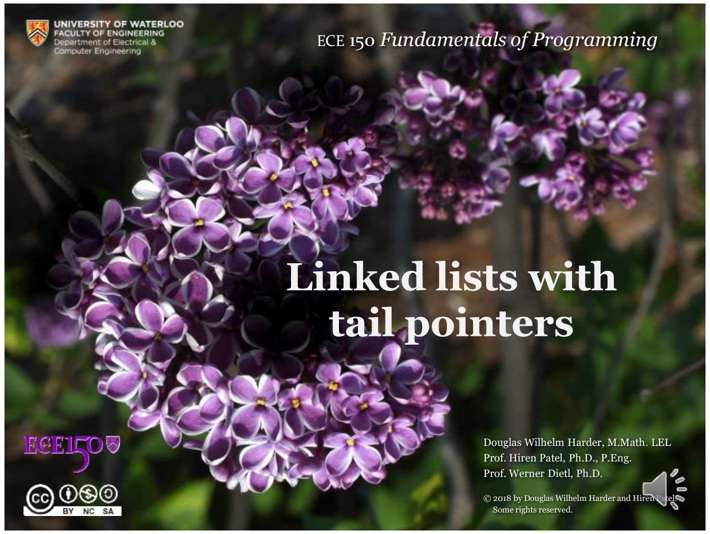


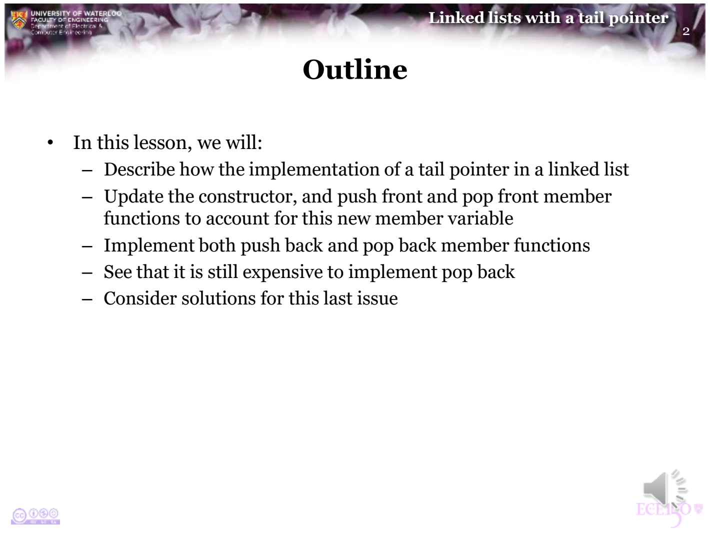
UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical & Computer Engineering

ECE 150 *Fundamentals of Programming*

# Linked lists with tail pointers

Douglas Wilhelm Harder, M.Math. LEL
Prof. Hiren Patel, Ph.D., P.Eng.
Prof. Werner Dietl, Ph.D.

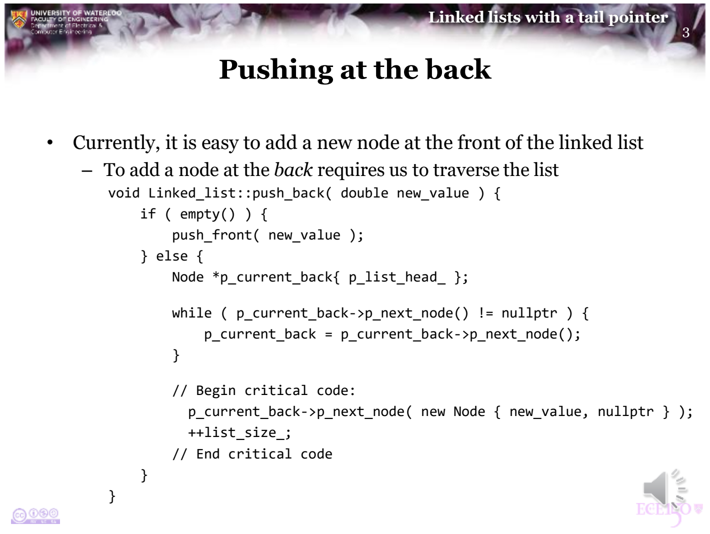
1



## Outline

- In this lesson, we will:
  - Describe how the implementation of a tail pointer in a linked list
  - Update the constructor, and push front and pop front member functions to account for this new member variable
  - Implement both push back and pop back member functions
  - See that it is still expensive to implement pop back
  - Consider solutions for this last issue

