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

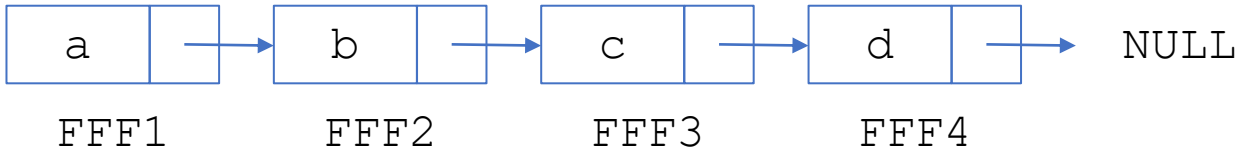
Array vs Linked List

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

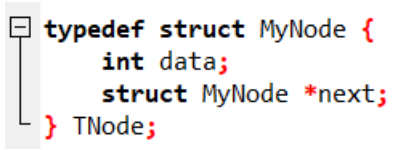


- Each node in a linked list has a field that contains a pointer to the next node, and also has a field that contains data
- The last node will point to `NULL` which will be used as a **stop condition** when reading the content of the linked list
- Keyword for `NULL` in C++ is `nullptr`

Single Linked List Non Circular: Create

- Node Declaration

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

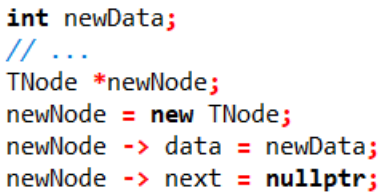


- Explanation
  - Creating a struct called `TNode` which contains 2 fields, namely the `data` field of type integer and the `next` field which is of type pointer from `TNode`
  - After creating the struct, create a `head` variable of type pointer from `TNode` which is useful as the **head** of the linked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



Pointer Allocation: Another Way

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

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

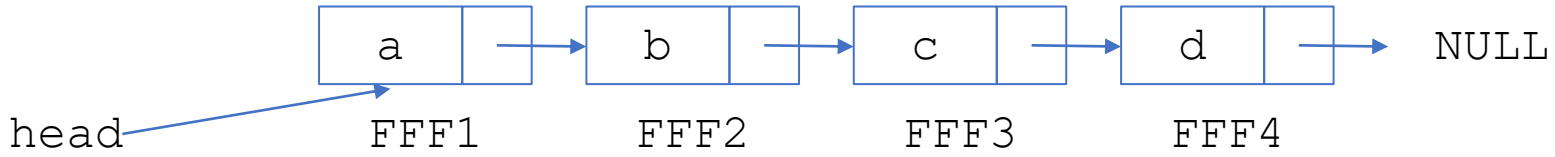
Pointer Program Example

```c
#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**



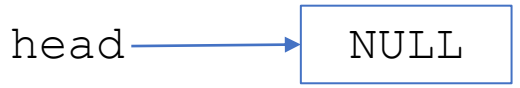
- Single Linked List Headed Pointer Declaration
  - Manipulation of linked list cannot be done directly to the destination node, but must use a pointer to the first node in the linked list (in this case is **head**)
  - The declaration is as follows

```
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)

```
1   #include <stdio.h>
2   #include <iostream>
3   using namespace std;
4
5   typedef struct MyNode {
6       int data;
7       struct MyNode *next;
8   } TNode;
9
10  TNode *head;
11
12  void init() {
13      head = nullptr;
14  }
15
16  int isEmpty() {
17      if (head == nullptr) return 1;
18      else return 0;
19  }
20
```

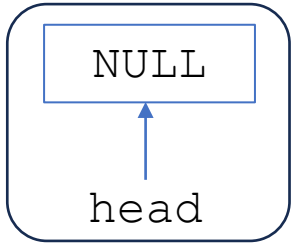
```
21  void frontInsertion(int newData) {
22      TNode *newNode;
23      newNode = new TNode;
24      newNode -> data = newData;
25      newNode -> next = nullptr;
26      if (isEmpty() == 1) {
27          head = newNode;
28          head -> next = nullptr;
29      } else {
30          newNode -> next = head;
31          head = newNode;
32      }
33      printf("%d has been inserted by frontInsertion()\n", newData);
34  }
35
36  int main() {
37      printf("Front insertion...\n");
38      frontInsertion(3);
39      return 0;
40  }
```

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

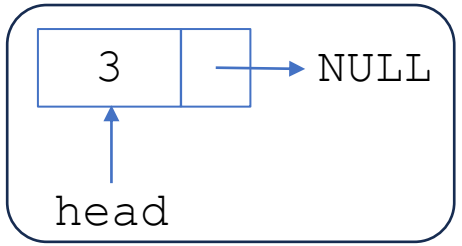
Headed SLLNC: `frontInsertion` (cont'd)

- Illustration

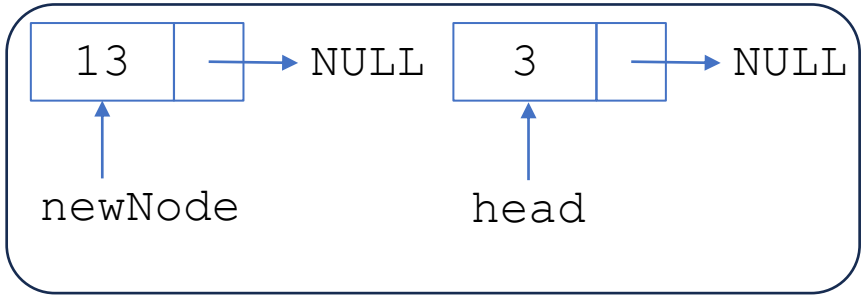
1. The list is still empty.

