Section 3
Essential Object-Oriented Techniques

1. Encapsulation
2. Inheritance
3. Design patterns
4. Polymorphism
5. Overloading
6. Templates
7. Exception handling

Section 3.1
Encapsulation

1. Composition
2. Constants
3. Friendship
4. Static class members
5. Linked lists

3.1.1 Composition

- An object can contain a data member that's an object of another class
  - "has-a" relationship
- On initialization:
  - *container* object calls constructor for *containee* object
    - otherwise the default constructor is called implicitly
  - use **member initializer syntax** to avoid temporary objects

Composition (cont.)

- Order of invocation:
  - constructors
    - objects are built from the inside out
    - containee object(s) are constructed first, in order of declaration
      - not in member initializer order
    - container object is created last
  - destructors
    - objects are destroyed from the outside in
    - container object is destroyed first, then the containee object(s)
    - destructors are invoked in the reverse order of constructors

3.1.2 Constants

- A constant is not the same thing as a *literal*

- In C++, **`const`** is a qualifier specified on declaration
  - its purpose is protection

- There are many uses for the **`const`** qualifier in C++:
  - constant objects
  - constant member functions
  - constant data members

Constant Objects

- Side-effects of constant variables in general:
  - the content of a constant variable can never be modified
  - it can never be used as lvalue
    - *lvalue*:
      - can be used on left-hand side (LHS) of assignment operation
      - can also be used on right-hand side (RHS) of assignment
    - *rvalue*:
      - can be used on RHS
      - **cannot** be used on LHS
      - example: a literal, or an expression
  - value must be assigned upon declaration
    - at initialization

Constant Objects (cont.)

- Why make an object constant?
  - once initialized, object's data member values cannot change
  - no part of the program is allowed to change the object
- Characteristics of constant objects:
  - data members are initialized in the constructor
  - only *constant member functions* can be used on constant object

Constant Objects (cont.)

- Applying the principle of least privilege:
  - think about the objects you create
  - if they don’t need to be modified, make them constant
  - this protects the objects from bad code
  - it guarantees the integrity of the objects

Constant Member Functions

- Guarantee that member function will not modify object
- Characteristics of constant member function:
  - it is the **only** type of function allowed on a *constant object*
  - it is not allowed to **change** the value of any data member
  - it is not allowed to **call** a non-constant member function
  - constructors and destructors cannot be constant

Constant Member Functions (cont.)

- Applying the principle of least privilege:
  - think about the member functions you create
  - if they don’t need to change the object, make them constant
  - this ensures the functions can be called on constant objects

Constant Data Members

- Data member that can never be modified
- Characteristics of constant data members:
  - they cannot be modified, even in the object constructor!
  - initialization:
    - must occur before the body of the constructor
    - must use member initializer syntax

Constant Data Members (cont.)

- Member initializer syntax
  - it is used between the constructor parameter list and the body
  - it can be used to initialize non-constant data members
  - it *must* be used to initialize constant data members
  - it *must* be used to initialize data members that are references
  - it executes before the body of the constructor
  - it can result in an empty constructor body

Constant Data Members (cont.)

- Applying the principle of least privilege:
  - think about the data members that you create
  - if they never need to be modified after initialization, make them constant
  - this guarantees the integrity of the data members
    - even from member functions!

3.1.3 Friendship

- A class can grant friendship to:
  - a global function
    - **not** a member function
  - another class

Friendship (cont.)

- Characteristics of friendship:
  - it gives away complete access to the class members
    - even the private and protected ones
  - it is granted, and it cannot be taken
  - it is neither symmetric nor transitive

Friendship (cont.)

- Friend function
  - a global function that is given complete access to the class
  - it can access all members
    - private, protected, and public
- Friend class
  - another class that is given complete access to the class
  - the friend class member functions can access all class members

3.1.4 Static Class Members

- Characteristics of static members:
  - only one copy of a static member exists for the class
  - static members exist even if no objects of the class are created
  - static members can be accessed:
    - from the class name
      - using the binary scope resolution operator
    - from any object of that class
- Be careful of the **`static`** keyword
  - used at file scope, identifier becomes invisible outside the file

Static Class Members (cont.)

- Static data member
  - is a property of the class as a whole
  - is a value shared by all objects of that class
  - must be initialized at file scope
    - in the source file, by convention

Static Class Members (cont.)

