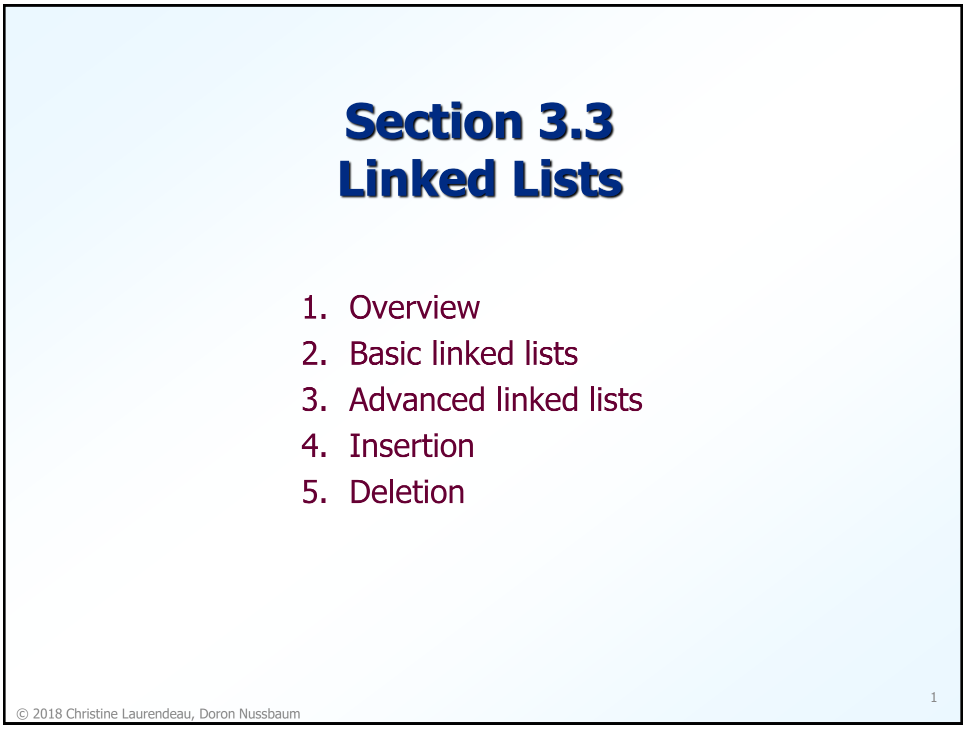
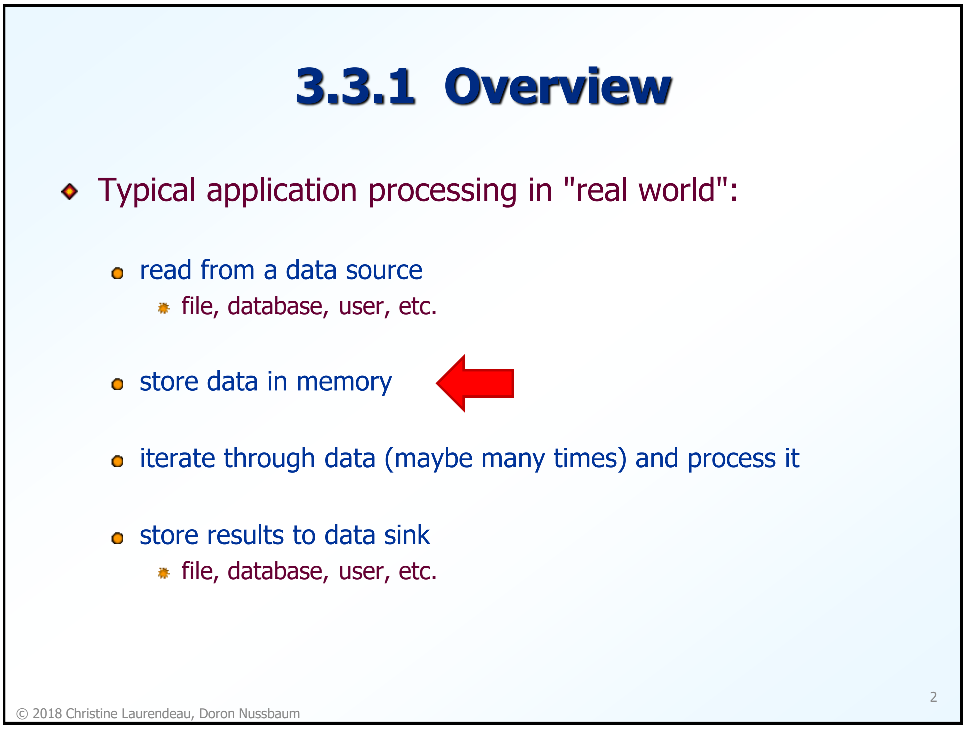


Section 3.3
Linked Lists

1. Overview
2. Basic linked lists
3. Advanced linked lists
4. Insertion
5. Deletion



3.3.1 Overview

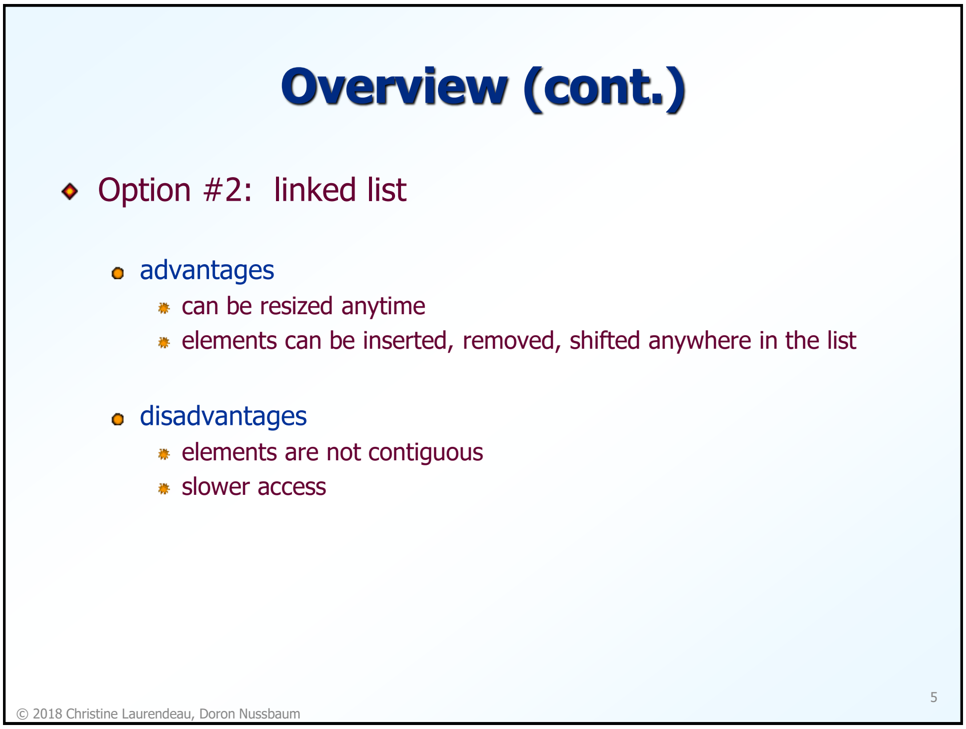
- Typical application processing in "real world":
  - read from a data source
    - file, database, user, etc.
  - store data in memory ⬅
  - iterate through data (maybe many times) and process it
  - store results to data sink
    - file, database, user, etc.