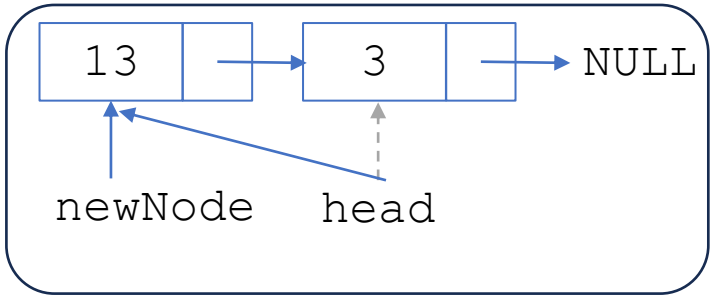


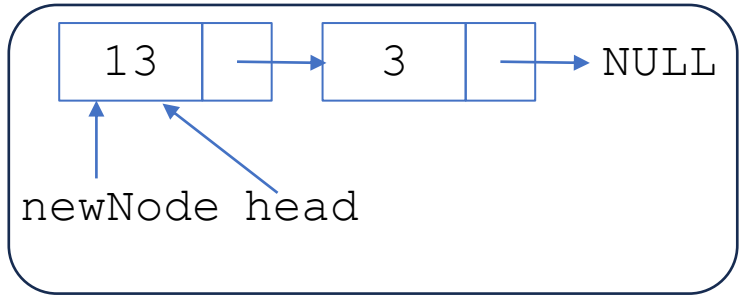
2. Enter the new data, e.g., 3.



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)

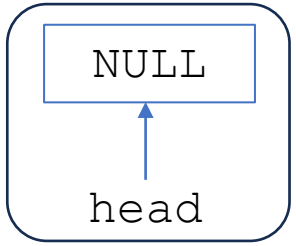
```
36  void backInsertion(int newData) {
37      TNode *newNode, *aux;
38      newNode = new TNode;
39      newNode -> data = newData;
40      newNode -> next = nullptr;
41      if (isEmpty() == 1) {
42          head = newNode;
43          head -> next = nullptr;
44      } else {
45          aux = head;
46          while (aux -> next != nullptr) {
47              aux = aux -> next;
48          }
49          aux -> next = newNode;
50      }
51      printf("%d has been inserted by backInsertion()\n", newData);
52  }
53
54  int main() {
55      printf("Back insertion...\n");
56      backInsertion(13);
57      return 0;
58  }
```

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

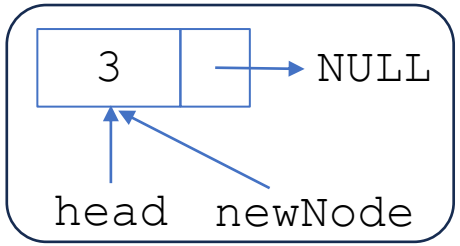
Headed SLLNC: `backInsertion` (cont'd)

- Illustration

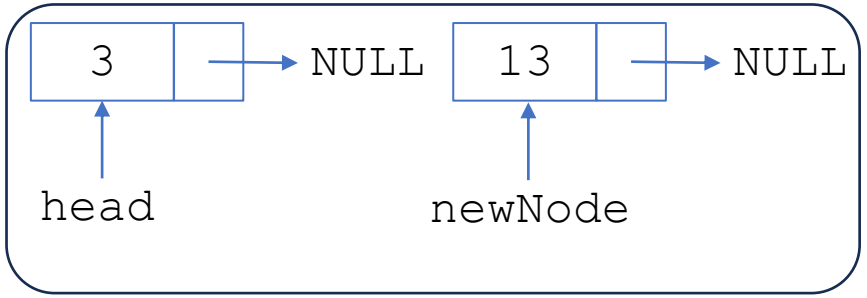
1. The list is still empty (head = NULL).

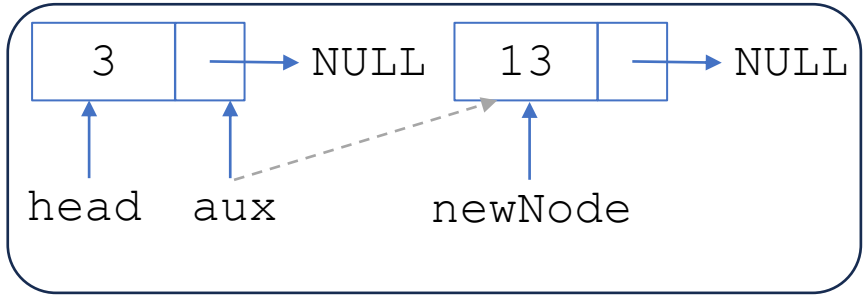


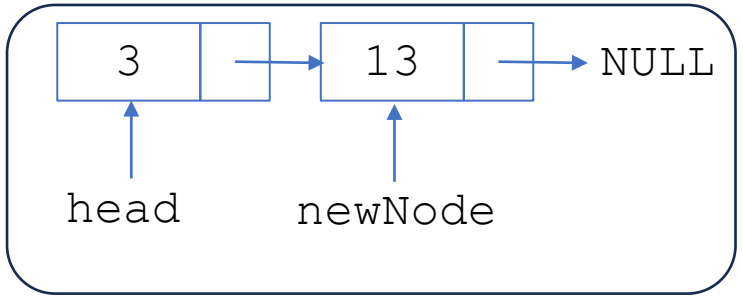
2. Enter the new data, e.g., 3.



