11 - Linked Lists

February 13, 2023

COMP2404

Darryl Hill

Contents

Encapsulation

1. Composition
2. Constants
3. Friendship
4. Static class members

Encapsulation example:
→ Linked lists

Linked Lists

Why Linked Lists?

- They demonstrate many useful concepts
- Rich source of examples:
    - Abstract interfaces
    - Separation of data and containers
    - Memory management

Linked Lists

Array-based collections have disadvantages

- Resizing is a pain
  - Make a new array
  - Copy everything over
- Adding or removing from middle not easy
  - have to move to make room or move to close gaps

Linked Lists

Linked List based collection

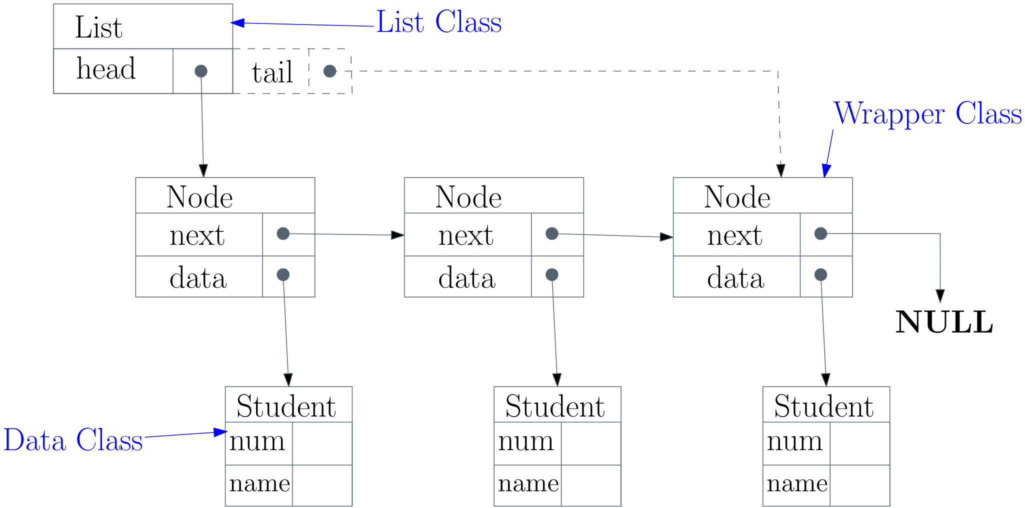
- allocate only memory we need
- If we have a reference to a Node, we can add and remove easily anywhere in the list
- data is not contiguous, so access can be slower
  - in an array we can access any index immediately - random access
  - linked list require traversals

Many data structures combine Linked Lists with arrays.

Linked Lists are composed of three components:

- List class
- Node class (a wrapper class for data)
- Data class

Linked Lists



Node class

```
class Node{

    public:
        Student* data;
        Node*    next;
};
```

- The Node is a small class with a clear purpose.
  - Might look something like this.
- Functionally it is a struct.
  - Two data types bound together.
- Recursive data structure
  - It has references to classes of its own type.
- It works very closely with the List class.

Linked Lists

```
class List{

    // some functions

    Node* head;
};
```

List class

- has a Node pointer to the head of the list
- sometimes a tail pointer as well

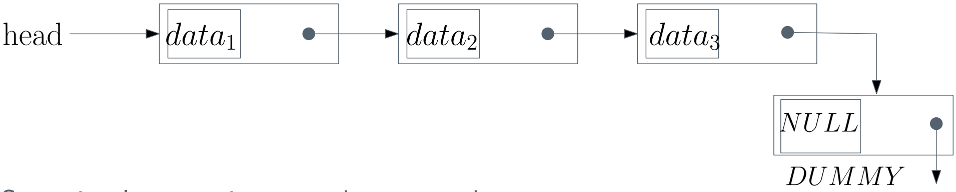
We have data class wrapped in our Node

- some class, ie Student
- we will hard-code the type until we learn templates

Nodes contain list information, data contains data information

- Student class should have no knowledge of the List or Node classes

(Singly) Linked Lists



Some implementations use dummy nodes

- We will not
- All nodes should point to actual data
- Last node has a next value of NULL
  - This indicates the end of the list.
- No tail pointer
  - (that is an assignment / tutorial exercise)

Linked Lists

Linked Lists in general (may) consist of:

- a sequence of nodes
    - pointer to data element
    - pointer to next node
    - (pointer to previous IF doubly-linked-list)
- head: pointer to the first node
- tail: pointer to the last node
    - not all linked lists
    - why would this be useful?
- Dummy nodes
    - No dummy nodes in this course!
    - Every node must have a data element

Linked Lists

Why do we separate list, wrapper, and data classes?

- The wrapper acts as an individual abstraction layer for each data element
    - Hides collection functions from the data.
- List elements need only know about themselves
    - not the data structures

Consider if we included a next pointer in the `Student` class

- Each Student could only be in one linked list.
- If we change the data structure implementation we must change `Student` class.
    - Not good encapsulation.

Linked List Interface

What sort of interface should our Linked List have?

- As generalized as possible.
- The interface will be the same as an array based list interface.

```
bool add(Student*);
Student* get(const string& name);
Student* remove(const string& name);
void honourRoll(List& hr);
void print() const;
```

- This is good ***abstraction*** - the same generalized interface irrespective of the implementation.
- Allows us to easily swap data structures.

