Linked Lists, Queues, and Priority Queues

黃世強 (Sai-Keung Wong)

College of Computer Science

National Yang Ming Chiao Tung University

Taiwan

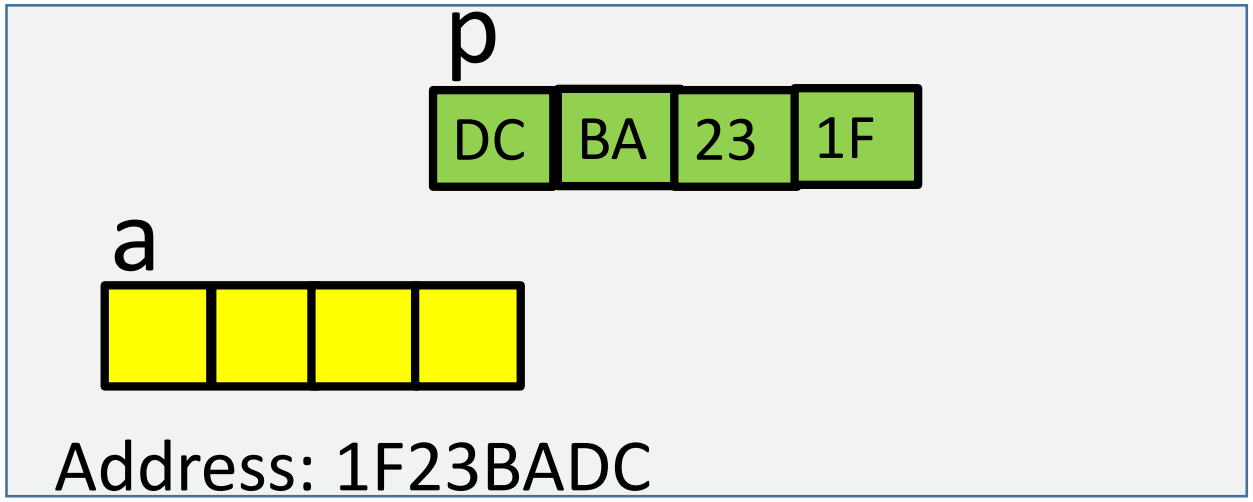
Linked List

Why do we need a linked list?

A linked list is efficient for storing and managing a varying number of elements.

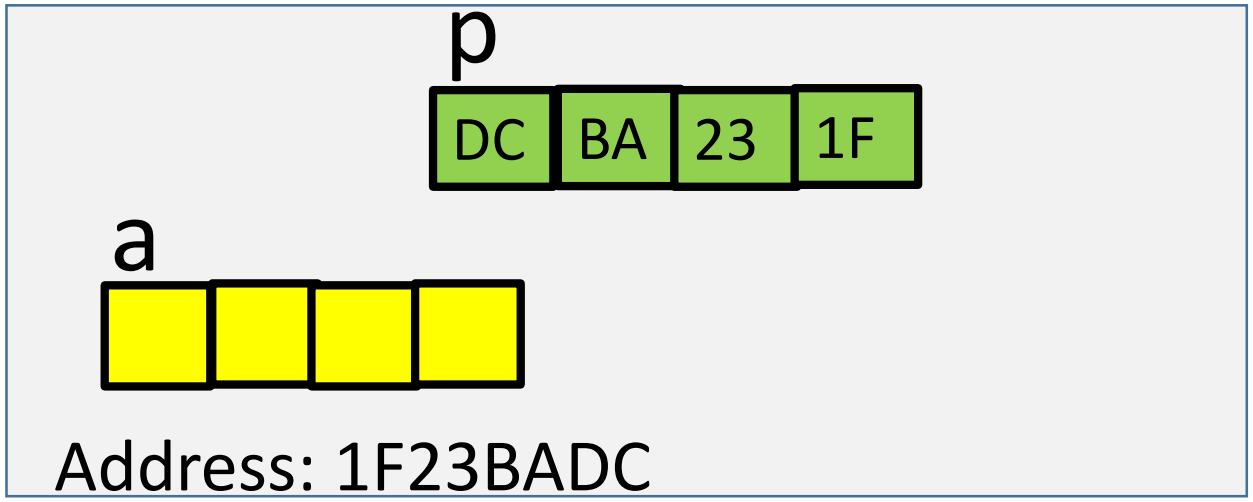
Motivation

```
int a;          // an integer variable
int *p;         // a pointer to an integer

p = &a;         // assign the address of a to p
```



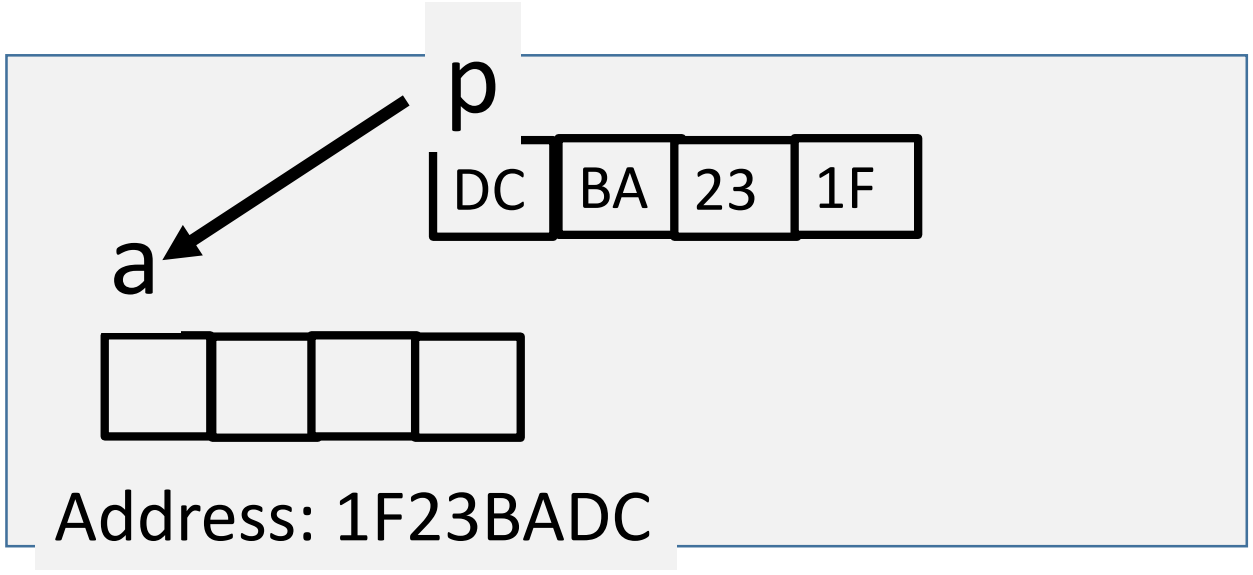
Motivation

```
int a;          // an integer variable
int *p;         // a pointer to an integer

p = &a;         // assign the address of a to p
```



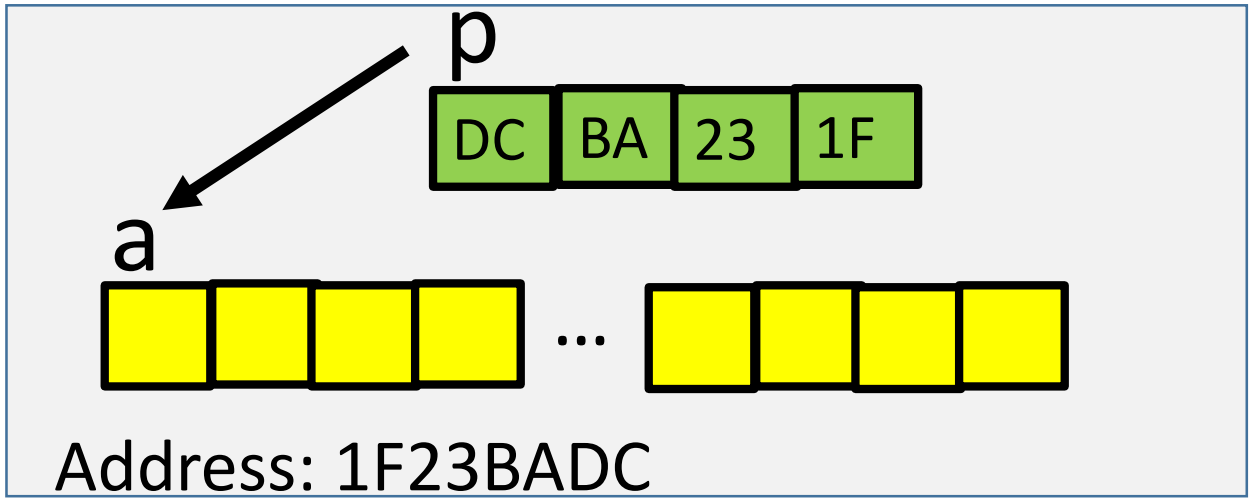
Motivation

```
int a;          // an integer variable
int *p;         // a pointer to an integer

p = &a;         // assign the address of a to p
```



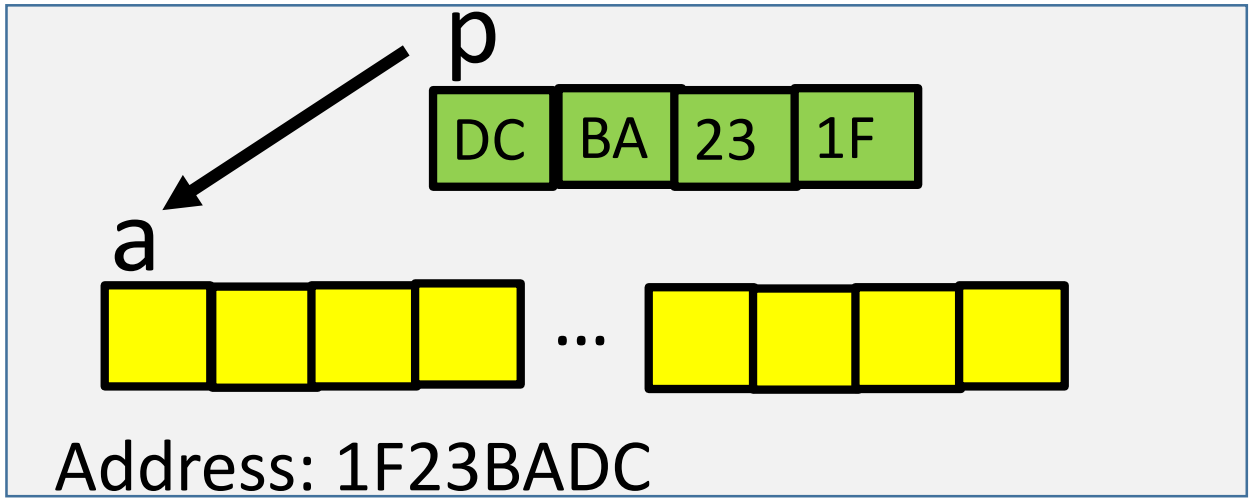
Motivation

```
X a;          // an object of X
X *p;         // a pointer to an object of X

p = &a;       // assign the address of a to p
```



Motivation

```
X a, b;          // objects of X
X *p;            // a pointer to an object of X

p = &a;          // assign the address of a to p
```




```
class X {
        ...
        void foo( );
        double d;
};
p = &a;
p-> foo( );
p->d;

p = &b;
p->...
```

Nodes in Linked Lists

- A linked list consists of nodes.
- Each node contains **an element**, and **a pointer** which **points to its next neighbor**.

Nodes in Linked Lists

- A linked list consists of nodes.
- Each node contains **an element**, and **a pointer** which **points to its next neighbor**.

```
template
<typename T = int>
class Node {
   T element;
   Node *next;
};
```

Nodes in Linked Lists

- A linked list consists of nodes.
- Each node contains **an element**, and **a pointer** which **points to its next neighbor**.

```
template
<typename T = int>
class Node {
  T element;
  Node *next;
};
```

```
Node *head, *tail;
Node node1, node2, …,node98, node99;
head = &node1;  node1.next = &node2; … node98.next = &node99; …; node99.next = 0;
tail = &node99.
```

Nodes in Linked Lists

- A linked list consists of nodes.
- Each node contains **an element**, and **a pointer** which **points to its next neighbor**.

```
template
<typename T = int>
class Node {
  T element;
  Node *next;
};
```

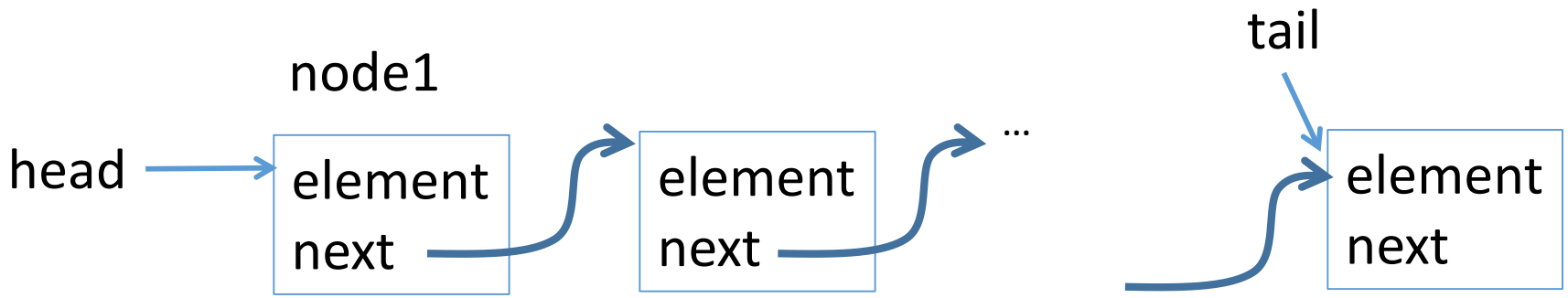
head and tail point to the first and the last nodes, respectively.

```
Node *head, *tail;
Node node1, node2, …,node98, node99;
head = &node1;  node1.next = &node2; … node98.next = &node99; …; node99.next = 0;
tail = &node99.
```

Nodes in Linked Lists

- A linked list consists of nodes.
- Each node contains **an element**, and **a pointer** which **points to its next neighbor**.

```
template
<typename T = int>
class Node {
  T element;
  Node *next;
};
```

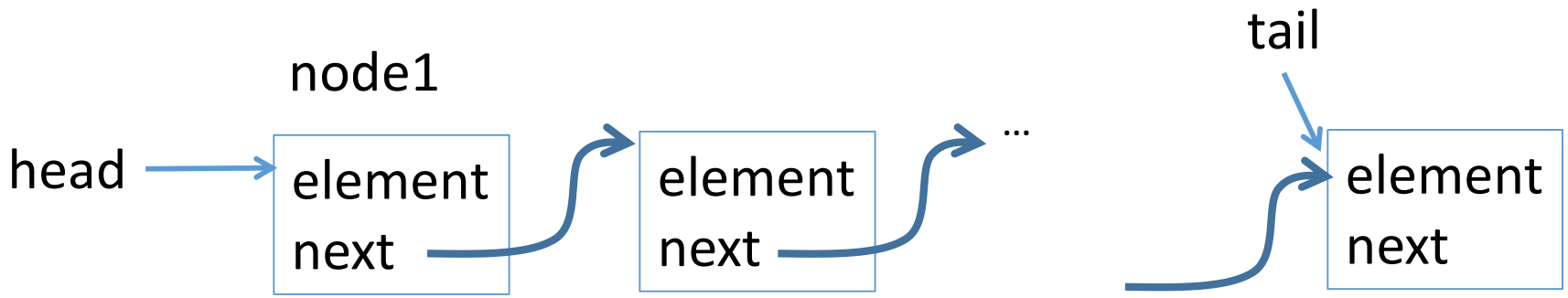



```
Node *head, *tail;
Node node1, node2, …,node98, node99;
head = &node1;  node1.next = &node2; … node98.next = &node99; …; node99.next = 0;
tail = &node99.
```

#define NULL 0

nullptr //C11

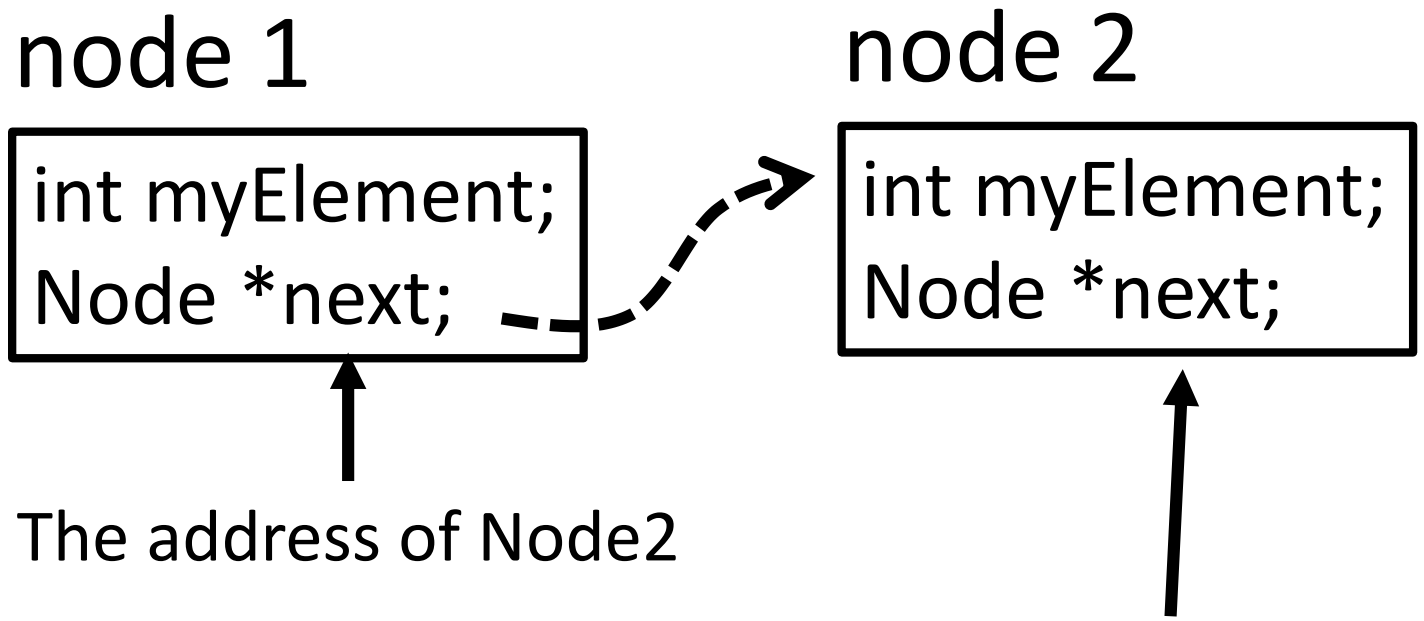
```
template
<typename T = int>
class Node {
  T element;
  Node *next;
};
```




```
Node *head, *tail;
Node node1, node2, …,node98, node99;
head = &node1;  node1.next = &node2; … node98.next = &node99; …; node99.next = 0;
tail = &node99.
```