Overview (cont.)

- How we store data in memory is important! ⚠
- We want
  - fastest possible access
  - least amount of memory
- Choice of data structure has major impact on performance

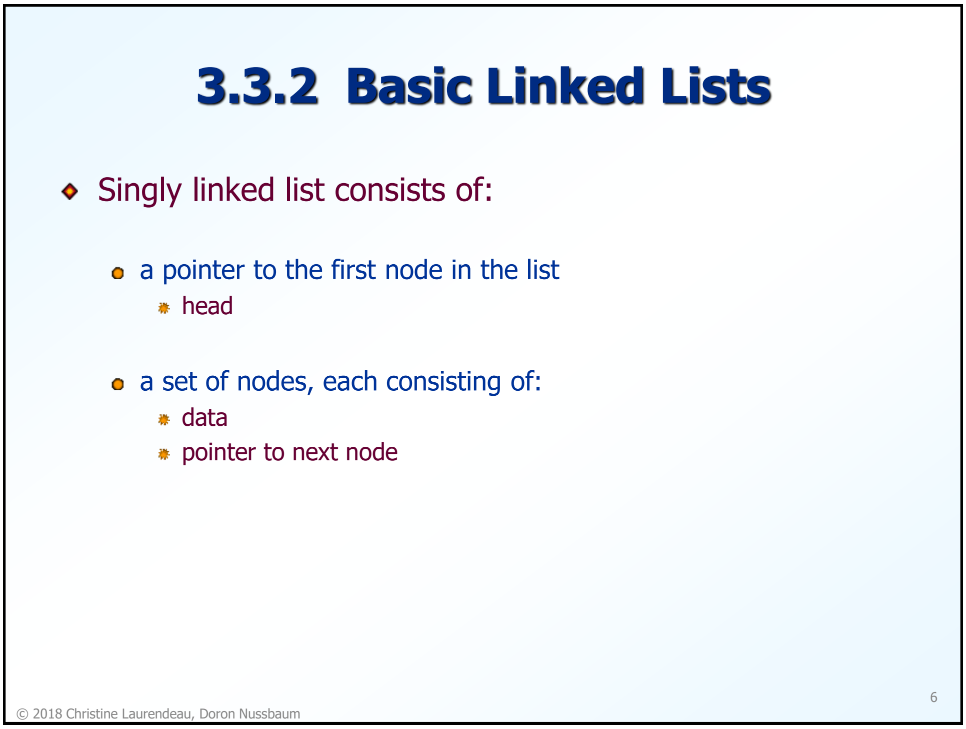
Overview (cont.)

- Option #1: array
  - advantages
    - elements are contiguous
    - faster access
  - disadvantages
    - once allocated, array cannot be resized
    - no growing, no shrinking
  - trade-offs
    - oversized array == waste of memory
    - undersized array == array overflow



Overview (cont.)

- Option #2: linked list
  - advantages
    - can be resized anytime
    - elements can be inserted, removed, shifted anywhere in the list
  - disadvantages
    - elements are not contiguous
    - slower access



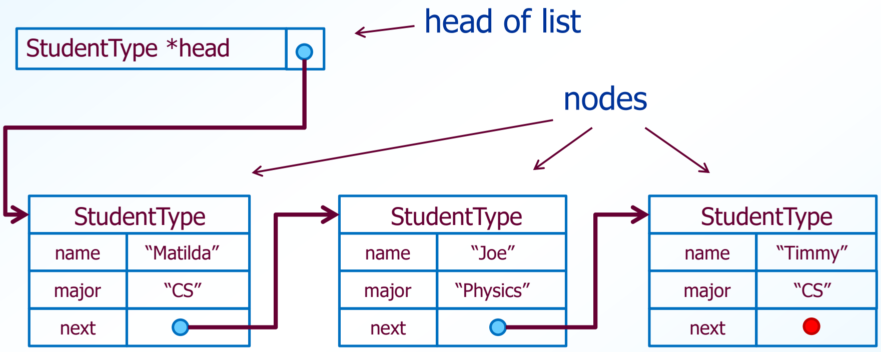
3.3.2 Basic Linked Lists

- Singly linked list consists of:
  - a pointer to the first node in the list
    - head
  - a set of nodes, each consisting of:
    - data
    - pointer to next node

Basic Linked Lists (cont.)

- Doubly linked list consists of:
  - a pointer to the first node in the list
    - head
  - a pointer to the last node in the list
    - tail
  - a set of nodes, each consisting of:
    - data
    - pointer to next node
    - pointer to previous node

Basic Linked Lists (cont.)



StudentType *head

head of list

nodes

StudentType	
name	"Matilda"
major	"CS"
next	●

StudentType	
name	"Joe"
major	"Physics"
next	●

StudentType	
name	"Timmy"
major	"CS"
next	●

Processing a Linked List

- Initialization
  - always initialize your pointers
  - use NULL or zero for empty pointers
    - check for NULL pointers in your code!
    - NULL is used as a sentinel
- Traversal
  - use an iteration pointer
- Do not lose the head of the list! ☠

3.3.3 Advanced Linked Lists

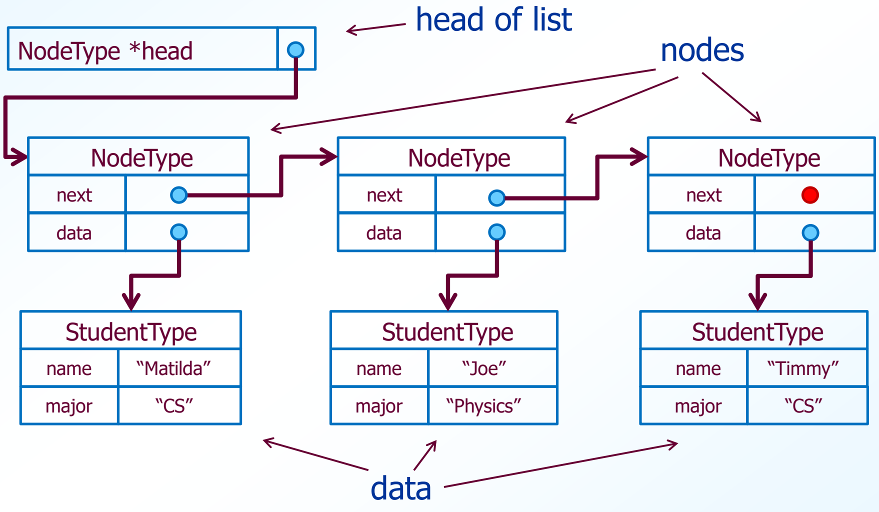
- Problems with basic linked lists
  - element data mixed with list data ⚠
    - no encapsulation!
    - bad design
  - each element is hard-coded to point to a specific other
    - what if we need the same element in multiple lists?

Advanced Linked Lists (cont.)

