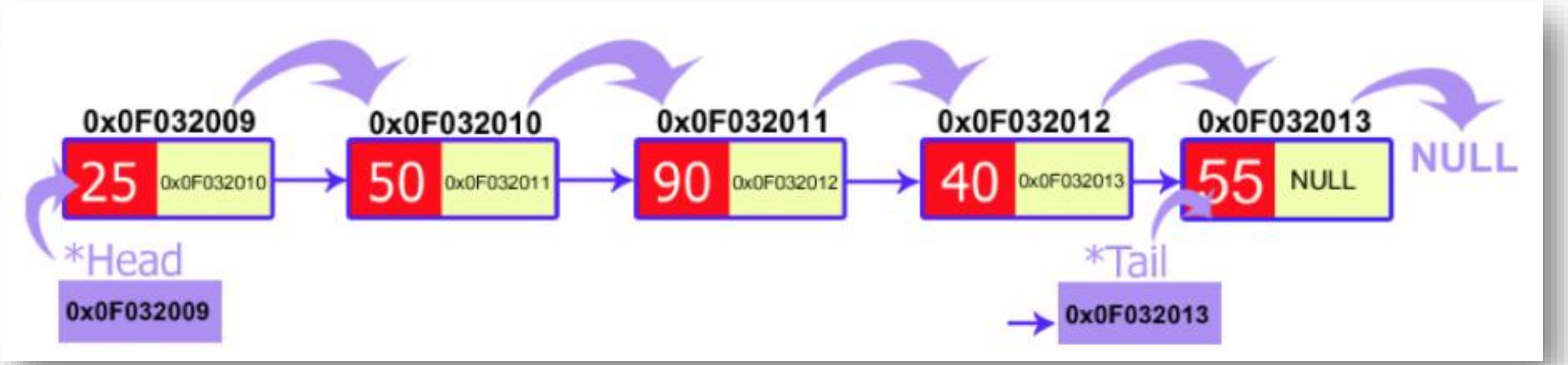
DATA STRUCTURE & PROGRAMMING II

Linked List data structure



Lecture overview

❑ Overall lectures

1. Introduction to algorithm
2. Basic data types and statements
3. Control structures and Loop
4. Array
5. Data structure
6. Sub-programs

7. Recursive
8. File IO
9. Pointers
10. *Linked Lists*
11. Stacks and Queues
12. Sorting algorithms
13. Trees

C++

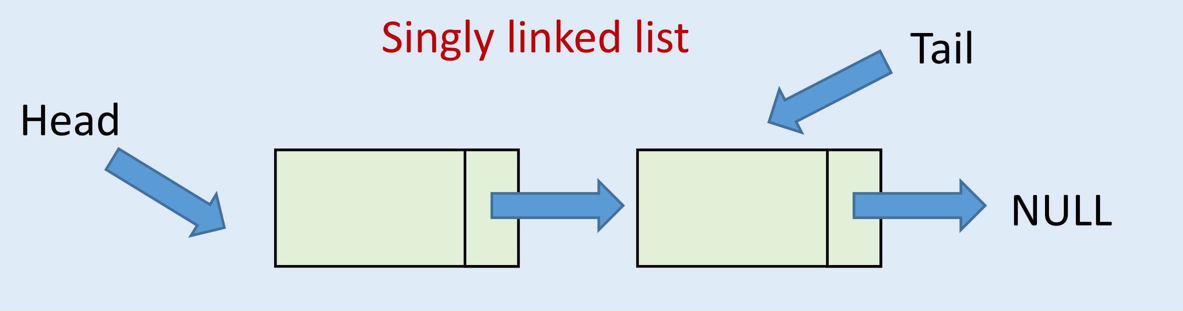
Outline

❑ A Brief of Outline

- What is linked list?
  - Single linked list? Double linked list?
- What are the advantages of using linked list and array?
- Linked list implementation in C++
  - Examples

What is Linked list?