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

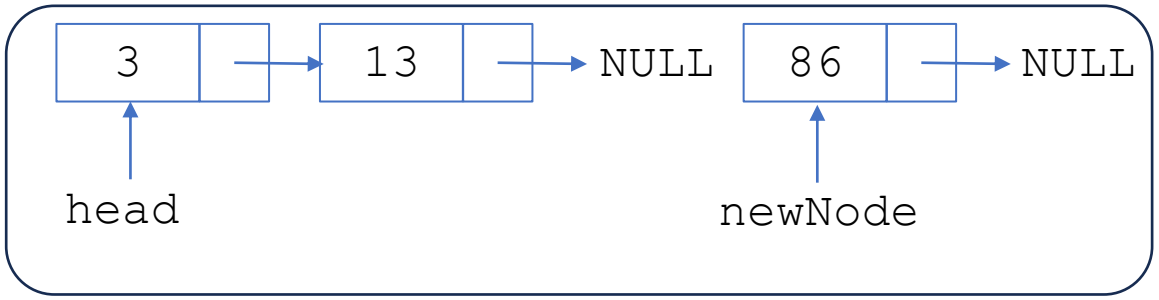


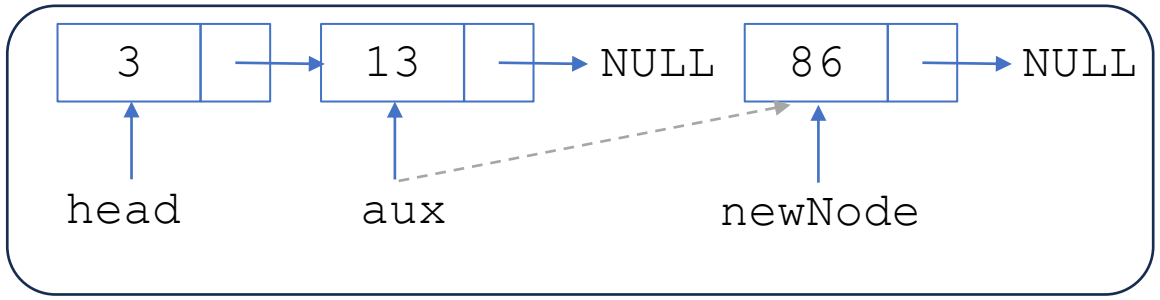


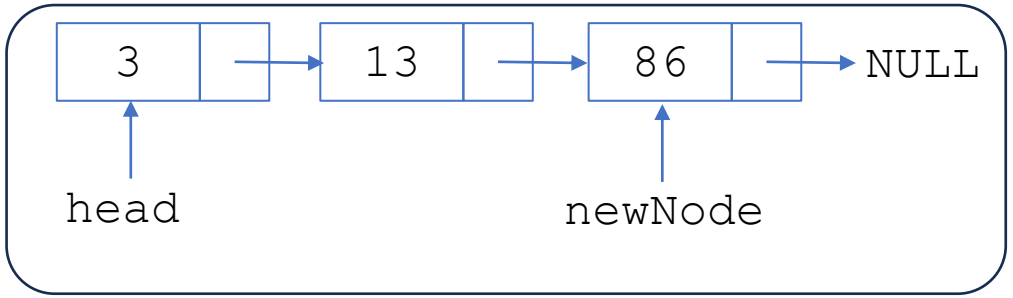


Headed SLLNC: `backInsertion` (cont'd)

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







How about the insertion at the middle?

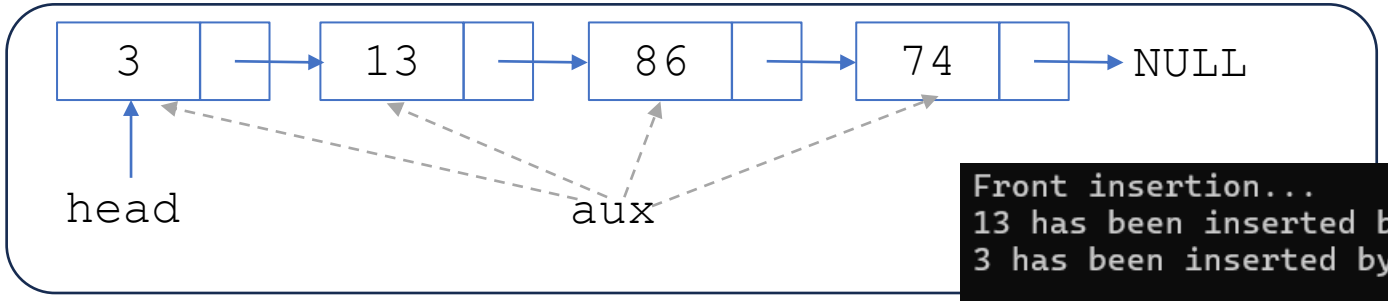
Headed SLLNC: show

```cpp
#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;
}
```

```cpp
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);
}
```

```cpp
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);
}
```

```cpp
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");
    }
}
```




```cpp
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
```

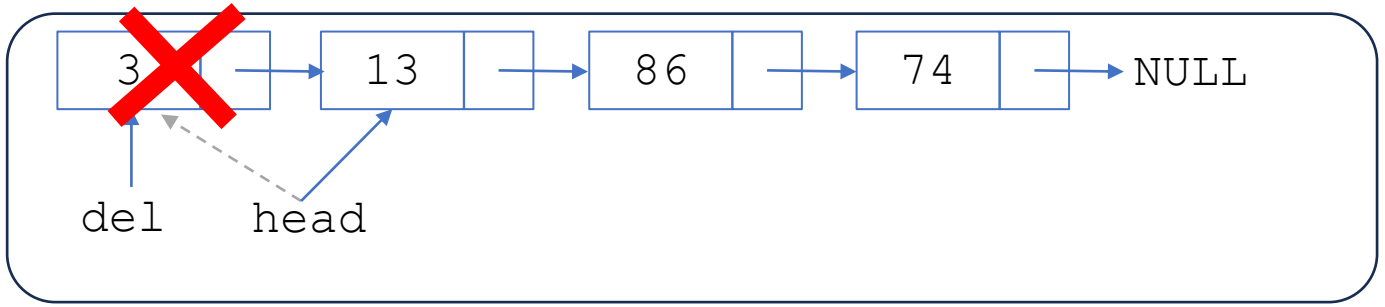
Headed SLLNC: show (continued)