- Static member function
  - is a service of the class as a whole
  - must work even if no objects of the class are ever created
  - may only access static data members
    - not any non-static data members, or non-static member functions
  - must be specified as static in the class definition only
    - not in the source file, otherwise it won't be visible

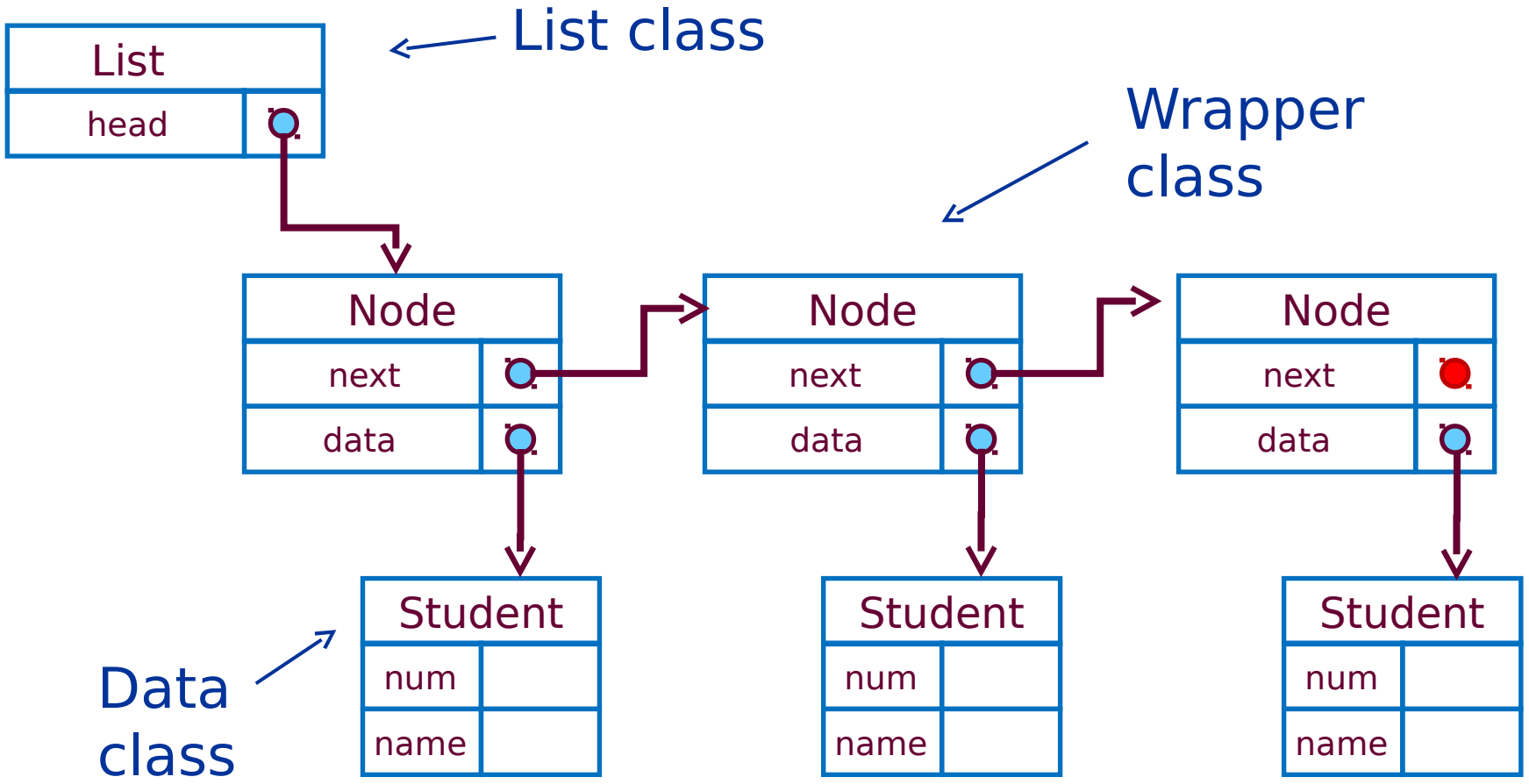
3.1.5 Linked Lists

- Array-based collections have disadvantages
  - we don’t always know how many elements we need
  - array may be oversized or undersized
    - dynamic resizing can be computationally expensive
  - it is very inefficient to add or remove elements in the middle
    - array elements are always contiguous in memory

Linked Lists (cont.)

- Linked list-based collections allow us to:
  - only allocate the amount of memory we need
  - efficiently add and remove elements anywhere in the list
  - easily shift elements around in the list
  - disadvantage: data is not contiguous, so access can be slower

Linked Lists (cont.)