❑ Definition

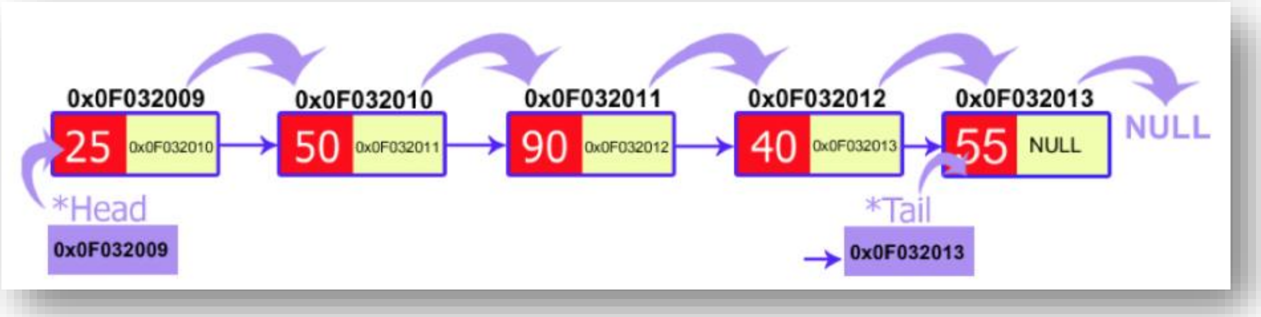


- **A linked list** is a data structure that can store an indefinite amount of elements (dynamic size)
- In a linked list, each element is linked with each other. Elements in a linked list are accessed sequentially.

```
struct Element
    data: integer
    *next: Element
End struct
```

```
struct List
    n: integer
    *head: Element
    *tail: Element
End struct
```

- Each element contains
  - ✓ **Data**
  - ✓ **A link (pointer)**
    - ✓ to its next element (successor)
    - ✓ and/or to its previous element (predecessor)



- Element = called a ***node***
- In linked list, the first element is ***head*** and the last element is ***tail***

Array Vs. Linked List

❑ Pros and Con

Array

- Fixed size
- Once created, can’t add or reduce number of elements to be stored
- Can random access
- Faster access
  - Elements in contiguous memory locations

Linked List

- Dynamically shrink and grow
- Dynamic memory management
- No random access is allowed
- Slower access
  - Elements not in contiguous memory locations

What is Linked list?

❑ Type of Linked List

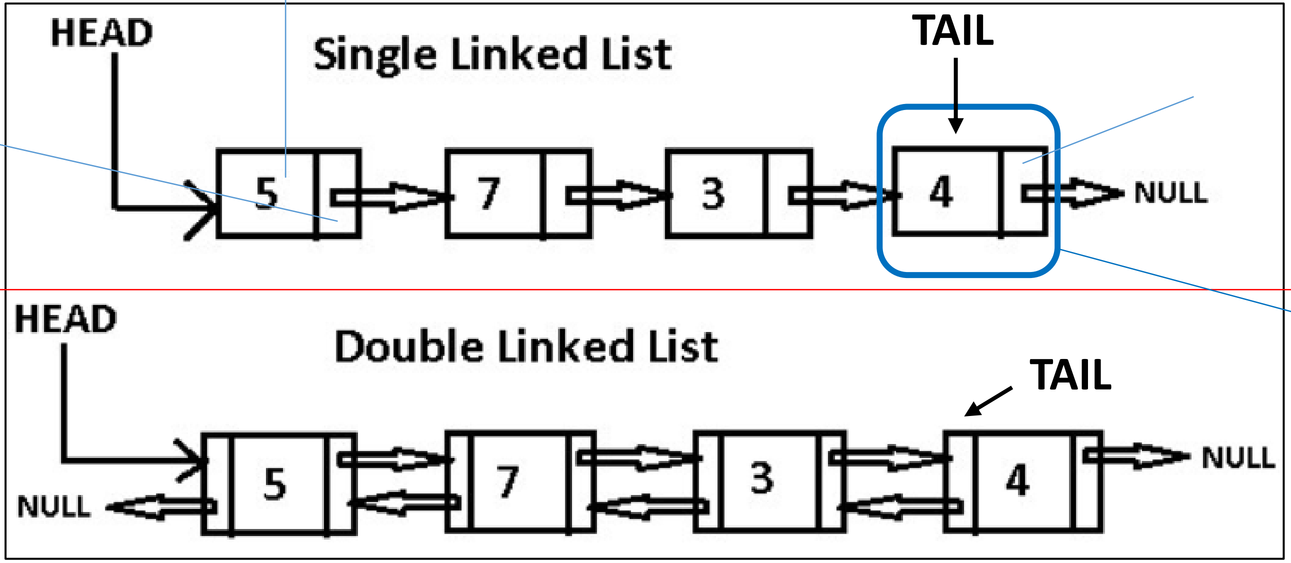
- There are two types of linked lists:
  - A single linked list is a linked list that has **a link to either its successor or predecessor**.
  - A double linked list is a linked list that has **both links** to successor and predecessor.

