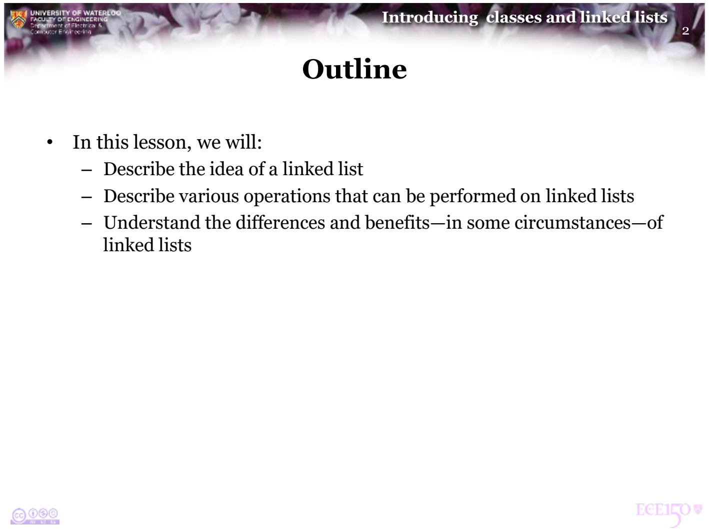
UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING
Department of Electrical & Computer Engineering

ECE 150 *Fundamentals of Programming*

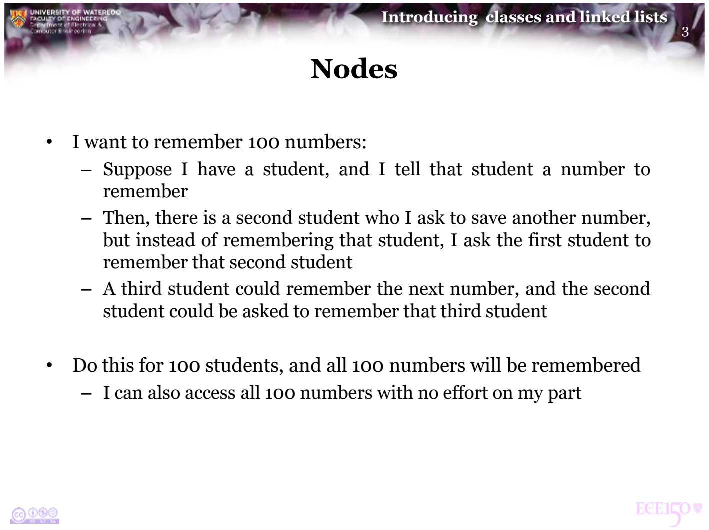
Introducing classes and linked lists

Douglas Wilhelm Harder, M.Math
Prof. Hiren Patel, Ph.D.
hdpatel@uwaterloo.ca dwharder@uwaterloo.ca



Outline

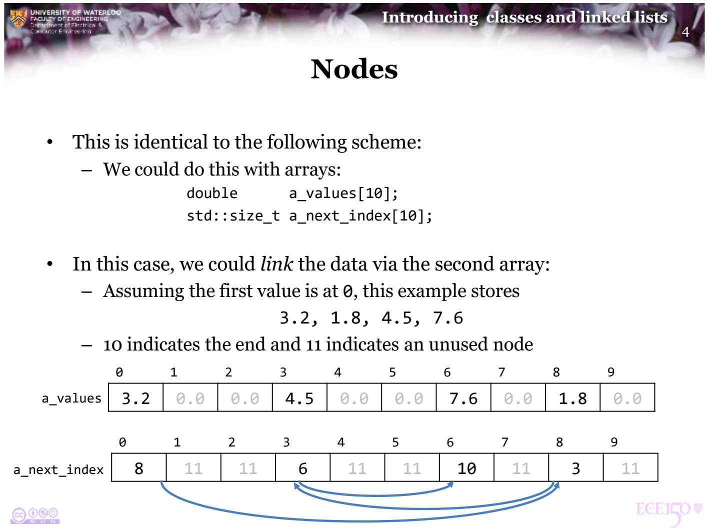
- In this lesson, we will:
  - Describe the idea of a linked list
  - Describe various operations that can be performed on linked lists
  - Understand the differences and benefits—in some circumstances—of linked lists



