Contemporary C++: Learning Modern C++ in a Modern Way

الماس فناوری ابری پاسارگاد- آلفا

مدرس: سعید امراللهی بیوکی

Agenda 17/24

Session 17. Design and Implementation of Doubly-Linked List (Part II)

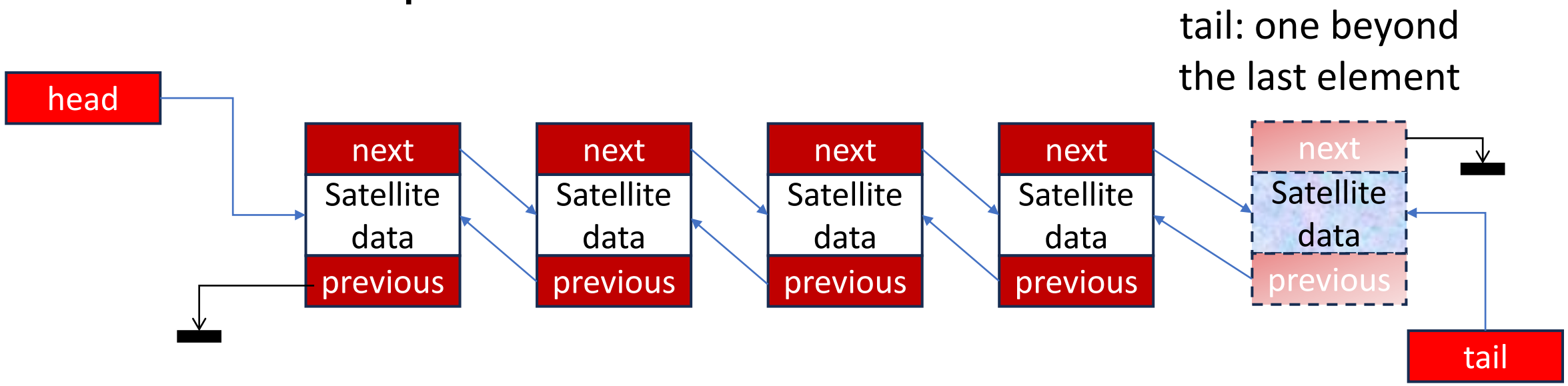
- Toward more standard linked list
- Reorganize the representation of the Linked List
- the tail of the linked list as one beyond the last node
- Re-implement the essential operations
- Toward standard list: push_back, pop_back, push_front and pop_front operations
- Re-implement the insert and remove operations
- Node wrapper: the implementation of bidirectional iterator
- Q&A

150 min (incl. Q & A)

Round 2

- → *Round 0*
- Improve list implementation
- Node as nested classes
- Re-implement append operation
- append as a private member function
- Re-implement essential operations: Special member functions + Ordinary constructor
- C++11: Delegating constructor
- More standard member function names: push_back, pop_back, push_front, pop_front
- More standard member function names: begin, end, and empty
- Remove with return type int
- Instantiate LinkedList with double, std::string, std::vector<int> and Point template parameters
- Using noexcept specifier