- Solution: separate the nodes from the data

- Why?
  - think “real world”
  - encapsulation
    - keep data-related stuff together, and list-related stuff together
    - compartmentalize what each element knows
      - should not know that it’s in a linked list
      - should only have information related to itself
  - reuse
    - one element may be included in multiple linked lists

Advanced Linked Lists (cont.)



head of list

nodes

NodeType *head

NodeType	
next	
data	

NodeType	
next	
data	

NodeType	
next	
data	

StudentType	
name	“Matilda”
major	“CS”

StudentType	
name	“Joe”
major	“Physics”

StudentType	
name	“Timmy”
major	“CS”

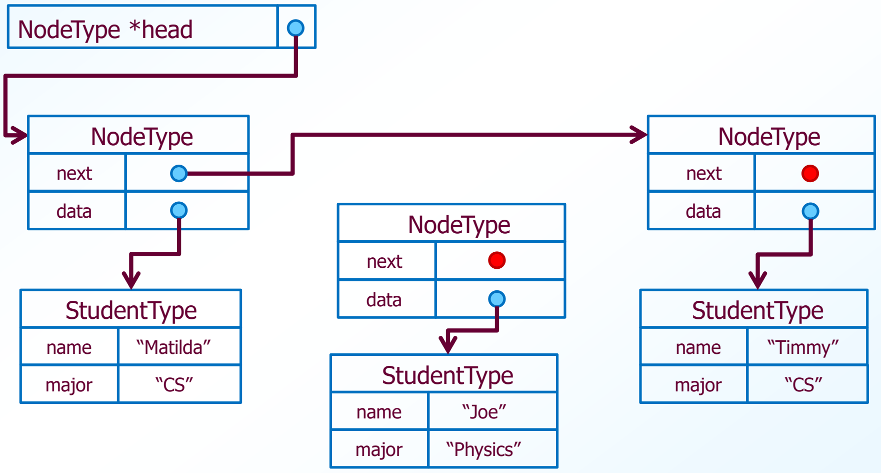
data

3.3.4 Linked List Insertion

- We can insert an element anywhere in the list
  - shift pointer values
- Always consider four cases:
  - element is the first to be added
  - element is to be added in first position
  - element is to be added in middle of the list
  - element is to be added in last position

Linked List Insertion (cont.)

- Original list:



NodeType *head

NodeType

next	○
data	○

StudentType

name	"Matilda"
major	"CS"

NodeType

next	●
data	○

StudentType

name	"Joe"
major	"Physics"

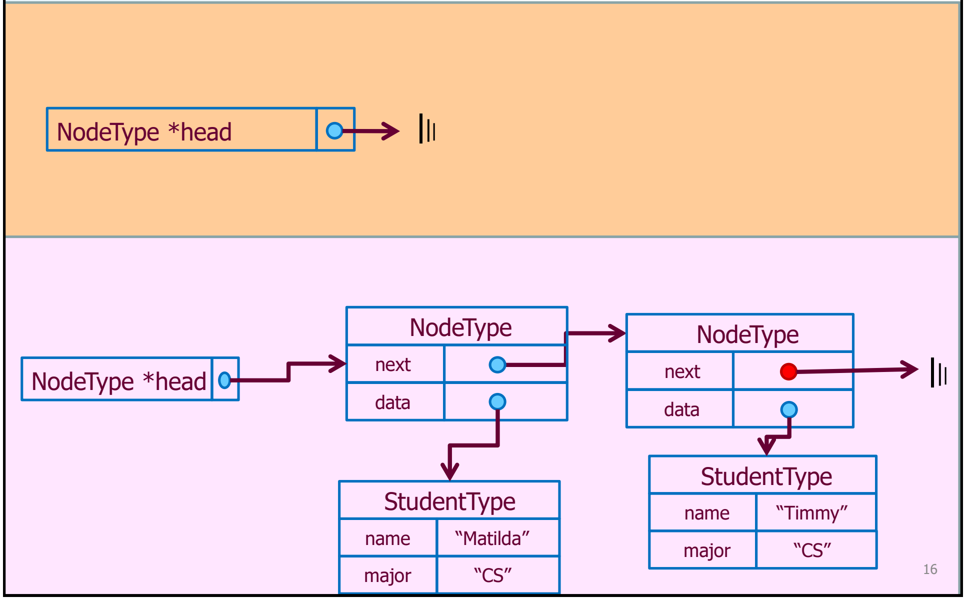
NodeType

next	●
data	○

StudentType

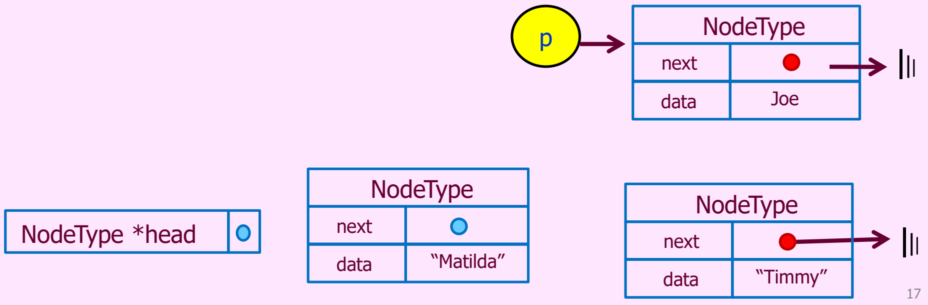
name	"Timmy"
major	"CS"

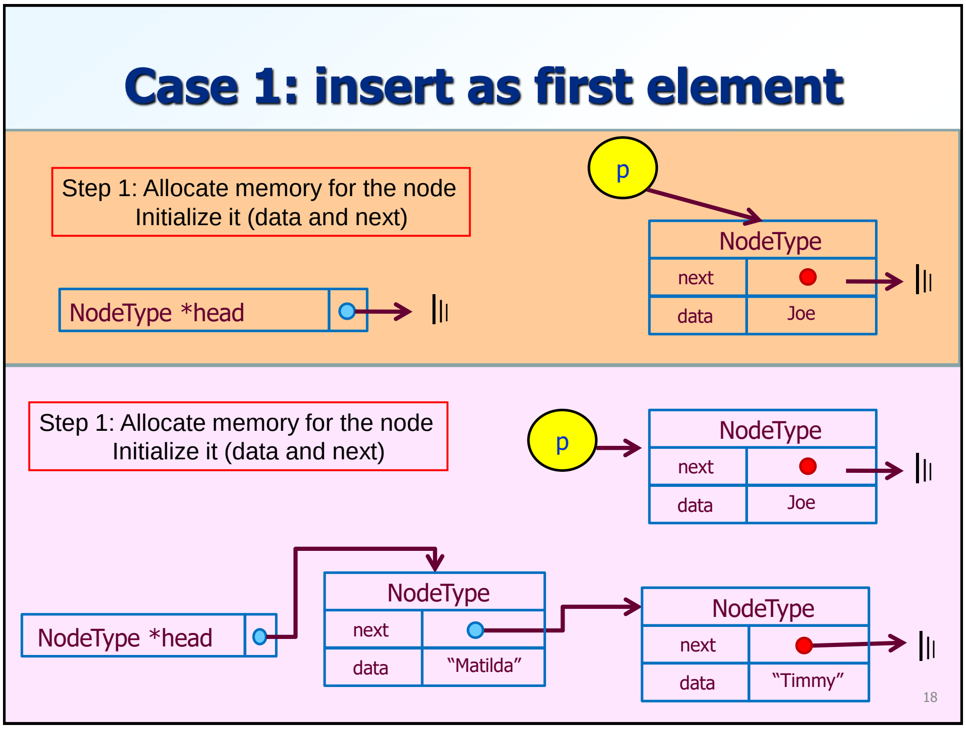
Case 1: insert as first element



Case 1: insert as first element

Step 1: Allocate memory for the node
Initialize it (data and next)