- 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

```
68  void frontDeletion() {
69      TNode *del;
70      int d;
71      if (isEmpty() == 0) {
72          if (head -> next != nullptr) {
73              del = head;
74              d = del -> data;
75              head = head -> next;
76              delete del;
77          } else {
78              d = head -> data;
79              head = nullptr;
80          }
81          printf("%d has been deleted by frontDeletion()\n", d);
82      } else {
83          printf("It's empty\n");
84      }
85  }
86
```

```
87  int main() {
88      printf("Front insertion...\n");
89      frontInsertion(13);
90      frontInsertion(3);
91
92      printf("\nBack insertion...\n");
93      backInsertion(86);
94      backInsertion(74);
95
96      printf("\nThe content of the linked list is in the following.\n");
97      show();
98
99      printf("\nFront deletion...\n");
100     frontDeletion();
101
102     printf("\nThe content of the linked list is in the following.\n");
103     show();
104     return 0;
105 }
```




```
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
```

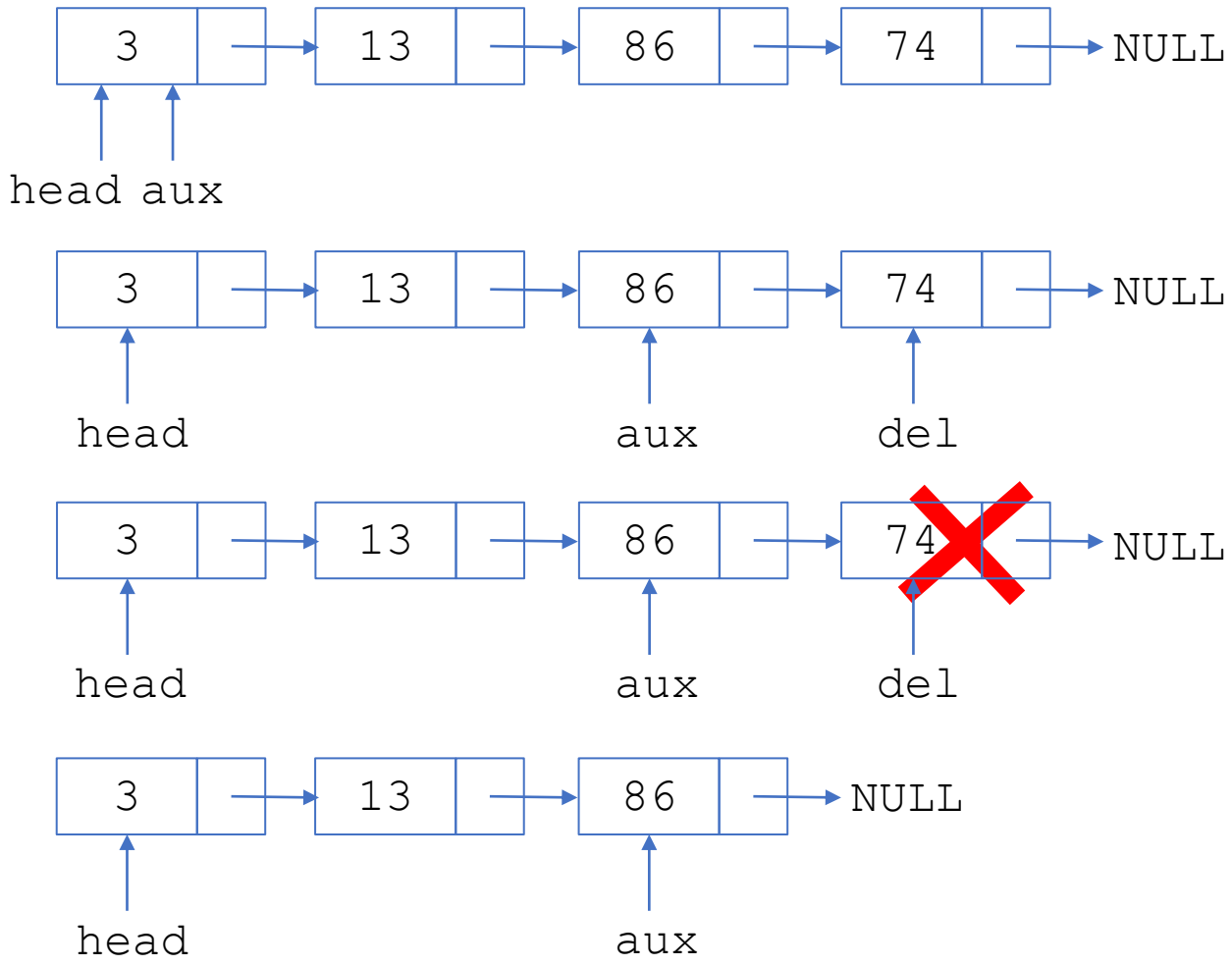
```
87  void backDeletion() {
88      TNode *del, *aux;
89      int d;
90      if (isEmpty() == 0) {
91          if (head -> next != nullptr) {
92              aux = head;
93              while (aux -> next -> next != nullptr) {
94                  aux = aux -> next;
95              }
96              del = aux -> next;
97              d = del -> data;
98              aux -> next = nullptr;
99              delete del;
100         } else {
101             d = head -> data;
102             head = nullptr;
103         }
104         printf("%d has been deleted by backDeletion()\n", d);
105     } else {
106         printf("It's empty\n");
107     }
108 }
109
```

```
110 int main() {
111     printf("Front insertion...\n");
112     frontInsertion(13);
113     frontInsertion(3);
114
115     printf("\nBack insertion...\n");
116     backInsertion(86);
117     backInsertion(74);
118
119     printf("\nThe content of the linked list is in the following.\n");
120     show();
121
122     printf("\nFront deletion...\n");
123     frontDeletion();
124     printf("\nBack deletion...\n");
125     backDeletion();
126
127     printf("\nThe content of the linked list is in the following.\n");
128     show();
129     return 0;
130 }
```