2



## Pushing at the back

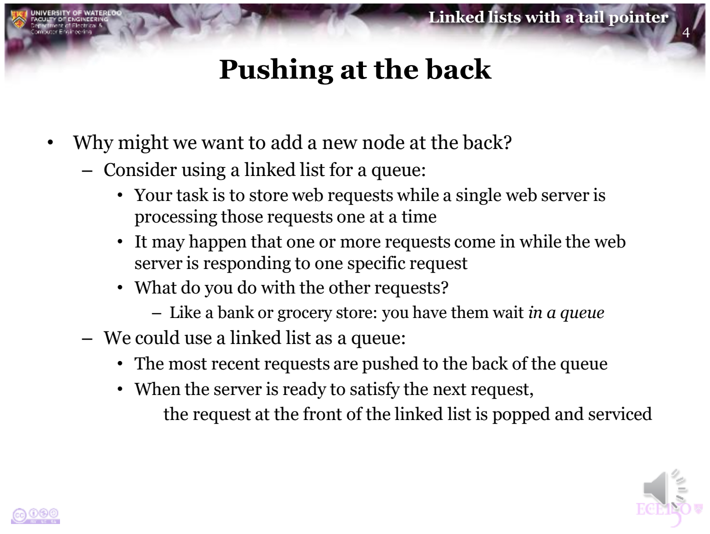
- Currently, it is easy to add a new node at the front of the linked list
  - To add a node at the *back* requires us to traverse the list

```
void Linked_list::push_back( double new_value ) {
    if ( empty() ) {
        push_front( new_value );
    } else {
        Node *p_current_back{ p_list_head_ };

        while ( p_current_back->p_next_node() != nullptr ) {
            p_current_back = p_current_back->p_next_node();
        }

        // Begin critical code:
          p_current_back->p_next_node( new Node { new_value, nullptr } );
          ++list_size_;
        // End critical code
    }
}
```

3



## Pushing at the back

- Why might we want to add a new node at the back?
  - Consider using a linked list for a queue:
    - Your task is to store web requests while a single web server is processing those requests one at a time
    - It may happen that one or more requests come in while the web server is responding to one specific request
    - What do you do with the other requests?
      - Like a bank or grocery store: you have them wait *in a queue*
  - We could use a linked list as a queue:
    - The most recent requests are pushed to the back of the queue
    - When the server is ready to satisfy the next request,
      the request at the front of the linked list is popped and serviced

4

## A pointer to the back

- How can we easily get to the last node in a linked list?
  - How about having a member variable `p_list_tail_` which we always keep assigned to the address of the last node?

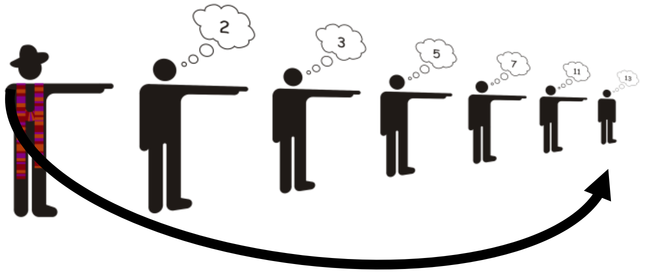
```
class Linked_list {
    public:
        // Public constructors, member functions
        // and the destructor

    private:
        Node *p_list_head_;
        Node *p_list_tail_;
        std::size_t list_size_;

    // A list of any friends
};
```

## A pointer to the back

- What will this look like?



## Initializing the pointer to the back

- An initial linked list is empty, so like the pointer to the list head, the pointer to the list tail must be initialized to the null pointer

```
Linked_list::Linked_list():
p_list_head_{ nullptr },
p_list_tail_{ nullptr },
list_size_{ 0 } {
    // Empty constructor
}
```

## Updating other member functions

- Push front now has to update the tail pointer, but only if the linked list is empty

```
void Linked_list::push_front( double new_value ) {
    // Begin critical code:
        p_list_head_ = new Node{ new_value, p_list_head_ };
        if ( p_list_tail_ == nullptr ) {
            p_list_tail_ = p_list_head_;
        }
        ++list_size_;
    // End critical code
}
```

## Simplifying push front

- We must also update pop front:

```
void Linked_list::pop_front() {
    if ( !empty() ) {
        Node *p_old_head{ p_list_head_ };

        // Begin critical code:
            p_list_head_ = p_list_head_->p_next_node();
            if ( p_list_head_ == nullptr ) {
                p_list_tail_ = nullptr;
            }
            --list_size_;
        // End critical code

        delete p_old_head;
        p_old_head = nullptr;
    }
}
```

9

## Simplifying push front

- Try not to use tests that anticipate a change has not yet been made:

```
void Linked_list::pop_front() {
    if ( !empty() ) {
        Node *p_old_head{ p_list_head_ };

        // Begin critical code
            p_list_head_ = p_list_head_->p_next_node();
            if ( size() == 1 ) {
                p_list_tail_ = nullptr;
            }
            --list_size_;
        // End critical code

        delete p_old_head;
        p_old_head = nullptr;
    }
}
```