Nodes

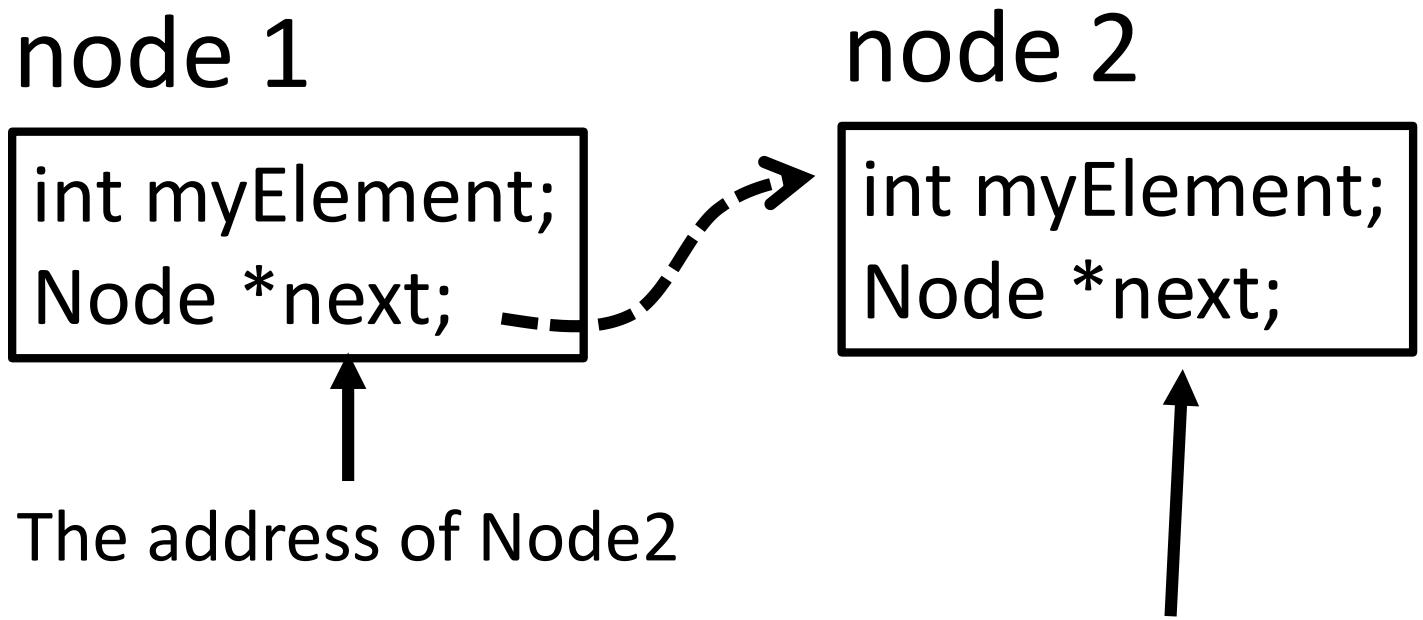
Each node contains **an element**, and **a pointer** which **points to its next neighbor**.



If next == 0, there is no more nodes.

Nodes

Each node contains **an element**, and **a pointer** which **points to its next neighbor**.

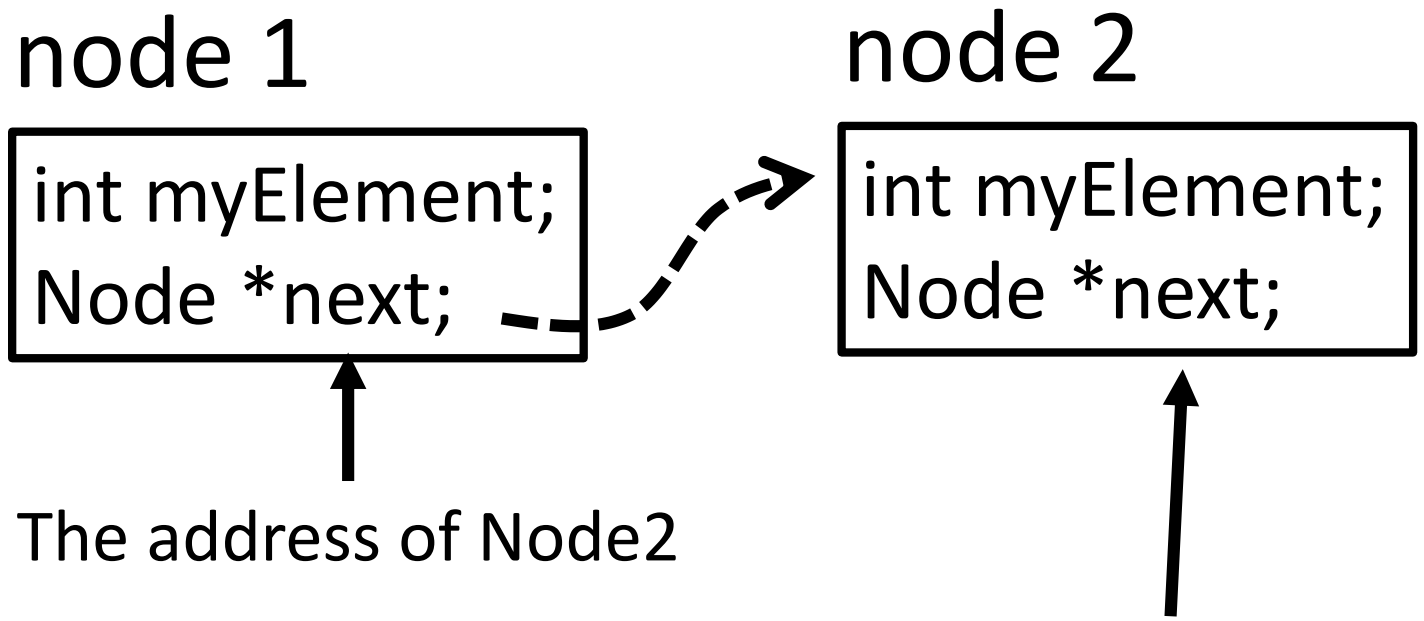



```
Node node1, node2;
node1.next = &node2;
node2.next = 0;
```

If next == 0, there is no more nodes.

Nodes

Each node contains **an element**, and **a pointer** which **points to its next neighbor**.

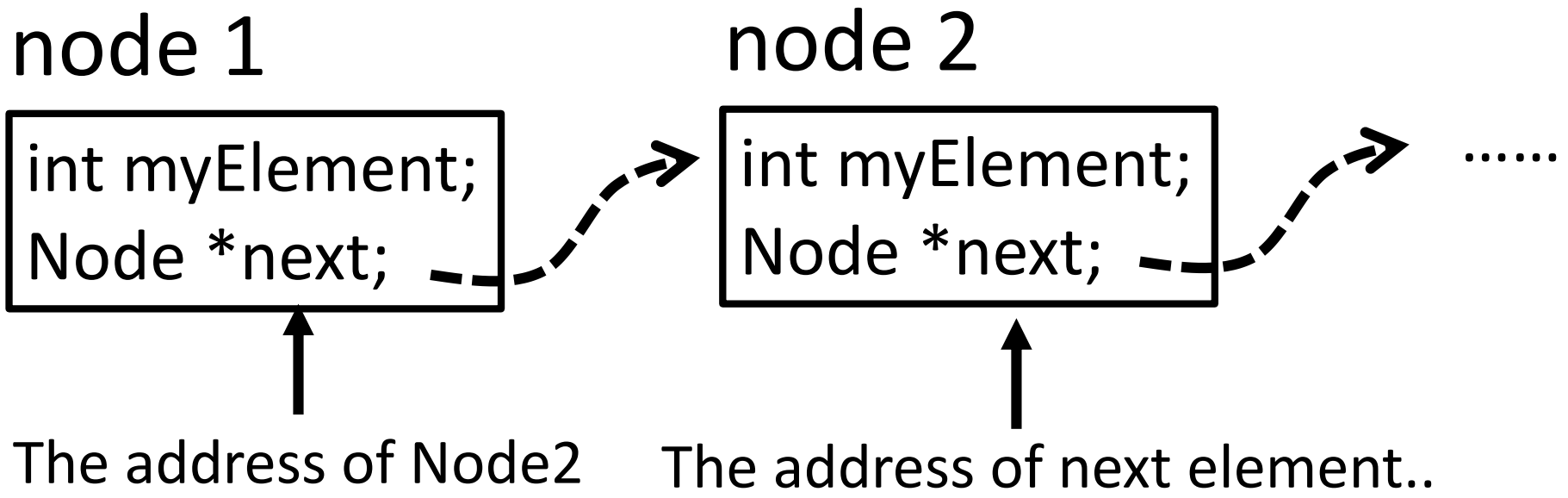


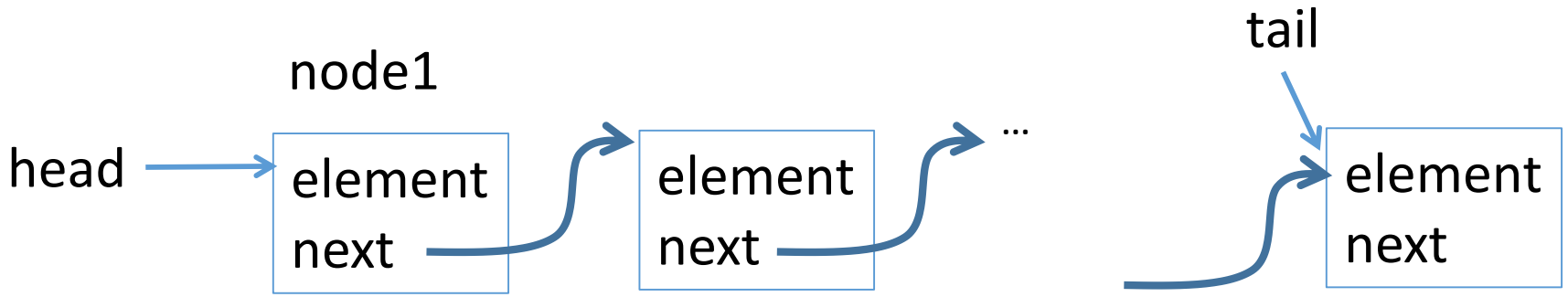

```
Node node1, node2;
node1.next = &node2;
node2.next = 0;
```

```
Node *node1, *node2;
node1 = new Node;
node2 = new Node;
node1->next = node2;
node2.next = 0;
```

If next == 0, there is no more nodes.

Nodes


node 1
int myElement;
Node *next;
node 2
int myElement;
Node *next;
.....
The address of Node2
The address of next element..

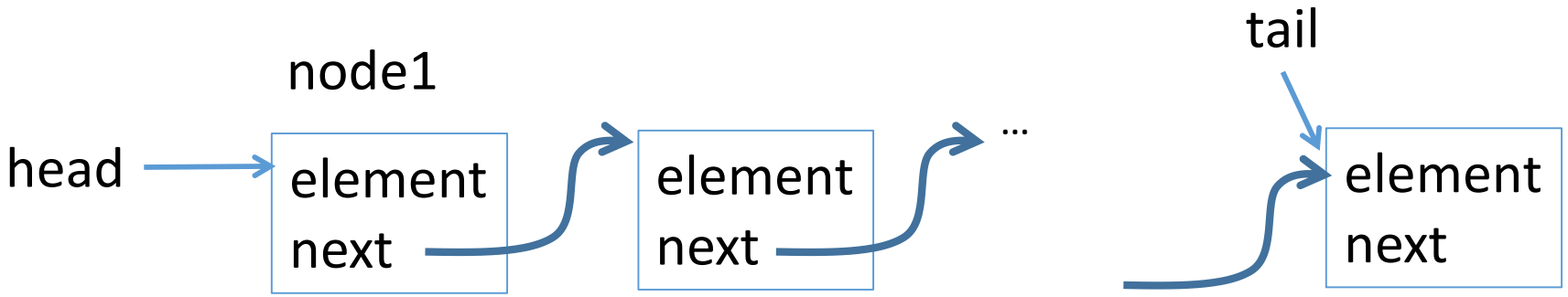

node1
tail
head
element
next
element
next
...
element
next

Nodes

How to tell a node is the last one? Maintain a tail pointer.

Node *tail; ….

bool flg_last_node = tail == &node;

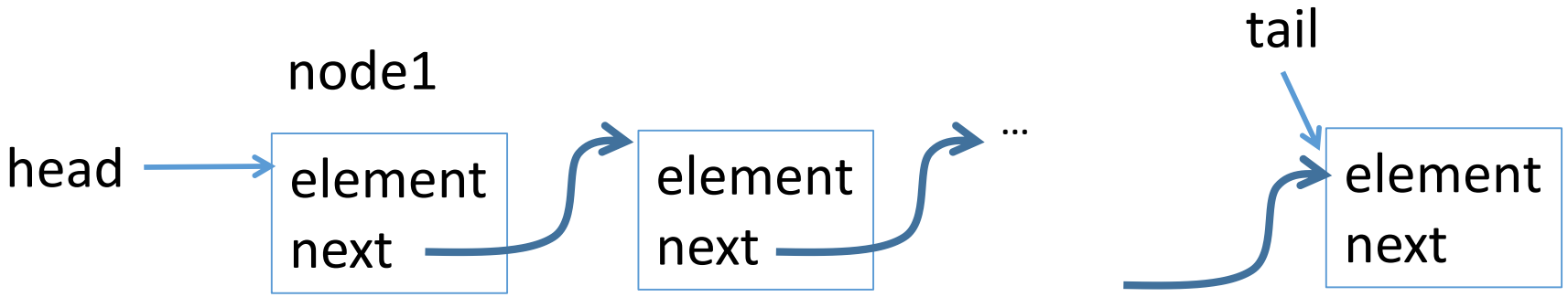


Nodes

How to tell a node is the last one? Maintain a tail pointer.

Node *tail; ….

bool flg_last_node = (tail == &node);

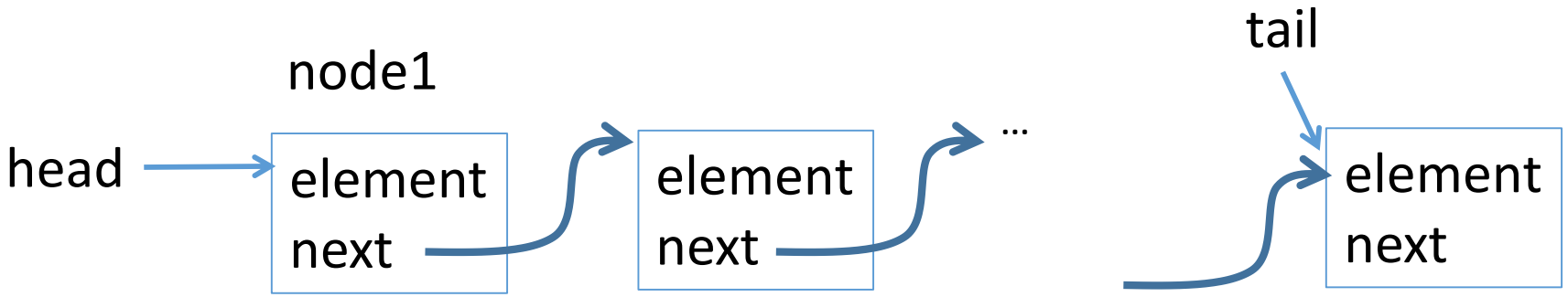


Nodes

How to tell a node is the last one? Maintain a tail pointer.

Node *tail; ….

bool flg_last_node = !node.next;



The Node Class

```
template < typename T >
class Node {
public:
  T element;  // The element contained in the node
  Node* next; // A pointer which points to the next node
  Node() { // No-arg constructor
    next = NULL;    // next = 0; or next = nullptr in C11
  }

  Node(const T &element) { // Constructor
    this->element = element;
    next = NULL;
  }
};
```

The head and tail pointers

- If the list is A1 both A2 and A3 should be A4. In this case, both pointers do not point to any node.

The head and tail pointers

- If the list is empty, both head and tail should be NULL. In this case, both pointers do not point to any node.

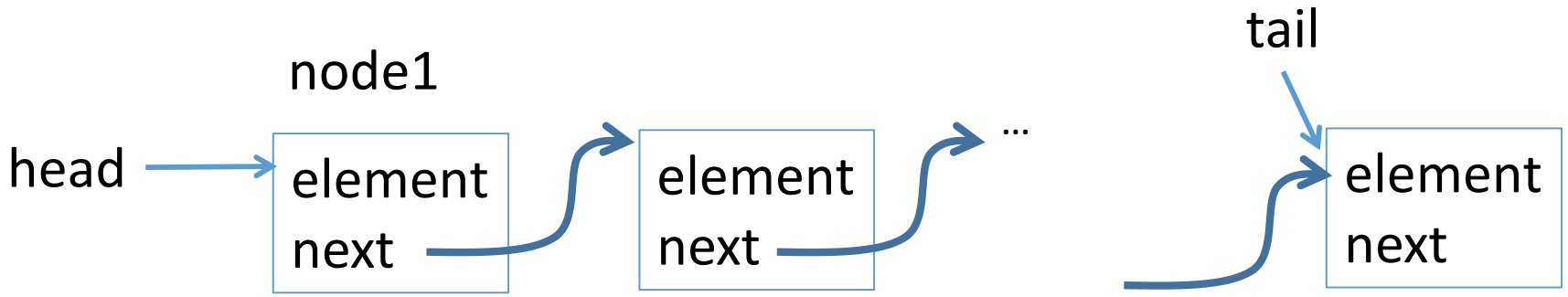
- NULL is defined in <iostream> and <cstddef>.