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

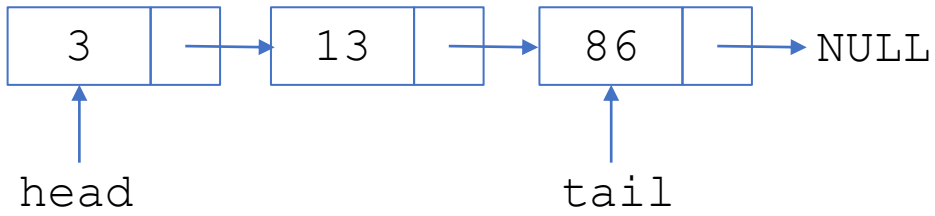
```
110 void clear() {
111     TNode *aux, *del;
112     aux = head;
113     while (aux != nullptr) {
114         del = aux;
115         aux = aux -> next;
116         delete del;
117     }
118     head = nullptr;
119 }
120
```

```
121 int main() {
122     printf("Front insertion...\n");
123     frontInsertion(13);
124     frontInsertion(3);
125
126     printf("\nBack insertion...\n");
127     backInsertion(86);
128     backInsertion(74);
129
130     printf("\nThe content of the linked list is in the following.\n");
131     show();
132
133     printf("\nFront deletion...\n");
134     frontDeletion();
135     printf("\nBack deletion...\n");
136     backDeletion();
137
138     printf("\nThe content of the linked list is in the following.\n");
139     show();
140
141     printf("\nClear all elements...\n");
142     clear();
143     printf("\nThe content of the linked list is in the following.\n");
144     show();
145
146     return 0;
147 }
```

```
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;
}
```

```
1  #include <stdio.h>
2  #include <iostream>
3  using namespace std;
4
5  typedef struct MyNode {
6      int data;
7      struct MyNode *next;
8  } TNode;
9
10 TNode *head, *tail;
11
12 void init() {
13     head = nullptr;
14     tail = nullptr;
15 }
16
17 int isEmpty() {
18     if (head == nullptr) return 1;
19     else return 0;
20 }
21
```

SLLNC with `head` & `tail`: `frontInsertion`

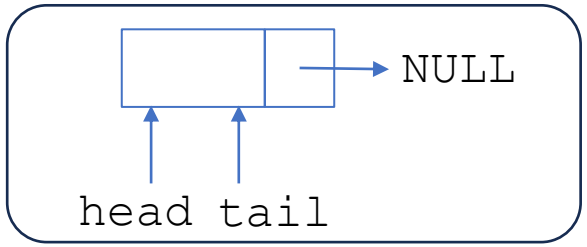
```
22  void frontInsertion(int newData) {
23      TNode *newNode;
24      newNode = new TNode;
25      newNode -> data = newData;
26      newNode -> next = nullptr;
27      if (isEmpty() == 1) {
28          head = tail = newNode;
29          tail -> next = nullptr;
30      } else {
31          newNode -> next = head;
32          head = newNode;
33      }
34      printf("%d has been inserted by frontInsertion()\n", newData);
35  }
36
```

```
37  int main() {
38      printf("Front insertion...\n");
39      frontInsertion(13);
40      frontInsertion(3);
41
42      return 0;
43  }
```

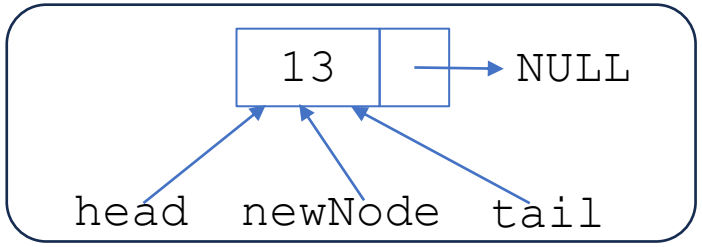
```
Front insertion...
13 has been inserted by frontInsertion()
3 has been inserted by frontInsertion()
```

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

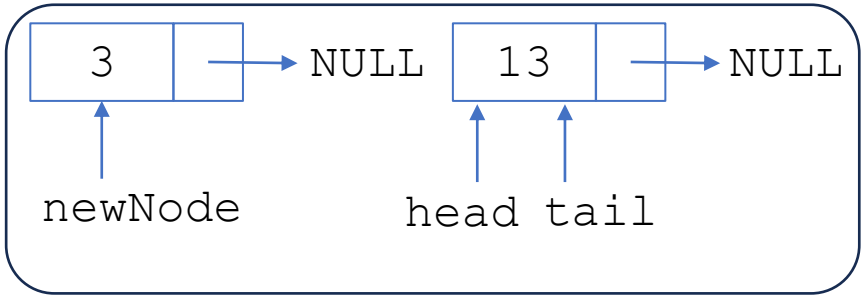
1. The list is still empty (head = tail = NULL).

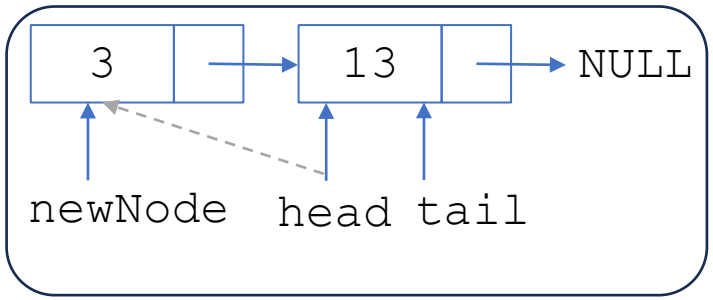


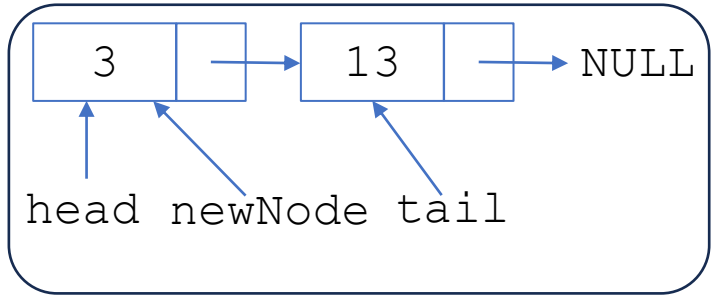
2. Enter the new data, e.g., 13.