Linked Lists (cont.)

- Linked list consists of:
  - a set of nodes, where each node contains:
    - a pointer to the data element
    - a pointer to the next node
    - a pointer to the previous node, if the list is *doubly-linked*
  - head: a pointer to the first node
  - tail: a pointer to the last node
    - not all linked lists store a tail
  - do **not** create dummy nodes!
    - dummy node: placeholder node that represents list head
    - every node must have a data element associated with it

Linked Lists (cont.)

- Why separate list, wrapper, and data classes?
  - encapsulation!
  - keep the data-related and the list-related knowledge separate
  - compartmentalize what each object knows
    - for example, data element should not know it's part of a list
  - reuse
    - data element may be included in multiple lists
    - that can't happen if each element keeps pointer to next element
  - it's good software engineering

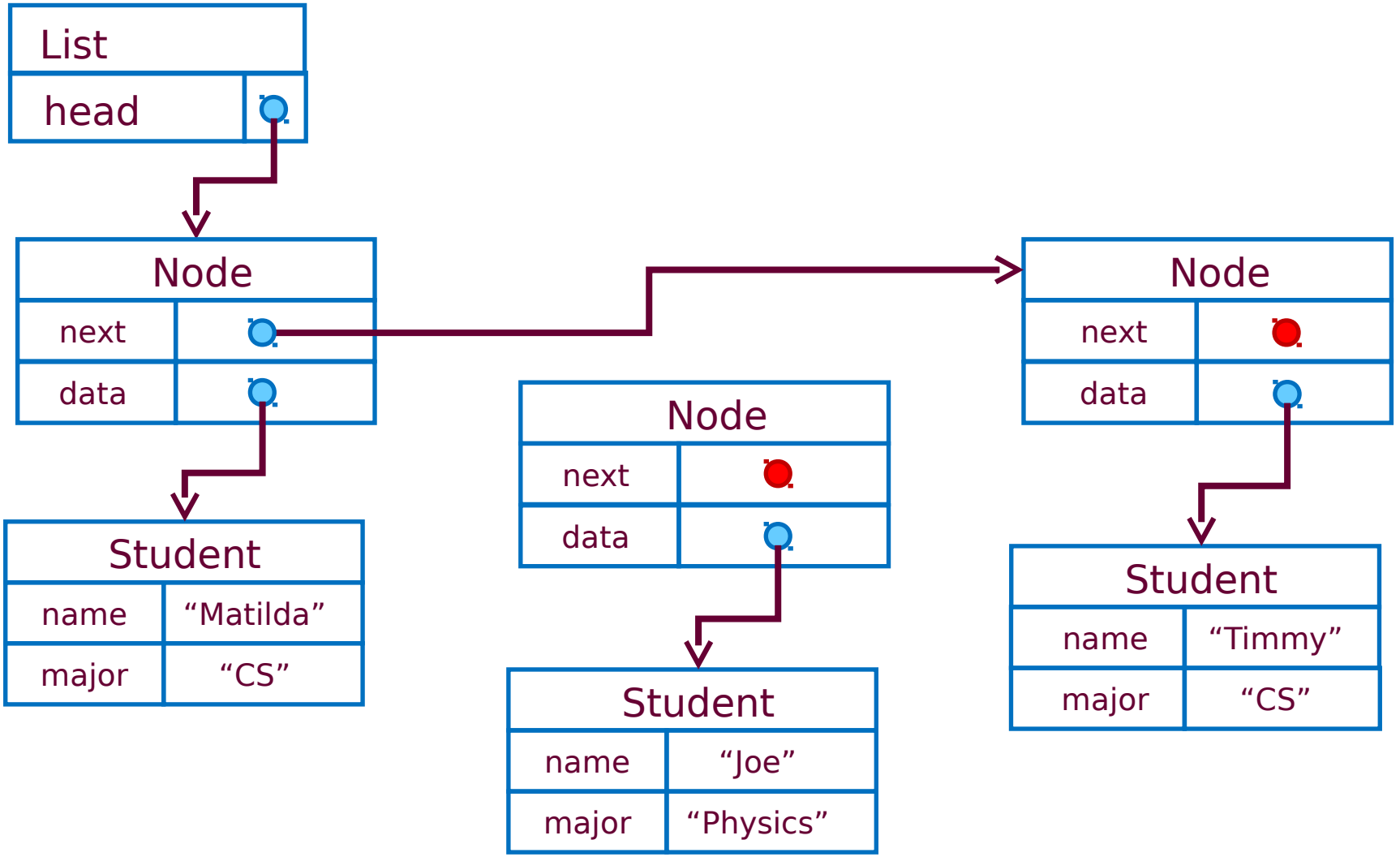
Element Insertion

- We can insert an element anywhere in a linked list
  - simply shift pointer values