Node class

```
class Node{

    public:
        Student* data;
        Node*    next;
};
```

- We want Node to be a part of List, but we do not want it used anywhere else.

Node class

```
class List{
    class Node{

        public:
            Student* data;
            Node*    next;
    };

    public:

    private:
        Node* head;

};
```

- We want `Node` to be a part of `List`, but we do not want it used anywhere else.

- We can make `Node` a ***private class*** inside the `List` class.

Node class

```
class List{
    class Node{

        public:
            Student* data;
            Node*    next;
    };

    public:

    private:
        Node* head;

};
```

- We want Node to be a part of List, but we do not want it used anywhere else.
- We can make Node a ***private class*** inside the List class.
- We can make Node members public.

Node class

```
class List{
    class Node{

        public:
            Student* data;
            Node*    next;
    };

    public:

    private:
        Node* head;

};
```

- We want Node to be a part of List, but we do not want it used anywhere else.
- We can make Node a ***private class*** inside the List class.
- We can make Node members public.
    - List will see them, but since Node is private, no one else can

Node class

```
class List{
    class Node{

        public:
            Student* data;
            Node*    next;
    };

    public:

    private:
        Node* head;

};
```

- We want Node to be a part of List, but we do not want it used anywhere else.
- We can make Node a ***private class*** inside the List class.
- We can make Node members public.
  - List will see them, but since Node is private, no one else can
- **Coding example <p1>** - make List.h

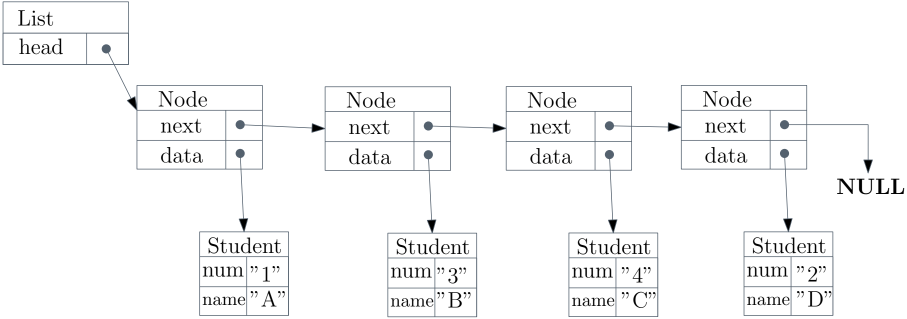
List Traversal

We will start with the "easiest" use case:

- Traversal

We want to visit every element of the `List`.

- Perhaps to call some operation on each data element
- For example: `print`



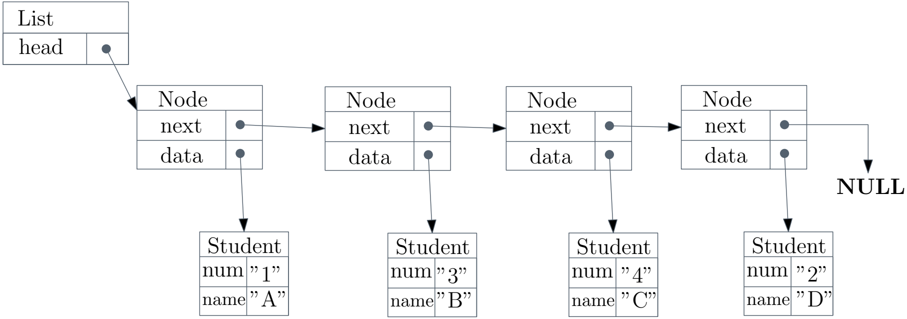
List Traversal

We will need a pointer to keep track of the `current Node`.

- We can get to the next node using

  ```
  current = current.next;
  ```

- Then call `print` on the `data`.



Element Insertion

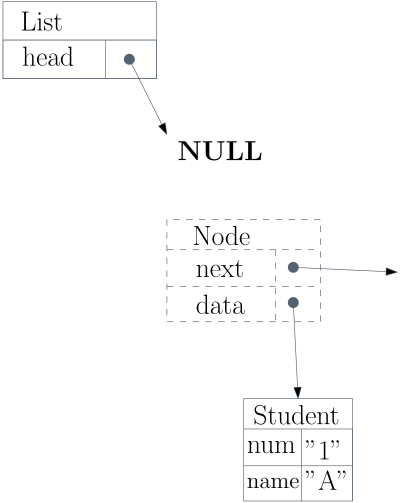
We can insert anywhere (but we need a pointer to that location)

- inserting is then shifting some pointer values

- Consider four cases:

1. an element is added to an empty list
2. an element is inserted at the beginning
3. an element is inserted in the middle
4. an element is inserted at the end

