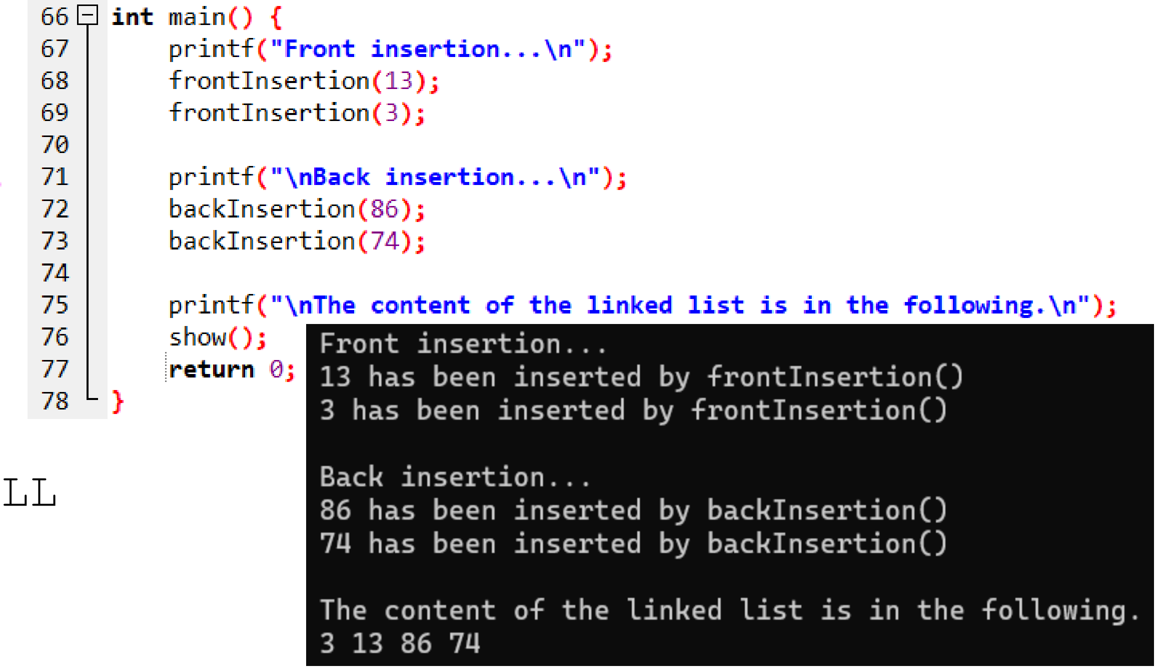


SLLNC with head & tail: show

- The advantage of a Single Linked List with head & tail is that when adding data at the **back**, only the tail is needed which binds the **new node** without having to use aux pointer loops
- Function to display the contents of a linked list

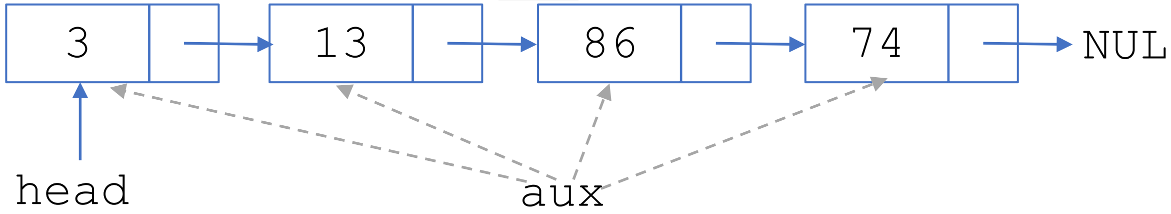


```
void show() {
    TNode *aux;
    aux = head;
    if (isEmpty() == 0) {
        while (aux != nullptr) {
            printf("%d ", aux -> data);
            aux = aux -> next;
        }
        printf("\n");
    } else {
        printf("It's empty\n");
    }
}