Linked list: new Representation



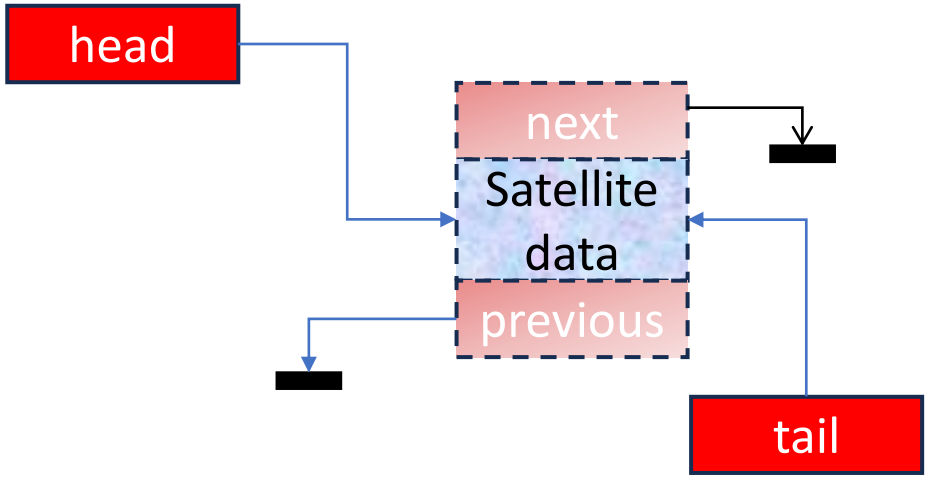
Node-based data structure

head: a pointer to the first node

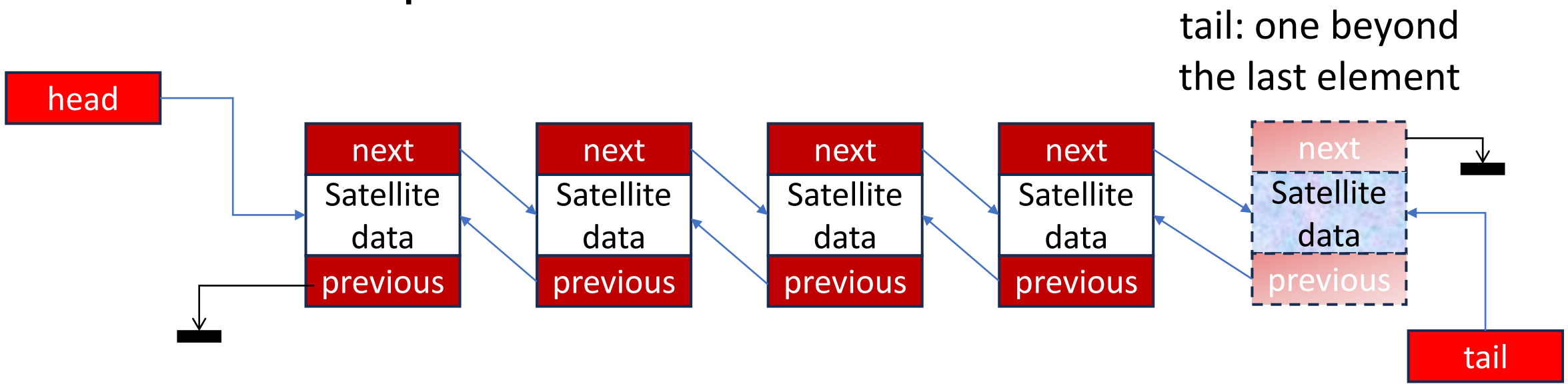
tail: a pointer to one beyond the last node

```
template<class T> class LinkedList {
    template<class T>
    struct Node {
        T data;
        Node* next_;
        Node* prev_;
    };
    Node<T>* head_; // pointer to first element
    Node<T>* tail_; // pointer to one beyond last element
};
```

special member functions: Default constructor



Linked list: new Representation



Node-based data structure

head: a pointer to the first node

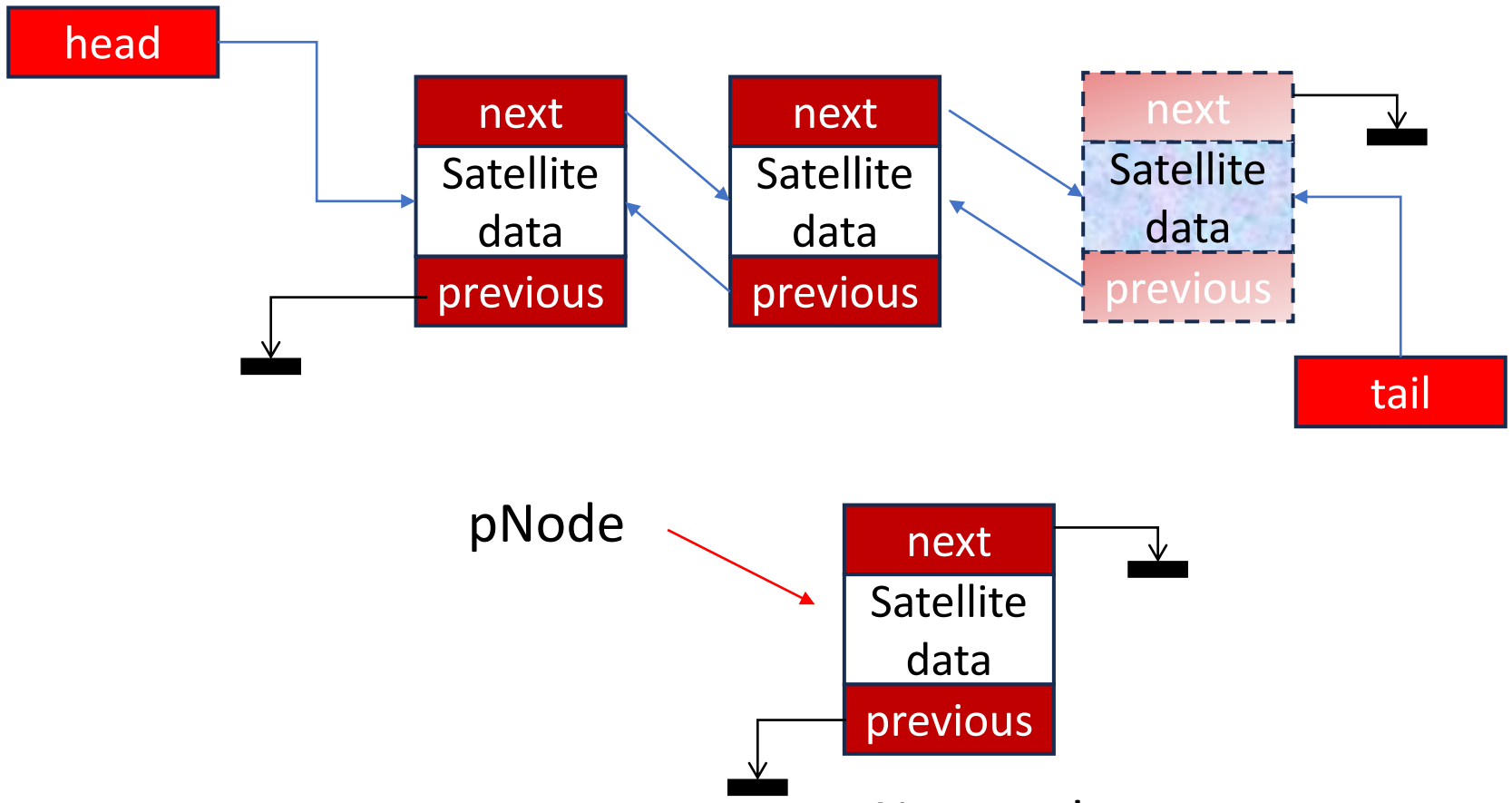
tail: a pointer to one beyond the last node

```
template<class T> class LinkedList {
    template<class T>
    struct Node {
        T data;
        Node* next_;
        Node* prev_;
    };
    Node<T>* head_; // pointer to first element
    Node<T>* tail_; // pointer to one beyond last element
};
```