- Always consider four cases:
  - element added to an empty list
  - element inserted at the beginning of the list
  - element inserted in middle
  - element added to the end

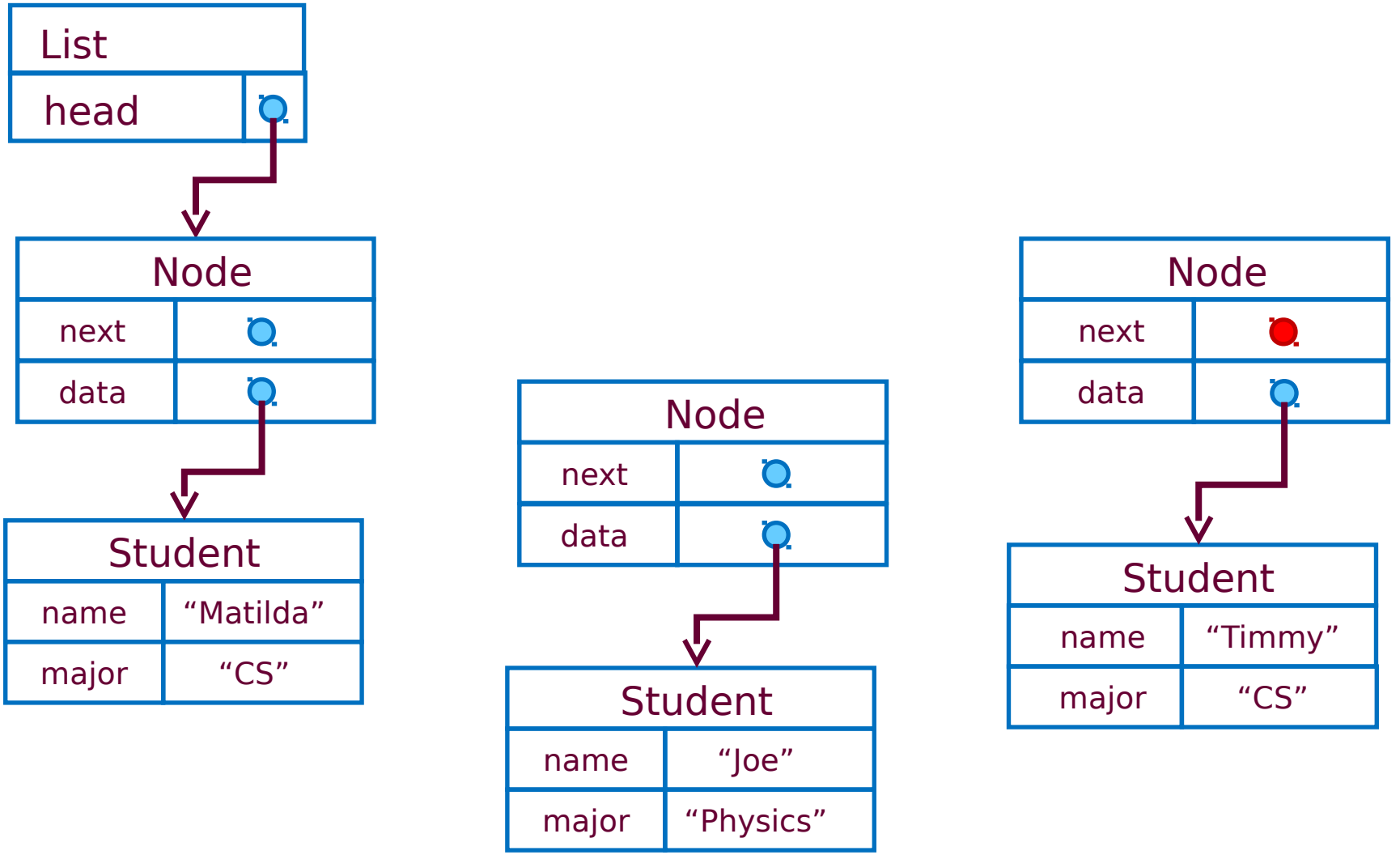
Element Insertion (cont.)