```

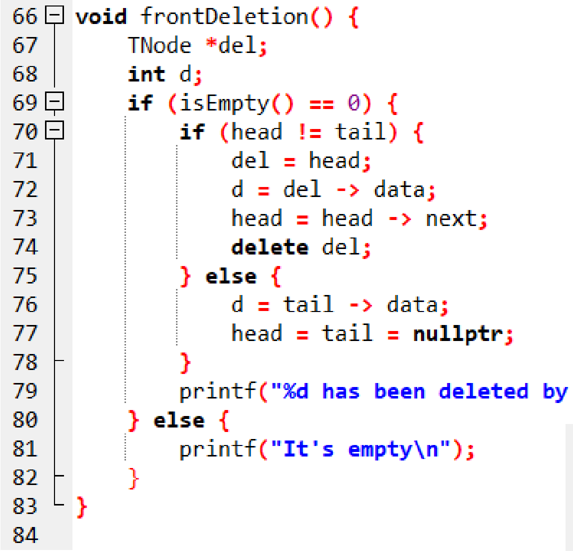


```
int main() {
    printf("Front insertion...\n");
    frontInsertion(13);
    frontInsertion(3);

    printf("\nBack insertion...\n");
    backInsertion(86);
    backInsertion(74);

    printf("\nThe content of the linked list is in the following.\n");
    show();
    return 0;
}
```

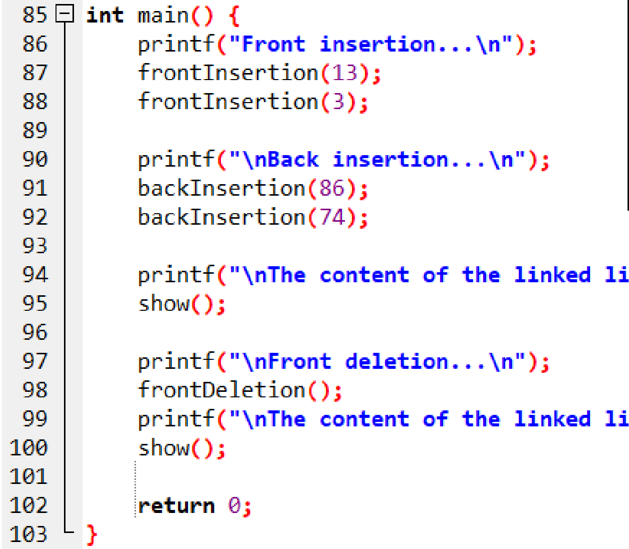
```
Front insertion...
13 has been inserted by frontInsertion()
3 has been inserted by frontInsertion()

Back insertion...
86 has been inserted by backInsertion()
74 has been inserted by backInsertion()

The content of the linked list is in the following.
3 13 86 74
```



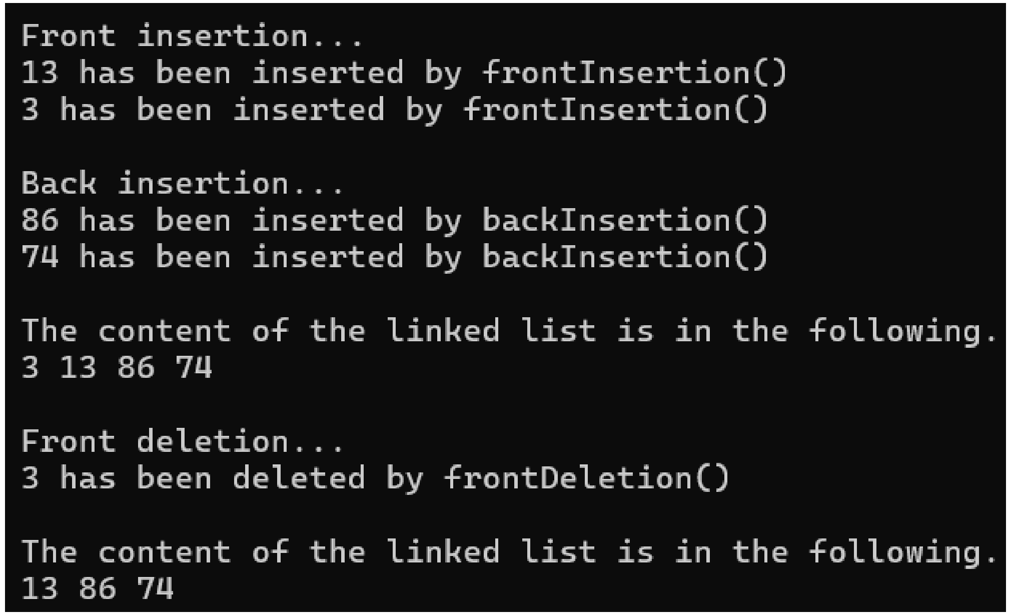
SLLNC with `head` & `tail`: `frontDeletion`



```
void frontDeletion() {
    TNode *del;
    int d;
    if (isEmpty() == 0) {
        if (head != tail) {
            del = head;
            d = del -> data;
            head = head -> next;
            delete del;
        } else {
            d = tail -> data;
            head = tail = nullptr;
        }
        printf("%d has been deleted by frontDeletion()\n", d);
    } else {
        printf("It's empty\n");
    }
}
```