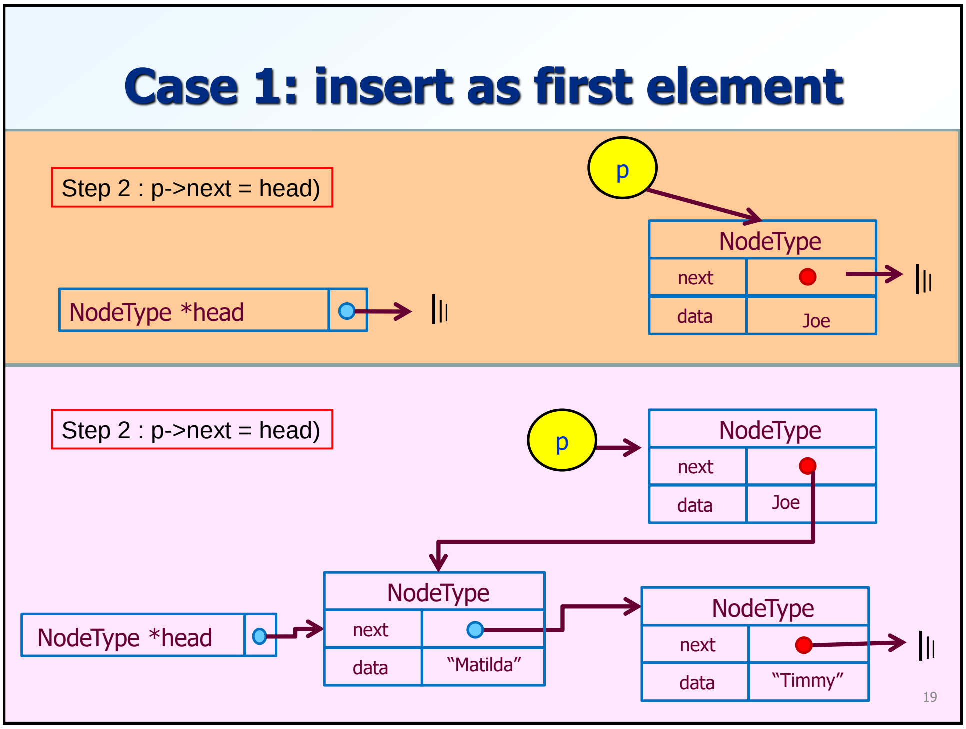




Case 1: insert as first element

Step 1: Allocate memory for the node
Initialize it (data and next)

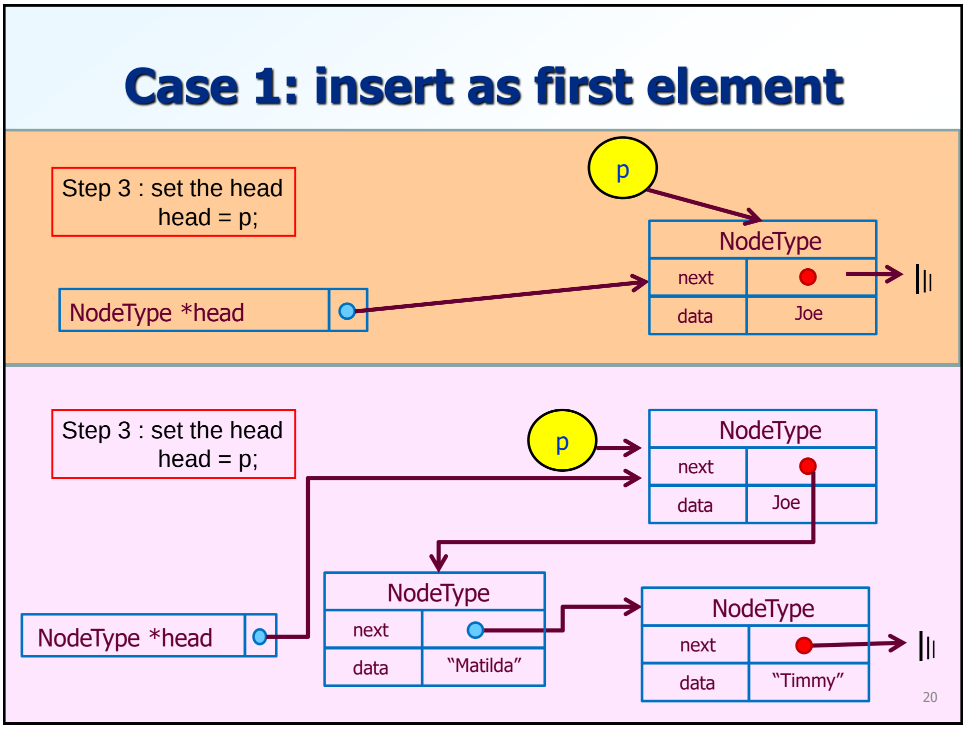
Step 1: Allocate memory for the node
Initialize it (data and next)



Case 1: insert as first element

Step 2 : p->next = head)

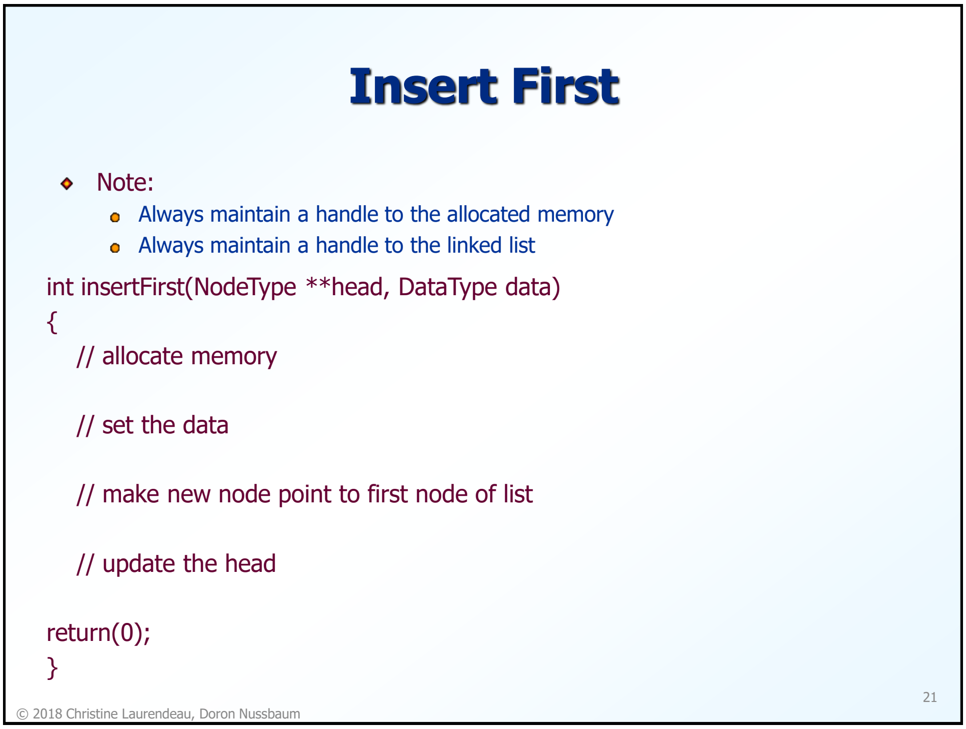
Step 2 : p->next = head)