Implementation of a linked list

- Add the first node
- Add the other node(s)

What should be done?

1. head points to the first element.
2. tail points to the last element.
3. Set tail->next = 0



Implementation of a linked list

- Add the first node
- Add the other node(s)

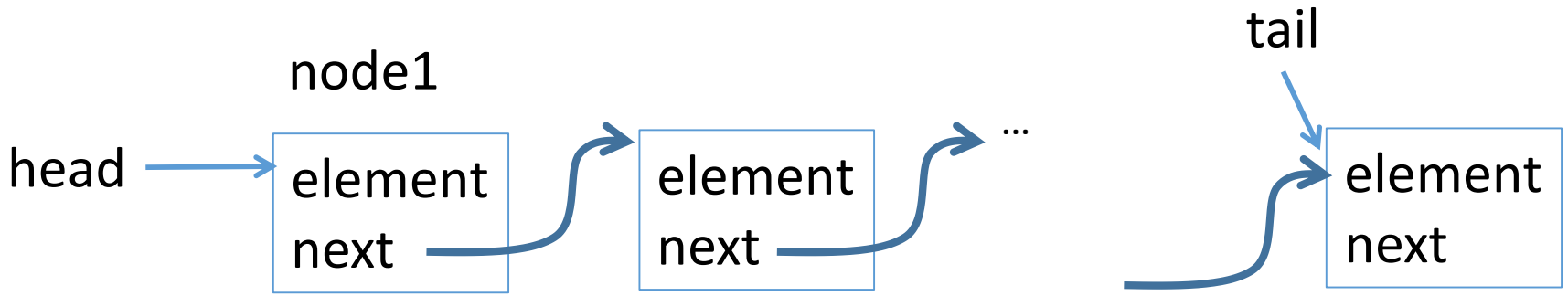
What should be done?

Maintain the head

Maintain the tail

Maintain the next pointer(s)

Set tail->next = 0



Node Class

```
template < typename T >
class Node {
public:
  T element;  // The element contained in the node
  Node* next; // A pointer which points to the next node
  Node() { // No-arg constructor
    next = NULL;
  }

  Node(T element) { // Constructor
    this->element = element;
    next = NULL;
  }
};
```

Example: Create a linked list for three nodes

The approach

1. Initialize the linked list
2. Add the first node (add as the last element)
3. Add the second node (add as the last element)
4. Add the third node (add as the last element)

When a node is added, the head and tail pointers must be updated properly.

Example: Create a linked list for three nodes

The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

```
template < typename T = int >
class Node {
public:
  T element; Node* next;
  Node() { next = NULL; }
  Node(T element) {
    this->element = element;
    next = NULL;
  }
};

Set next = 0;
```

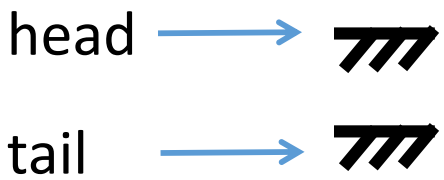
Example: Create a linked list for three nodes

The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

```
Initialize( );
```

```
Node *head, *tail;
void initialize( ) {
  head = tail = 0; // NULL
}
```



Example: Create a linked list for three nodes

The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

```
Initialize( );
Node *node1 = new Node; …
addNode(node1);
```

```
void addNode( Node *node ) {
  if (head == 0) {
     head = node;
  }
  tail = node;
  tail->next = 0;
}
```

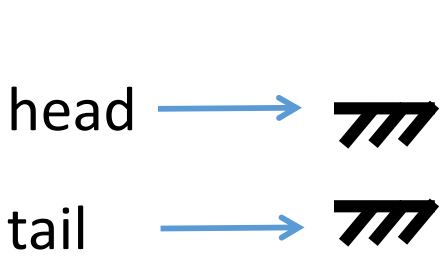
Example: Create a linked list for three nodes

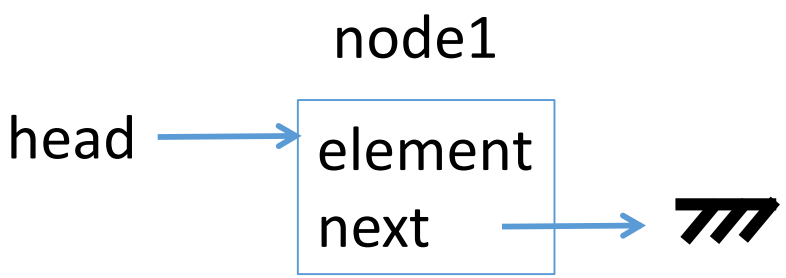
The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

```
Initialize( );
Node *node1 = new Node; …
addNode(node1);
```

```
void addNode( Node *node ) {
  if (head == 0) {
     head = node;
  }
  tail = node;
  tail->next = 0;
}
```





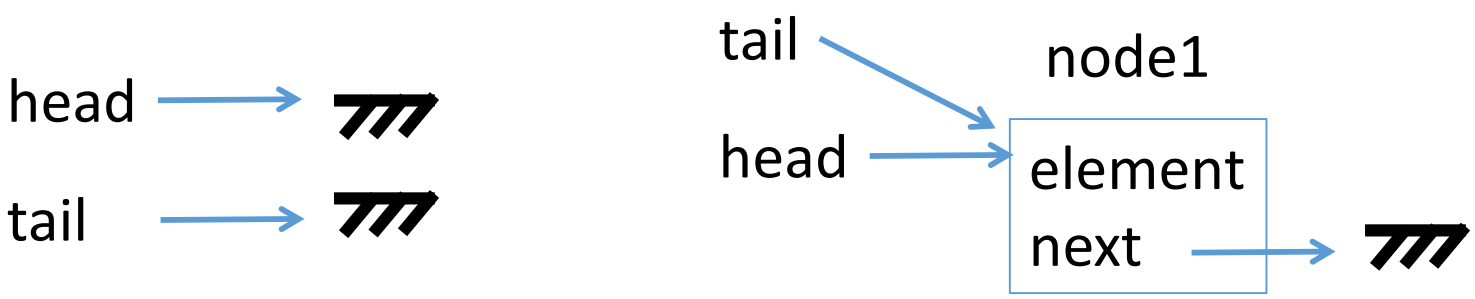
Example: Create a linked list for three nodes

The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

```
void addNode( Node *node ) {
  if (head == 0) {
     head = node;
  }
  tail = node;
  tail->next = 0;
}
```

```
Initialize( );
Node *node1 = new Node; …
addNode(node1);
```



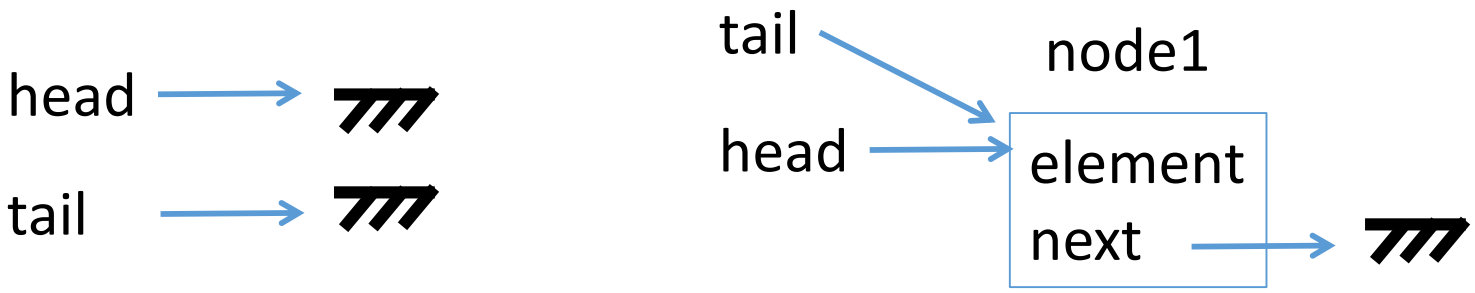
Example: Create a linked for three nodes

The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

```
void addNode( Node *node ) {
  if (head == 0) {
     head = node;
  }
  tail = node;
  tail->next = 0; // required! Why?
}
```

```
Initialize( );
Node *node1 = new Node; …
addNode(node1);
```



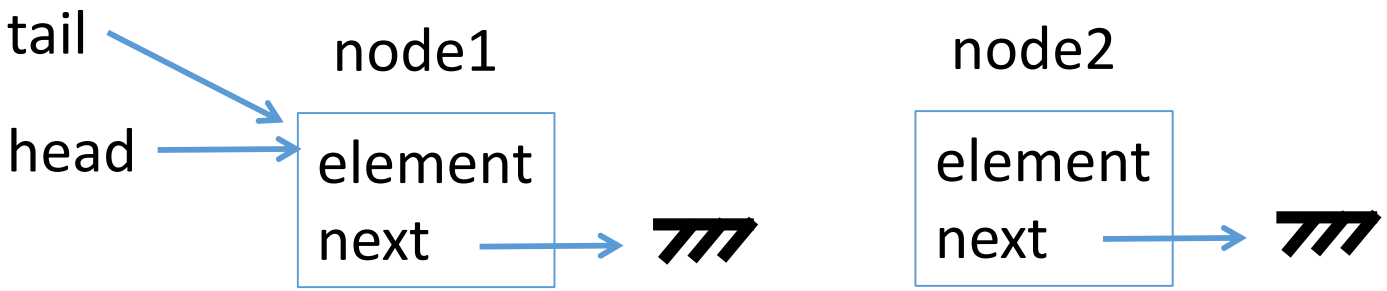
Example: Create a linked for three nodes

The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

```
void addNode( Node *node ) {
  if (head == 0) {
     head = node;
  }
  tail = node;
  tail->next = 0;
}
```

```
…
Node *node2 = new Node; …
addNode(node2);
```



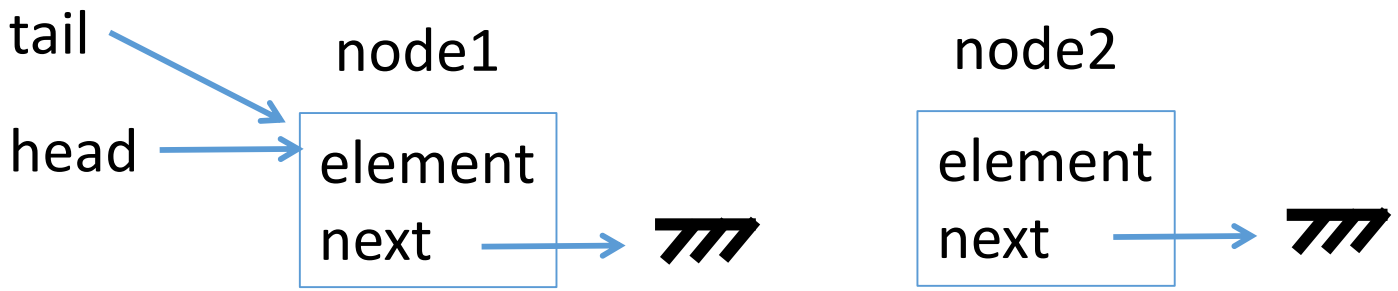
Example: Create a linked for three nodes

The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

```
…
Node *node2 = new Node; …
addNode(node2);
```

```
void addNode( Node *node ) {
  if (head == 0) {
     head = node;
  }
  tail = node;
  tail->next = 0;
}
```



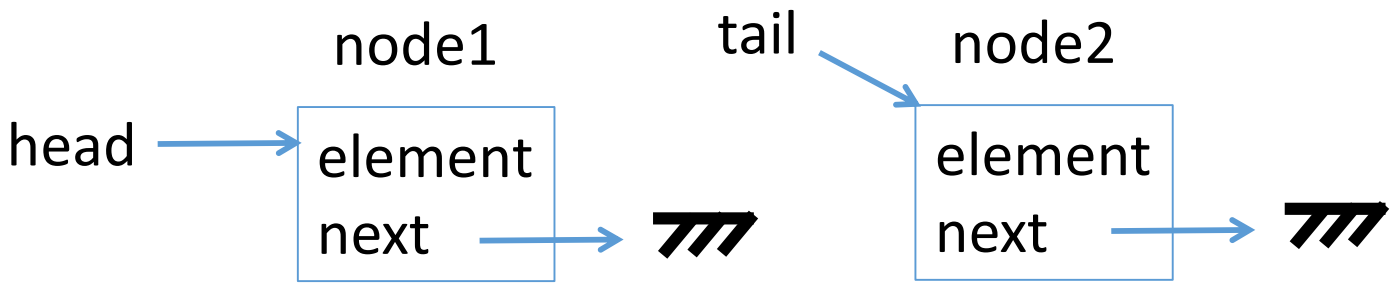
Example: Create a linked for three nodes

The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

```
…
Node *node2 = new Node; …
addNode(node2);
```

```
void addNode( Node *node ) {
  if (head == 0) {
     head = node;
  }
  tail = node;
  tail->next = 0;
}
```



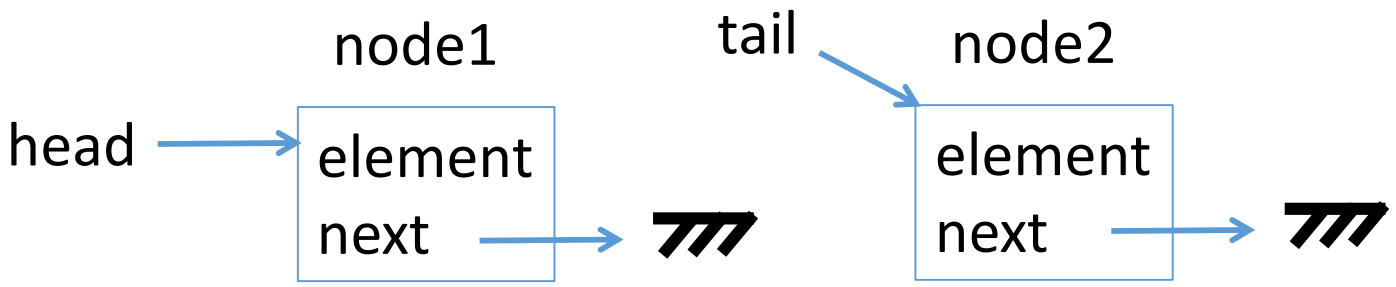
Example: Create a linked for three nodes

The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

```
…
Node *node2 = new Node; …
addNode(node2);
```

```
void addNode( Node *node ) {
  if (head == 0) {
     head = node;
  }
  tail = node;
  tail->next = 0;
}
```



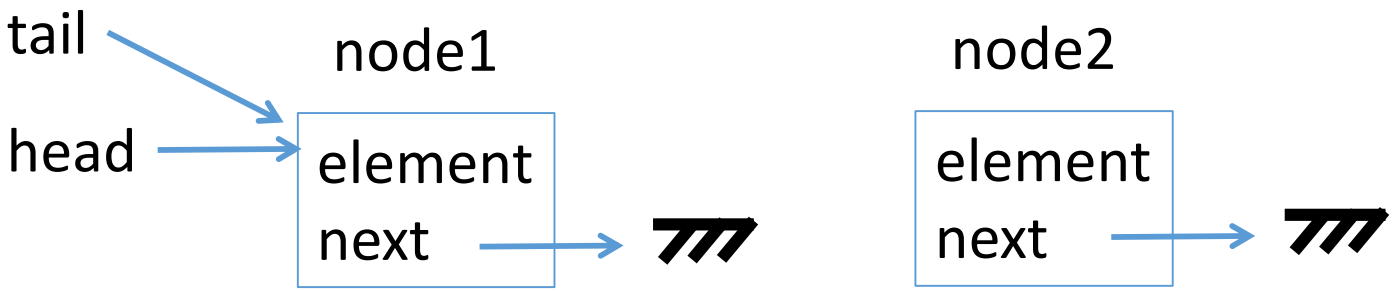
Example: Create a linked for three nodes

The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

```
…
Node *node2 = new Node; …
addNode(node2);
```

```
void addNode( Node *node ) {
  if (head == 0) {
     head = node;
  }

  tail = node;
  tail->next = 0;
}
```



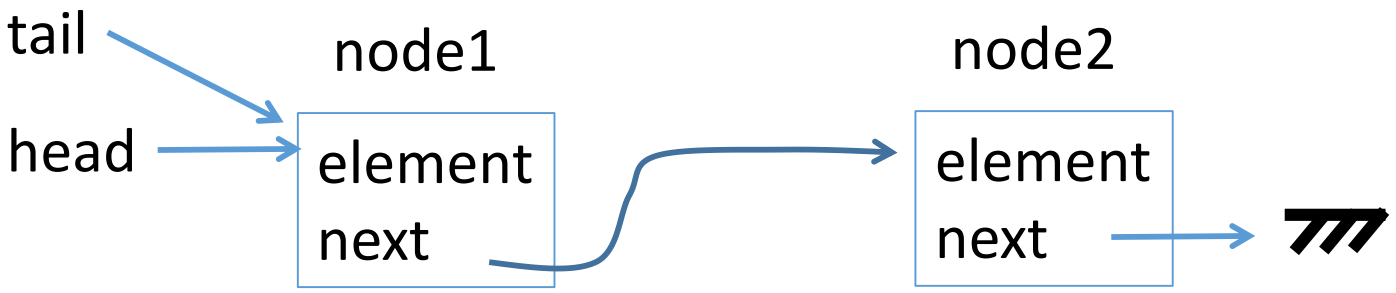
Example: Create a linked for three nodes