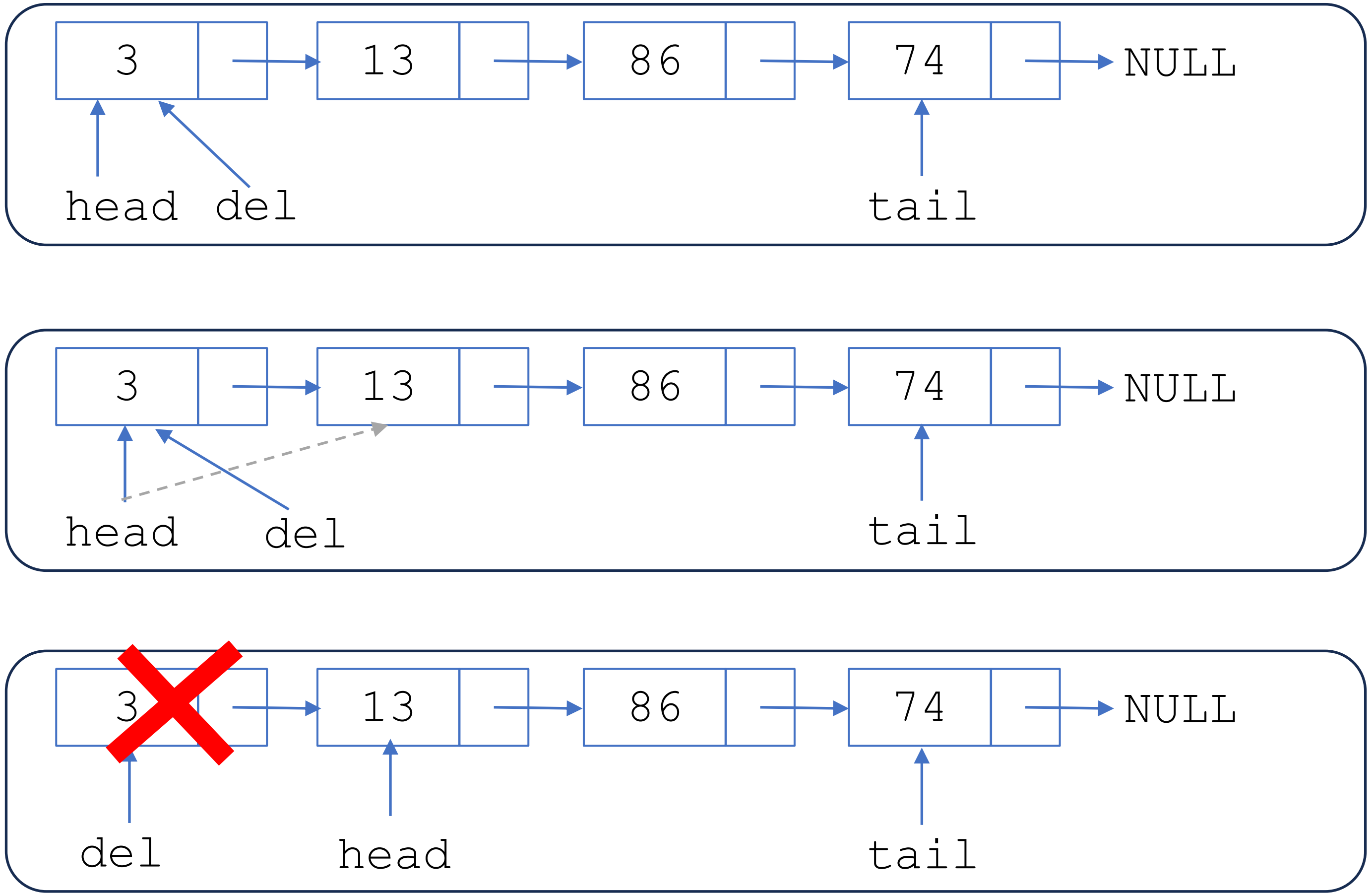


```
int main() {
    printf("Front insertion...\n");
    frontInsertion(13);
    frontInsertion(3);

    printf("\nBack insertion...\n");
    backInsertion(86);
    backInsertion(74);

    printf("\nThe content of the linked list is in the following.\n");
    show();

    printf("\nFront deletion...\n");
    frontDeletion();
    printf("\nThe content of the linked list is in the following.\n");
    show();

    return 0;
}
```



```
Front insertion...
13 has been inserted by frontInsertion()
3 has been inserted by frontInsertion()

Back insertion...
86 has been inserted by backInsertion()
74 has been inserted by backInsertion()

The content of the linked list is in the following.
3 13 86 74

Front deletion...
3 has been deleted by frontDeletion()

The content of the linked list is in the following.
13 86 74
```