10

## Adding a back member function

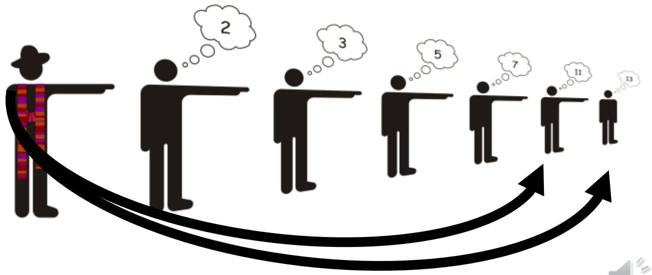
- We can also access the last entry of the linked list:

```
double Linked_list::back() const {
    if ( p_list_tail_ != nullptr ) {
        return p_list_tail_->value();
    } else {
        assert( p_list_tail_ == nullptr );
        throw std::out_of_range{ "The linked list is empty" };
    }
}
```

11

## Pushing at the back

- How do we add a new node at the back?
  – Create a new node and have the last node point to it
  – Update the tail pointer



12

## Pushing at the back

- Implementing this in code:

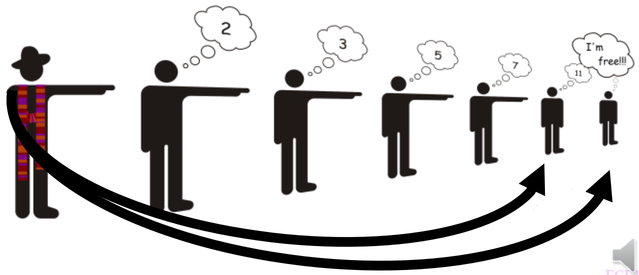
```
void Linked_list::push_back( double new_value ) {
    if ( empty() ) {
        push_front();
    } else {
        // Begin critical code:
            p_list_tail_->p_next_node(
                                 new Node{ new_value, nullptr } );
            p_list_tail_ = p_list_tail->p_next_node();
            ++list_size_;
        // End critical code
    }
}
```

## Pushing at the back

- Notice now that we can:
  - Easily push and pop at the front of the linked list
  - Push at the back of the linked list
- Can we pop at the back of the linked list?

## Popping at the back

- How do we pop the last node?
  - Update the tail pointer
  - Free the last node
  - Update what used to be the second-last node



## Popping at the back

- Implementing this in code:

```
void Linked_list::pop_back() {
    if ( size() == 1 ) {
        pop_front();
    } else if ( !empty() ) {
        Node *p_new_tail{ p_list_head_ };

        while ( p_new_tail->p_next_node() != p_list_tail_ ) {
            p_new_tail = p_new_tail->p_next_node();
        }

        // Begin critical code:
            // No need to update p_list_head_ as size() > 1
            p_list_tail_ = p_new_tail;
            delete p_list_tail_->p_next_node();
            p_list_tail_->p_next_node( nullptr );
            --list_size_;
        // End critical code
    }
}
```

## Discussion

- Popping the last node requires to access the second-last node
  - Finding that second-last node requires us to walk through the nodes
  - This will, of course, be slow for large linked lists
- Fortunately, this is not required for a queue:
  - For a queue, we push new requests at the back (fast)
  - When we are ready to service a request, we pop that request that has been in the queue the longest: the one at the front
- Never-the-less, in order to pop the last node efficiently, we now must change the node class
  - The node class must now not only store the address of the next node, but also store the address of the previous node

17

## Summary

- Following this lesson, you now
  - Know how to implement a tail pointer
  - Understand how to update the existing member functions to correctly update for the tail pointer
  - Know how to implement a push back member function
  - Understand how to implement a pop back member function
    - With the understanding this is an expensive operation
  - Are aware that a linked list could have nodes with two pointers:
    - One for the next node in the linked list and one for the previous node

18

## References

[1] https://en.wikipedia.org/wiki/Linked_list
[2] https://en.wikipedia.org/wiki/Node_(computer_science)

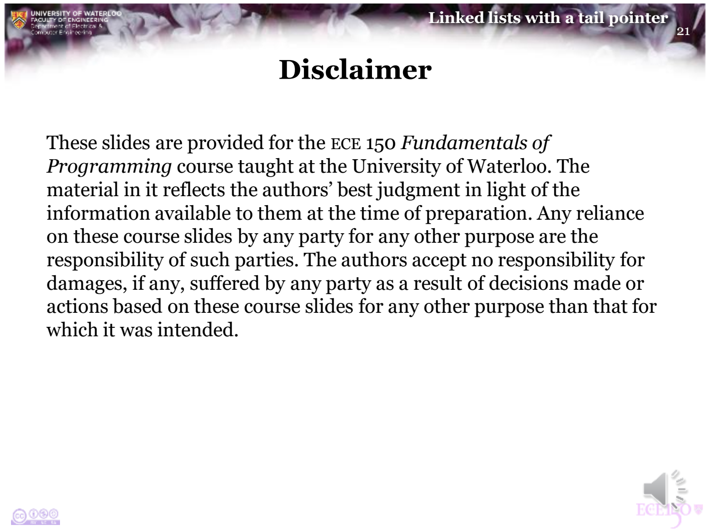
19

## Colophon

These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using `Consolas`.

The photographs of lilacs in bloom appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens on May 27, 2018 by Douglas Wilhelm Harder. Please see
https://www.rbg.ca/
for more information.

20



## Disclaimer

21