Case 1: insert as first element

Step 3 : set the head
head = p;

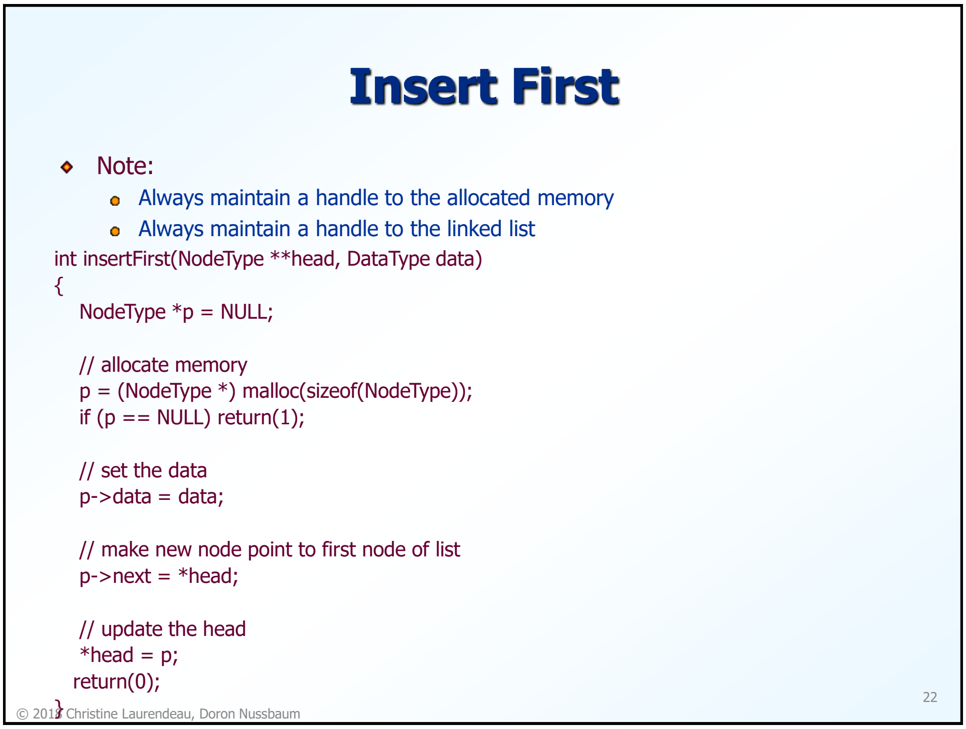
Step 3 : set the head
head = p;



Insert First

- Note:
  - Always maintain a handle to the allocated memory
  - Always maintain a handle to the linked list

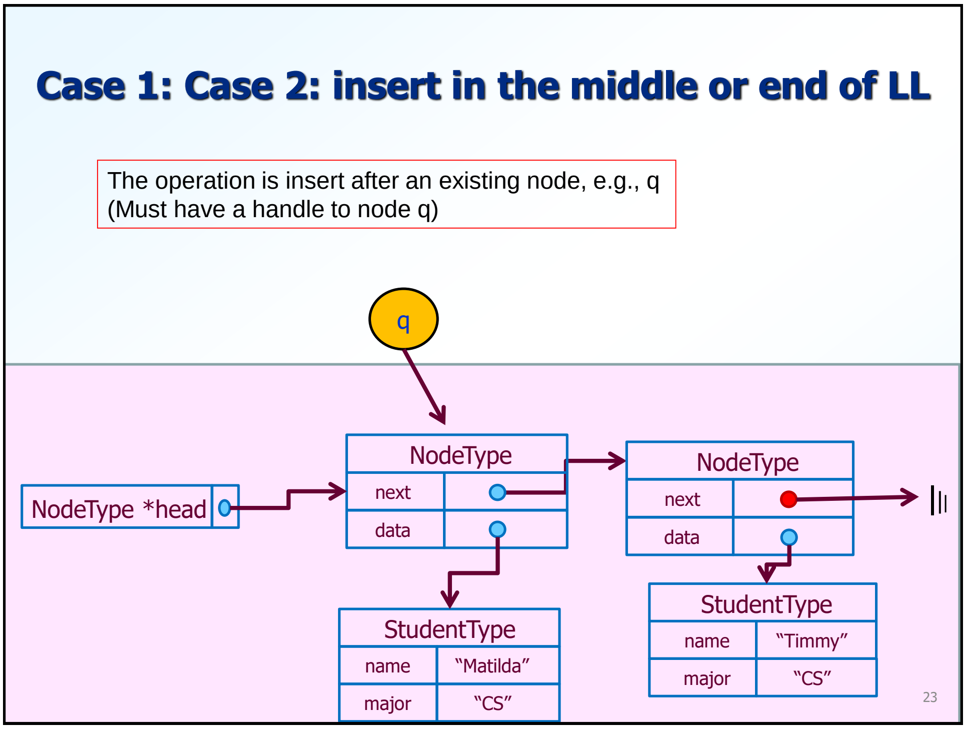
```
int insertFirst(NodeType **head, DataType data)
{
   // allocate memory

   // set the data

   // make new node point to first node of list

   // update the head

return(0);
}
```



Insert First

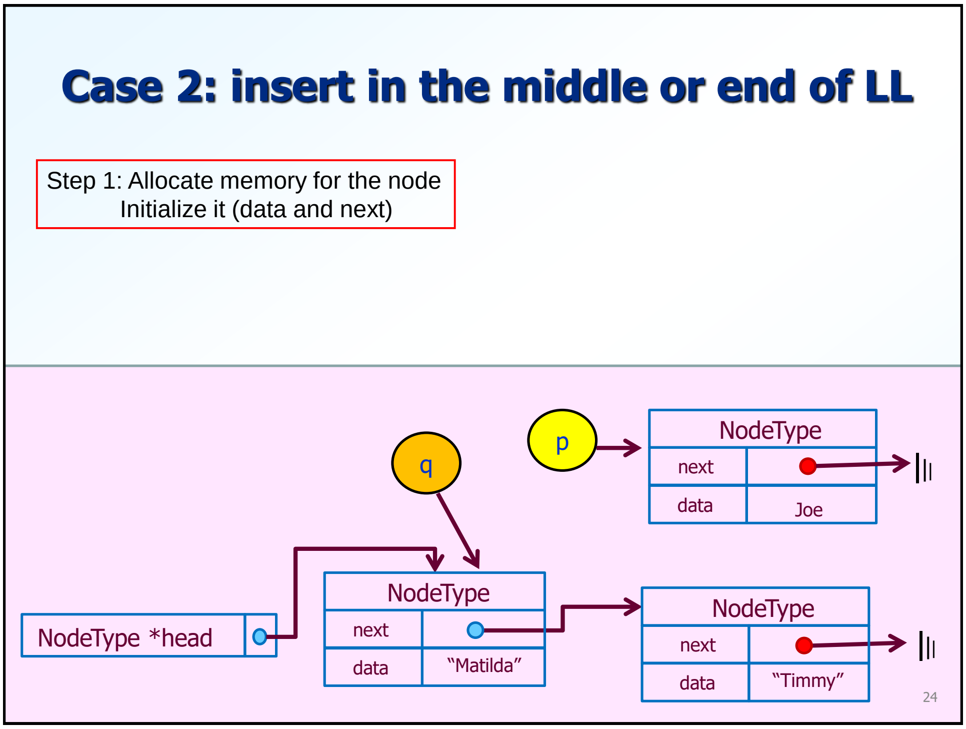
- Note:
  - Always maintain a handle to the allocated memory
  - Always maintain a handle to the linked list

```
int insertFirst(NodeType **head, DataType data)
{
    NodeType *p = NULL;

    // allocate memory
    p = (NodeType *) malloc(sizeof(NodeType));
    if (p == NULL) return(1);

    // set the data
    p->data = data;

    // make new node point to first node of list
    p->next = *head;

    // update the head
    *head = p;
  return(0);
}
```