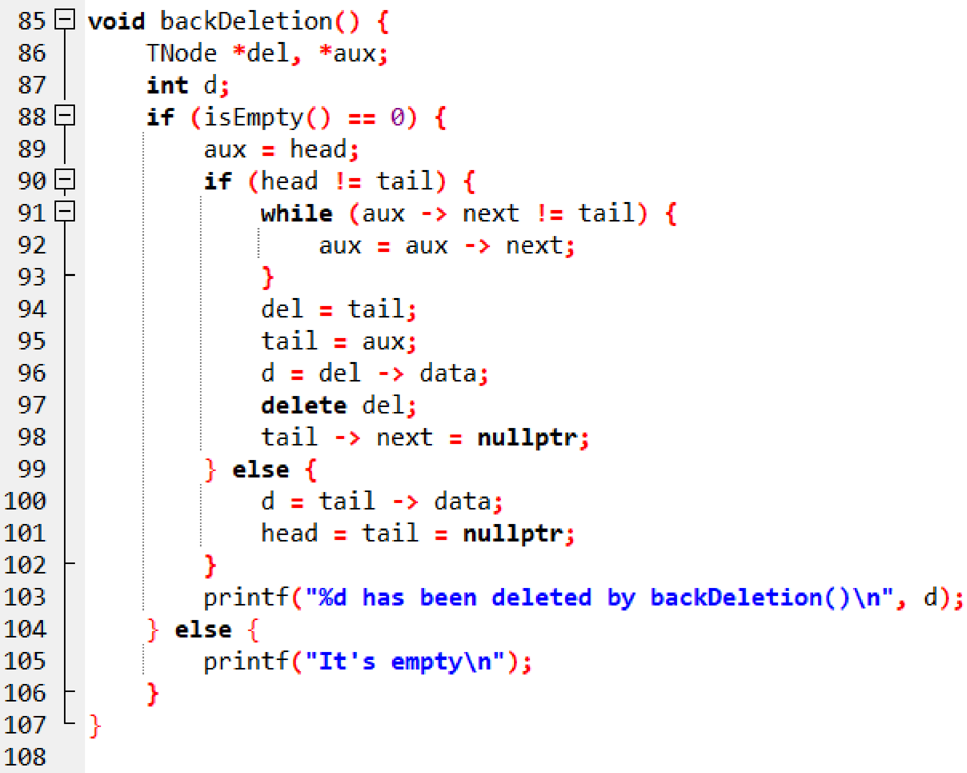
SLLNC with `head` & `tail`: `frontDeletion` (continued)



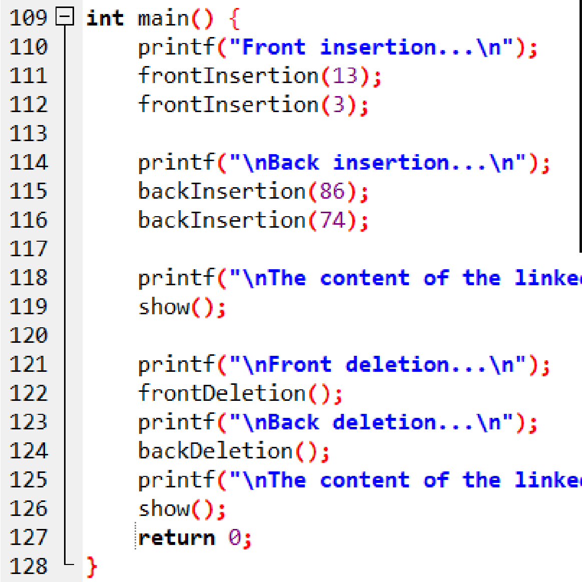
SLLNC with `head` & `tail`: `frontDeletion` (continued)