Data

A pointer points to the next element (successor)

myList1

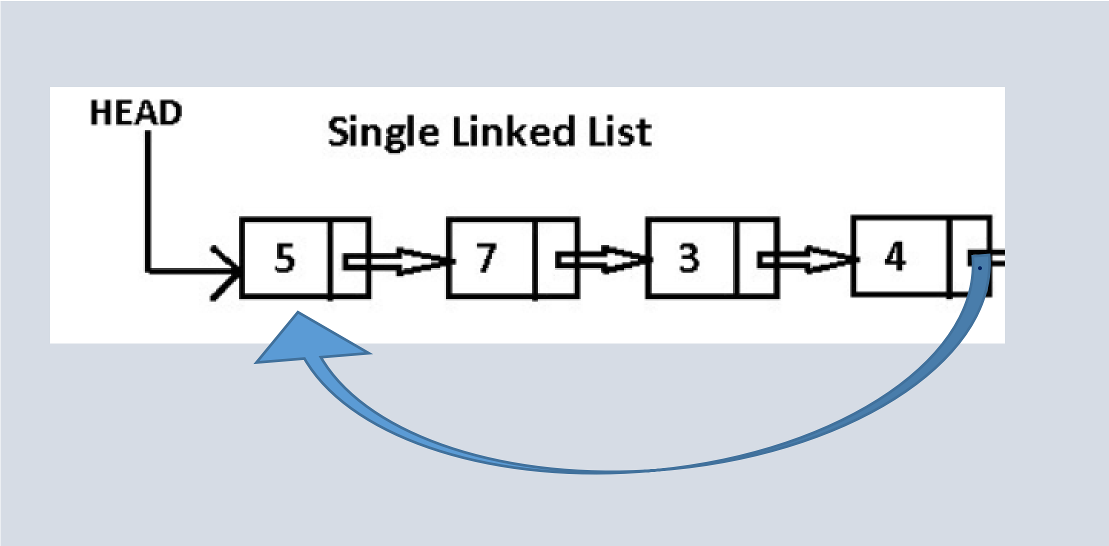
myList2



Last node of the list points to NULL

Remark

- A single or double linked list can be called **a circular linked list** when the last element (tail) points to the first element (head).