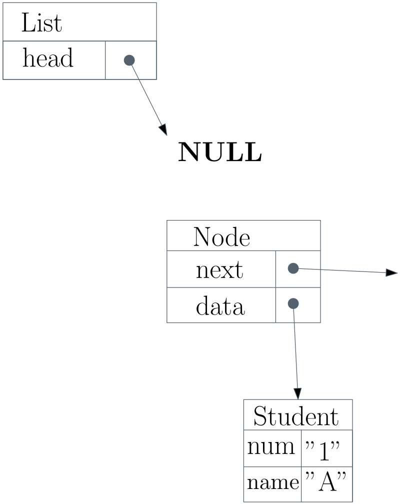
Element Insertion - Case 1 - Empty



Call to `add(Student*)` on an empty `List`

We make a new `Node`

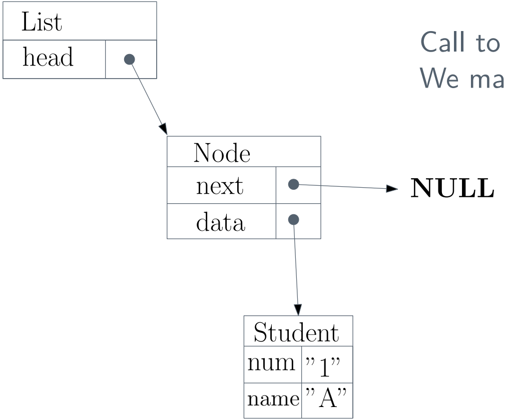
Element Insertion - Case 1 - Empty



Call to `add(Student*)` on an empty `List`

We make a new `Node`

Element Insertion - Case 1 - Empty

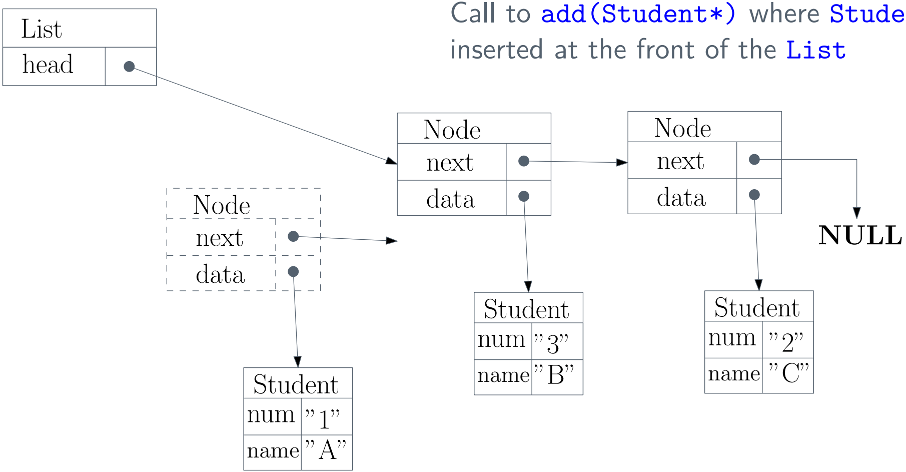


Call to add(Student*) on an empty List

We make a new Node

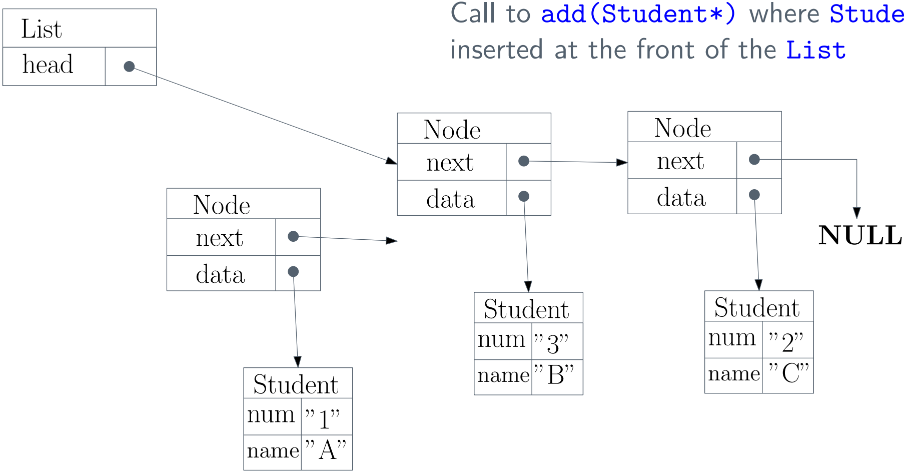