- The function above will delete the **top** (**first**) data pointed by the **head** in the linked list
- Deleting a node cannot be done if the node is being pointed at by a pointer, so it must be pointed first with the `del` pointer on the `head`, then shift the `head` to the next node so that the data after the `head` becomes the **new head**, then delete the `del` pointer using the **delete** command.
- If `tail` is still NULL then it means the data is still empty

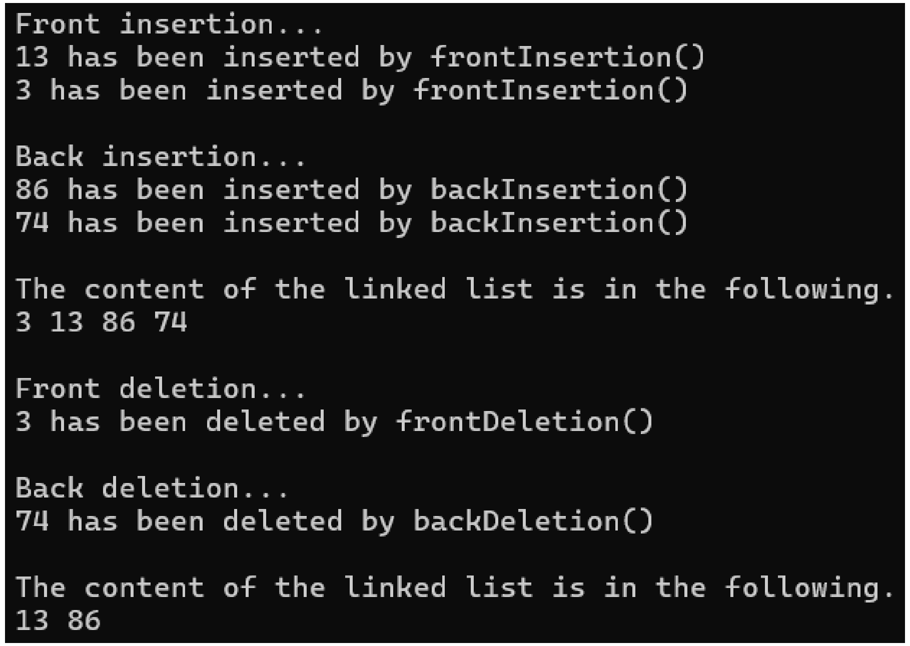
SLLNC with head & tail: backDeletion



```
void backDeletion() {
    TNode *del, *aux;
    int d;
    if (isEmpty() == 0) {
        aux = head;
        if (head != tail) {
            while (aux -> next != tail) {
                aux = aux -> next;
            }
            del = tail;
            tail = aux;
            d = del -> data;
            delete del;
            tail -> next = nullptr;
        } else {
            d = tail -> data;
            head = tail = nullptr;
        }
        printf("%d has been deleted by backDeletion()\n", d);
    } else {
        printf("It's empty\n");
    }
}
```