Nodes

- I want to remember 100 numbers:
  - Suppose I have a student, and I tell that student a number to remember
  - Then, there is a second student who I ask to save another number, but instead of remembering that student, I ask the first student to remember that second student
  - A third student could remember the next number, and the second student could be asked to remember that third student

- Do this for 100 students, and all 100 numbers will be remembered
  - I can also access all 100 numbers with no effort on my part



Nodes

- This is identical to the following scheme:
  - We could do this with arrays:

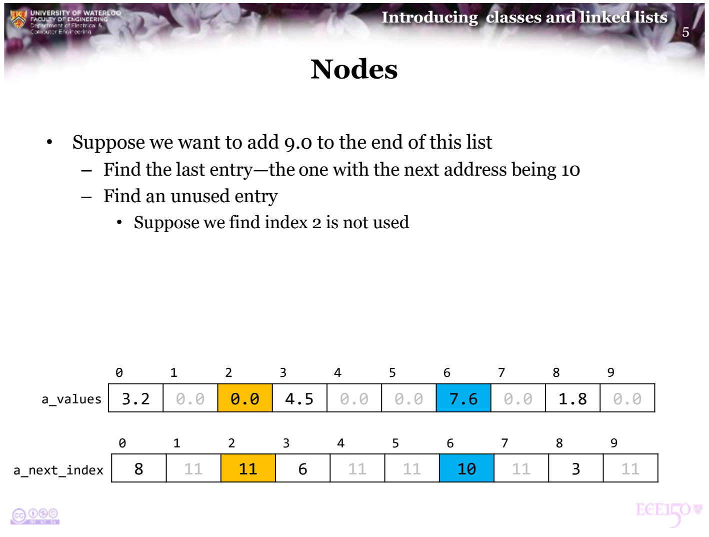
```
double      a_values[10];
std::size_t a_next_index[10];
```

- In this case, we could *link* the data via the second array:
  - Assuming the first value is at 0, this example stores

    3.2, 1.8, 4.5, 7.6
  - 10 indicates the end and 11 indicates an unused node

	0	1	2	3	4	5	6	7	8	9
a_values	3.2	0.0	0.0	4.5	0.0	0.0	7.6	0.0	1.8	0.0

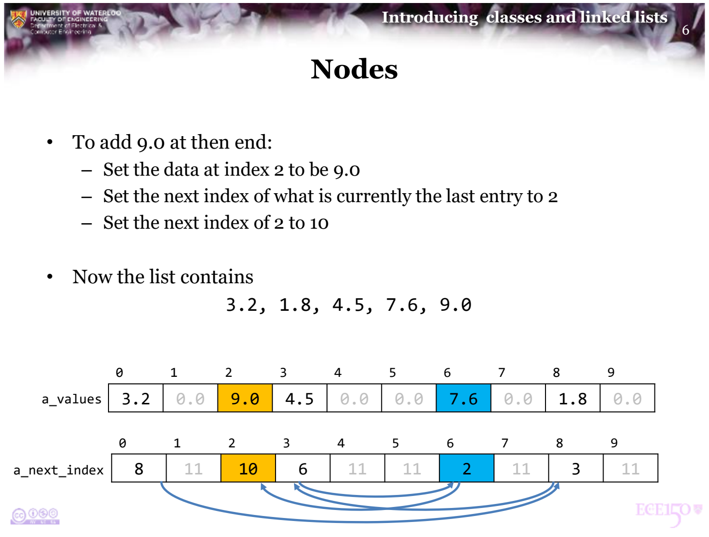
	0	1	2	3	4	5	6	7	8	9
a_next_index	8	11	11	6	11	11	10	11	3	11