The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

```
…
Node *node2 = new Node; …
addNode(node2);
```

```
void addNode( Node *node ) {
  if (head == 0) {
     head = node;
  }
  if (tail) tail->next = node;
  tail = node;
  tail->next = 0;
}
```



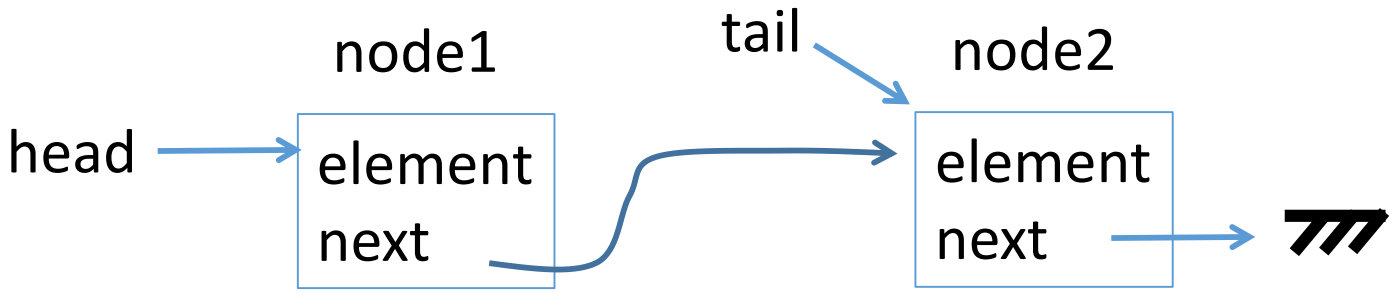
Example: Create a linked for three nodes

The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

```
…
Node *node2 = new Node; …
addNode(node2);
```

```
void addNode( Node *node ) {
  if (head == 0) {
     head = node;
  }
  if (tail) tail->next = node;
  tail = node;
  tail->next = 0;
}
```



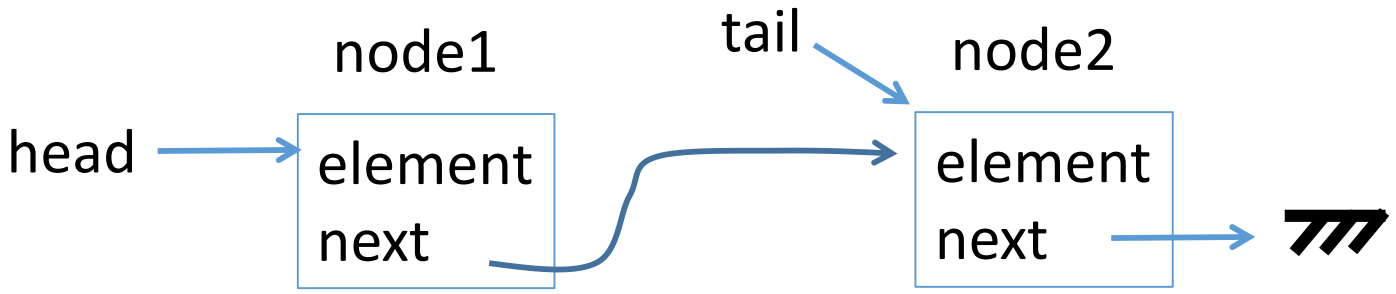
Example: Create a linked for three nodes

The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

```
…
Node *node2 = new Node; …
addNode(node2);
```

```
void addNode( Node *node ) {
  if (head == 0) {
     head = node;
  }
  if (tail) tail->next = node;
  tail = node;
  tail->next = 0;
}
```



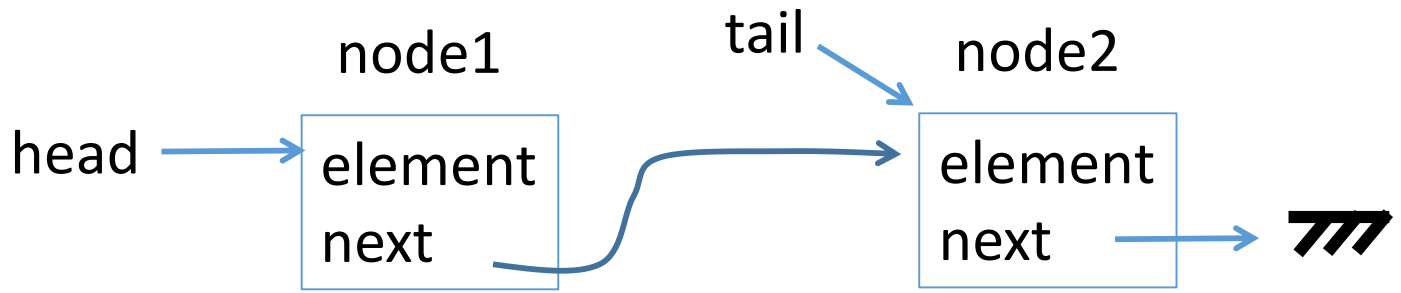
Example: Create a linked for three nodes

The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

```
…
Node *node2 = new Node; …
addNode(node2);
```

```
void addNode( Node *node ) {
  if (head == 0) {
     head = node;
  }
  if (tail) tail->next = node;
  tail = node;
  tail->next = 0;
}
```



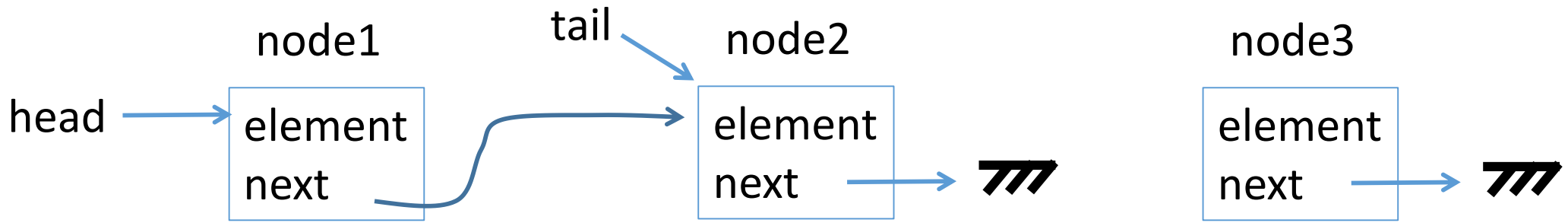
Example: Create a linked for three nodes

The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

```
…
Node *node3 = new Node; …
addNode(node3);
```

```
void addNode( Node *node ) {
  if (head == 0) {
     head = node;
  }
  if (tail) tail->next = node;
  tail = node;
  tail->next = 0;
}
```



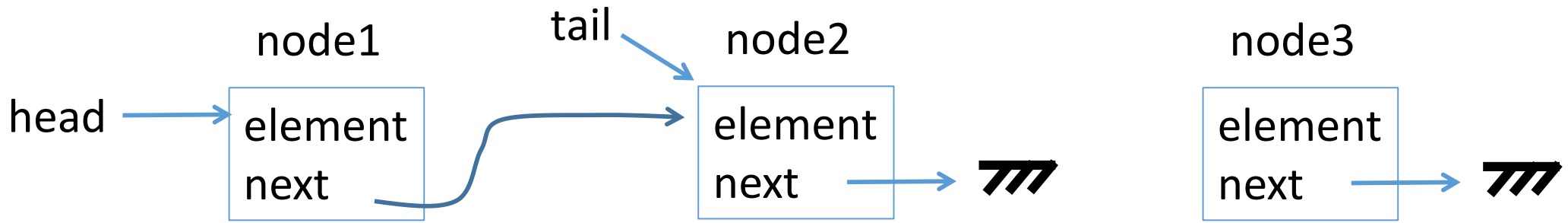
Example: Create a linked for three nodes

The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

```
…
Node *node3 = new Node; …
addNode(node3);
```

```
void addNode( Node *node ) {
  if (head == 0) {
     head = node;
  }
  if (tail) tail->next = node;
  tail = node;
  tail->next = 0;
}
```



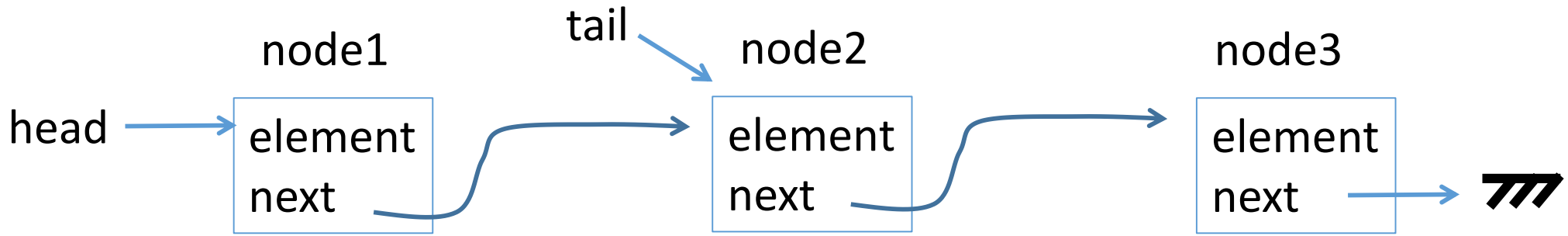
Example: Create a linked for three nodes

The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

```
void addNode( Node *node ) {
  if (head == 0) {
     head = node;
  }
  if (tail) tail->next = node;
  tail = node;
  tail->next = 0;
}
```

```
…
Node *node3 = new Node; …
addNode(node3);
```



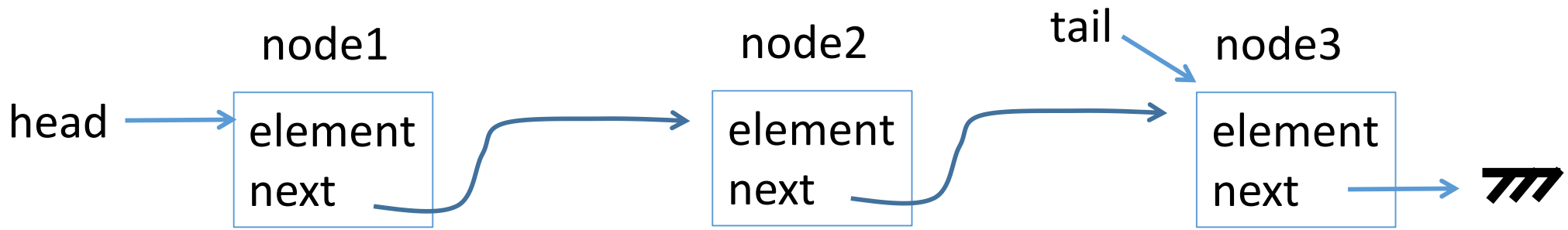
Example: Create a linked for three nodes

The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

```
…
Node *node3 = new Node; …
addNode(node3);
```

```
void addNode( Node *node ) {
  if (head == 0) {
     head = node;
  }
  if (tail) tail->next = node;
  tail = node;
  tail->next = 0;
}
```



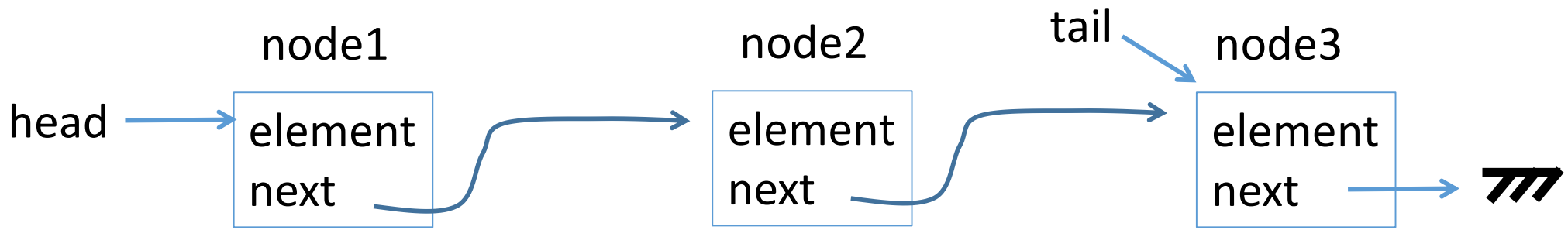
Example: Create a linked for three nodes

The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

```
…
Node *node3 = new Node; …
addNode(node3);
```

```
void addNode( Node *node ) {
  if (head == 0) {
     head = node;
  }
  if (tail) tail->next = node;
  tail = node;
  tail->next = 0;
}
```



Example: Create a linked for three nodes

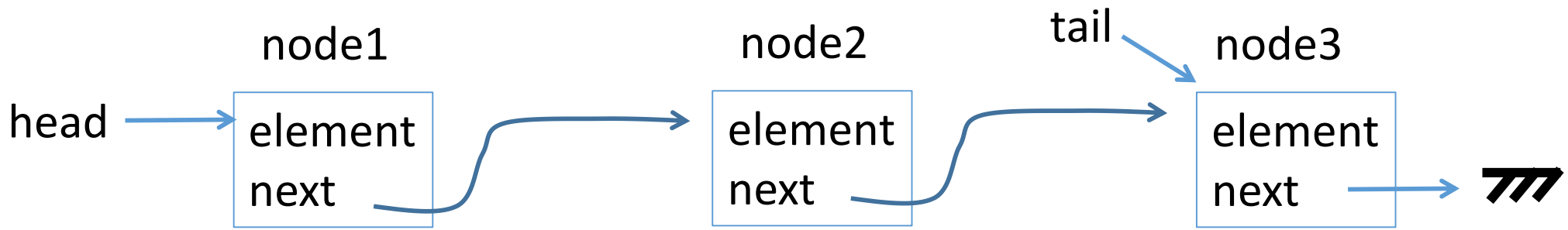
The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

```
…
Node *node3 = new Node; …
addNode(node3);
```

```
void addNode( Node *node ) {
  if (head == 0) {
     head = node;
  }
  if (tail) tail->next = node;
  tail = node;
  tail->next = 0;
}
```

How to eliminate the if-structure?



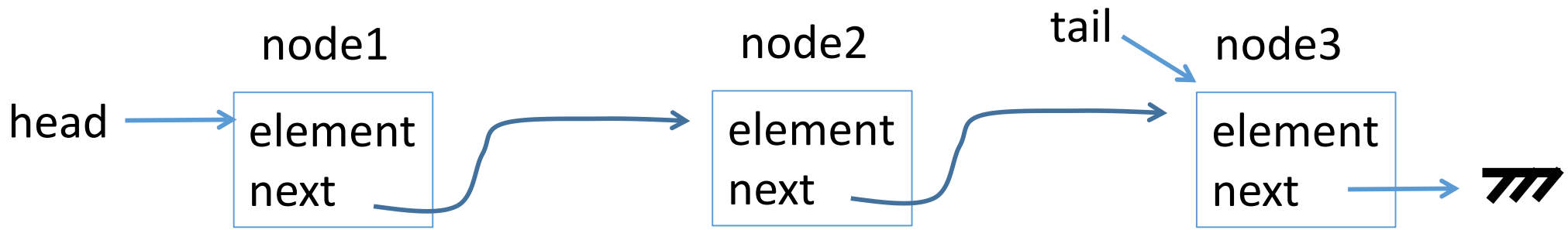
Example: Create a linked for three nodes

The approach

1. Initialize the linked list
2. Add the first node
3. Add the second node
4. Add the third node

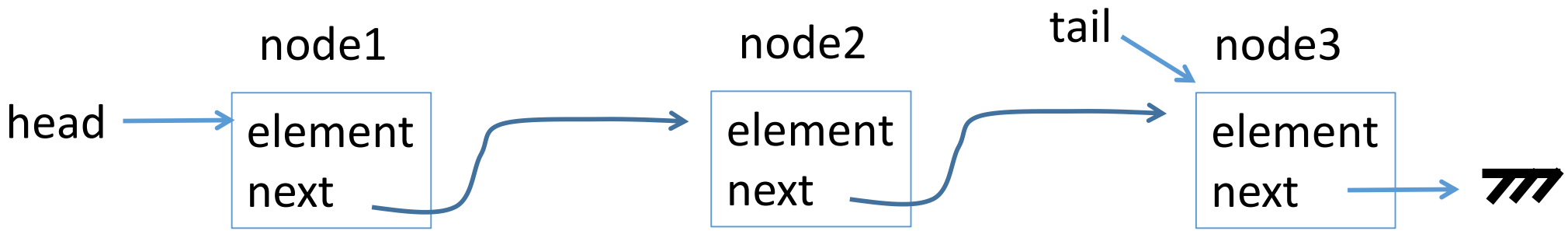
```
…
Node *node3 = new Node; …
addNode(node3);
```

```
void addNode( Node *node ) {
  if (head == 0) {
     head = tail = node;
     tail->next = 0;
     return;
  }
  tail->next = node;
  tail = node;
  tail->next = 0;
}
```



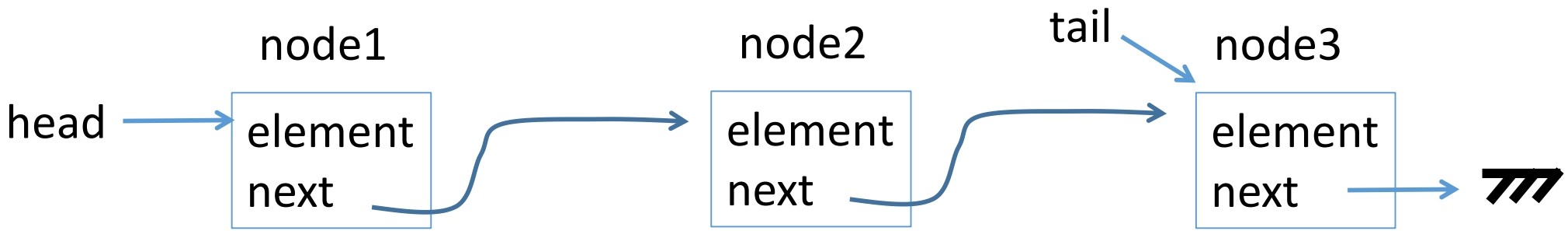
Traversing Elements in the Linked List

```
Node* cur = head; // current
while (cur != NULL)
{
  cout << cur->element << endl;
  cur = cur->next;
}
```