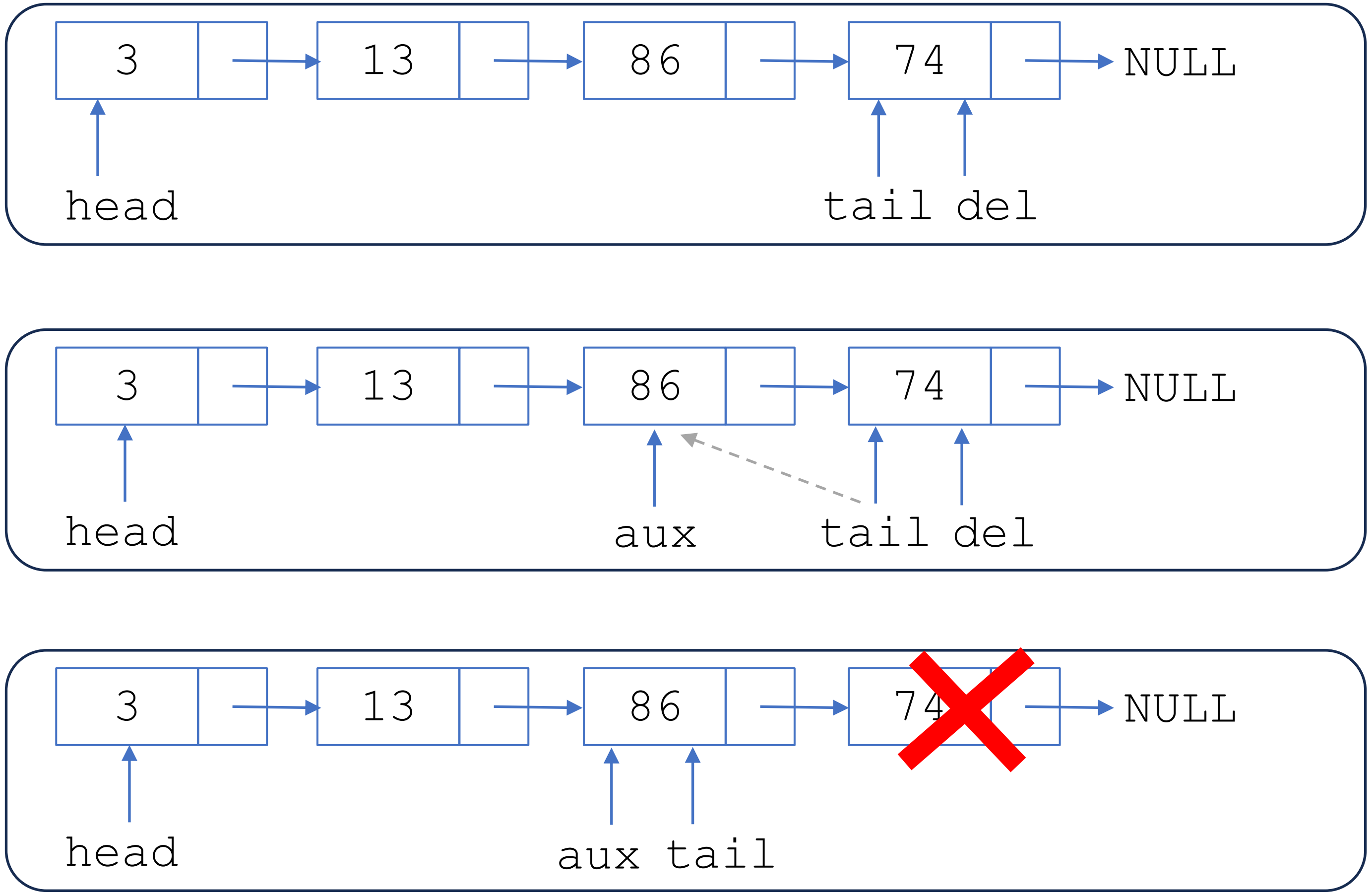


```
int main() {
    printf("Front insertion...\n");
    frontInsertion(13);
    frontInsertion(3);

    printf("\nBack insertion...\n");
    backInsertion(86);
    backInsertion(74);

    printf("\nThe content of the linked list is in the following.\n");
    show();

    printf("\nFront deletion...\n");
    frontDeletion();
    printf("\nBack deletion...\n");
    backDeletion();
    printf("\nThe content of the linked list is in the following.\n");
    show();
    return 0;
}
```



```
Front insertion...
13 has been inserted by frontInsertion()
3 has been inserted by frontInsertion()

Back insertion...
86 has been inserted by backInsertion()
74 has been inserted by backInsertion()

The content of the linked list is in the following.
3 13 86 74

Front deletion...
3 has been deleted by frontDeletion()

Back deletion...
74 has been deleted by backDeletion()

The content of the linked list is in the following.
13 86
```