Nodes

- Suppose we want to add 9.0 to the end of this list
  - Find the last entry—the one with the next address being 10
  - Find an unused entry
    - Suppose we find index 2 is not used

	0	1	2	3	4	5	6	7	8	9
a_values	3.2	0.0	0.0	4.5	0.0	0.0	7.6	0.0	1.8	0.0
a_next_index	8	11	11	6	11	11	10	11	3	11

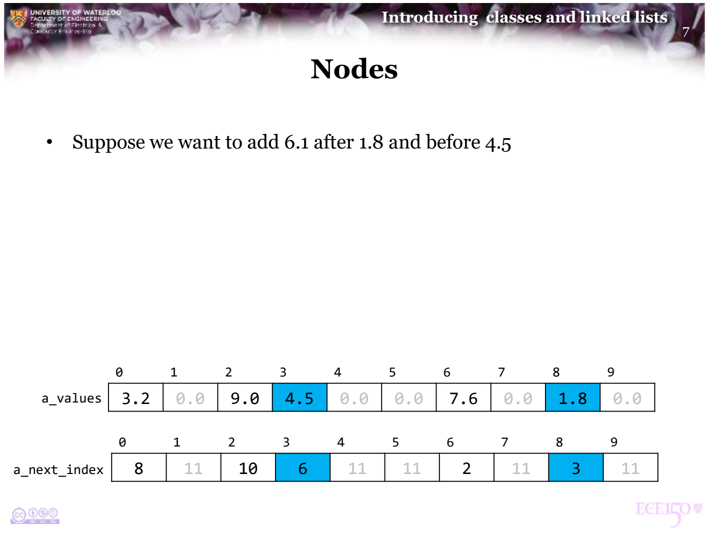


Nodes

- To add 9.0 at then end:
  - Set the data at index 2 to be 9.0
  - Set the next index of what is currently the last entry to 2
  - Set the next index of 2 to 10
- Now the list contains

```
3.2, 1.8, 4.5, 7.6, 9.0
```

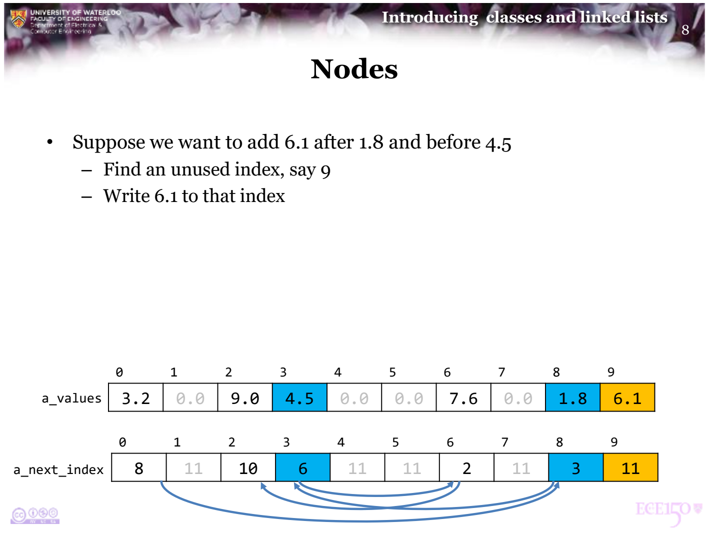
	0	1	2	3	4	5	6	7	8	9
a_values	3.2	0.0	9.0	4.5	0.0	0.0	7.6	0.0	1.8	0.0
a_next_index	8	11	10	6	11	11	2	11	3	11