Linked list initial operation: append

- Append

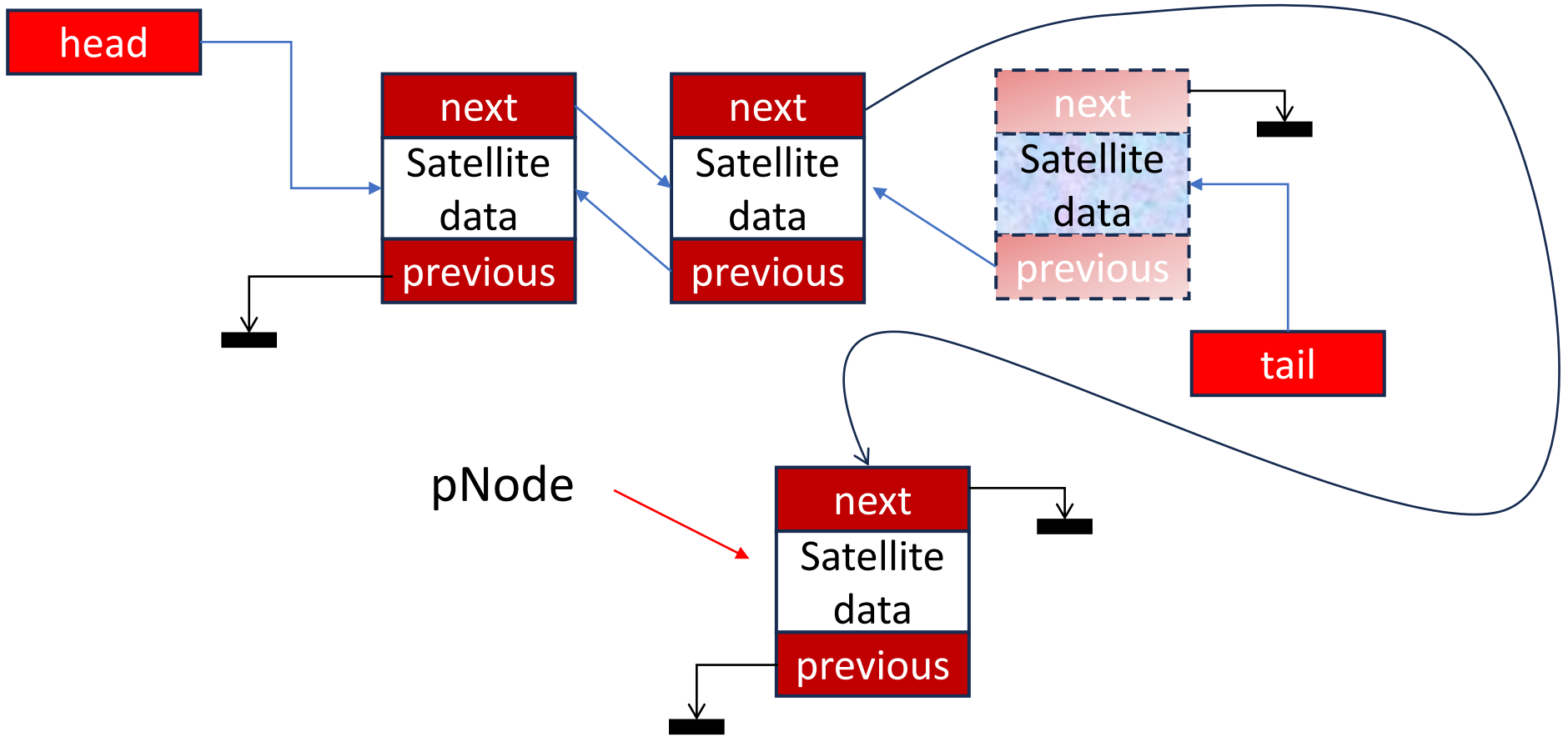
```
template<class T> class LinkedList {
public:
    void append(const T& t); // insert at the end of list
};
```