Circular linked list

List Operations

❑ Operations with a list

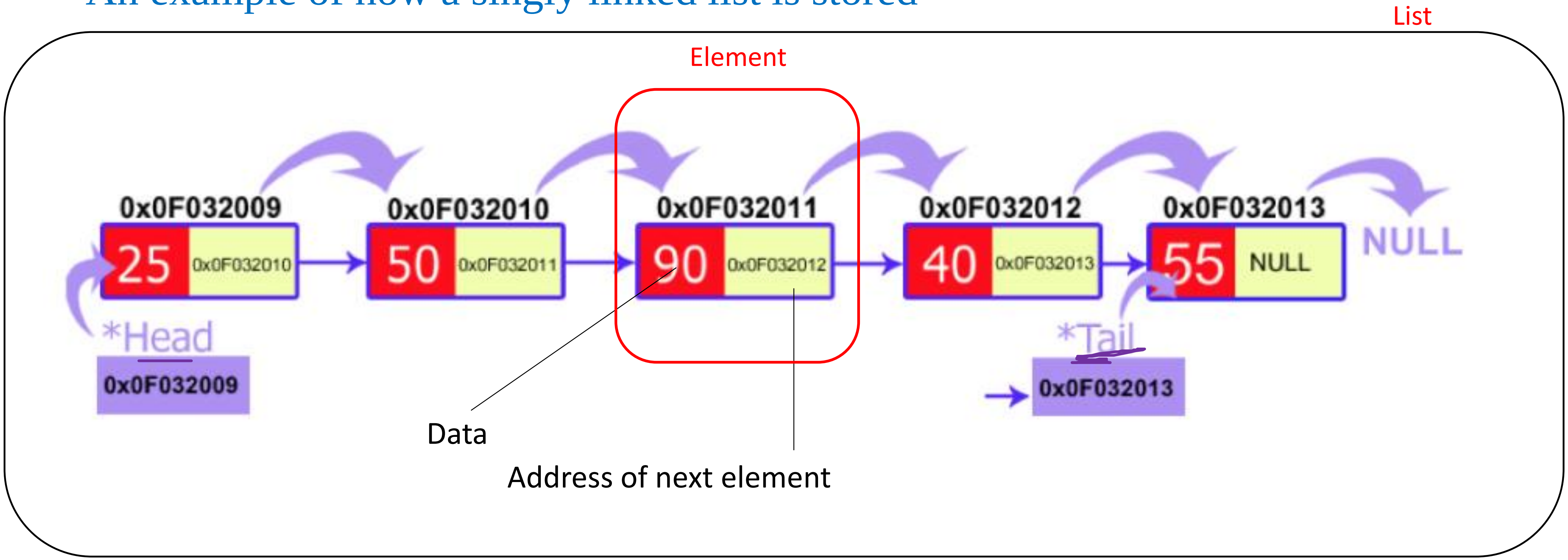
- ✓Creating a list
- ✓Insert a new element to a list
  - ✓ Insert to beginning, end, at a position
- ✓Delete an element from a list
  - ✓ Delete to beginning, end, at a position
- ✓Search an element in a list
- ✓Update an element in a list
- ✓Display data in list
- ✓Reverse a list
- ✓Combine two lists
- ✓… etc.

Singly Linked List (SLL)

Singly linked list

❑ Overview

- An example of how a singly linked list is stored



List operation

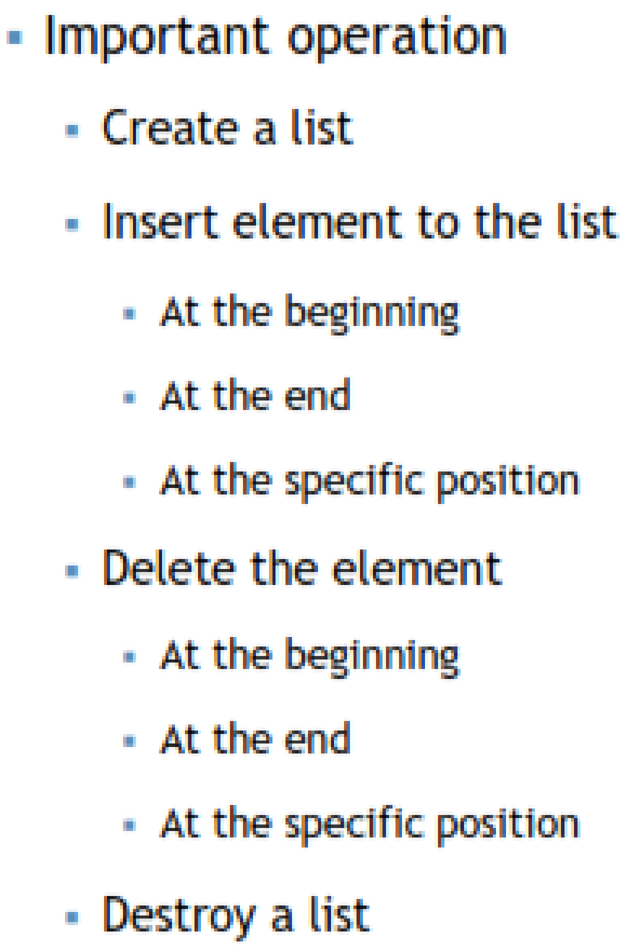
❑ Operation with a list

- All elements of a linked list can be accessed by
    - First setup a pointer pointing to the first element (node) of the list
    - Loop to traverse the list until NULL