Nodes

- Suppose we want to add 6.1 after 1.8 and before 4.5

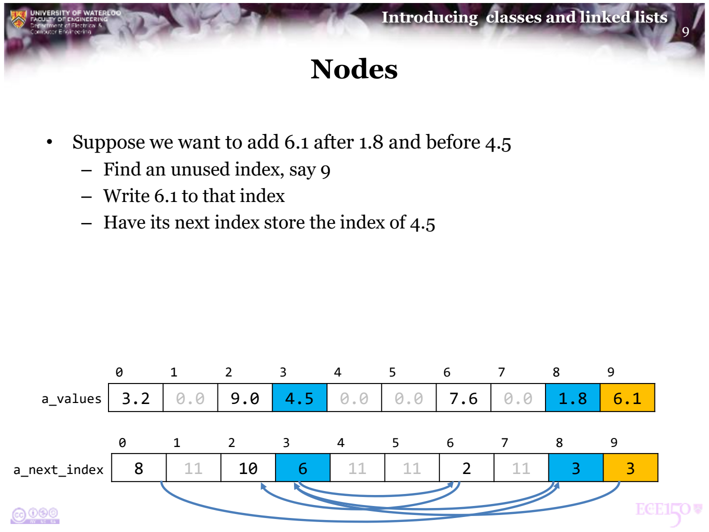
	0	1	2	3	4	5	6	7	8	9
a_values	3.2	0.0	9.0	4.5	0.0	0.0	7.6	0.0	1.8	0.0
a_next_index	8	11	10	6	11	11	2	11	3	11



Nodes

- Suppose we want to add 6.1 after 1.8 and before 4.5
  - Find an unused index, say 9
  - Write 6.1 to that index

	0	1	2	3	4	5	6	7	8	9
a_values	3.2	0.0	9.0	4.5	0.0	0.0	7.6	0.0	1.8	6.1
a_next_index	8	11	10	6	11	11	2	11	3	11

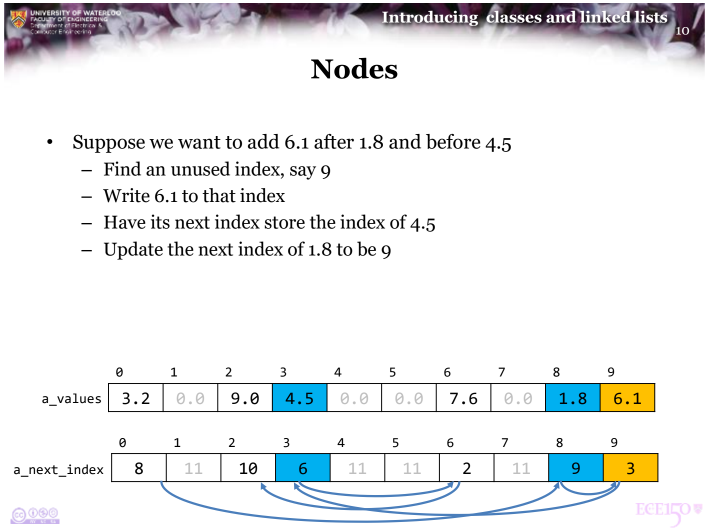


Introducing classes and linked lists 9

Nodes

- Suppose we want to add 6.1 after 1.8 and before 4.5
  - Find an unused index, say 9
  - Write 6.1 to that index
  - Have its next index store the index of 4.5

	0	1	2	3	4	5	6	7	8	9
a_values	3.2	0.0	9.0	4.5	0.0	0.0	7.6	0.0	1.8	6.1
a_next_index	8	11	10	6	11	11	2	11	3	3