Linked list initial operation: append

- Append

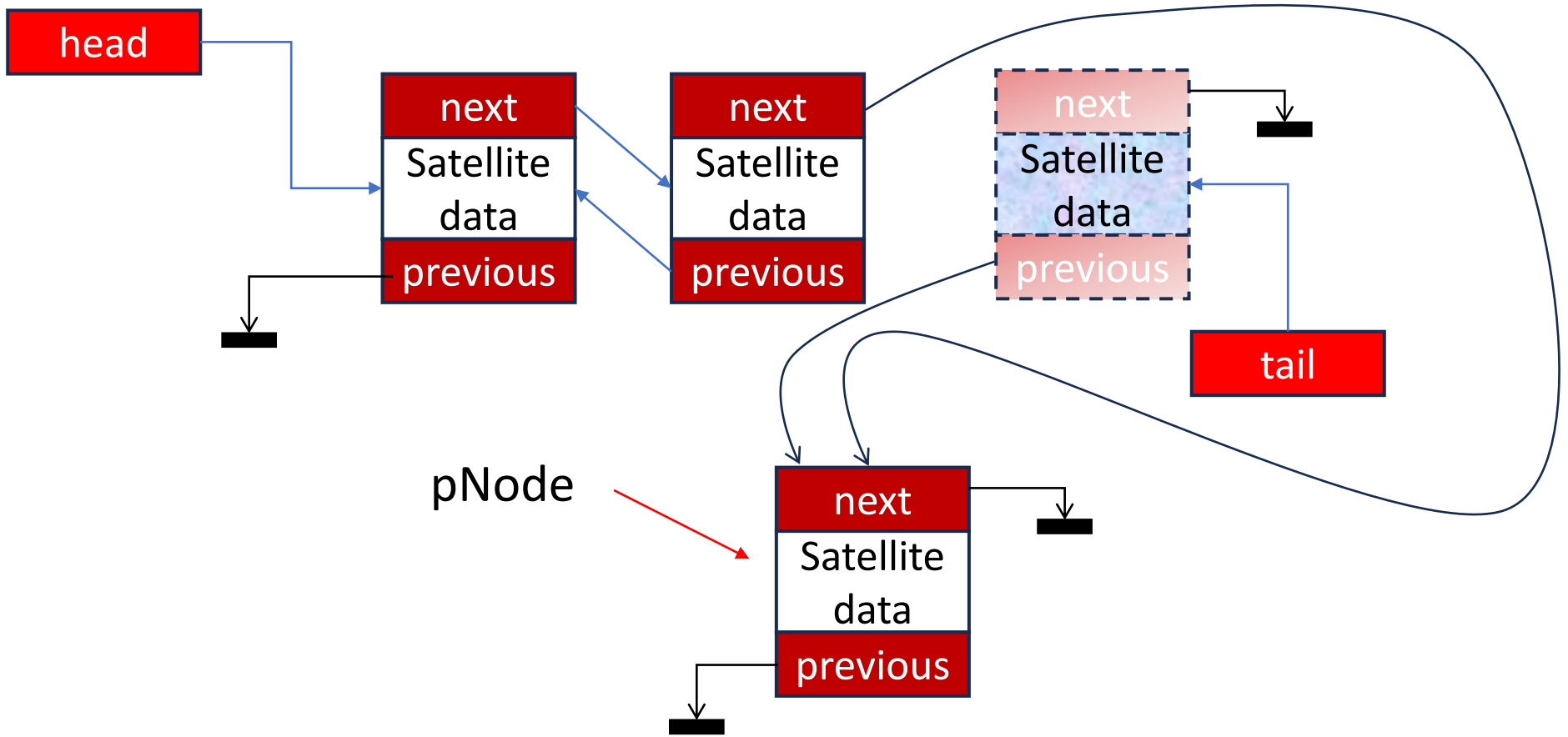
```
template<class T> class LinkedList {
public:
    void append(const T& t); // insert at the end of list
};
```