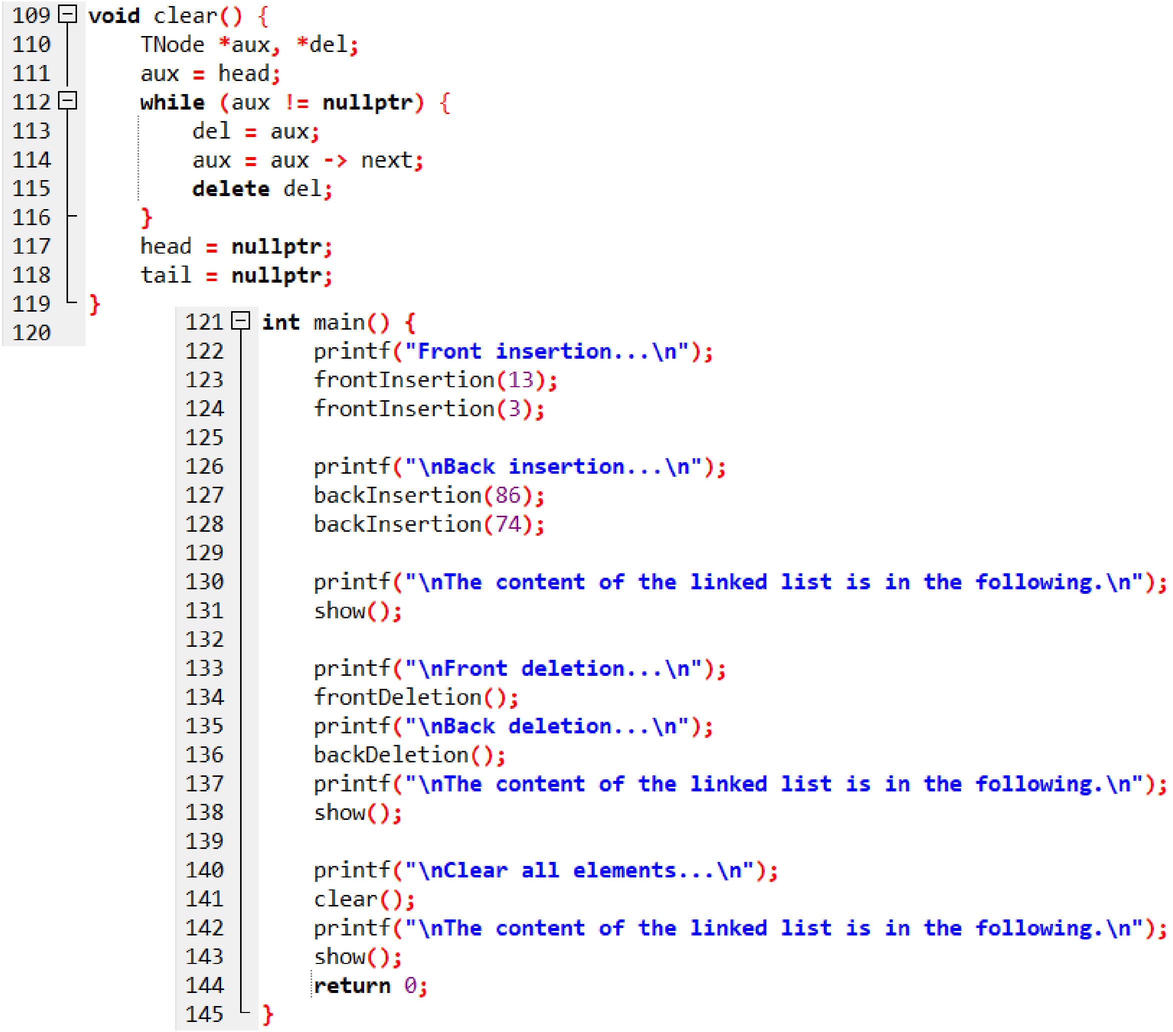
SLLNC with head & tail: backDeletion (continued)