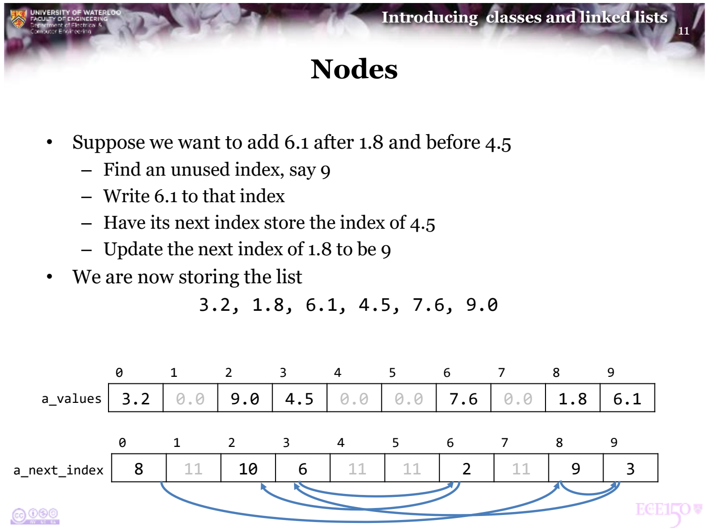


Introducing classes and linked lists 10

Nodes

- Suppose we want to add 6.1 after 1.8 and before 4.5
  - Find an unused index, say 9
  - Write 6.1 to that index
  - Have its next index store the index of 4.5
  - Update the next index of 1.8 to be 9

	0	1	2	3	4	5	6	7	8	9
a_values	3.2	0.0	9.0	4.5	0.0	0.0	7.6	0.0	1.8	6.1
a_next_index	8	11	10	6	11	11	2	11	9	3



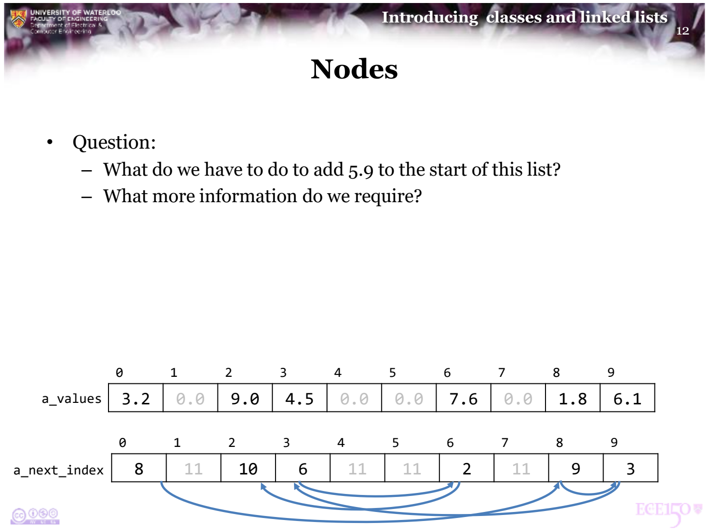
Introducing classes and linked lists 11

Nodes

- Suppose we want to add 6.1 after 1.8 and before 4.5
  - Find an unused index, say 9
  - Write 6.1 to that index
  - Have its next index store the index of 4.5
  - Update the next index of 1.8 to be 9
- We are now storing the list

```
3.2, 1.8, 6.1, 4.5, 7.6, 9.0
```

	0	1	2	3	4	5	6	7	8	9
a_values	3.2	0.0	9.0	4.5	0.0	0.0	7.6	0.0	1.8	6.1
a_next_index	8	11	10	6	11	11	2	11	9	3



Introducing classes and linked lists 12

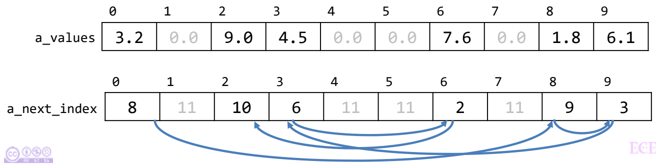
Nodes

- Question:
  - What do we have to do to add 5.9 to the start of this list?
  - What more information do we require?