Linked list initial operation: append

- Append

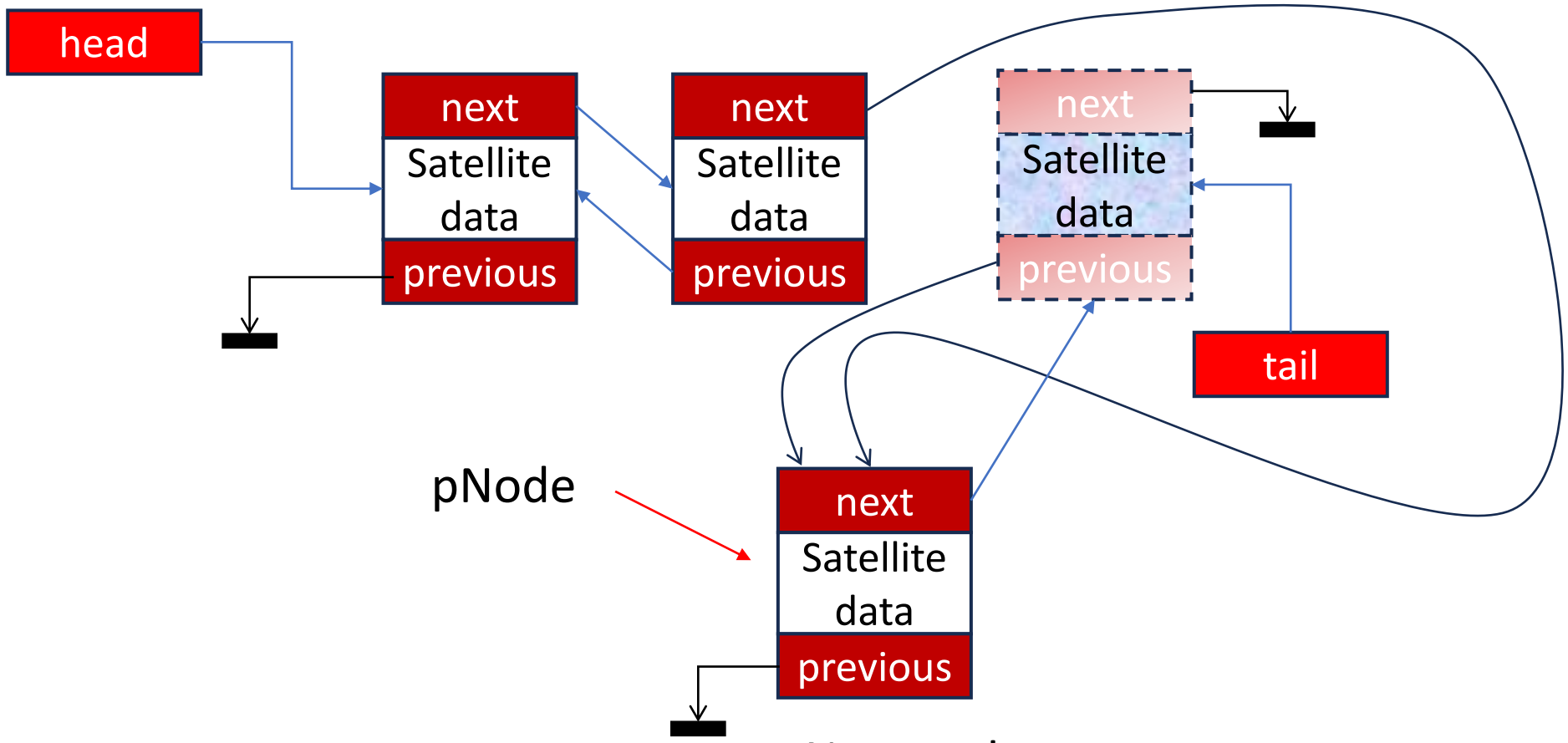
```
template<class T> class LinkedList {
public:
    void append(const T& t); // insert at the end of list
};
```