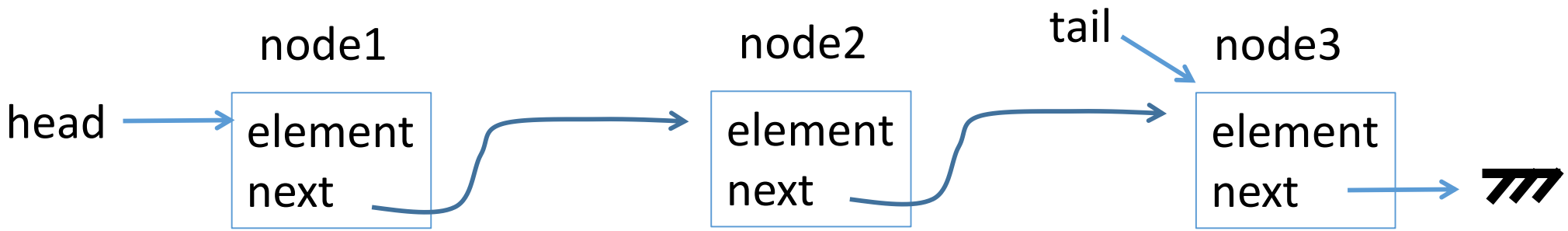
Traversing Elements in the Linked List

```
Node* cur = head; // current
while (current)
{
  cout << cur->element << endl;
  cur = cur->next;
}
```



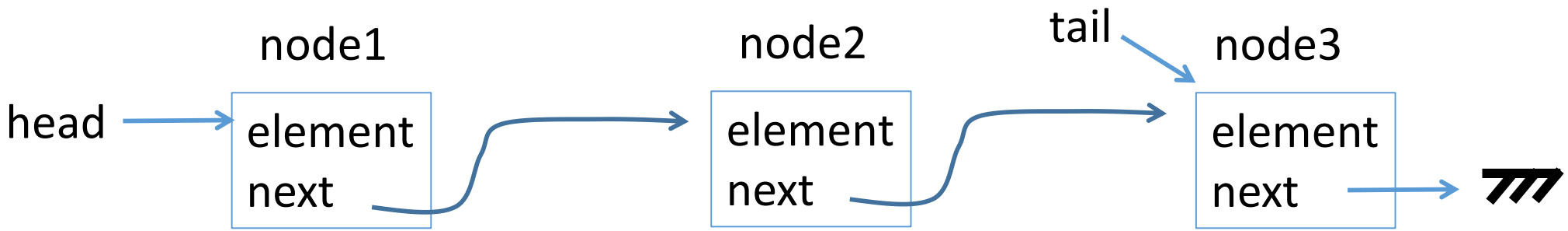
Traversing Elements in the Linked List

```
for (Node* cur = head; cur != NULL; cur = cur->next)
{
  cout << cur->element << endl;
}
```



Traversing Elements in the Linked List

```
for (Node* cur = head; cur; cur = cur->next)
{
  cout << cur->element << endl;
}
```

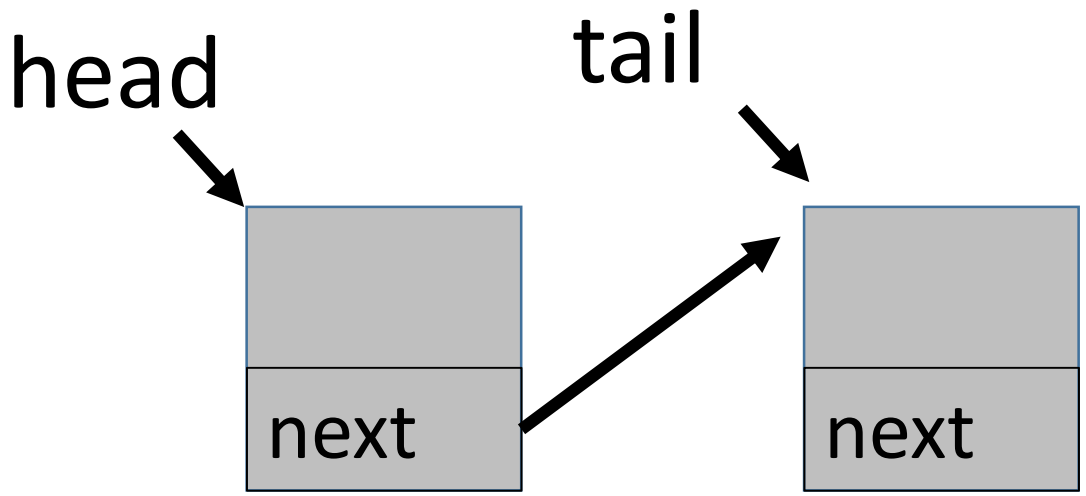


LinkedList

```
template<typename T>
class LinkedList {
public:
  LinkedList( );

  void addNode( Node<T> *node );
  Node<T> *getLast( ) const;
  Node<T> *getFirst( ) const;
  bool removeFirst( ) const;
  bool removeLast( ) const;
  bool empty( ) const;
};
```

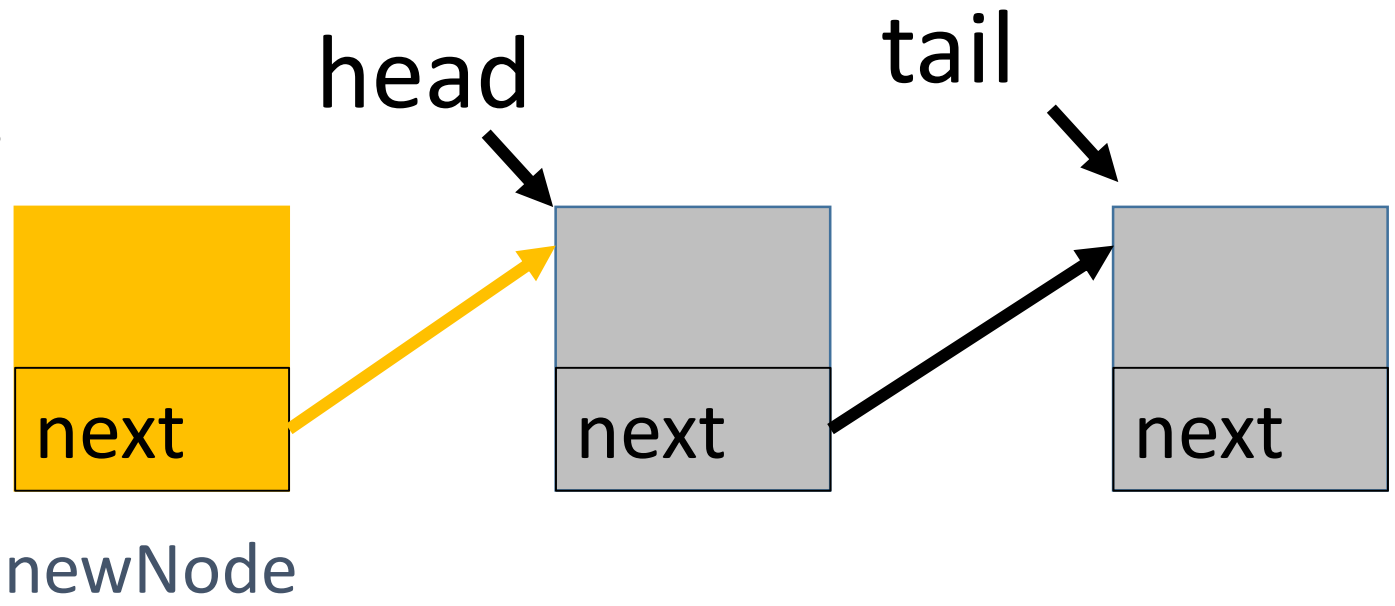
addFirst(T element).
Add a new object in front of the first object.

```
template<typename T>
void LinkedList<T>::addFirst(T element)
{
  Node<T>* newNode = new Node<T>(element);
  newNode->next = head;
  head = newNode;
  size++;
  if (!tail) tail = head;
}
```



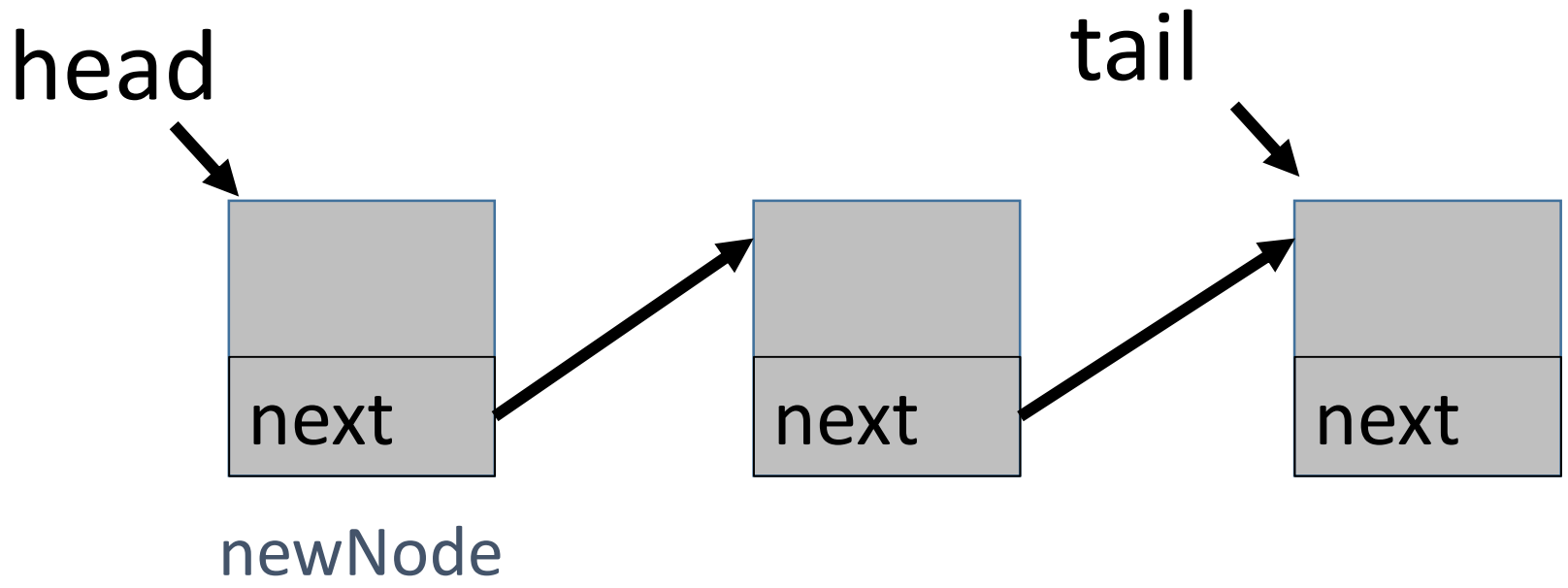
addFirst(T element).
Add a new object in front of the first object.

```
template<typename T>
void LinkedList<T>::addFirst(T element)
{
  Node<T>* newNode = new Node<T>(element);
  newNode->next = head;
  head = newNode;
  size++;
  if (!tail) tail = head;
}
```



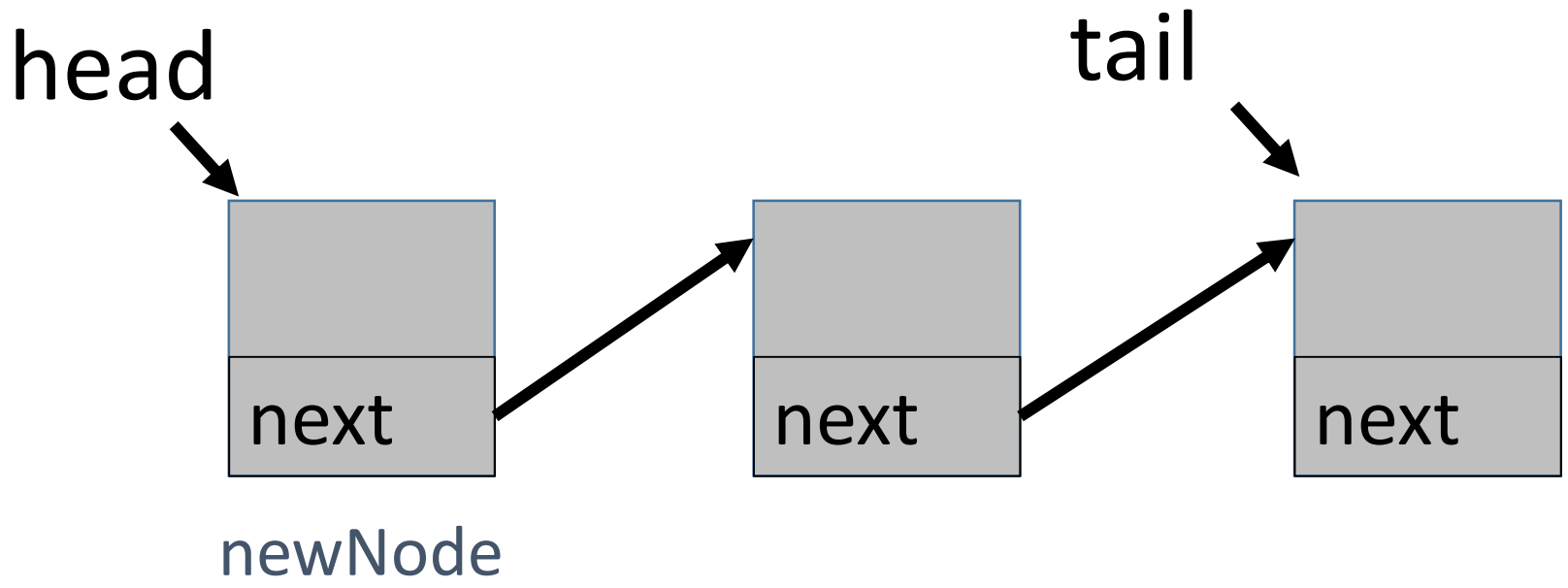
addFirst(T element).
Add a new object in front of the first object.

```
template<typename T>
void LinkedList<T>::addFirst(T element)
{
  Node<T>* newNode = new Node<T>(element);
  newNode->next = head;
  head = newNode;
  size++;
  if (!tail)
    tail = head;
}
```



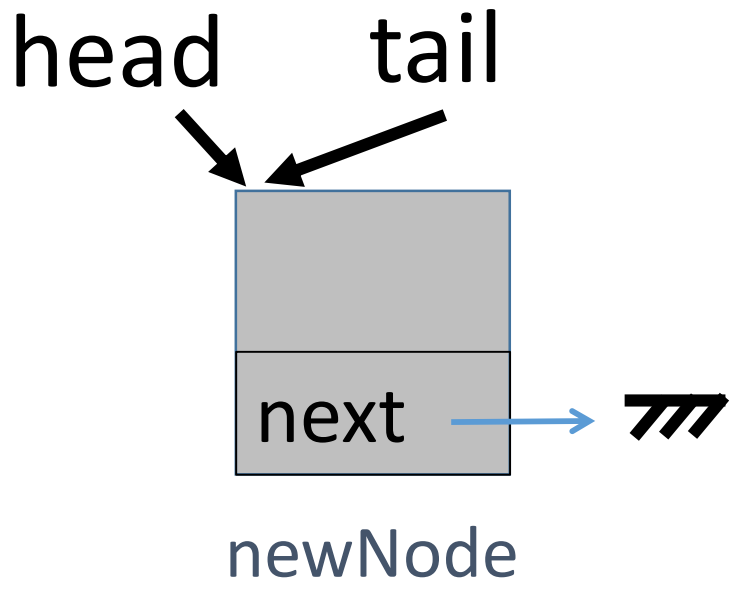
addFirst(T element).
Add a new object in front of the first object.

```
template<typename T>
void LinkedList<T>::addFirst(T element)
{
  Node<T>* newNode = new Node<T>(element);
  newNode->next = head;
  head = newNode;
  size++;
  if (!tail)
    tail = head;
}
```



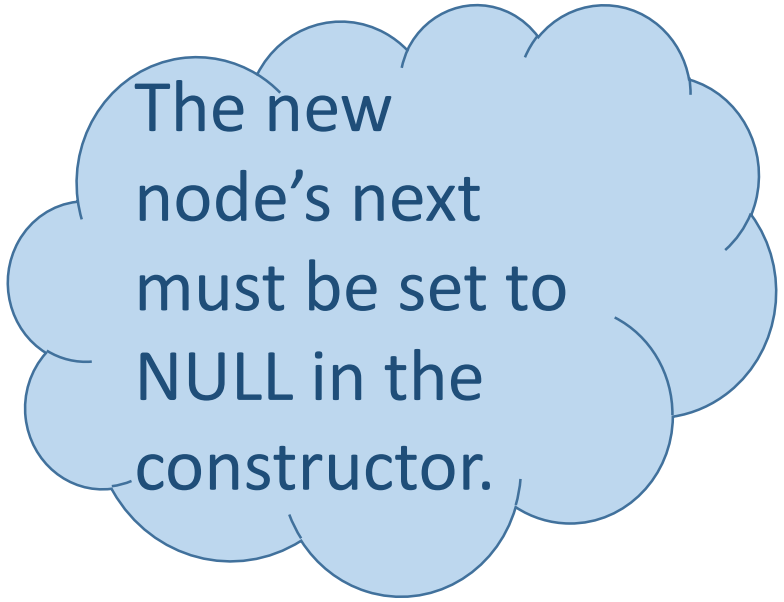
addFirst(T element).
Add a new object in front of the first object.

```
template<typename T>
void LinkedList<T>::addFirst(T element)
{
  Node<T>* newNode = new Node<T>(element);
  newNode->next = head;
  head = newNode;
  size++;
  if (!tail)
    tail = head;
}

// head and tail
// are NULL
```



Implementing addLast(T element).
Add a new object behind the last object.