Element Insertion - Case 2 - Beginning

Call to add(Student*) where Student* is inserted at the front of the List



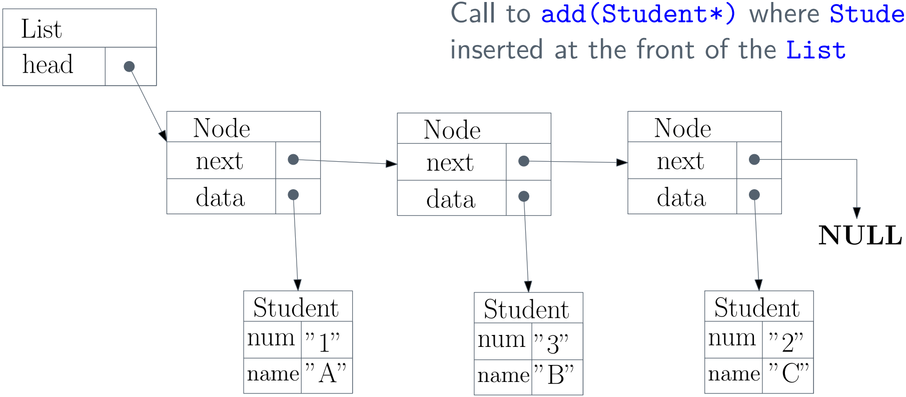
Element Insertion - Case 2 - Beginning

Call to add(Student*) where Student* is inserted at the front of the List

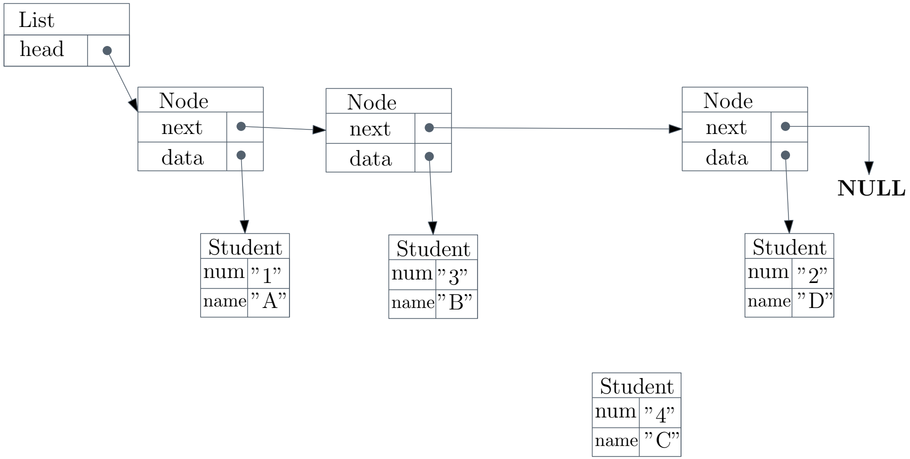


Element Insertion - Case 2 - Beginning

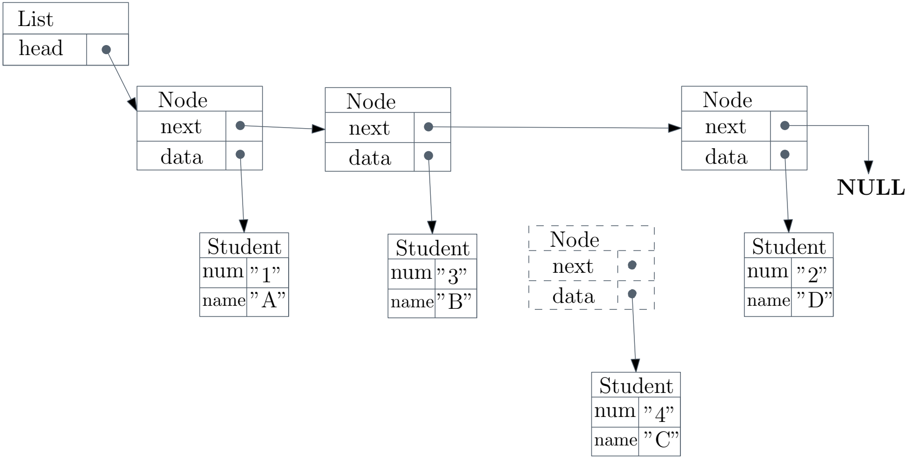
Call to `add(Student*)` where `Student*` is inserted at the front of the `List`