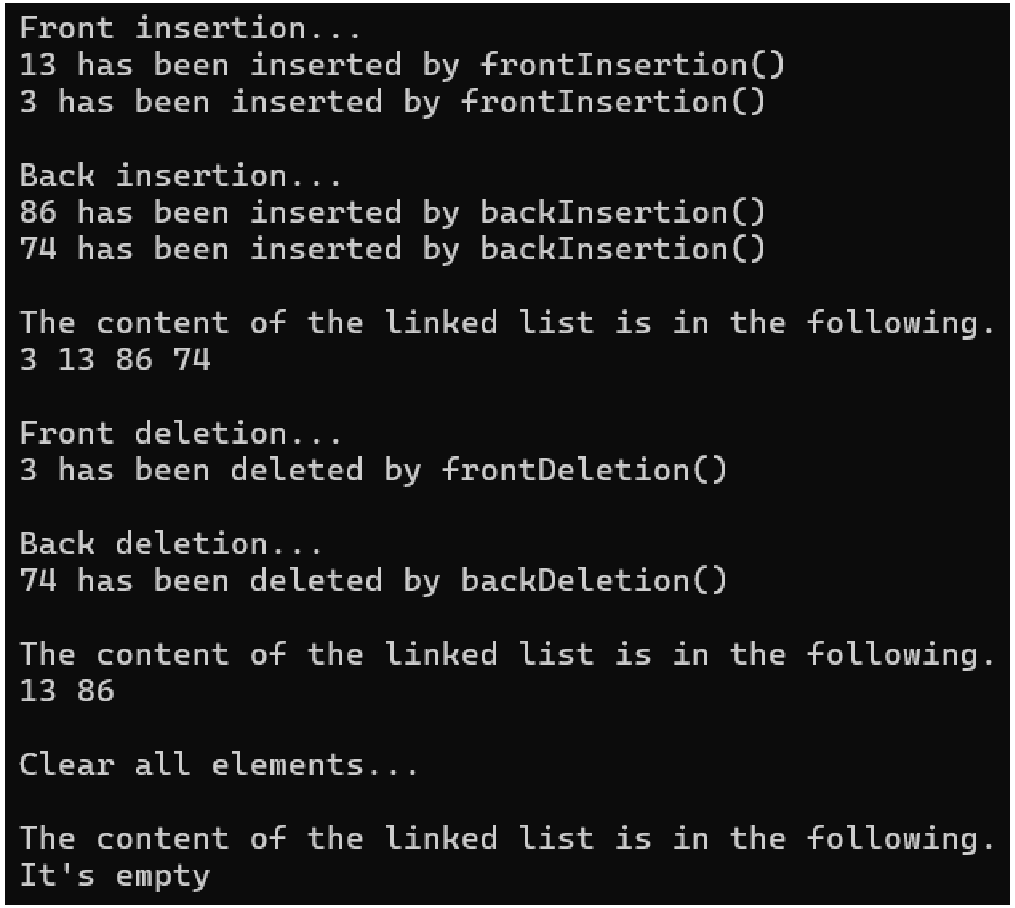
SLLNC with `head` & `tail`: `backDeletion` (continued)

- The function above will delete the **last data** indicated by `tail` in the linked list
- Deleting a node cannot be done if the node's state is being pointed to by a pointer, so it must be pointed out first with the `del` variable in the `tail`, then an `aux` pointer is needed to help shift from the `head` to the next node until before the `tail`, so that the `tail` can be pointed to the `aux`, and the `aux` will become the **new tail**. Furthermore, delete the `del` pointer using the **delete** command.
- If `tail` is still NULL then it means the data is still empty

SLLNC with head & tail: clear



```
void clear() {
    TNode *aux, *del;
    aux = head;
    while (aux != nullptr) {
        del = aux;
        aux = aux -> next;
        delete del;
    }
    head = nullptr;
    tail = nullptr;
}

int main() {
    printf("Front insertion...\n");
    frontInsertion(13);
    frontInsertion(3);

    printf("\nBack insertion...\n");
    backInsertion(86);
    backInsertion(74);

    printf("\nThe content of the linked list is in the following.\n");
    show();

    printf("\nFront deletion...\n");
    frontDeletion();
    printf("\nBack deletion...\n");
    backDeletion();
    printf("\nThe content of the linked list is in the following.\n");
    show();

    printf("\nClear all elements...\n");
    clear();
    printf("\nThe content of the linked list is in the following.\n");
    show();
    return 0;
}
```



```
Front insertion...
13 has been inserted by frontInsertion()
3 has been inserted by frontInsertion()

Back insertion...
86 has been inserted by backInsertion()
74 has been inserted by backInsertion()

The content of the linked list is in the following.
3 13 86 74

Front deletion...
3 has been deleted by frontDeletion()

Back deletion...
74 has been deleted by backDeletion()

The content of the linked list is in the following.
13 86

Clear all elements...

The content of the linked list is in the following.
It's empty
```

Exercise

- Make a **complete program** from all the algorithms and functions above in the form of a menu to add data, view data, and delete data
- Create an **additional function** that is useful for searching for data in a linked list either with a **head** or with **head** & **tail**
- Create a function to **delete certain data** in a linked list
- Make **insertion** nodes **after** or **before** certain data

- NEXT
  - Single Linked List Circular (**SLLC**) with **head** & **tail**