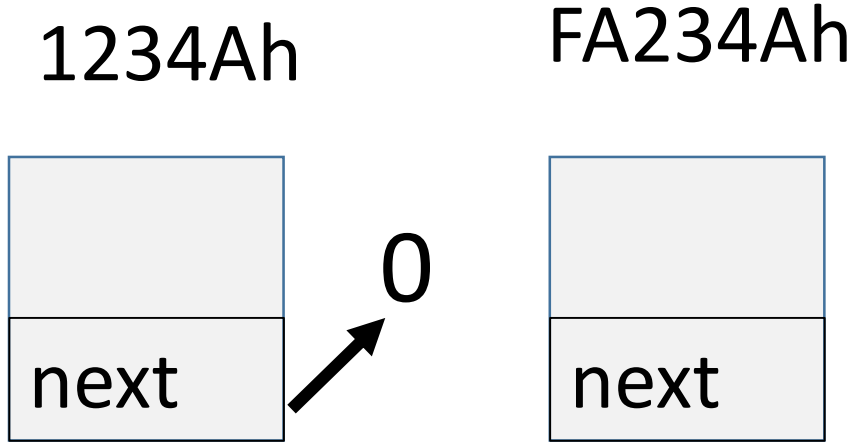
```
template<typename T>
void LinkedList<T>::addLast(T element)
{
  if (tail == NULL)
  {
    head = tail = new Node<T>(element);
  }
  else {
    tail->next = new Node<T>(element);
    tail = tail->next;
  }
  size++;
}
```



Visualizing the memory content
addLast(T *node)

&node := FA234Ah

```
//Assume the list is non-empty.
tail->next = node;
tail = node;
tail->next = 0;
```



tail = 1234Ah

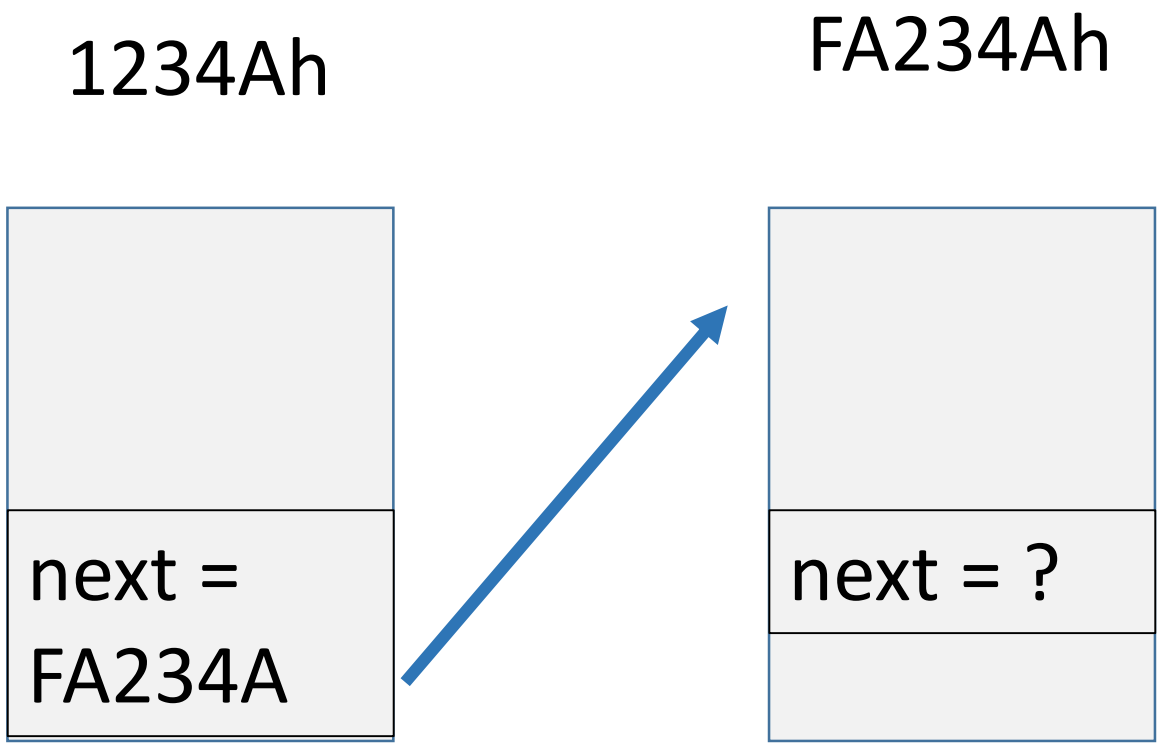
addLast(T *node)

node = FA234Ah

tail->next = node;

tail = node;

tail->next = 0;



tail = 1234Ah

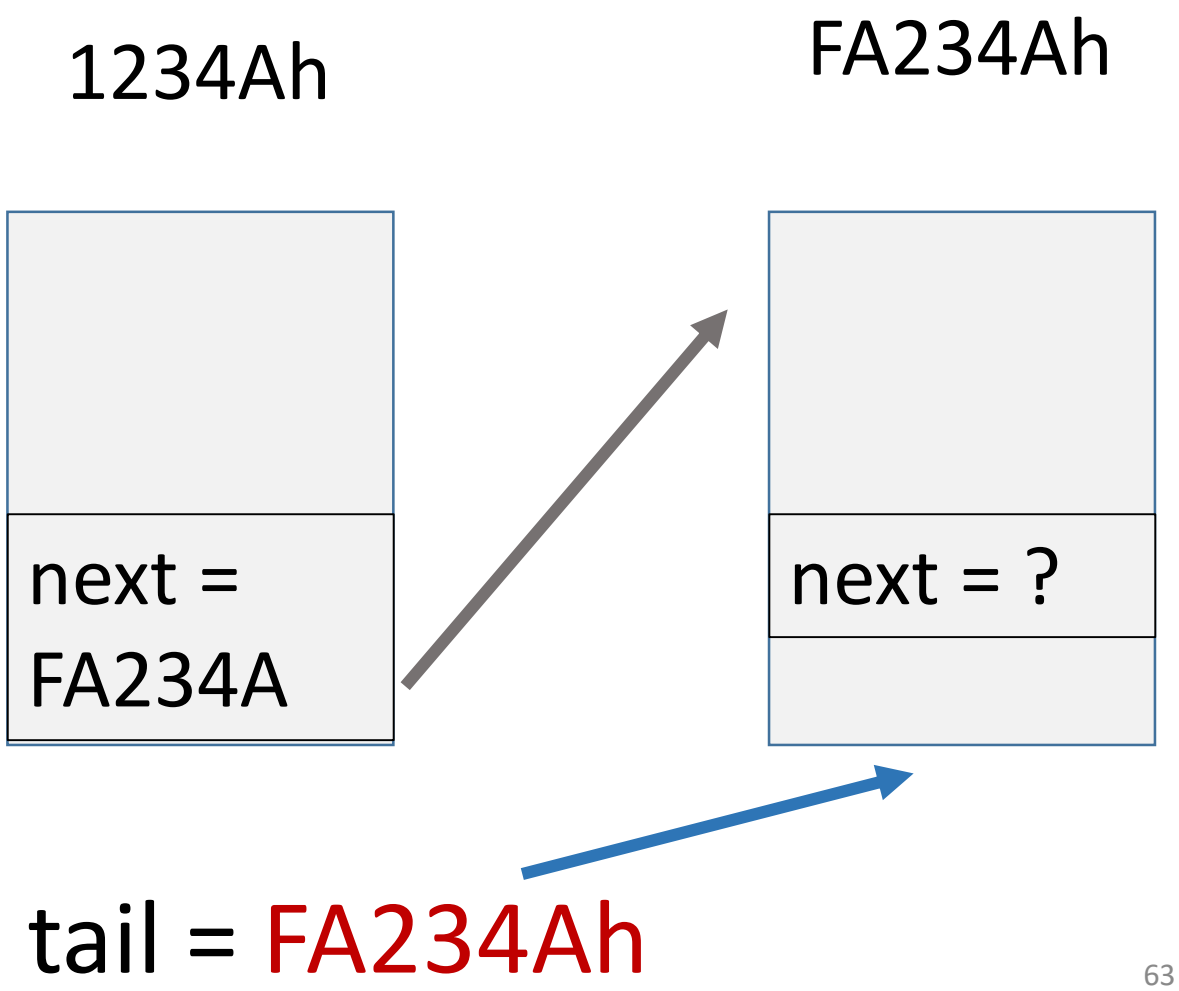
addLast(T *node)

node = FA234Ah

tail->next = node;

tail = node;

tail->next = 0;



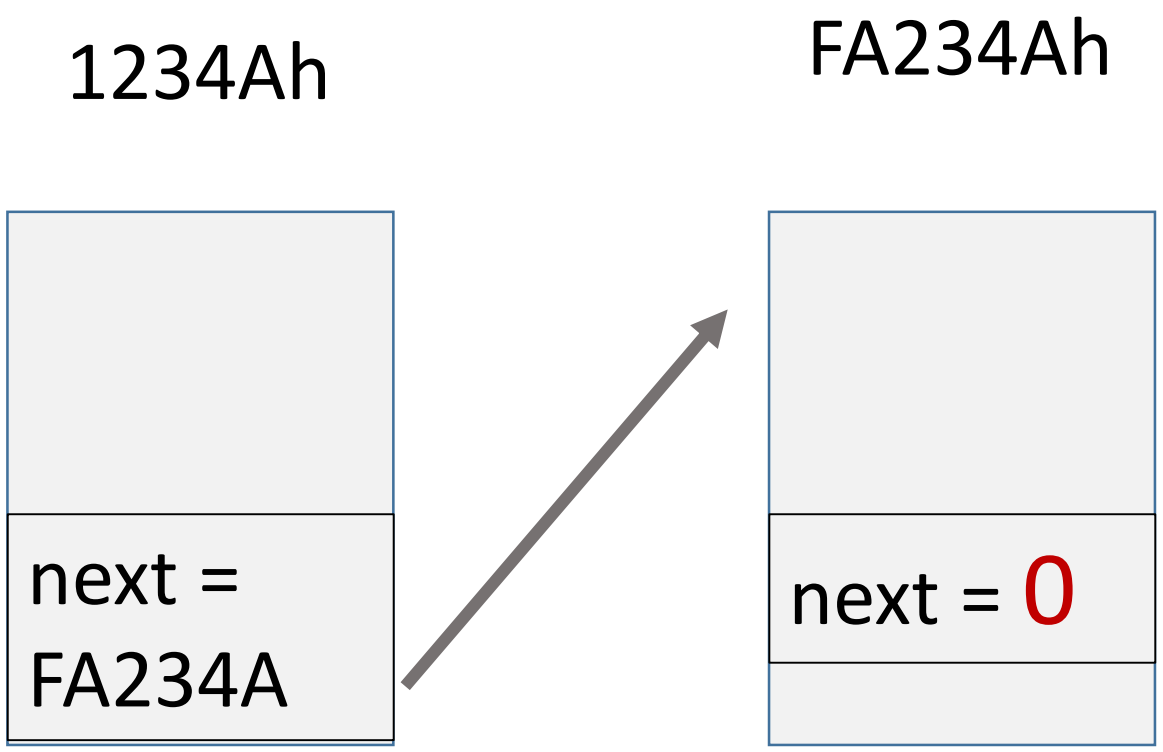
addLast(T *node)

node = FA234Ah

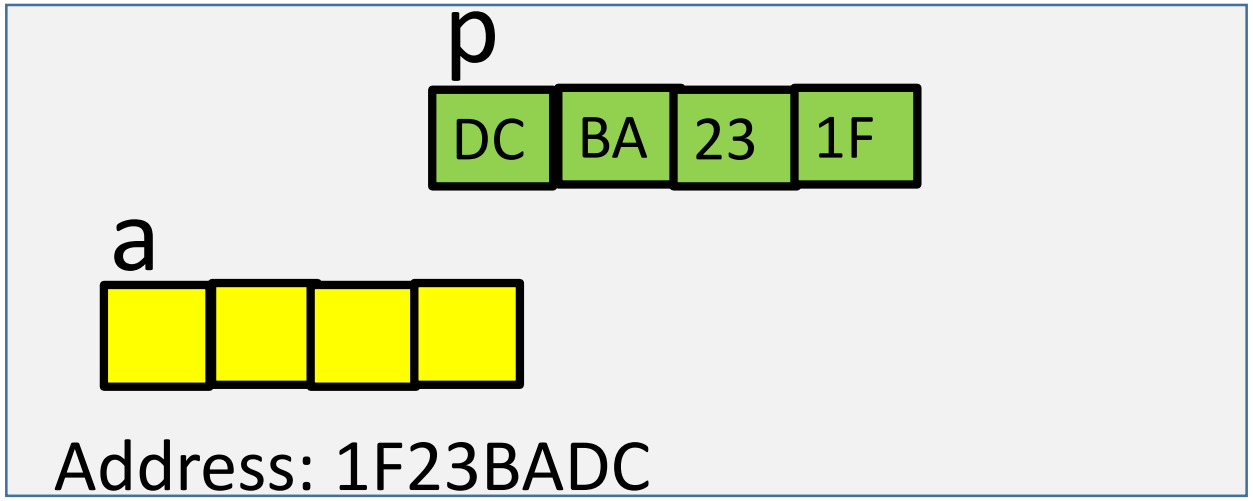
tail->next = node;

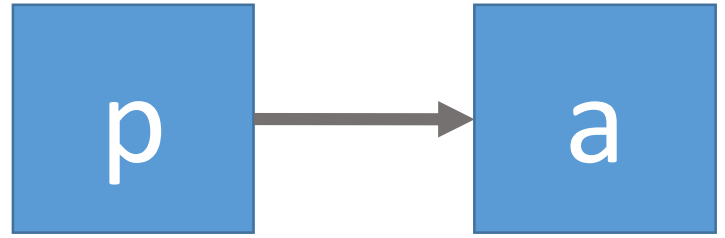
tail = node;

tail->next = 0;



tail = FA234Ah

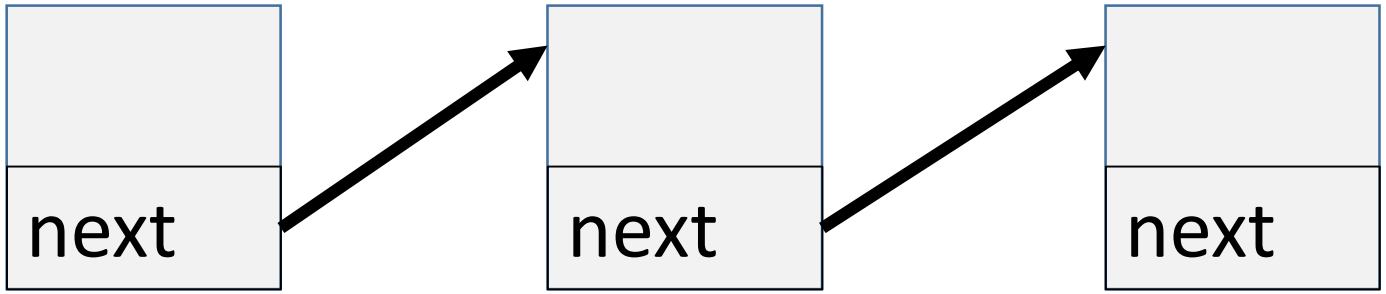

p
DC
BA
23
1F
a
Address: 1F23BADC


p
a

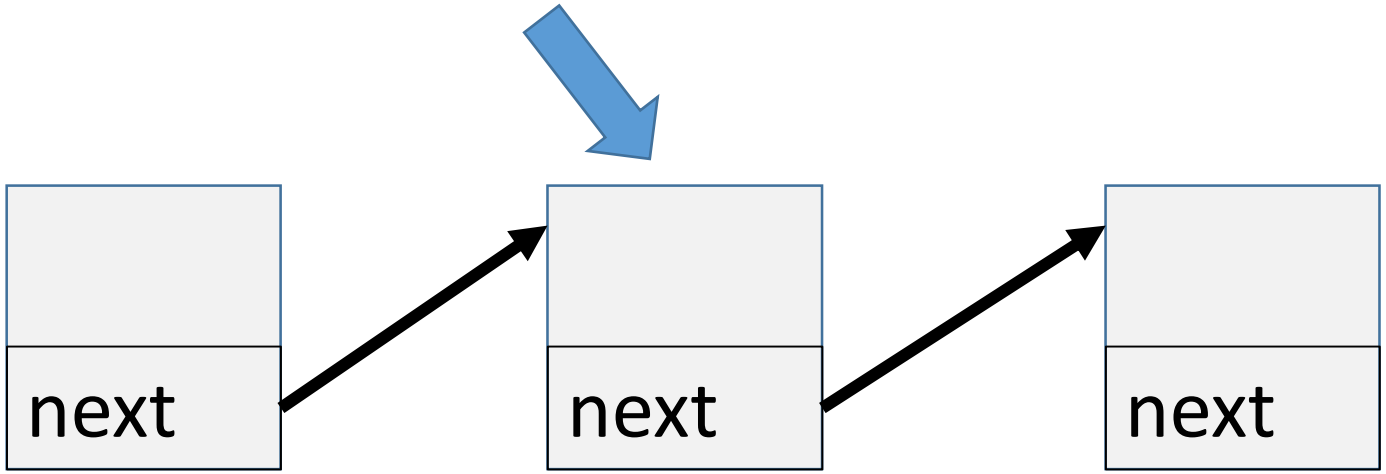
Implementing removeFirst(). How do you remove the first object? Do this exercise on your own.

```
template<typename T>
T LinkedList<T>::removeFirst() {
  if (size == 0) return NULL;
  else  {
    Node<T>* temp = head;
    head = head->next;
    size--;
    if (!head) tail = NULL;
    T element = temp->element;
    delete temp;    // do we need to delete it or not?
    return element;
  }
}
```