Linked list initial operation: append

- Append

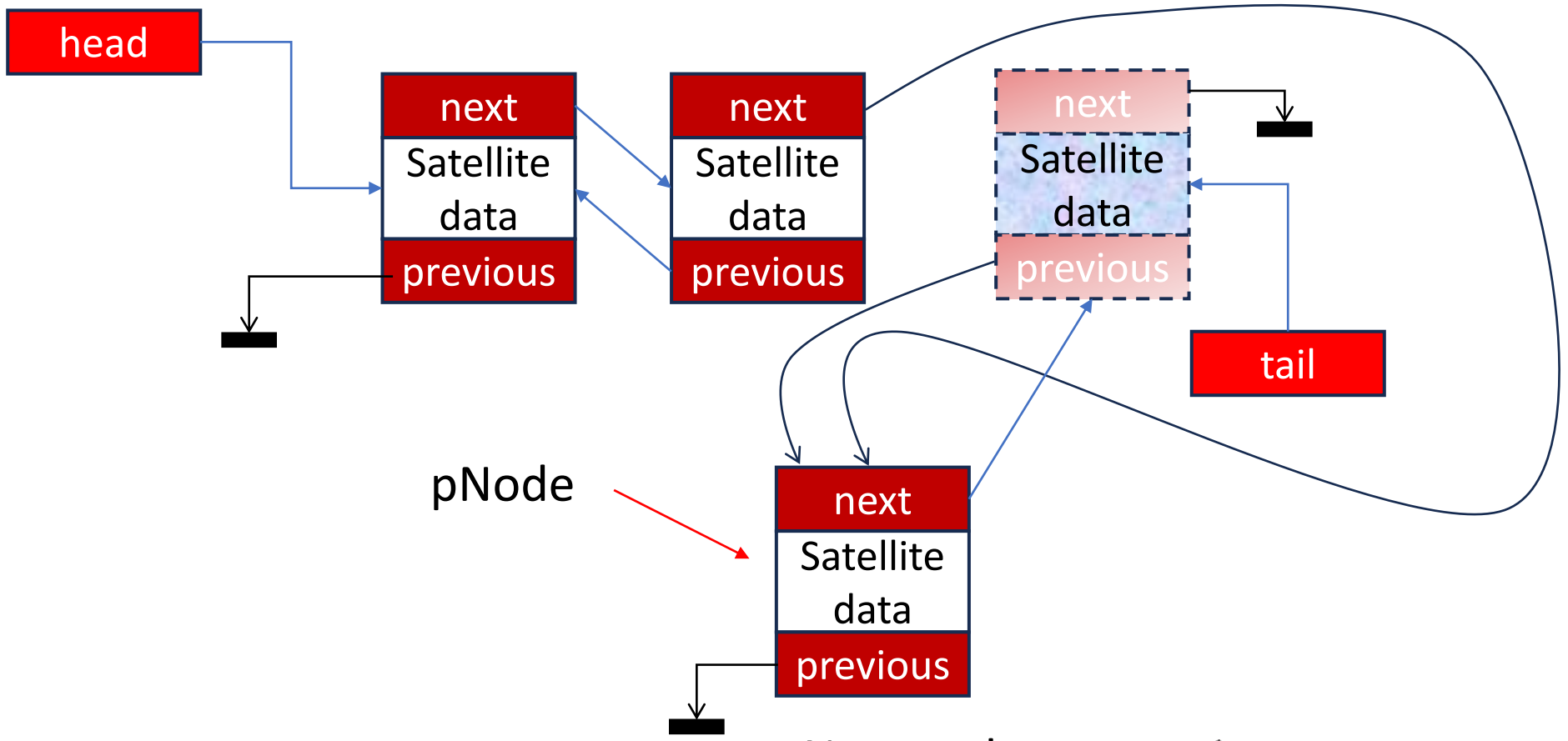
```
template<class T> class LinkedList {
public:
    void append(const T& t); // insert at the end of list
};
```