- One of the disadvantage of the single linked list is
    - Given a pointer A to a node, we can not reach any of the nodes that precede the node (previous element) to which A is pointing

Operation on linked list

❑ Operations



- Important operation
  - Create a list
  - Insert element to the list
    - At the beginning
    - At the end
    - At the specific position
  - Delete the element
    - At the beginning
    - At the end
    - At the specific position
  - Destroy a list

```
Struct Element
        data: data_type
        *next: Element
End struct
```

```
Struct List
        *head: Element
        *tail: Element
        n: Integer
End struct
```

- **n** store number of elements in list.
- **n** is zero when list is first created. Then n is incremented by 1 when there is an element added to list.

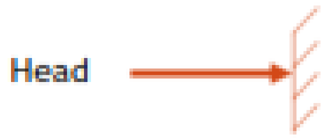
Examples

❑ Create an element

```
Var *head, *tmp : Element
```

- Create an empty list

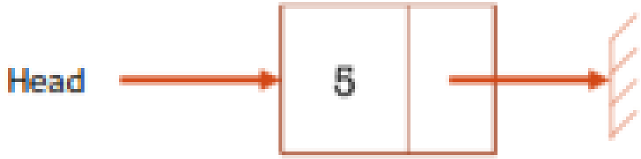
```
head ← null
```



- Add an element of the list with value 5

```
tmp ← new(size(Element))
tmp→ data ← 5
tmp→ next ← null
head ← tmp
```

Reserve/allocate
memory for this element

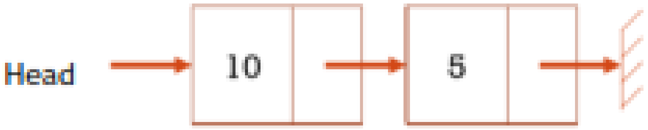


Examples

❑ Add and remove element

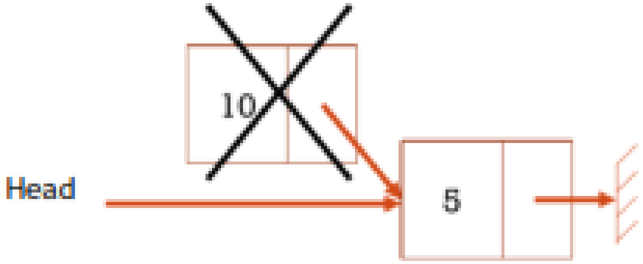
- Add a new element containing value 10 to the beginning of the list

```
tmp ← new(size(Element))
tmp→ data ← 10
tmp→ next ← head
head ← tmp
```



- Delete the first element from the list

```
tmp  ← head
head ← head → next
free(tmp)
```



Create a list

❑ A function to create an empty list

```
Function create_list( ) : Pointer of List
            var  *ls :  List

            ls ← new(size(List))
            ls→n  ←  0
            ls→head ← null
            ls→tail ← null

            return ls

End function
```

Steps to create an empty list:

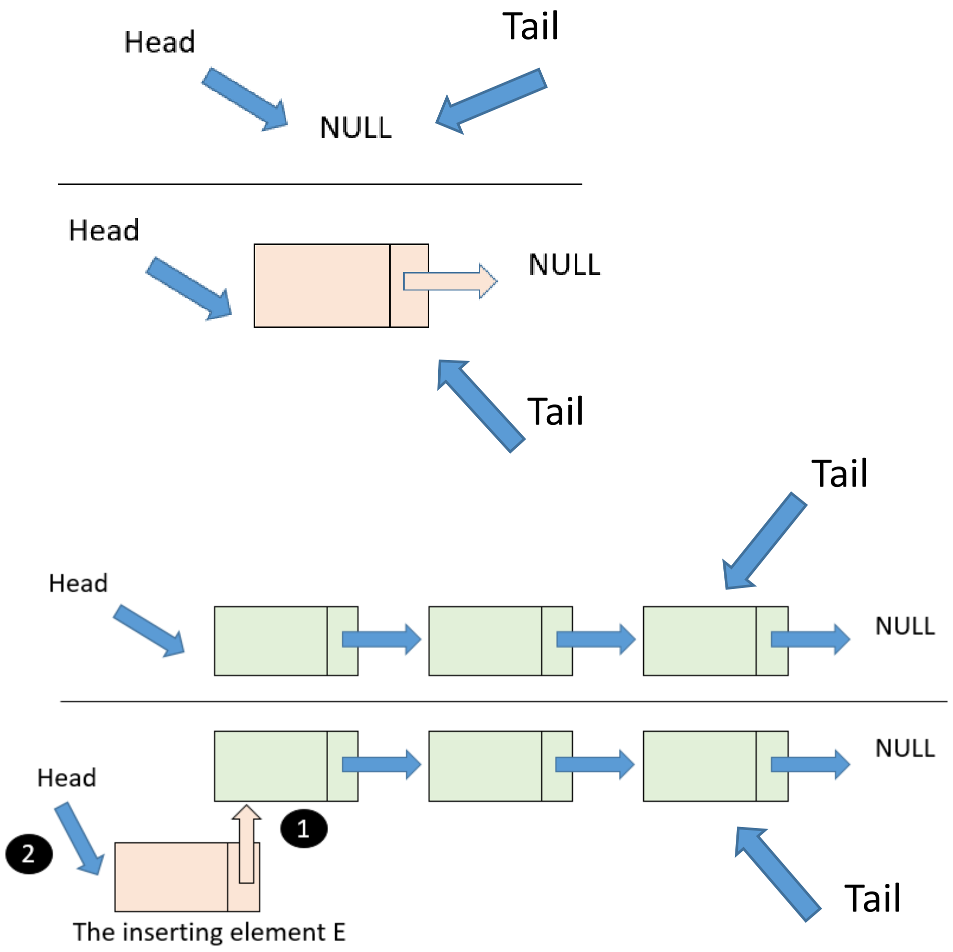
1. Create a list variable
2. Allocate memory
3. Set 0 to n since we are creating an empty list
4. Head points to **null**
5. Tail points to **null**

Insertion

❑ Insert an element to the beginning of the list

```
Procedure insert_be(*ls: List, d: data_type)
        var  *E: Element
   1    E ← new(size(Element))
        E→data ← d

   2    E→next ← ls→head
   3    ls→head ← E
        if(ls→n ==0) then
   4        ls→tail ← E
        end if
   5    ls→n ← ls→n + 1
End procedure
```



Steps to add element to beginning of list

1. Create a new element E
2. Make next pointer of E points to head of list
3. Update E to be head of list
4. Update tail if needed
5. Increase n by 1 (n is number of elements in list)