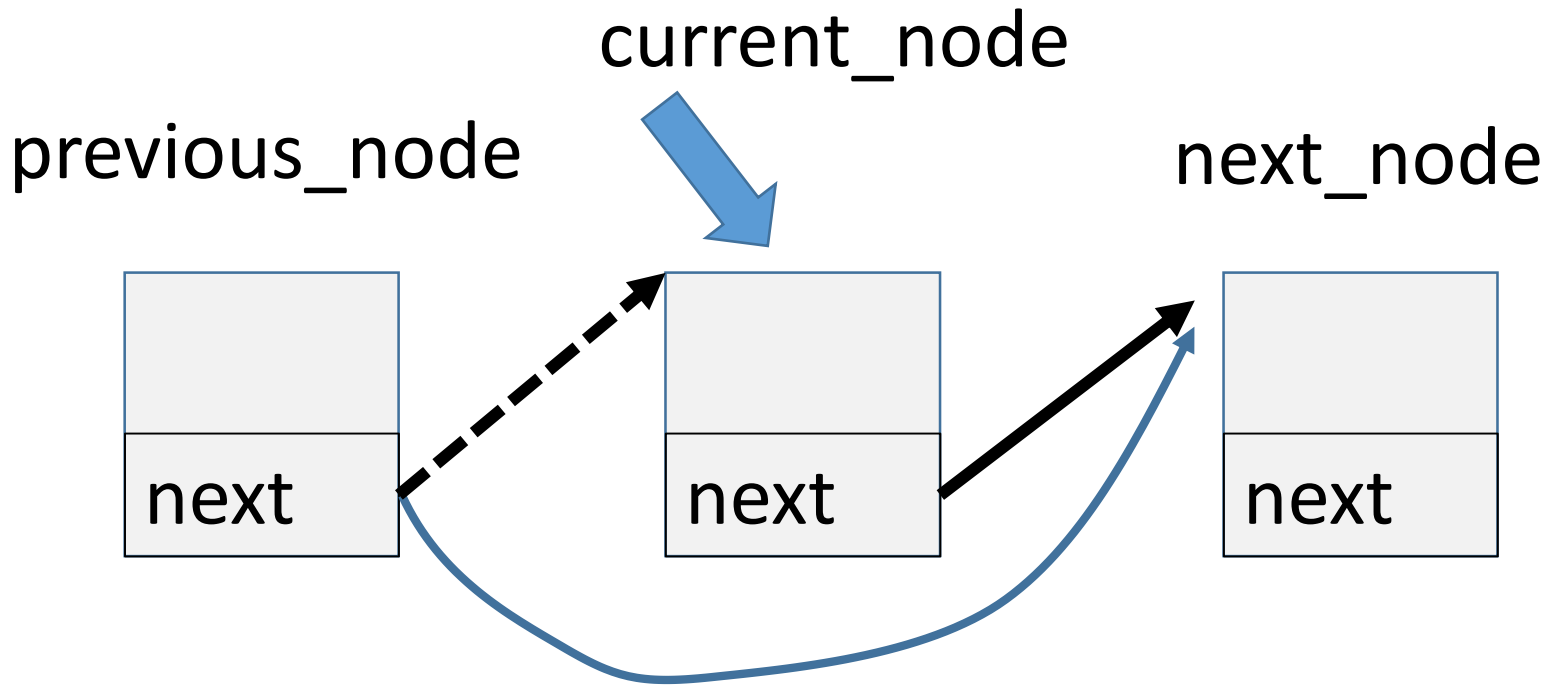
Node removal



Node removal

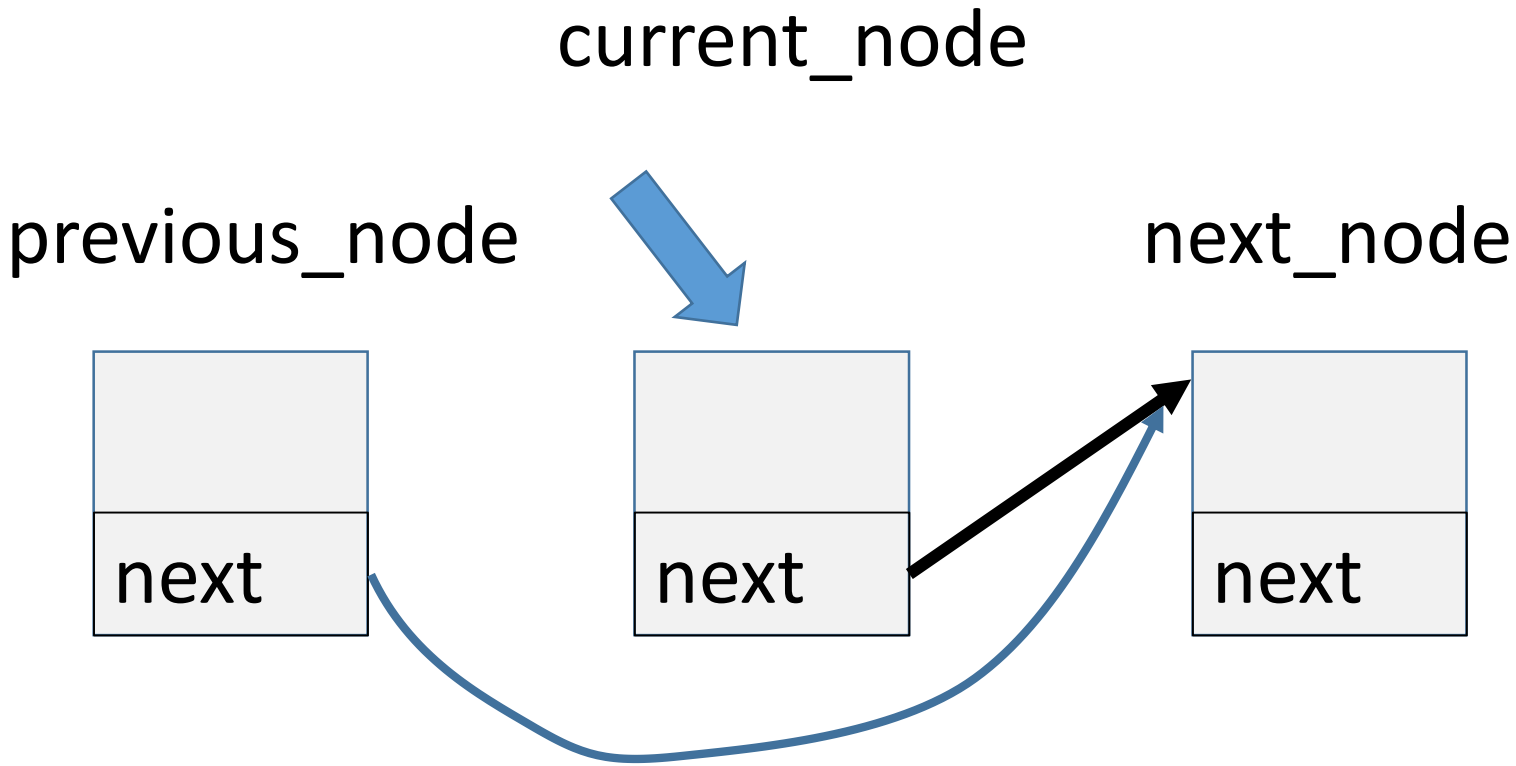


Remove a node?



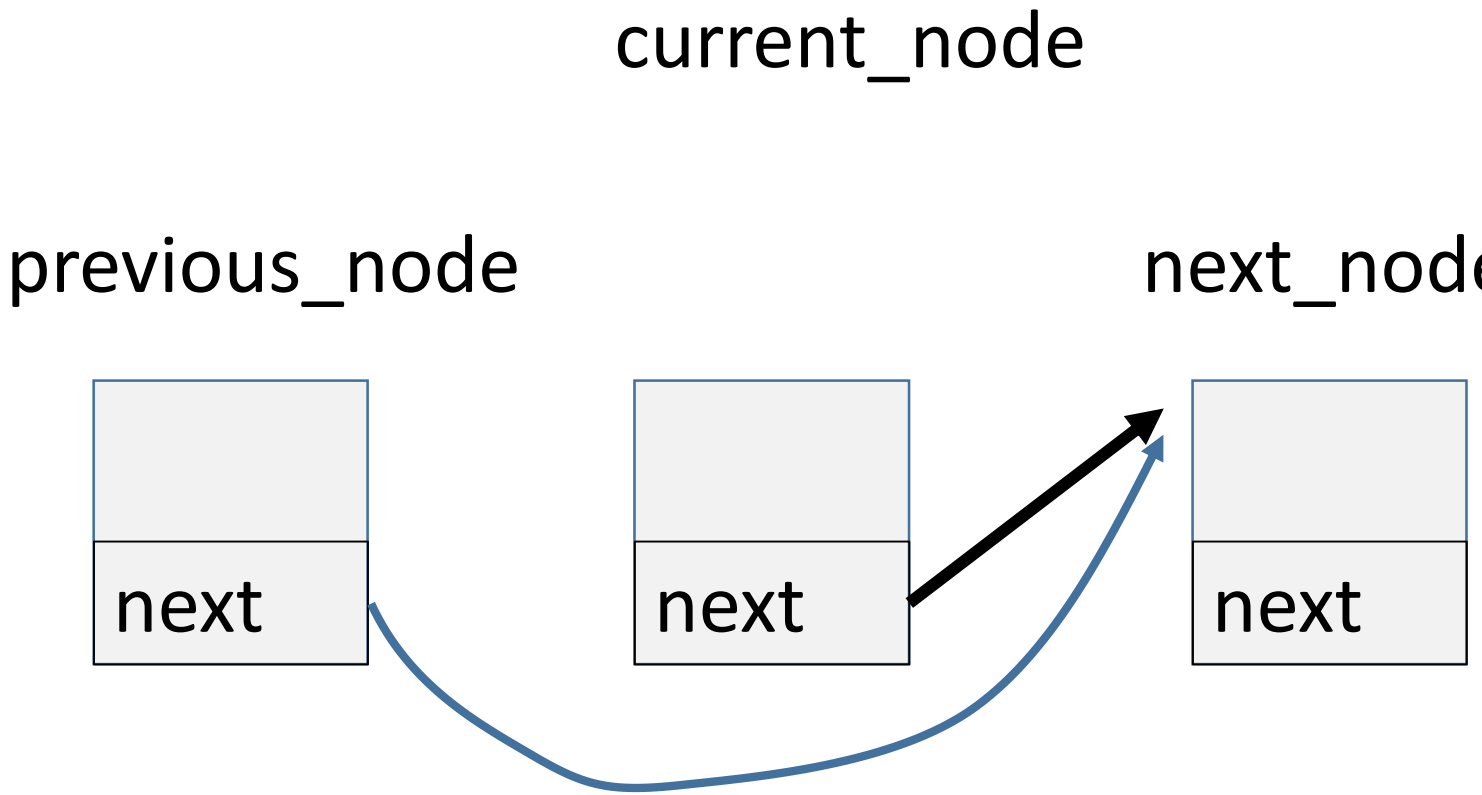
previous_node->next = current_node->next

Remove a node?



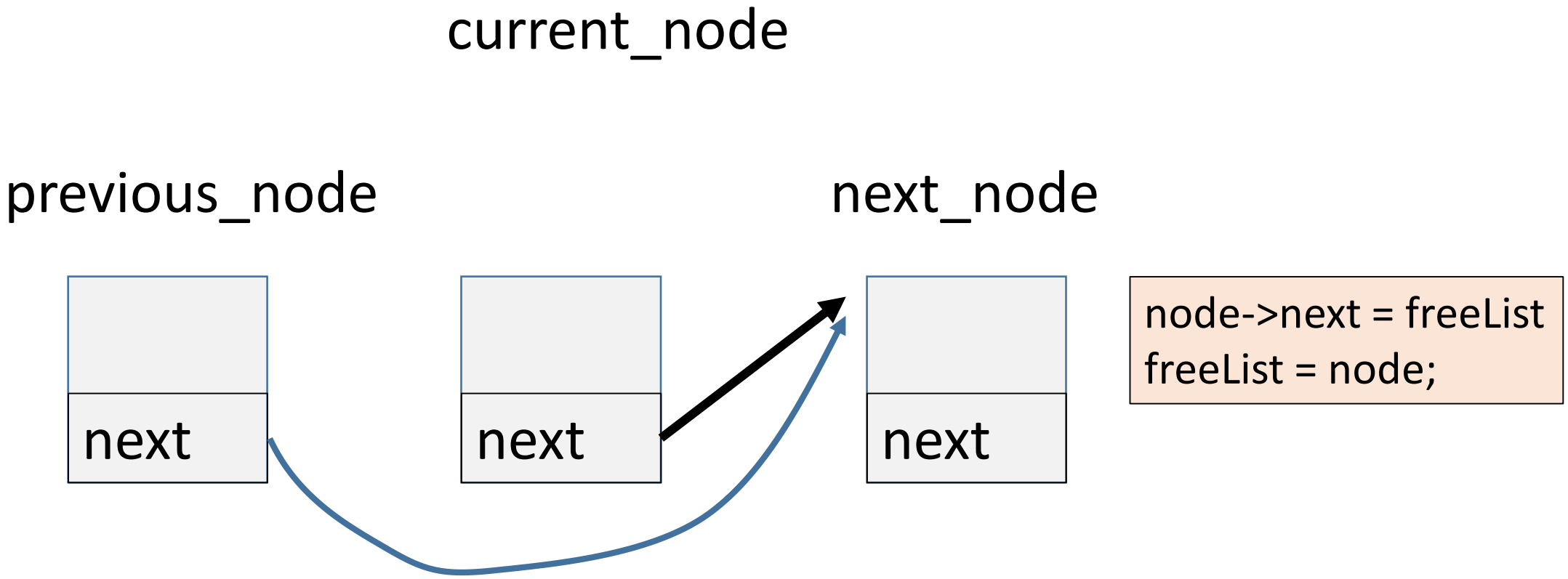
previous_node->next = current_node->next

Remove a node?



recycle the node: delete it or put it to a free list.

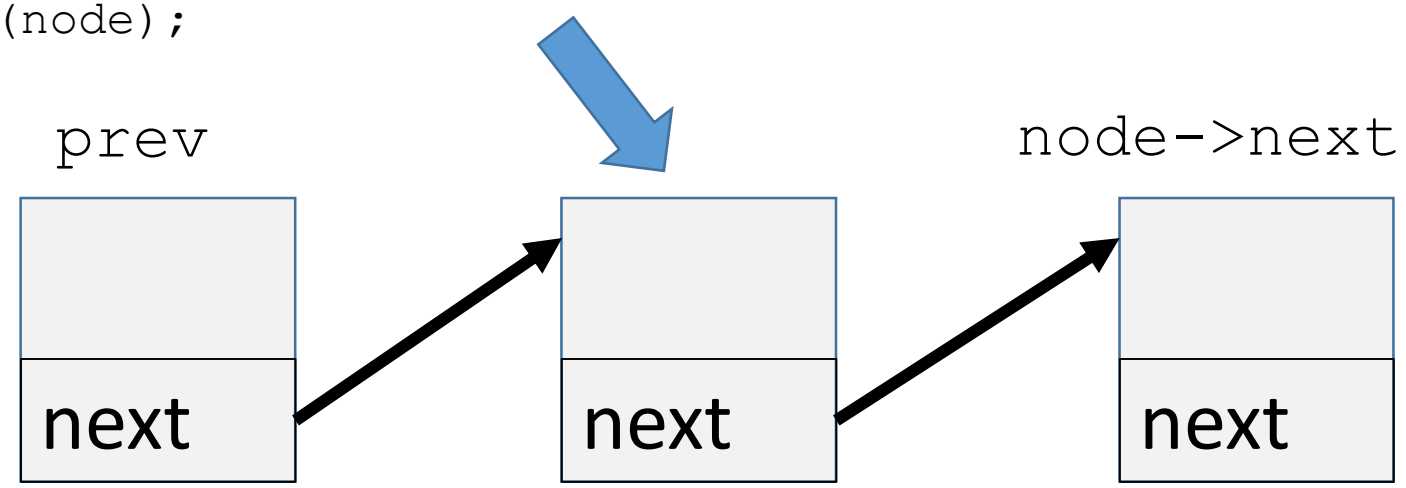
Remove a node?



recycle the node: delete it or put it to a free list.