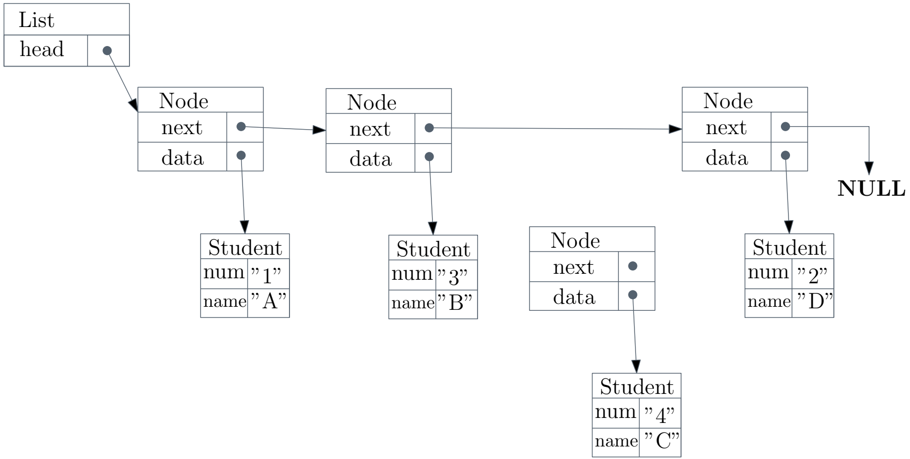
Element Insertion - Case 3 - Middle



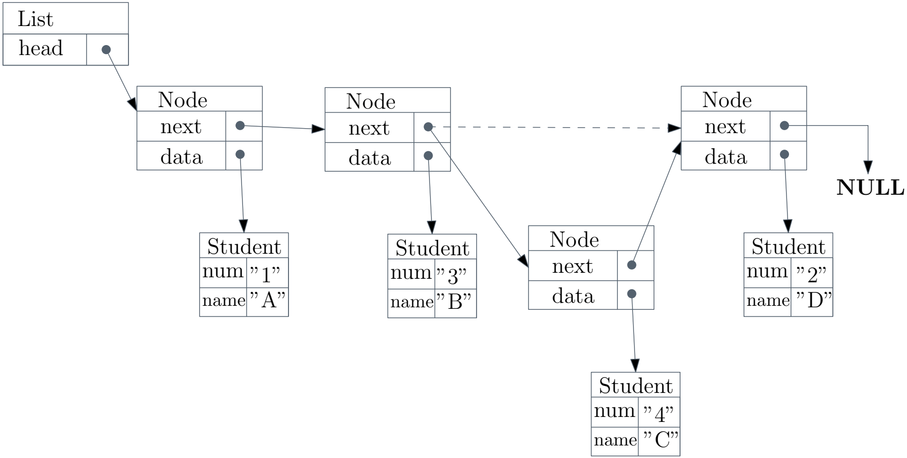
Element Insertion - Case 3 - Middle



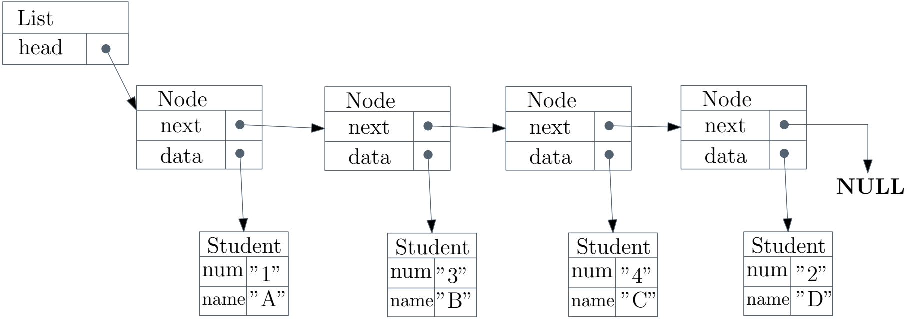
Element Insertion - Case 3 - Middle



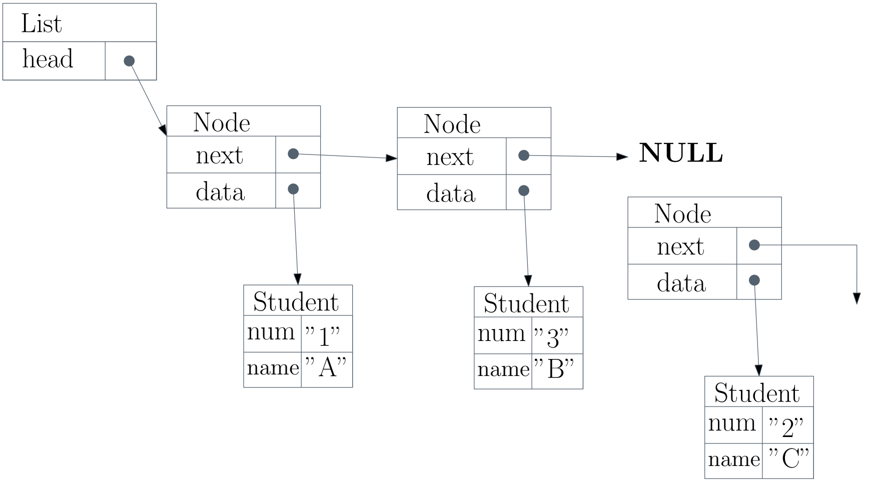
Element Insertion - Case 3 - Middle



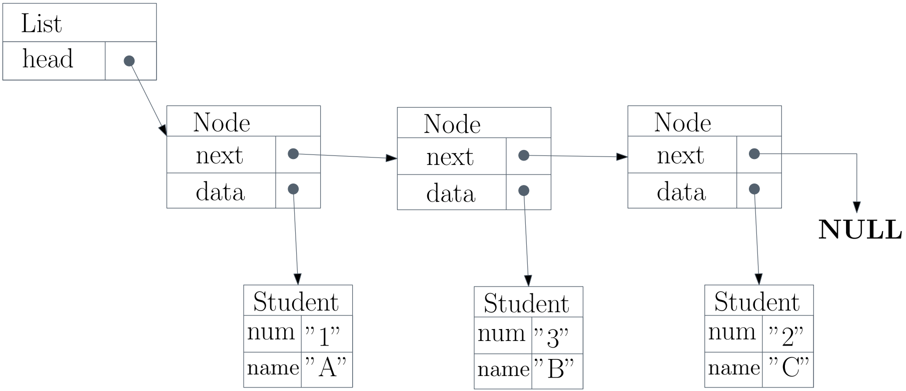
Element Insertion - Case 3 - Middle



Element Insertion - Case 4 - End



Element Insertion - Case 4 - End