	0	1	2	3	4	5	6	7	8	9
a_values	3.2	0.0	9.0	4.5	0.0	0.0	7.6	0.0	1.8	6.1
a_next_index	8	11	10	6	11	11	2	11	9	3

Nodes

- Problems with this approach:
  - We can still only store 10 values in this node
  - It's better to think of a *node* being:
    - A value
    - The index of the next item in the list
  - These two pieces of information are, however, stored in different arrays



	0	1	2	3	4	5	6	7	8	9
a_values	3.2	0.0	9.0	4.5	0.0	0.0	7.6	0.0	1.8	6.1

	0	1	2	3	4	5	6	7	8	9
a_next_index	8	11	10	6	11	11	2	11	9	3

Nodes

- A class is a data structure that allows you to store related data together

```
class Index_node {
    public:
        double      value_;
        std::size_t next_index_;
};
```

- The entries `value_` and `next_index_` are called *member variables* of the class
- The identifier `Index_node` becomes a type no different from `int`, `long`, `double`, etc.

Nodes

- It is possible to declare either an instance or an array of this class:

```
int main() {
    Index_node value{0.0, 0};
    Index_node a_values[10];    // uninitialized

    return 0;
}
```

- Each instance of this class (be it the local variable value or each entry of the array) is described as an *object*
- Like local variables, each object occupies memory
  - In this case, 16 bytes are required:
    - 8 bytes for the double
    - 8 bytes for the index
  - The memory is allocated on the stack and is cleaned up when the function exits

```
class Index_node {
    public:
        double      value_;
        std::size_t next_index_;
};
```

Nodes

- The data associated with the class are called *member variables*

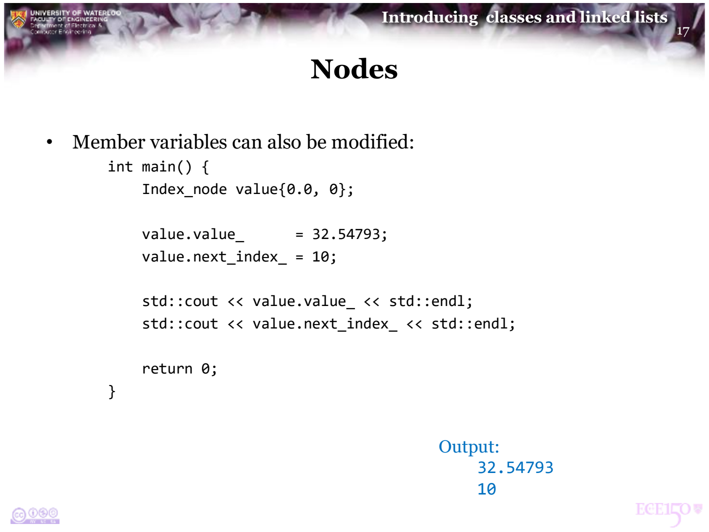
```
class Index_node {
    public:
        double      value_;
        std::size_t next_index_;
};
```

- Member variables can be accessed:

```
int main() {
    Index_node value{0.0, 0};
    std::cout << value.value_ << std::endl;
    std::cout << value.next_index_ << std::endl;

    return 0;
}
```

Output:

```
0
0
```



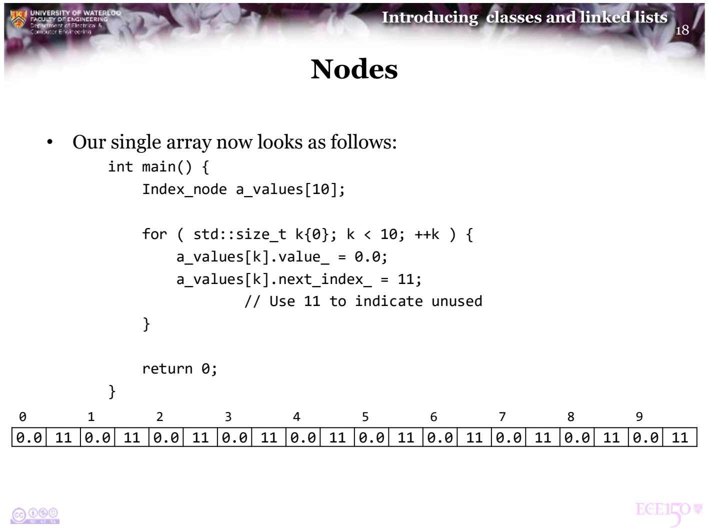
Nodes

- Member variables can also be modified:

```
int main() {
    Index_node value{0.0, 0};

    value.value_      = 32.54793;
    value.next_index_ = 10;

    std::cout << value.value_ << std::endl;
    std::cout << value.next_index_ << std::endl;

    return 0;
}
```