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







SLLNC with `head` & `tail`: `backInsertion`

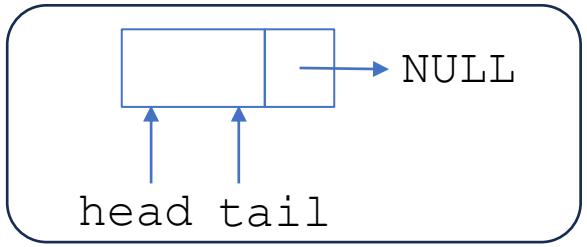
```
37  void backInsertion(int newData) {
38      TNode *newNode;
39      newNode = new TNode;
40      newNode -> data = newData;
41      newNode -> next = nullptr;
42      if (isEmpty() == 1) {
43          head = tail = newNode;
44          tail -> next = nullptr;
45      } else {
46          tail -> next = newNode;
47          tail = newNode;
48      }
49      printf("%d has been inserted by backInsertion()\n", newData);
50  }
51
```

```
52  int main() {
53      printf("Front insertion...\n");
54      frontInsertion(13);
55      frontInsertion(3);
56
57      printf("\nBack insertion...\n");
58      backInsertion(86);
59      backInsertion(74);
60
61      return 0;
62  }
```

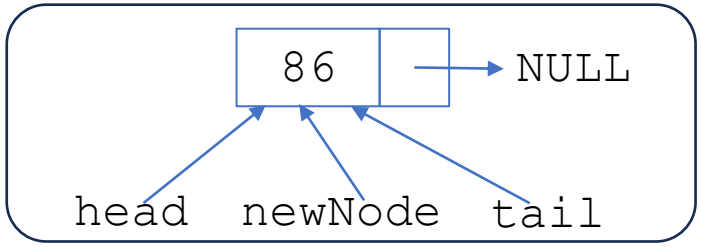
```
Front insertion...
13 has been inserted by frontInsertion()
3 has been inserted by frontInsertion()

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

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

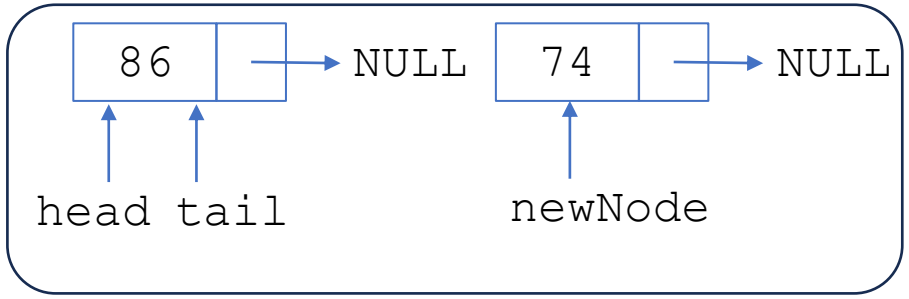
1. The list is still empty (head = tail = NULL).

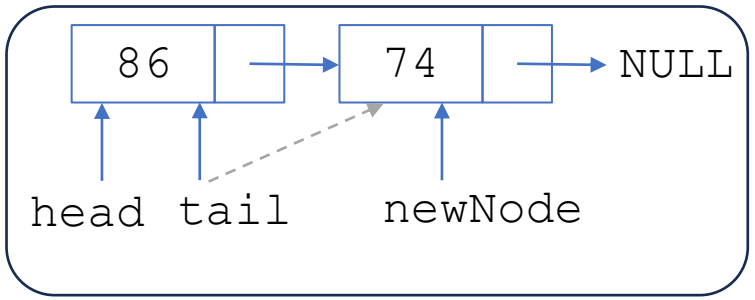


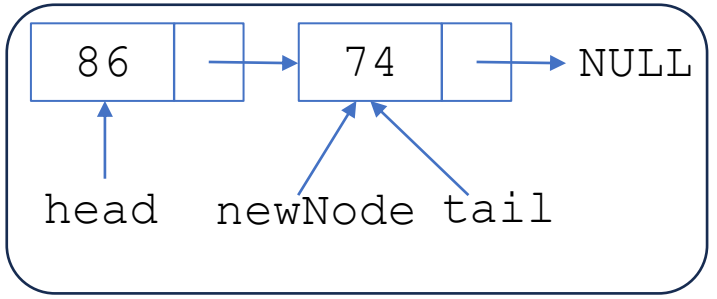
2. Enter the new data, e.g., 86.



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





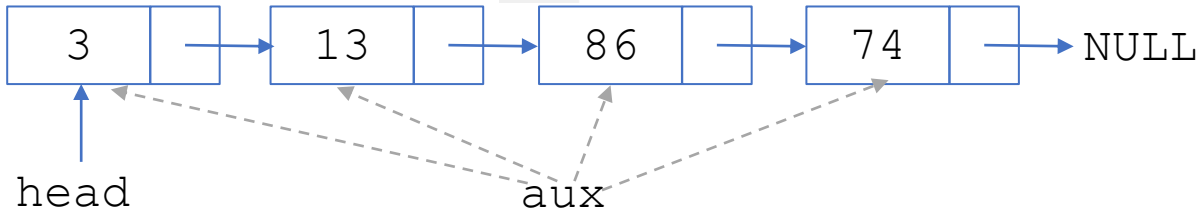


SLLNC with `head` & `tail`: `show`

- 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