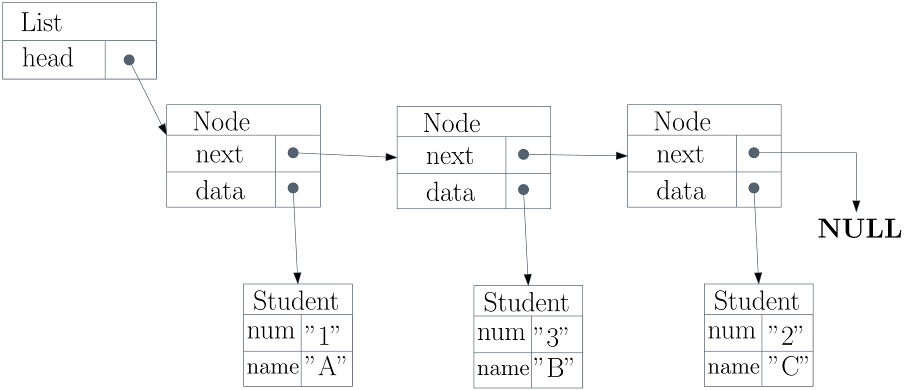
Element Deletion

We can remove an element from anywhere

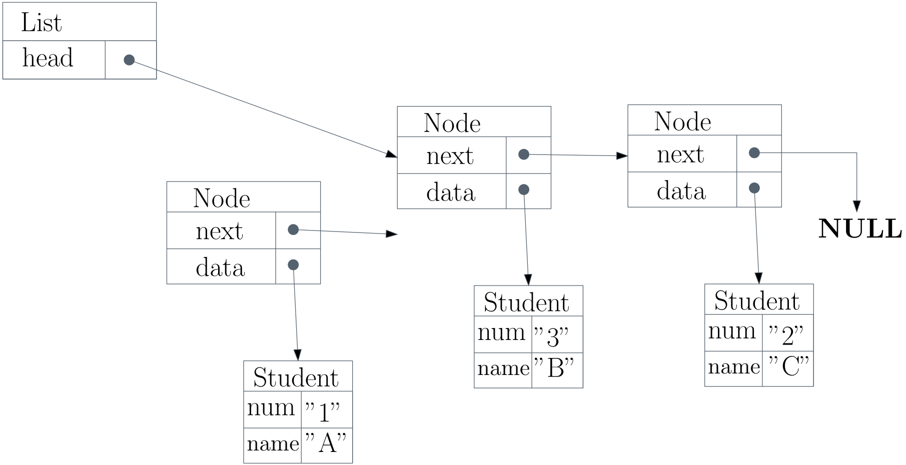
- pointer operations are add in reverse
- though we allocated memory - what to delete?
  - node, data or both?
- Consider five cases:

1. the list is empty!
2. an element is removed from the beginning
3. an element is removed from the middle
4. an element is removed from the end
5. Both 2 and 4 above (last element is removed)