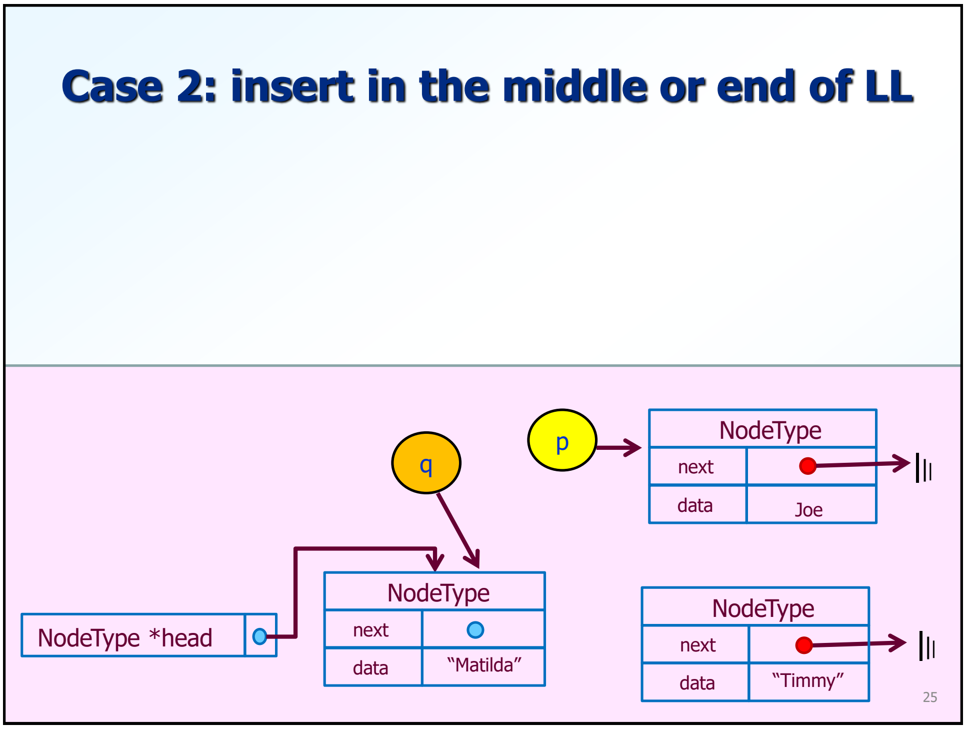
Case 1: Case 2: insert in the middle or end of LL

The operation is insert after an existing node, e.g., q
(Must have a handle to node q)

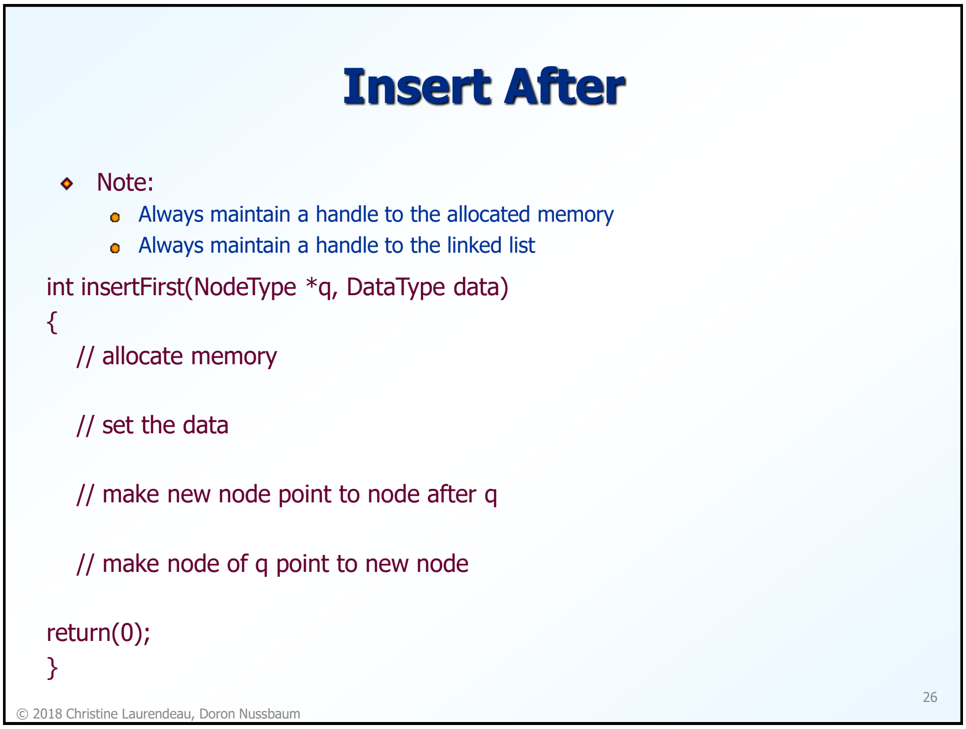


Case 2: insert in the middle or end of LL

Step 1: Allocate memory for the node
Initialize it (data and next)



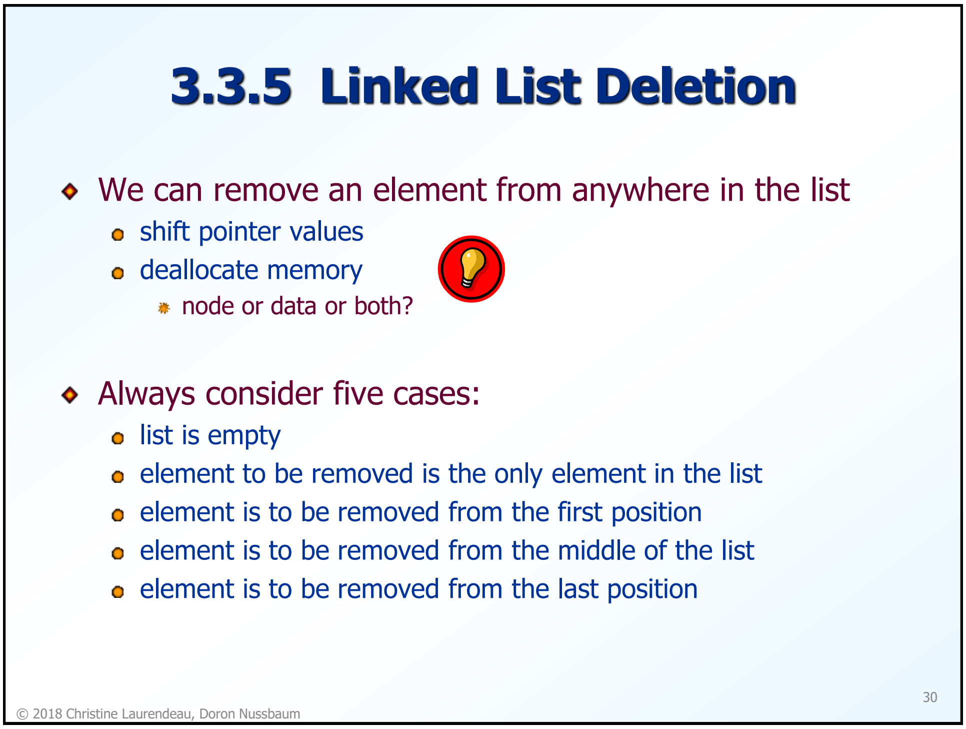
Case 2: insert in the middle or end of LL