```
52  void show() {
53      TNode *aux;
54      aux = head;
55      if (isEmpty() == 0) {
56          while (aux != nullptr) {
57              printf("%d ", aux -> data);
58              aux = aux -> next;
59          }
60          printf("\n");
61      } else {
62          printf("It's empty\n");
63      }
64  }
65
```

```
66  int main() {
67      printf("Front insertion...\n");
68      frontInsertion(13);
69      frontInsertion(3);
70
71      printf("\nBack insertion...\n");
72      backInsertion(86);
73      backInsertion(74);
74
75      printf("\nThe content of the linked list is in the following.\n");
76      show();
77      return 0;
78  }
```




```
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
```

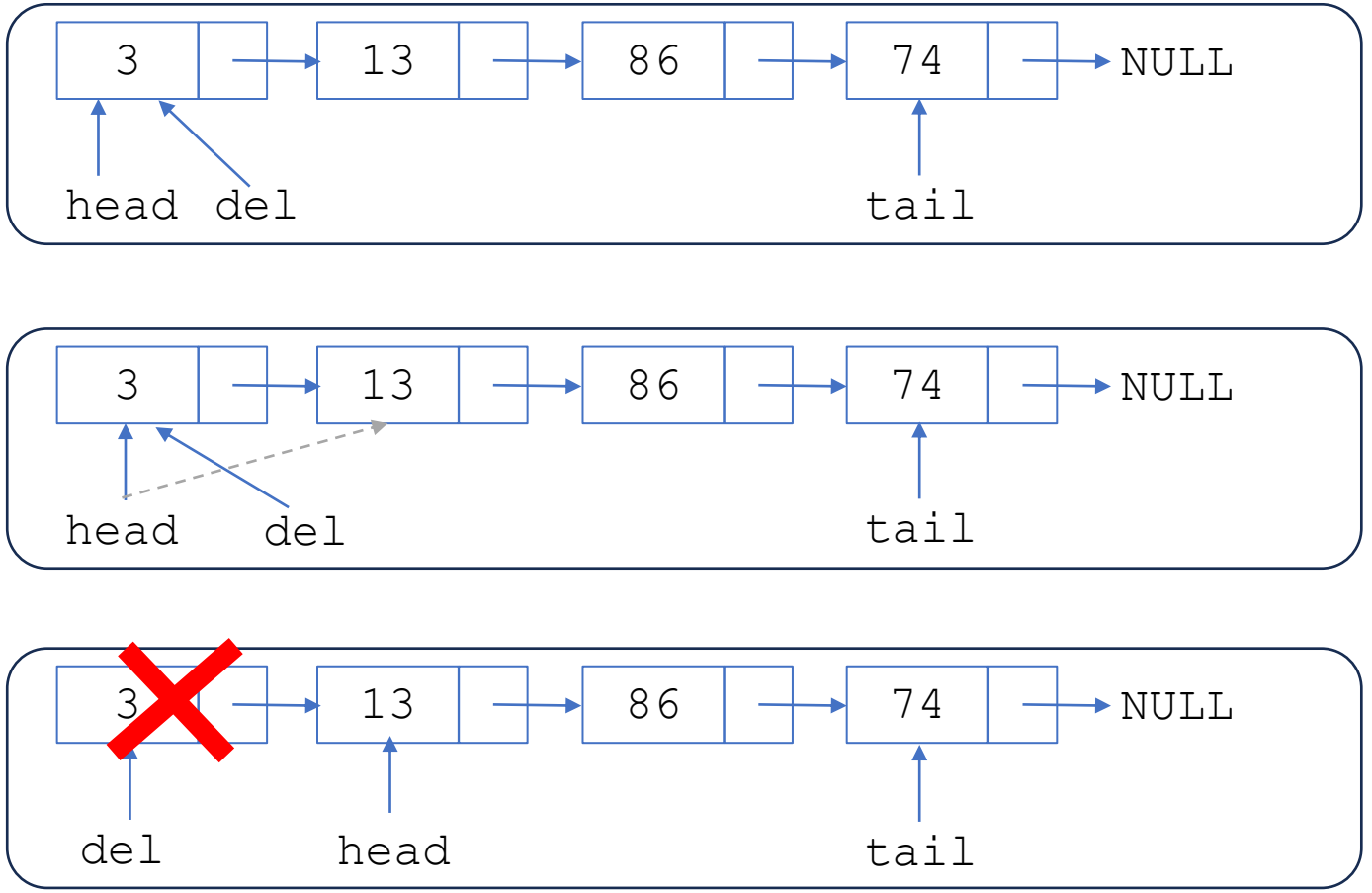
SLLNC with `head` & `tail`: `frontDeletion`

```
66  void frontDeletion() {
67      TNode *del;
68      int d;
69      if (isEmpty() == 0) {
70          if (head != tail) {
71              del = head;
72              d = del -> data;
73              head = head -> next;
74              delete del;
75          } else {
76              d = tail -> data;
77              head = tail = nullptr;
78          }
79          printf("%d has been deleted by frontDeletion()\n", d);
80      } else {
81          printf("It's empty\n");
82      }
83  }
84
```

```
85  int main() {
86      printf("Front insertion...\n");
87      frontInsertion(13);
88      frontInsertion(3);
89
90      printf("\nBack insertion...\n");
91      backInsertion(86);
92      backInsertion(74);
93
94      printf("\nThe content of the linked list is in the following.\n");
95      show();
96
97      printf("\nFront deletion...\n");
98      frontDeletion();
99      printf("\nThe content of the linked list is in the following.\n");
100     show();
101
102     return 0;
103 }
```