Linked list initial operation: append

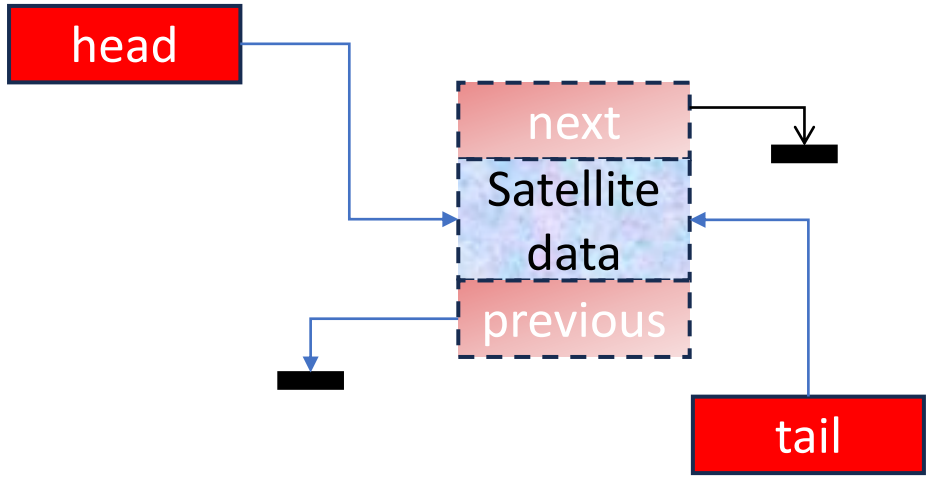
- Append

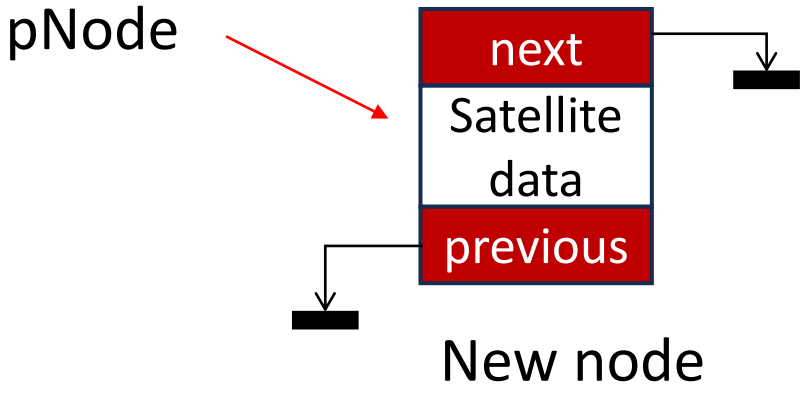
```
template<class T> class LinkedList {
public:
    void append(const T& t); // insert at the end of list
};
```