Output:

```
32.54793
10
```

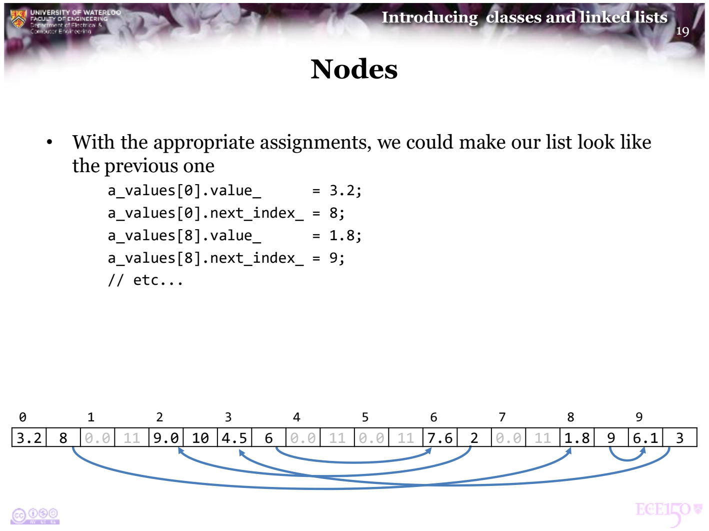


Nodes

- Our single array now looks as follows:

```
int main() {
    Index_node a_values[10];

    for ( std::size_t k{0}; k < 10; ++k ) {
        a_values[k].value_ = 0.0;
        a_values[k].next_index_ = 11;
                // Use 11 to indicate unused
    }

    return 0;
}
```

0		1		2		3		4		5		6		7		8		9	
0.0	11	0.0	11	0.0	11	0.0	11	0.0	11	0.0	11	0.0	11	0.0	11	0.0	11	0.0	11

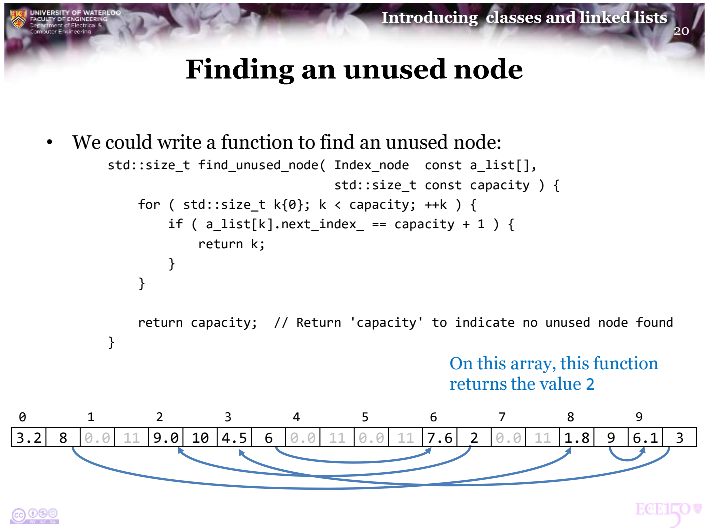


Nodes

- With the appropriate assignments, we could make our list look like the previous one

```
a_values[0].value_      = 3.2;
a_values[0].next_index_ = 8;
a_values[8].value_      = 1.8;
a_values[8].next_index_ = 9;
// etc...
```

0		1		2		3		4		5		6		7		8		9	
3.2	8	0.0	11	9.0	10	4.5	6	0.0	11	0.0	11	7.6	2	0.0	11	1.8	9	6.1	3