- Original list:



Element Insertion (cont.)

- After insertion in the middle:



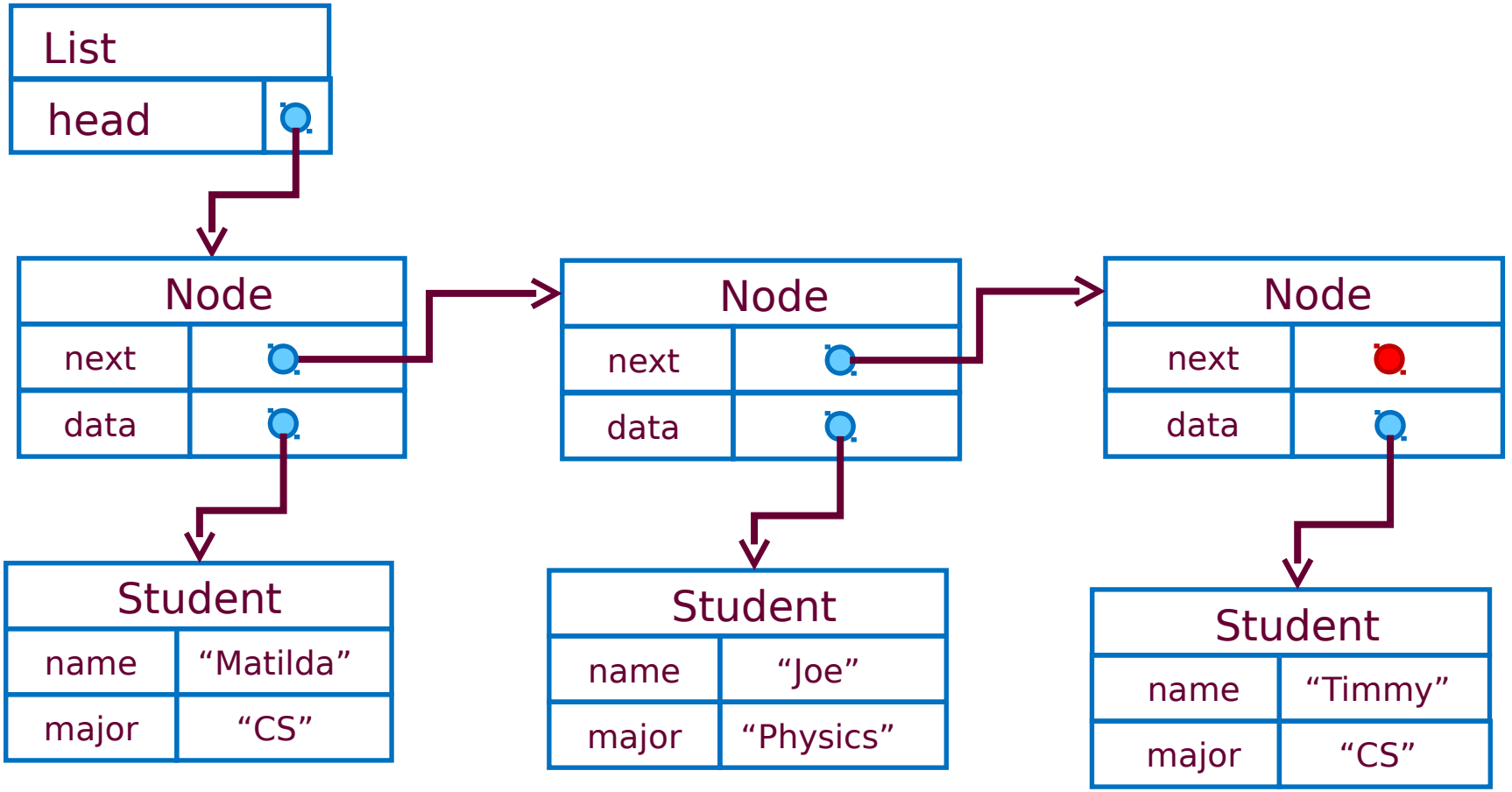
Element Deletion

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

- Always consider five cases:
  - list is empty!
  - element removed from the beginning of the list
  - element removed from middle
  - element removed from the end
  - element removed is the last remaining element in the list