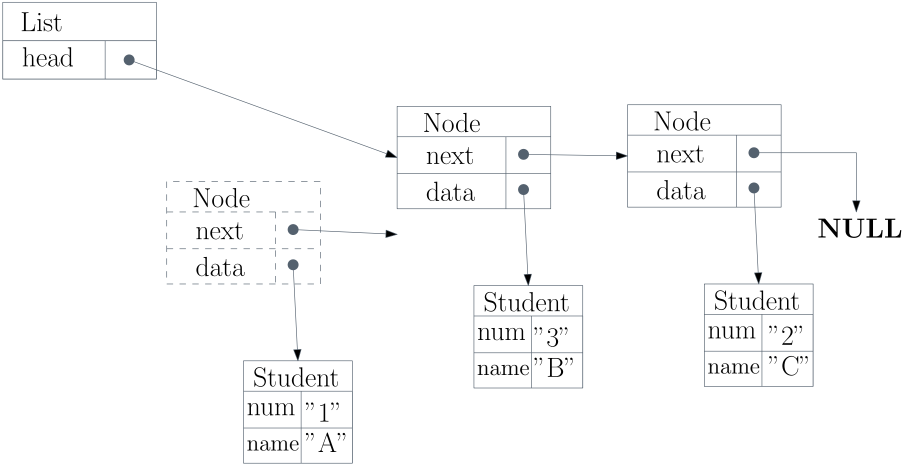
Element Deletion - Case 2 - Beginning



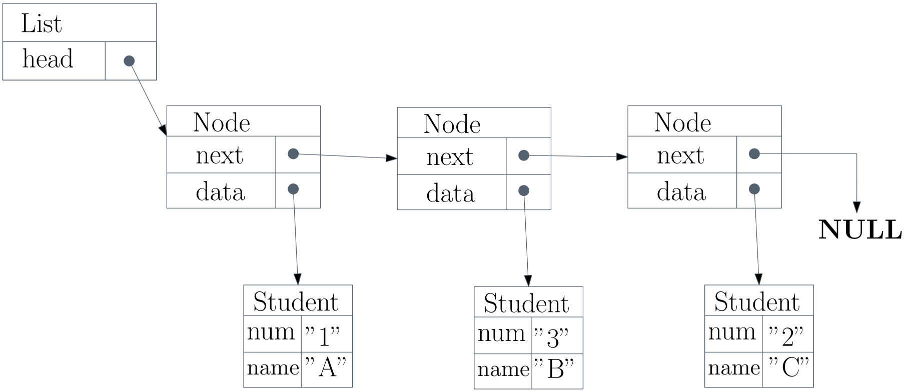
Element Deletion - Case 2 - Beginning



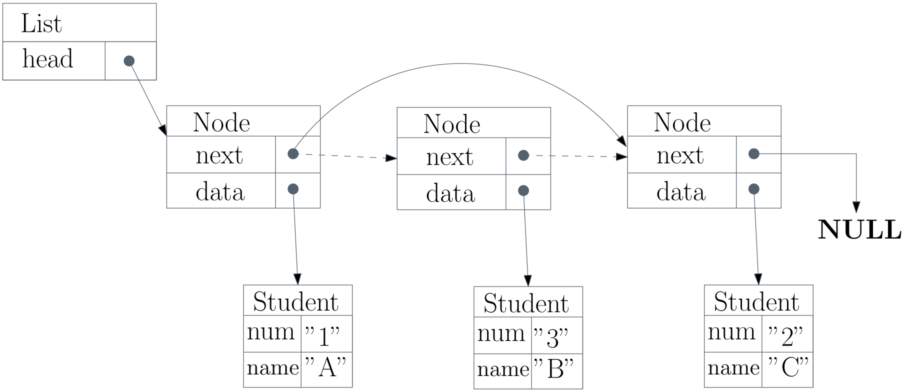
Element Deletion - Case 2 - Beginning



Element Deletion - Case 3 - Middle



Element Deletion - Case 3 - Middle



List

head

Node

next

data

Node

next

data

Node

next

data

NULL

Student

num ”1”

name ”A”

Student

num ”3”

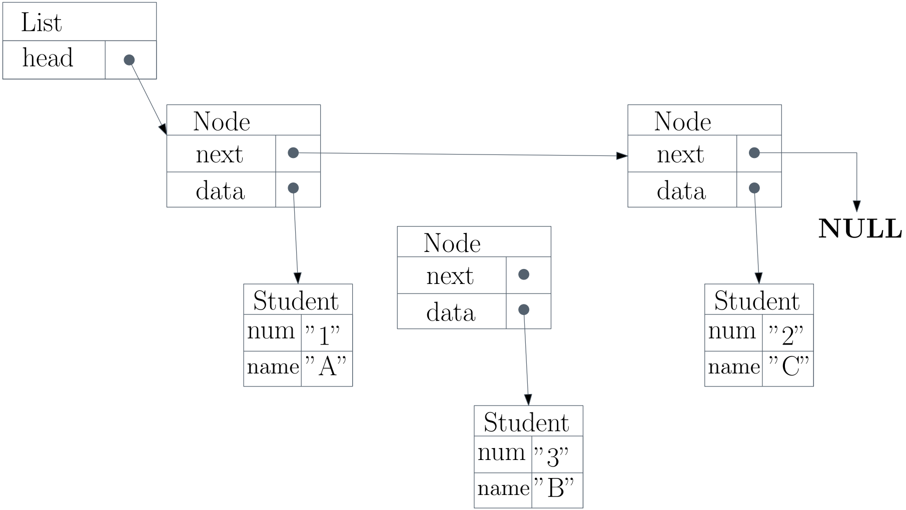
name ”B”

Student

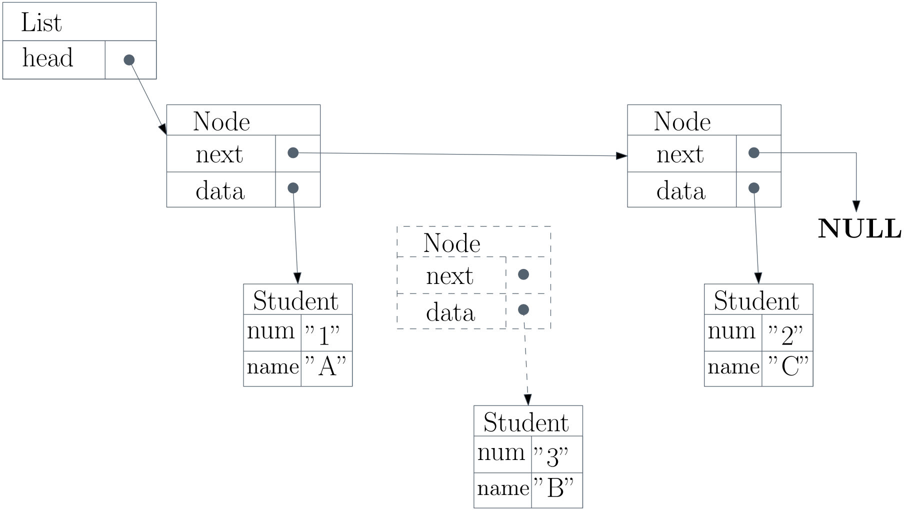
num ”2”

name ”C”