Insert After

- Note:
  - Always maintain a handle to the allocated memory
  - Always maintain a handle to the linked list

```
int insertFirst(NodeType *q, DataType data)
{
   // allocate memory

   // set the data

   // make new node point to node after q

   // make node of q point to new node

return(0);
}
```



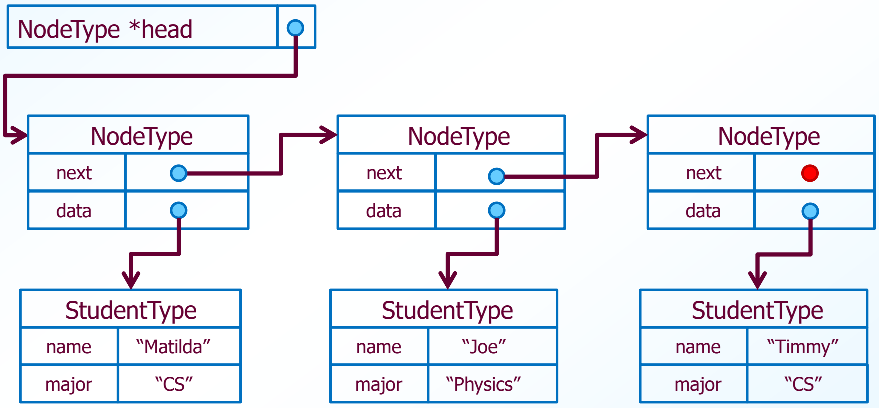
3.3.5 Linked List Deletion

- We can remove an element from anywhere in the list
  - shift pointer values
  - deallocate memory
    - node or data or both?

- Always consider five cases:
  - list is empty
  - element to be removed is the only element in the list
  - element is to be removed from the first position
  - element is to be removed from the middle of the list
  - element is to be removed from the last position

Linked List Deletion Delete First

- Original list:



NodeType *head

NodeType
next
data

NodeType
next
data

NodeType
next
data

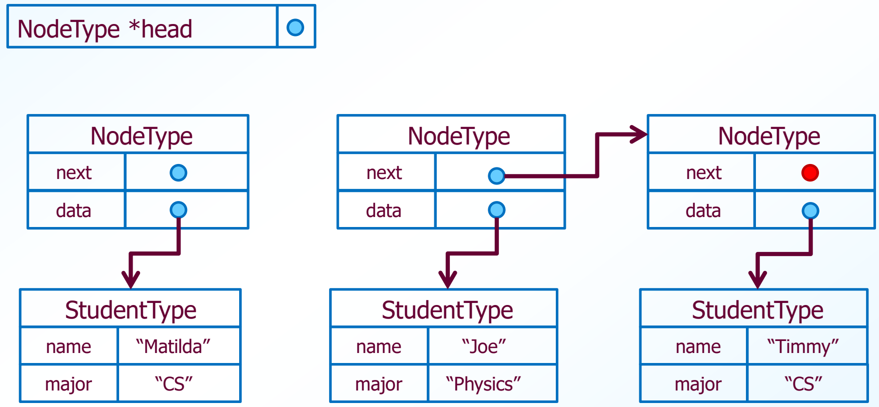
StudentType
name "Matilda"
major "CS"

StudentType
name "Joe"
major "Physics"

StudentType
name "Timmy"
major "CS"

Linked List Deletion Delete First

- Original list:



NodeType *head

NodeType
next
data

NodeType
next
data

NodeType
next
data

StudentType
name "Matilda"
major "CS"

StudentType
name "Joe"
major "Physics"

StudentType
name "Timmy"
major "CS"

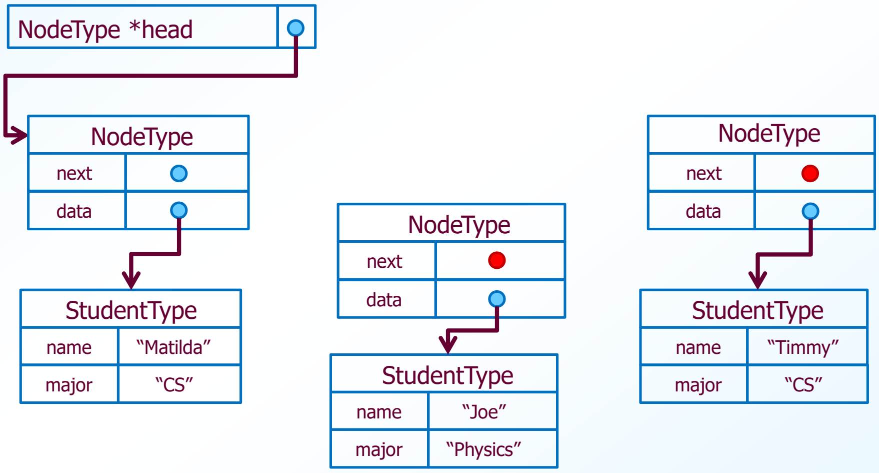
Linked List Deletion Delete First

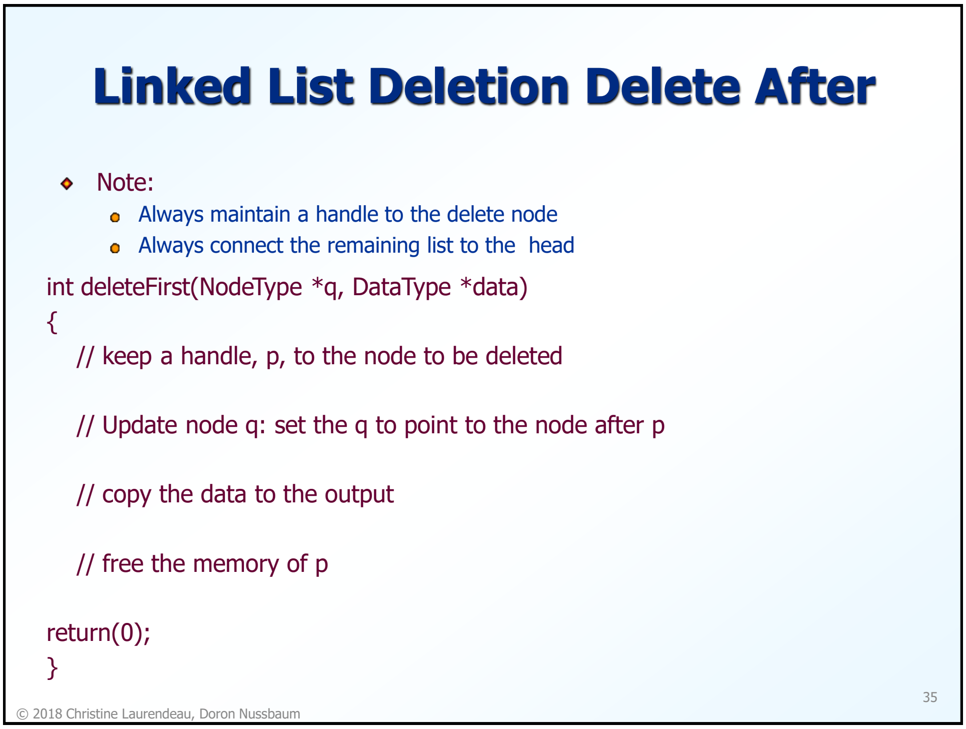
- Note:
  - Always maintain a handle to the delete node
  - Always connect the remaining list to the head

```
int deleteFirst(NodeType **head, DataType *data)
{
   // keep a handle, p, to the node to be deleted

   // Update the head: set the head to the node after p

   // copy the data to the output

   // free the memory of p

return(0);
}
```