```
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

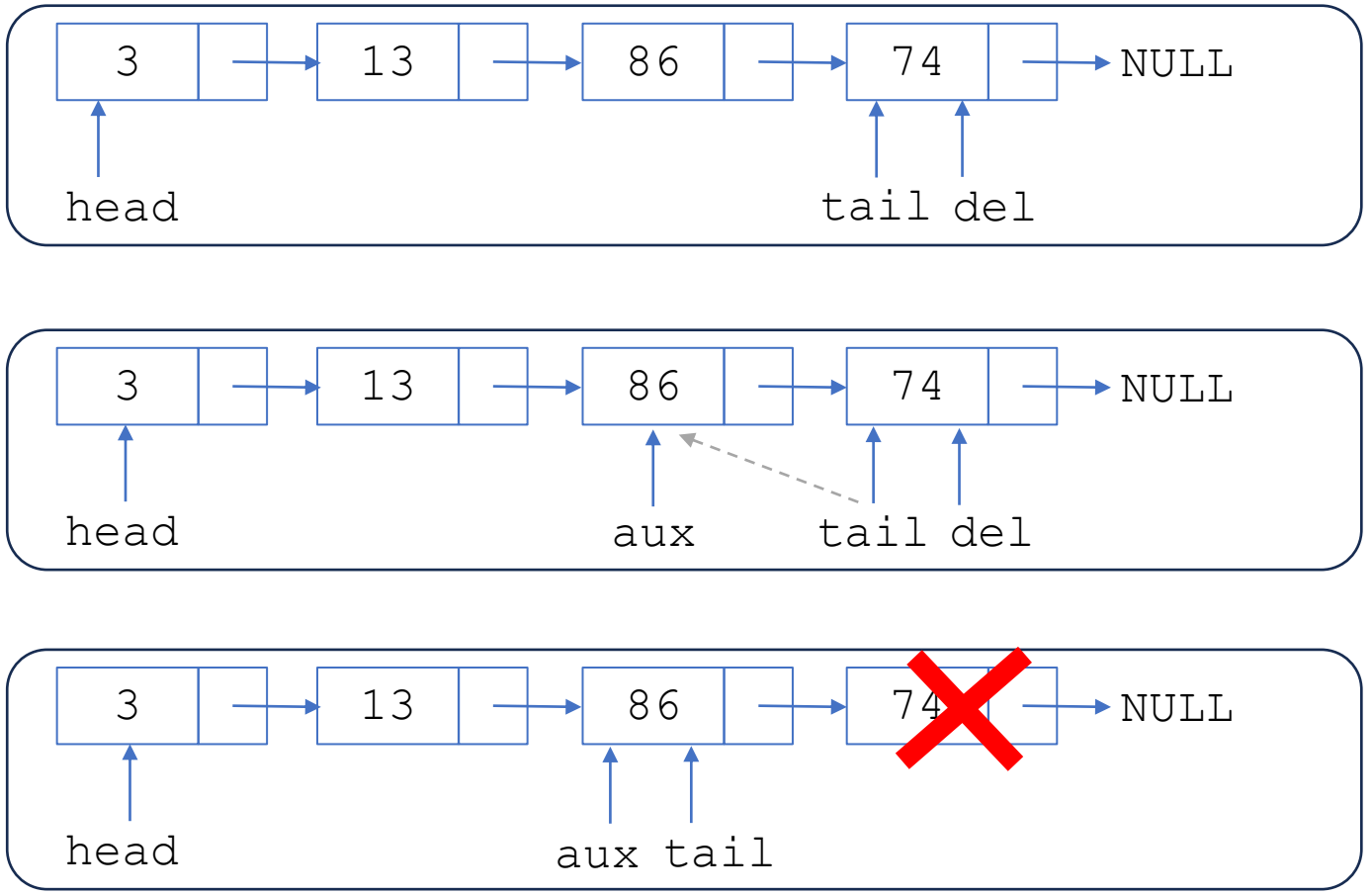
SLLNC with `head` & `tail`: `backDeletion`

```
85  void backDeletion() {
86      TNode *del, *aux;
87      int d;
88      if (isEmpty() == 0) {
89          aux = head;
90          if (head != tail) {
91              while (aux -> next != tail) {
92                  aux = aux -> next;
93              }
94              del = tail;
95              tail = aux;
96              d = del -> data;
97              delete del;
98              tail -> next = nullptr;
99          } else {
100             d = tail -> data;
101             head = tail = nullptr;
102         }
103         printf("%d has been deleted by backDeletion()\n", d);
104     } else {
105         printf("It's empty\n");
106     }
107 }
108
```

```
109 int main() {
110     printf("Front insertion...\n");
111     frontInsertion(13);
112     frontInsertion(3);
113
114     printf("\nBack insertion...\n");
115     backInsertion(86);
116     backInsertion(74);
117
118     printf("\nThe content of the linked list is in the following.\n");
119     show();
120
121     printf("\nFront deletion...\n");
122     frontDeletion();
123     printf("\nBack deletion...\n");
124     backDeletion();
125     printf("\nThe content of the linked list is in the following.\n");
126     show();
127     return 0;
128 }
```

```
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`

```
109 void clear() {
110     TNode *aux, *del;
111     aux = head;
112     while (aux != nullptr) {
113         del = aux;
114         aux = aux -> next;
115         delete del;
116     }
117     head = nullptr;
118     tail = nullptr;
119 }
120
```

```
121 int main() {
122     printf("Front insertion...\n");
123     frontInsertion(13);
124     frontInsertion(3);
125
126     printf("\nBack insertion...\n");
127     backInsertion(86);
128     backInsertion(74);
129
130     printf("\nThe content of the linked list is in the following.\n");
131     show();
132
133     printf("\nFront deletion...\n");
134     frontDeletion();
135     printf("\nBack deletion...\n");
136     backDeletion();
137     printf("\nThe content of the linked list is in the following.\n");
138     show();
139
140     printf("\nClear all elements...\n");
141     clear();
142     printf("\nThe content of the linked list is in the following.\n");
143     show();
144     return 0;
145 }
```

```
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**