Node removal

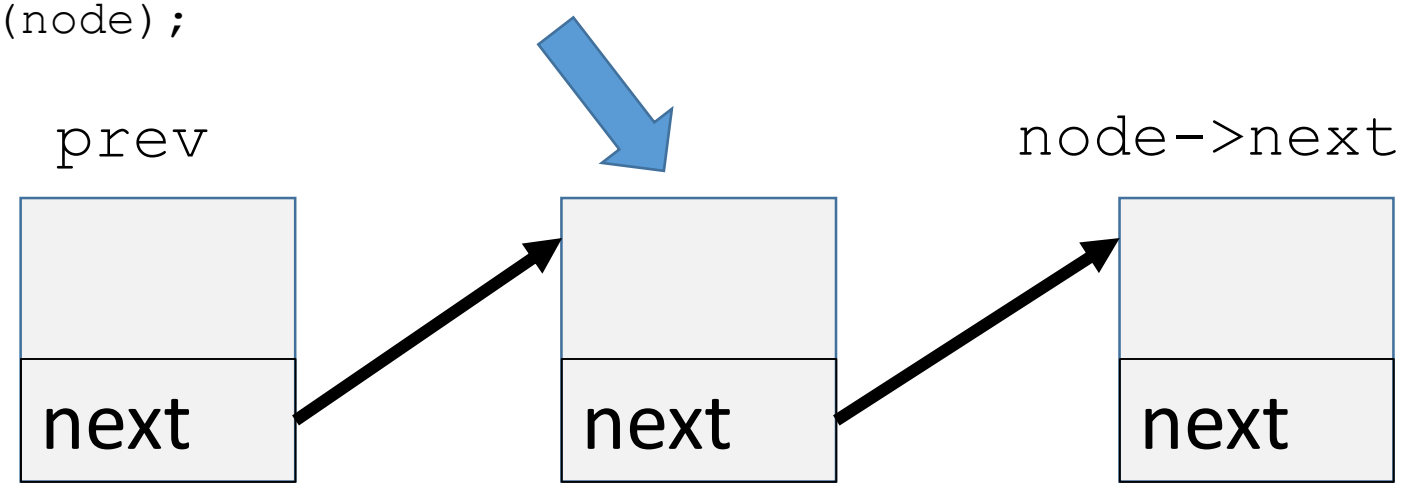
```
void removeNode( Node<T> *node) {
  if (node==head) {
     removeFirstNode( );
     return;
  }
  if (node==tail) {
     removeLastNode( );
     return;
  }
  Node<T> *prev = findPreviousNode(node);
  prev->next = node->next;
}
```



Node removal

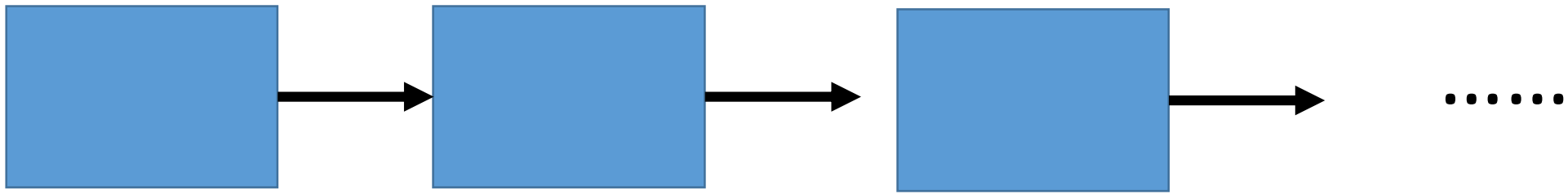
```
void removeNode( Node<T> *node) {
  if (node==head) {
     removeFirstNode( );
     return;
  }
  if (node==tail) {
     removeLastNode( );
     return;
  }
  Node<T> *prev = findPreviousNode(node);
  prev->next = node->next;
}
```

Implement findPreviousNode



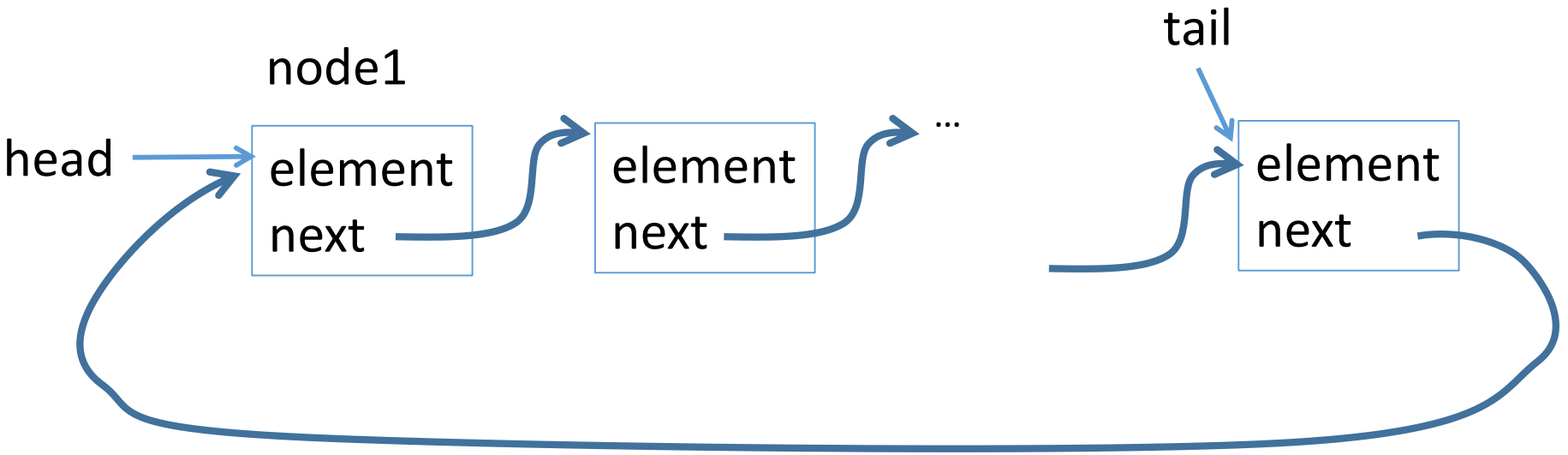
Variations of Linked Lists

A singly linked list has one pointer, i.e., next, which points to the next node.



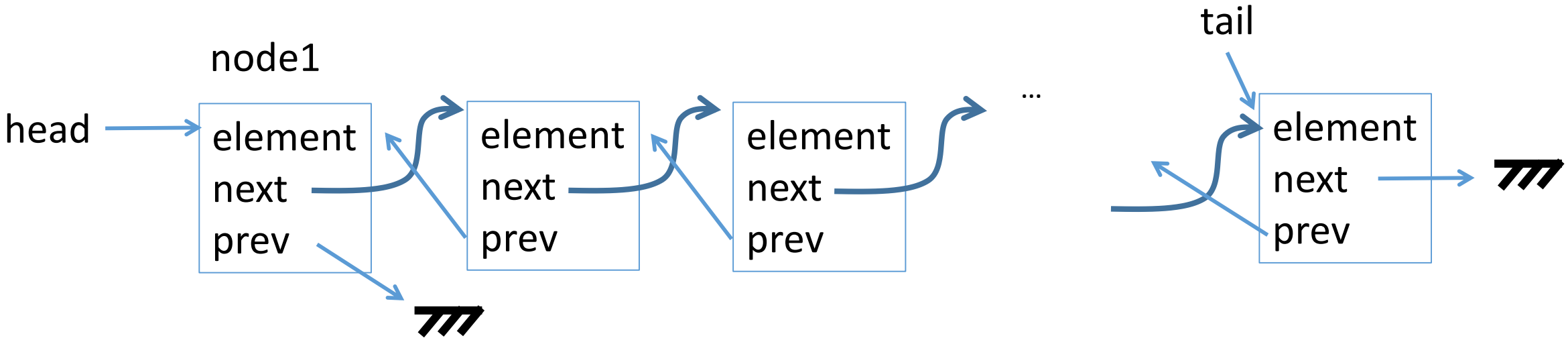
Circular Linked Lists

- A circular, singly linked list
- The last node points back to the first node.



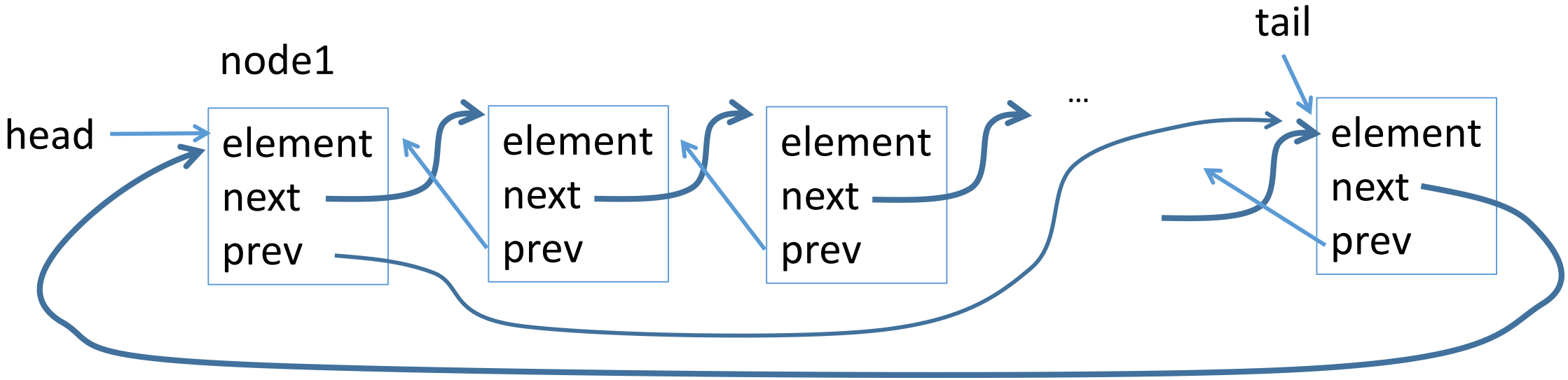
Doubly Linked Lists

- A doubly linked list contains the nodes with two pointers.
- The forward pointer points to the next node.
- The backward pointer points to the previous node.



Circular, Doubly Linked Lists

- A linked list contains the nodes with two pointers.
- The forward pointer points to the next node.
- The backward pointer points to the previous node.
- The last node (forward pointer) points to the first node.
- The first node (backward pointer) points to the last node.

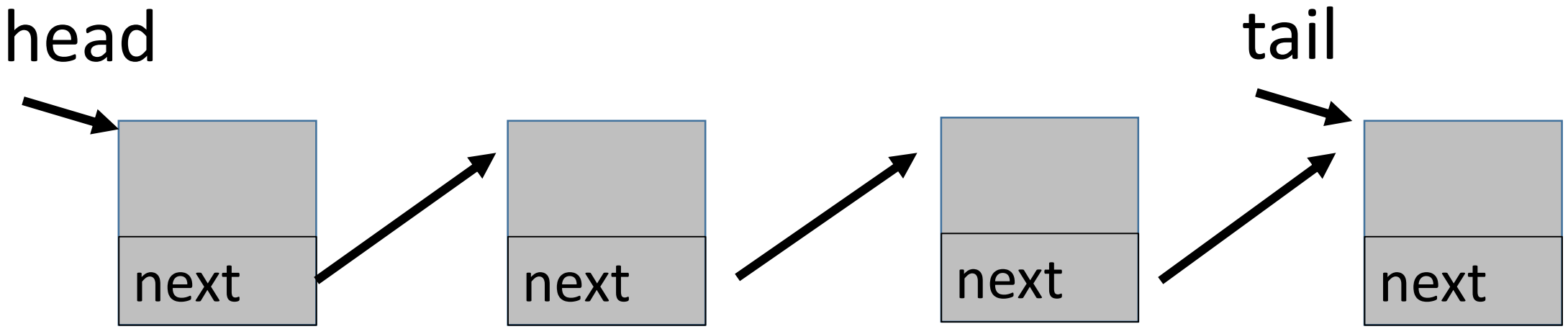


Stacks

Stack can be implemented using an array

or

a linked list.



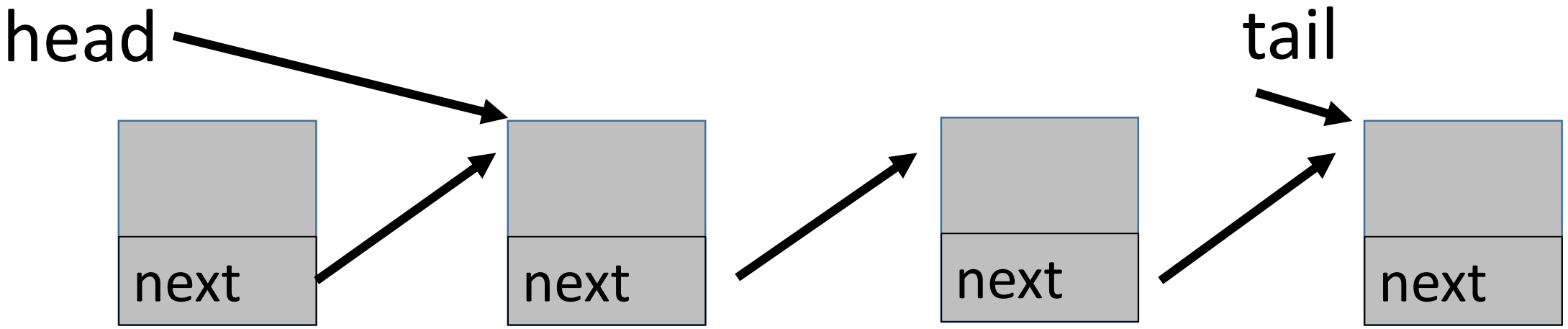
head = head->next

Stacks

Stack can be implemented using an array

or

a linked list.



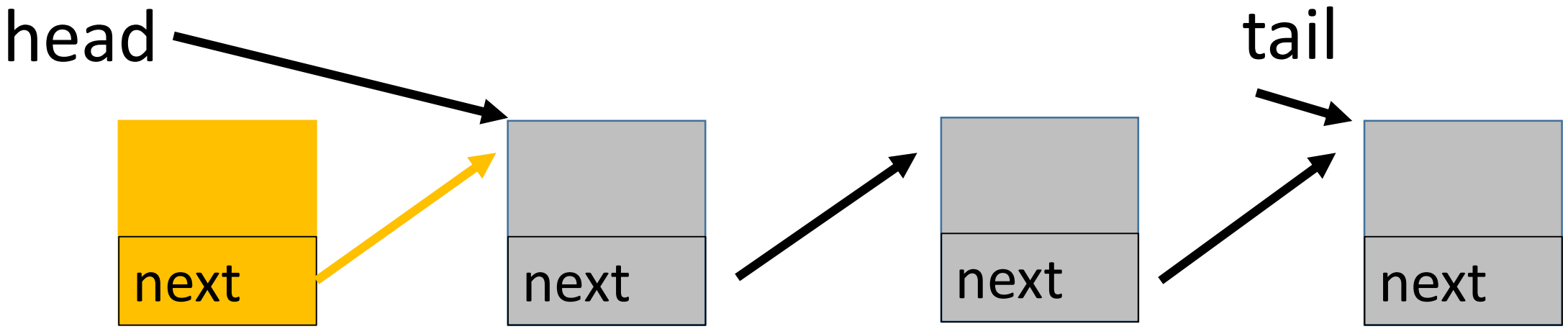
head = head->next

Stacks (Last-In-First-Out LIFO)

Stack can be implemented using an array

or

a linked list.



head = head->next

Queues (First-In-First-Out FIFO)

A queue: 1) remove the first node; 2) add a node as the last node.

We can adopt two ways to implement a queue.

- Using inheritance: You can declare the queue class by extending the linked list class.

- Using composition: You can declare a linked list as a data field in the queue class.

Queues (First-In-First-Out FIFO)

We can adopt two ways to implement a queue.

- Using inheritance: We can derive the queue class from the linked list class.

- Using composition: We can declare a linked list as a data field in the queue class.

```
template<typename E>
class Queue: public LinkedList<E> {
  ...
};
```

Queues (First-In-First-Out FIFO)

We can adopt two ways to implement a queue.

- Using inheritance: We can derive the queue class from the linked list class.

```
template<typename E>
class Queue: public LinkedList<E> {
  …
};
```

- Using composition: We can declare a linked list as a data field in the queue class.

```
template<typename E>
class Queue {
 …
 LinkedList L;
 Node *addLast( const E &e );
 Node *removeFirst( );
};
```

Queues (First-In-First-Out FIFO)

We can adopt two ways to implement a queue.

- Using inheritance: We can derive the queue class from the linked list class.

```
template<typename E>
class Queue: public LinkedList<E> {
  …
};
```

- Using composition: We can declare a linked list as a data field in the queue class.

```
template<typename E>
class Queue {
 …
 LinkedList L;
 Node *enqueue( const E &e ); //last
 Node *dequeue( ); // remove first
 int getSize( ) const;
};
```

Supplemental Materials

Iterators

We can use iterators to perform **a uniform way for traversing elements** in various types of containers.

Traverse elements using the for-loop

```
vector<int>::iterator p1;
 cout << "Traverse the vector: ";
 for (p1 = intVector.begin(); p1 != intVector.end(); p1++)
 {

   cout << *p1 << " ";

 }
```

Traverse elements using the for-loop

```
vector<int>::iterator p1;
 cout << "Traverse the vector: ";
 for (p1 = intVector.begin(); p1 != intVector.end(); p1++)
 {
```

*p1

```
  cout <<                                  << " ";
 }
```

Traverse elements using the for-loop

```
vector<int>::iterator p1;
 cout << "Traverse the vector: ";
 for (p1 = intVector.begin(); p1 != intVector.end(); p1++)
 {

   cout <<  *p1  << " ";

 }
```

Traverse elements using the for-loop

Datatype::iterator p1;

```
cout << "Traverse the vector: ";
for (p1 = intVector.begin(); p1 != intVector.end(); p1++)
{
  cout << *p1 << " ";
}
```

Datatype::iterator p1;

```
cout << "Traverse the vector: ";
for (p1 = intVector.begin(); p1 != intVector.end(); p1++)
{
  cout << *p1 << " ";
}
```

```
vector<int> a(10, 1);
for ( int i = 0; i < a.size( ); ++i ) {
    cout << "i:" << "\t" << a[i] << endl;
}
```

```
vector<int>::iterator it;
for ( it = a.begin( ); it != a.end( ); ++it) {
    cout << "i:" << "\t" << (*it) << endl;
}
```

The Iterator Class

Iterators can be viewed as encapsulated pointers. In a linked list, you can use pointers to traverse the list.

But iterators have more functions than pointers. Iterators are objects.

Iterators contain functions for accessing and manipulating elements.

The Iterator Class for LinkedList

Operation	Description
operator++( ): Iterator	Get the iterator for the next pointer
operator*( ): T &	Return the element from the node pointed by the iterator
operator==(itr:interator<T>&: bool	
operator!=(itr:interator<T>&: bool	

Advantages of Iterators

- An iterator function likes a pointer.
- We may use pointers, array indexes, or other data structures to implement iterator functions.

Priority Queues

- **Elements are assigned with priorities**.
- The element **with the highest priority** is **accessed or removed first**.
- A priority queue has a largest-in, first-out behavior.

Use vector to implement priority queue

```
class PQ {
protected:
vector<int> q;
public:
    int getElement( ) {
        int key = -1;

        return key;
    }
};
```

Use vector to implement priority queue

```
vector<int> q;
public:
	// assume that a valid key is non-negative
	// return -1: no more element
	int getElement( ) {
		int key = -1;
		int key_index;
		for ( int i = 0; i < q.size( ); ++i ) {
			if ( key < 0 || q[i] > key ) {
				key = q[i];
				key_index = i;
			}
		}

		return key;
	}
```

Use a heap structure to implement a priority queue structure

- Exercise

Use a structure with a sorting ability to implement a priority queue structure

- Exercise