Linked list initial operation: append

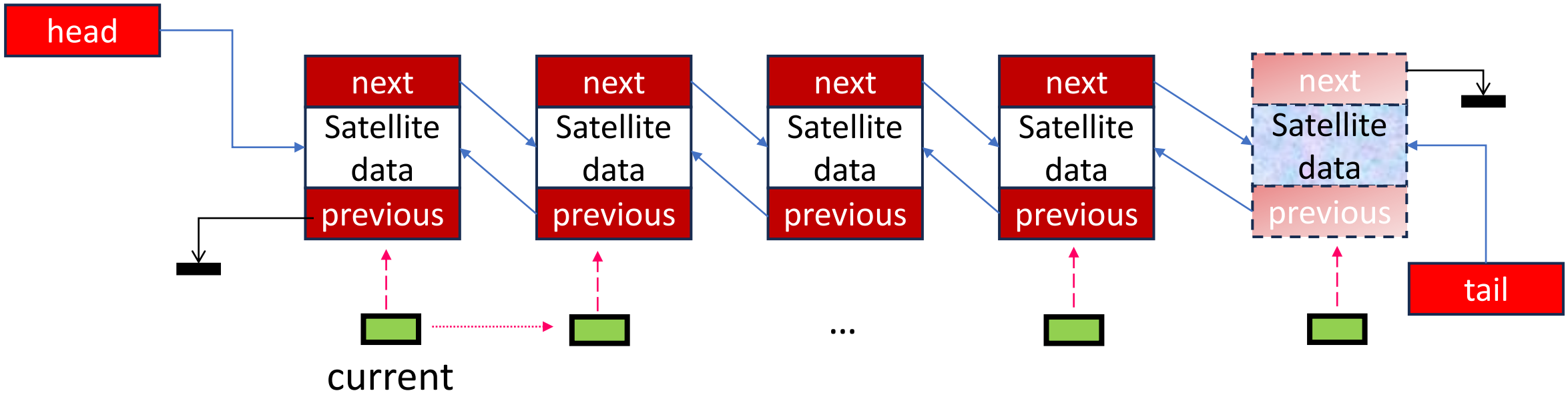
- Append to the empty list





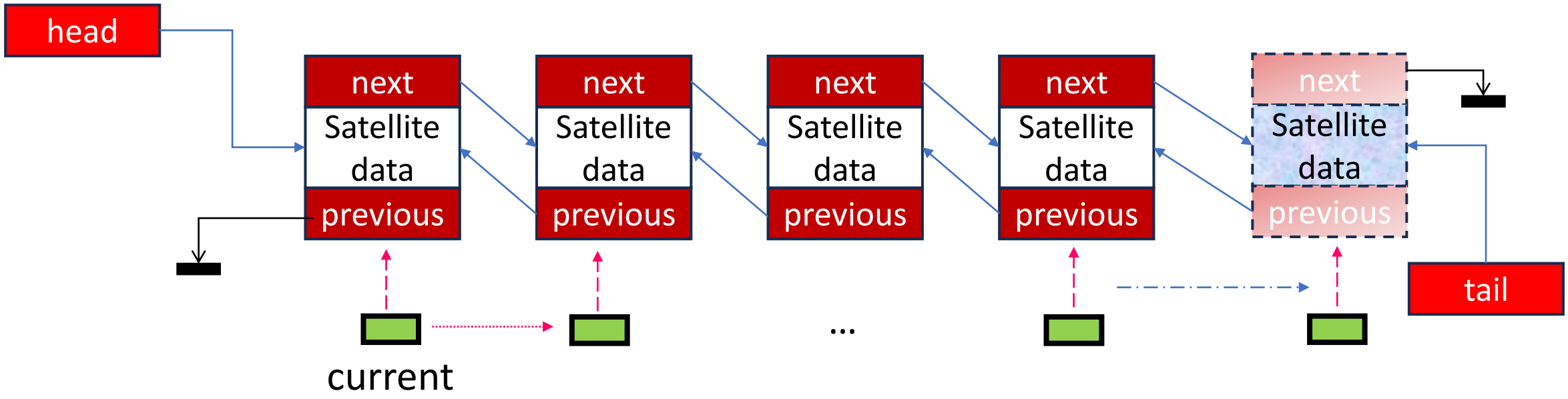
special member functions: Destructor

- Traversing the linked list and deleting nodes:

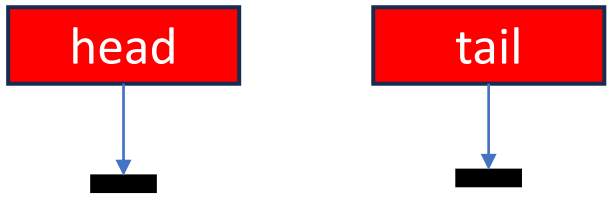


special member functions: Destructor

- Traversing the linked list and deleting nodes:



- Destructed list



using Delegating constructor

- Calling default constructor in copy constructor
- Before using delegating constructor

```
LinkedList() : head_{ nullptr }, tail_{ new Node<T>{T{}} }
{
    head_ = tail_; // the invariant for empty list: head_ == tail_
}
```

```
LinkedList(const LinkedList& linked_list) : head_{ nullptr }, tail_{ new Node<T>{T{}} }
{
    head_ = tail_;
    for (auto p = linked_list.head_; p != linked_list.tail_; p = p->next_) {
        append(p->data_);
    }
}
```

using Delegating constructor

- Calling default constructor in copy constructor
- Before using delegating constructor

```
LinkedList() : head_{ nullptr }, tail_{ new Node<T>{T{}} }
{
    head_ = tail_; // the invariant for empty list: head_ == tail_
}
```

- Copy constructor

```
LinkedList(const LinkedList& linked_list) : head_{ nullptr }, tail_{ new Node<T>{T{}} }
{
    head_ = tail_;
    for (auto p = linked_list.head_; p != linked_list.tail_; p = p->next_) {
        append(p->data_);
    }
}
```

using Delegating constructor

- Calling default constructor in copy constructor
- Before using delegating constructor

```
LinkedList() : head_{ nullptr }, tail_{ new Node<T>{T{}} }
{
    head_ = tail_; // the invariant for empty list: head_ == tail_
}
```

```
LinkedList(const LinkedList& linked_list) : head_{ nullptr }, tail_{ new Node<T>{T{}} }
{
    head_ = tail_;
    for (auto p = linked_list.head_; p != linked_list.tail_; p = p->next_) {
        append(p->data_);
    }
}
```

- After using delegating constructor

```
LinkedList(const LinkedList& linked_list) : LinkedList()
{
    for (auto p = linked_list.head_; p != linked_list.tail_; p = p->next_) {
        append(p->data_);
    }
}
```

Linked list modifier operations: push_back and push_front

- push_back

  Round0 → append

```
template<class T>
class LinkedList {
public:
    void push_back(const T& t) { append(t); }
};
```

- push_front

  Round1 → prepend

```
template<class T>
class LinkedList {
public:
    void push_front(const T& t) { prepend(t); }
};
```

Linked list modifier operations: pop_back and pop_front

- pop_back

```
template<class T>
class LinkedList {
public:
    void pop_back() { remove(tail_); }
};
```

- pop_front

```
template<class T>
class LinkedList {
public:
    void pop_front() { remove(head_); }
};
```

Thanks for your patience …

A man who asks a question is a fool for minute,
The man who does not ask, is a fool for a life.
- Confucius

Learning to ask the right (often hard) questions is an essential part of learning to think as a programmer.
- Bjarne Stroustrup *programming Principles and Practice Using C++, page 4.*

There is no stupid question, but there is stupid answer.
- Howard Hinnant