Finding an unused node

- We could write a function to find an unused node:

```
std::size_t find_unused_node( Index_node  const a_list[],
                              std::size_t const capacity ) {
    for ( std::size_t k{0}; k < capacity; ++k ) {
        if ( a_list[k].next_index_ == capacity + 1 ) {
            return k;
        }
    }

    return capacity;  // Return 'capacity' to indicate no unused node found
}
```

On this array, this function returns the value 2

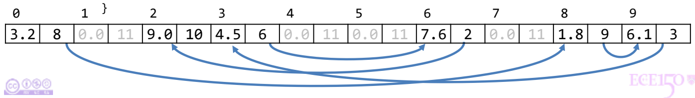
0		1		2		3		4		5		6		7		8		9	
3.2	8	0.0	11	9.0	10	4.5	6	0.0	11	0.0	11	7.6	2	0.0	11	1.8	9	6.1	3

Appending a new node

- We could write a function to append a value:
  - Find a unused entry
  - Find the last entry in the list
  - Update the last entry and the unused entry

Appending a new node

```
bool push_back( Index_node a_list[], std::size_t const capacity,
                double const new_value ) {
    std::size_t new_index{ find_unused_node( a_list, capacity ) };

    if ( new_index == capacity ) {
        return false;  // We could not append the new value
    }

    a_list[new_index].value_ = new_value;
    a_list[new_index].next_index_ = capacity;

    std::size_t last{0}; // Assume the first entry is at index 0...

    while ( a_list[last].next_index_ != capacity ) {
        last = a_list[last].next_index_;
    }

    a_list[last].next_index_ = new_index;

    return true;
}
```



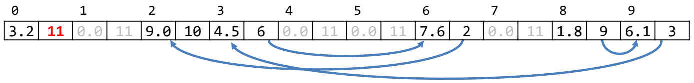
0		1		2		3		4		5		6		7		8		9	
3.2	8	0.0	11	9.0	10	4.5	6	0.0	11	0.0	11	7.6	2	0.0	11	1.8	9	6.1	3

Nodes

- There are two weakness with this approach:
  - There are only a finite number of entries
  - The user has access to everything and can make mistakes...

```
std::size_t find_unused_node( Index_node a_list[],
                              std::size_t const capacity ) {
    for ( std::size_t k{0}; k < capacity; ++k ) {
        if ( a_list[k].next_index_ = capacity + 1 ) {
            return k;
        }
    }

    return capacity;
}
```

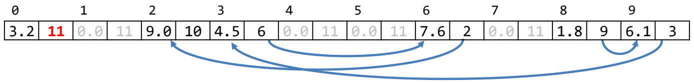
Anytime anyone uses this class, that programmer could make trivial but serious mistakes...



0		1		2		3		4		5		6		7		8		9	
3.2	11	0.0	11	9.0	10	4.5	6	0.0	11	0.0	11	7.6	2	0.0	11	1.8	9	6.1	3

Nodes

- The problem here is any user can access and modify any entry
  - This means anytime anyone uses this data structure, there can be mistakes



0		1		2		3		4		5		6		7		8		9	
3.2	11	0.0	11	9.0	10	4.5	6	0.0	11	0.0	11	7.6	2	0.0	11	1.8	9	6.1	3

Summary

- Following this lesson, you now
  - Understand how classes can be used to combine related material
  - Know what a node is and how a link list is structured
  - Recognize there are problems when information is available

References

[1] No references?

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.

Disclaimer

These slides are provided for the ECE 150 *Fundamentals of Programming* course taught at the University of Waterloo. The material in it reflects the authors' best judgment in light of the information available to them at the time of preparation. Any reliance on these course slides by any party for any other purpose are the responsibility of such parties. The authors accept no responsibility for damages, if any, suffered by any party as a result of decisions made or actions based on these course slides for any other purpose than that for which it was intended.