Linked List Deletion after

- After deletion from the middle:

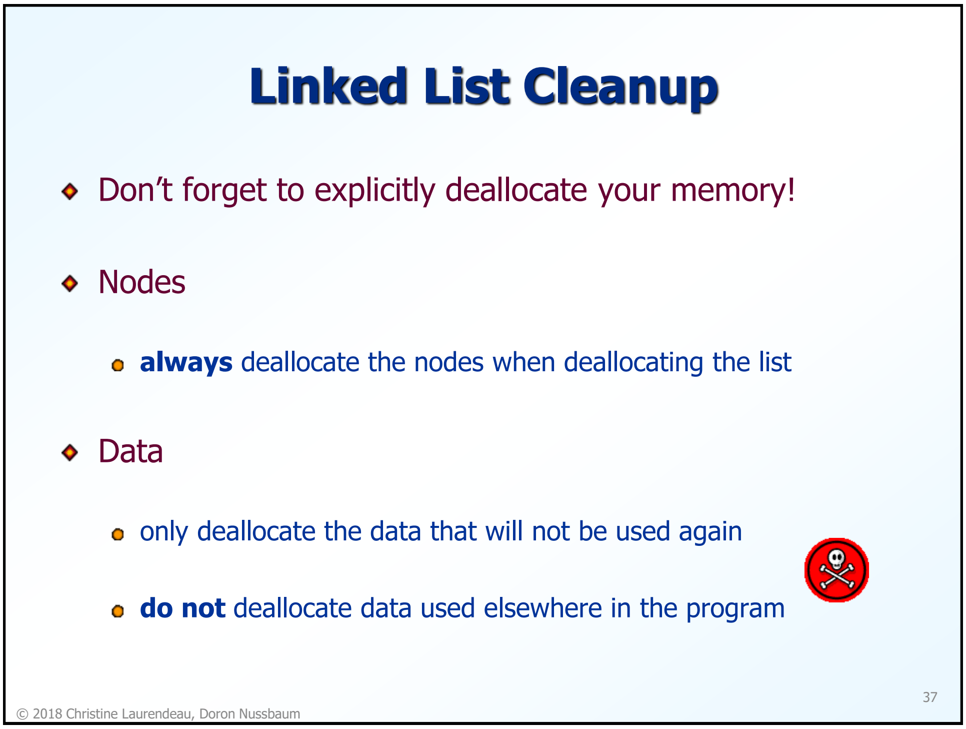




Linked List Deletion Delete After

- Note:
  - Always maintain a handle to the delete node
  - Always connect the remaining list to the head

```
int deleteFirst(NodeType *q, DataType *data)
{
   // keep a handle, p, to the node to be deleted

   // Update node q: set the q to point to the node after p

   // copy the data to the output

   // free the memory of p

return(0);
}
```



Linked List Cleanup

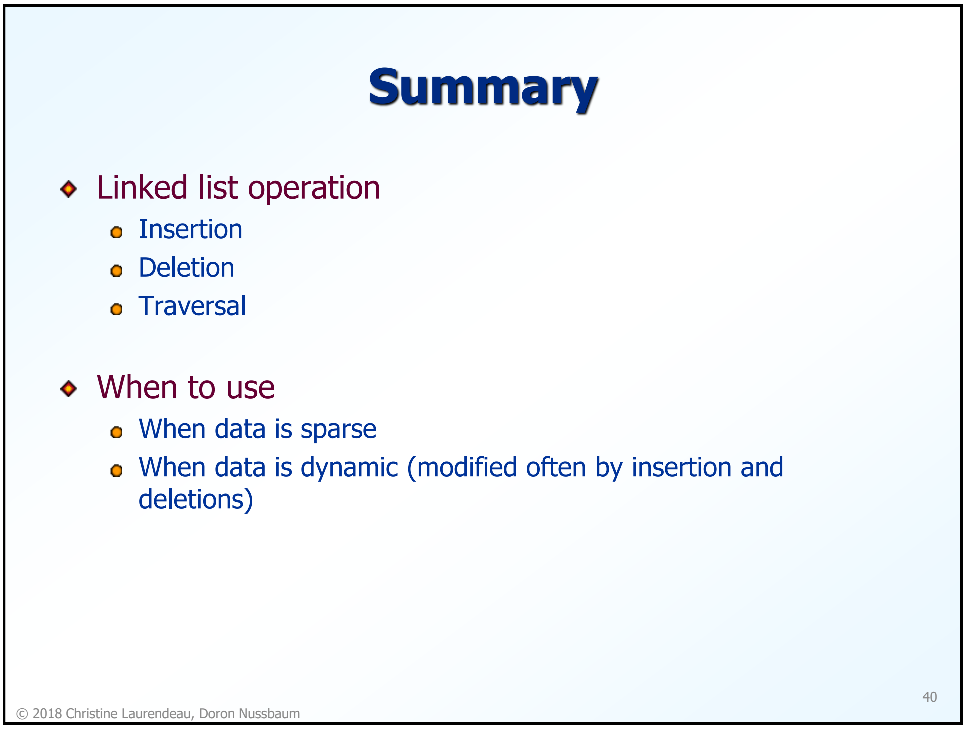
- Don’t forget to explicitly deallocate your memory!
- Nodes
  - **always** deallocate the nodes when deallocating the list
- Data
  - only deallocate the data that will not be used again
  - **do not** deallocate data used elsewhere in the program

Linked List Traversal

- Iterative
  - Start from head
    - Process the node
  - If list was not exhausted then move to next node
- Recursive
  - Check boundary condition
  - If boundary condition is not met then
    - Process node
    - Call yourself recursively with next node

Doubly Linked List

- Node Characteristics
  - Next pointer
  - Previous pointer
  - Data
- List access characteristics
  - Head
  - Tail
- Processing
  - Can traverse the list in both directions!!
- Expense
  - Additional pointer



Summary

- Linked list operation
  - Insertion
  - Deletion
  - Traversal
- When to use
  - When data is sparse
  - When data is dynamic (modified often by insertion and deletions)