Display elements in list

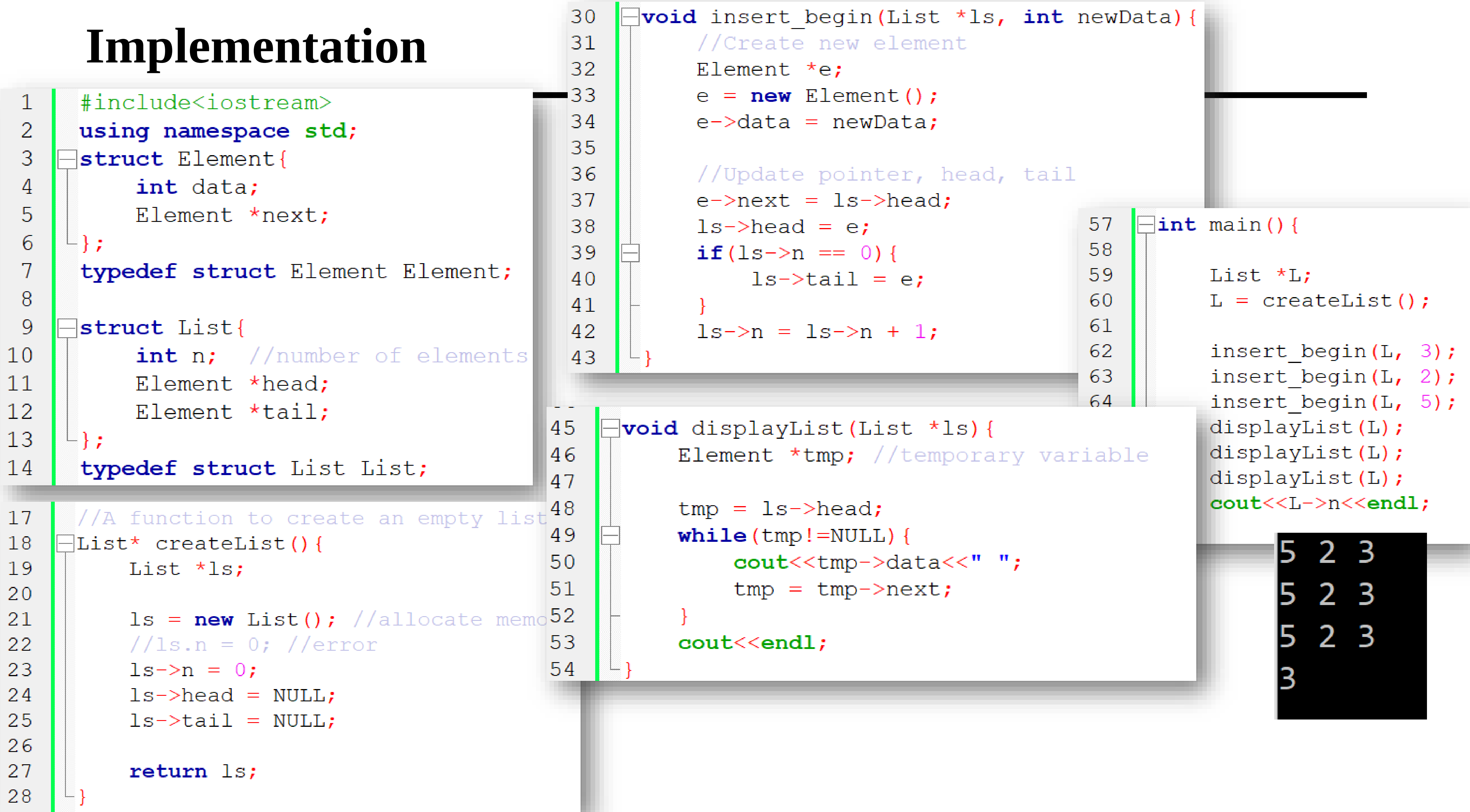
```
Procedure void(*ls: List)
        var *tmp: Element
        tmp ← ls→head

        while(tmp!=NULL) do
                write(tmp→data)
                tmp ← tmp→next
        end while
End procedure
```

Steps to display element in list

1. Start from head
2. Move to each element each time
3. …
4. …

Implementation



```
#include<iostream>
using namespace std;
struct Element{
    int data;
    Element *next;
};
typedef struct Element Element;

struct List{
    int n;  //number of elements
    Element *head;
    Element *tail;
};
typedef struct List List;
```

```
//A function to create an empty list
List* createList(){
    List *ls;

    ls = new List(); //allocate memo
    //ls.n = 0; //error
    ls->n = 0;
    ls->head = NULL;
    ls->tail = NULL;

    return ls;
}
```

```
void insert_begin(List *ls, int newData){
    //Create new element
    Element *e;
    e = new Element();
    e->data = newData;

    //Update pointer, head, tail
    e->next = ls->head;
    ls->head = e;
    if(ls->n == 0){
        ls->tail = e;
    }
    ls->n = ls->n + 1;
}
```

```
void displayList(List *ls){
    Element *tmp; //temporary variable

    tmp = ls->head;
    while(tmp!=NULL){
        cout<<tmp->data<<" ";
        tmp = tmp->next;
    }
    cout<<endl;
}
```

```
int main(){

    List *L;
    L = createList();

    insert_begin(L, 3);
    insert_begin(L, 2);
    insert_begin(L, 5);
    displayList(L);
    displayList(L);
    displayList(L);
    cout<<L->n<<endl;
```

```
5 2 3
5 2 3
5 2 3
3
```

Q&A