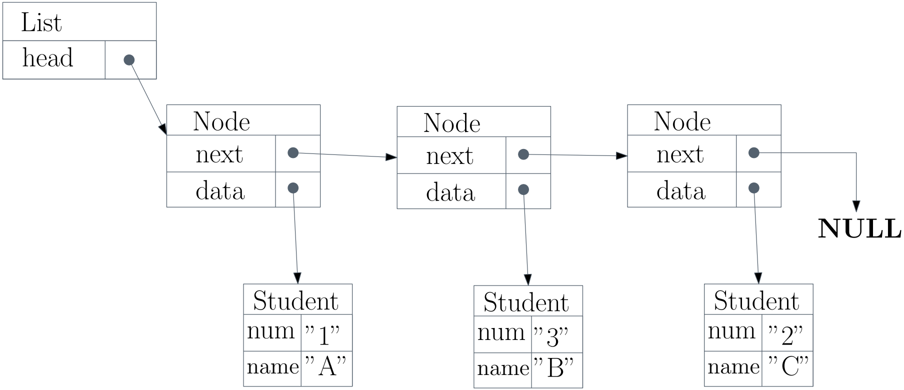
Element Deletion - Case 3 - Middle



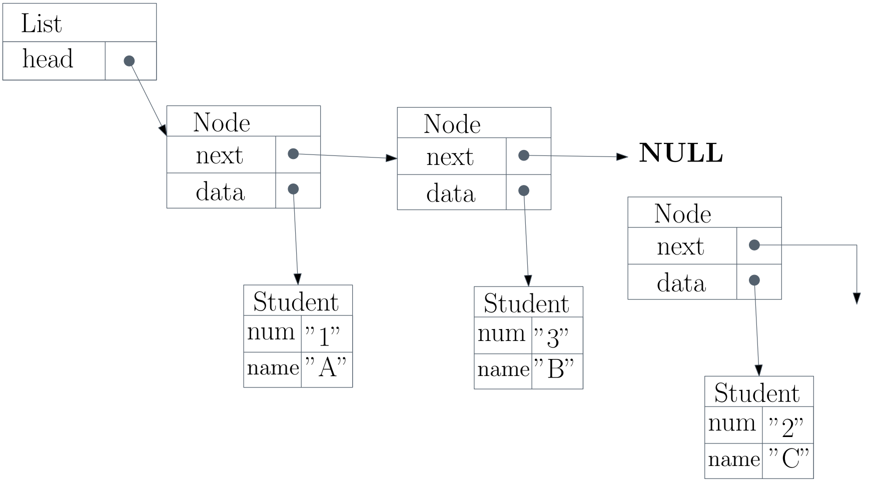
Element Deletion - Case 3 - Middle



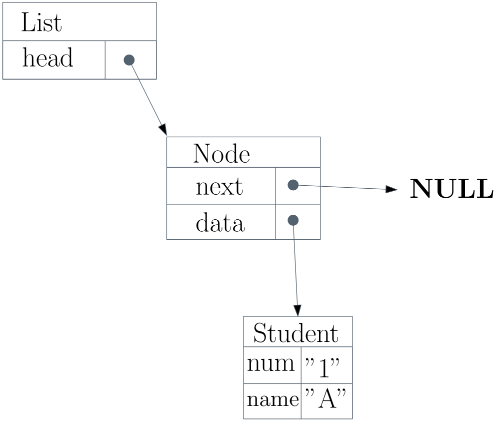
Element Deletion - Case 4 - End



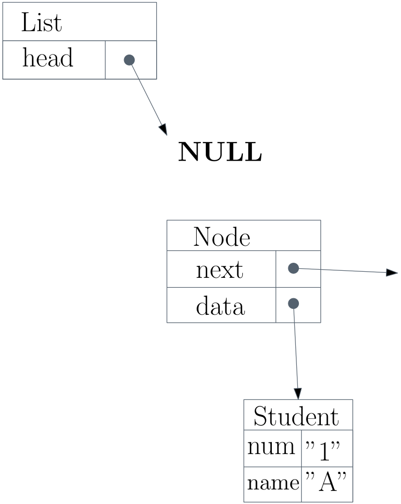
Element Deletion - Case 4 - End



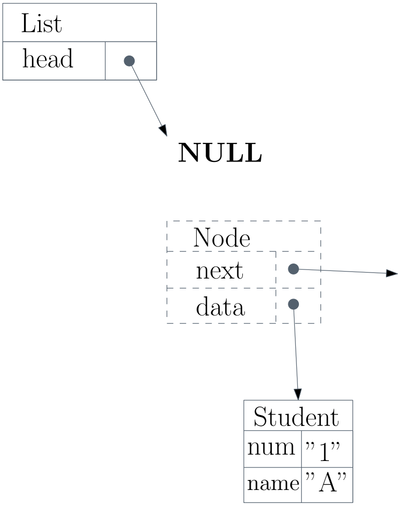
Element Deletion - Case 5 - Only Element



Element Deletion - Case 5 - Only Element



Element Deletion - Case 5 - Only Element



List Cleanup (Deleting this List)

Explicitly deallocate any allocated memory in the destructor

- If we call `new` within the class, we call `delete` within the class.

Deleting Nodes

- Always deallocate the remaining `Nodes` when `List` is deleted.
  - they are the responsibility of the `List` class
- Always deallocate the `Node` of deleted data
  - unless - it might be more efficient to use a pool of `Nodes`.

Deleting Data

- Depends on the application
- Default is to **NOT** delete stored data
  - It might also be stored somewhere else

Linked List Variants

Doubly-Linked-List

- each `Node` has `Node*` previous and `Node*` next
- can also make it circular

Indexed List

- how could we implement a `Linked List` using indices?
  - `add(int, data)`, `remove(int)`

Tail pointer

- `addLast` becomes simpler and more efficient

These depend on your needs