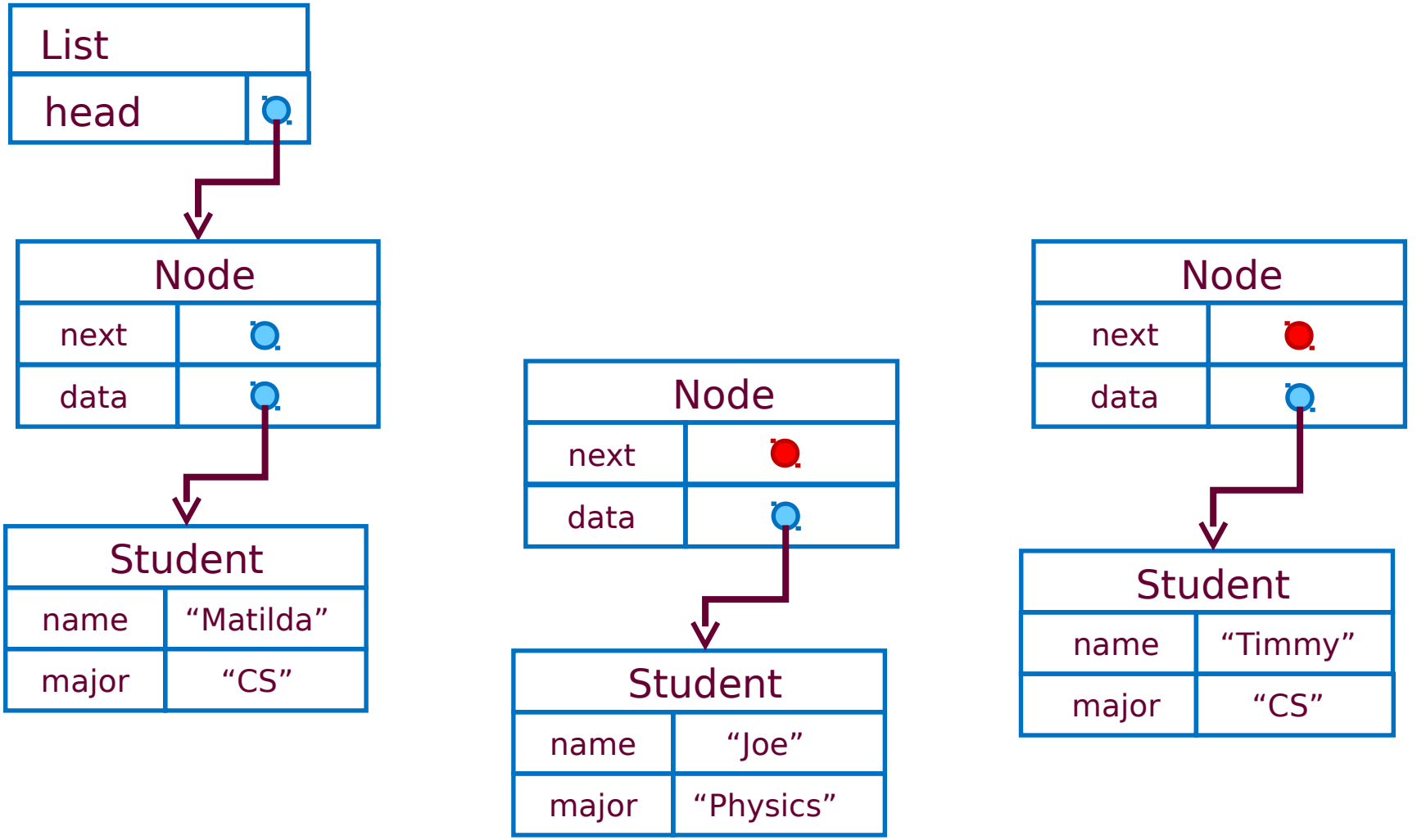
Element Deletion (cont.)

- Original list:



Element Deletion (cont.)

- After deletion from the middle:



List Cleanup

- Don’t forget to explicitly deallocate your memory!
  - destructors are the correct place to do this work
- Deallocating nodes
  - always deallocate the nodes when deallocating the list
  - always deallocate the node associated with a deleted element
- Deallocating data
  - only deallocate data that will not be used again
  - do not deallocate data used elsewhere in the program

Doubly Linked Lists

- Doubly linked list consists of:
  - a set of nodes, where each node contains:
    - a pointer to the data element
    - a pointer to the next node
      - the last element's next pointer is NULL
    - a pointer to the previous node
      - the first element's previous pointer is NULL
  - head: a pointer to the first node
  - tail: a pointer to the last node
    - not all doubly linked lists store a tail