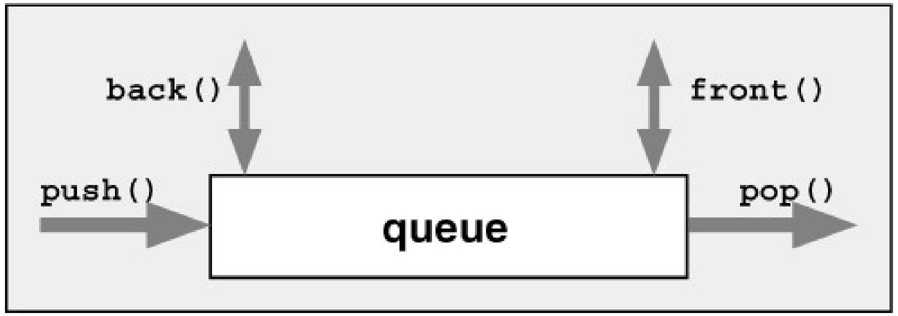
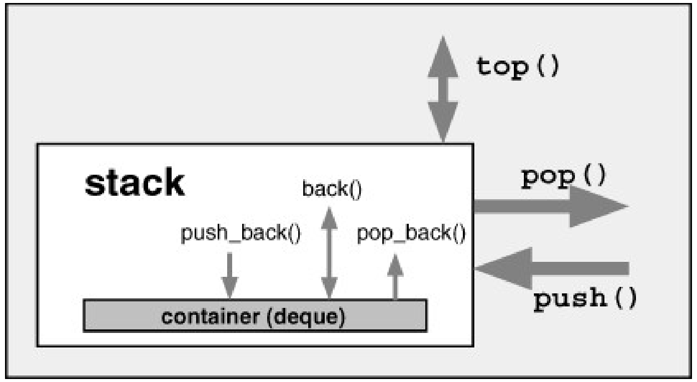
CPSC 131 – Data Structures

Stack and Queue
Abstract Data Types

Professor T. L. Bettens
Fall 2020





Key terms

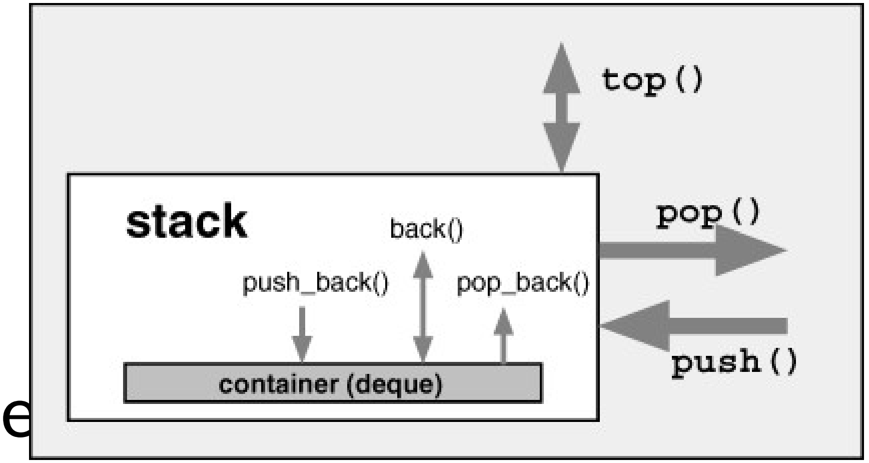
- An abstract data type (ADT) is an abstraction of a data structure that specifies
  - Data stored
  - Operations on the data
  - Error conditions associated with operations
- Does not talk about implementation

- Container Adapter (Wrapper class) implementation
  - Provides a public interface, but
  - Delegates the implementation to some other data structure

You must tell it what other structure to use

Stack Concepts

- LIFO – Last In, First Out (Stack)
  - push() - insert elements into the stack
  - pop() - remove elements in the opposite order in which they were inserted (“last in, first out”)
  - top() - view the most recent element inserted

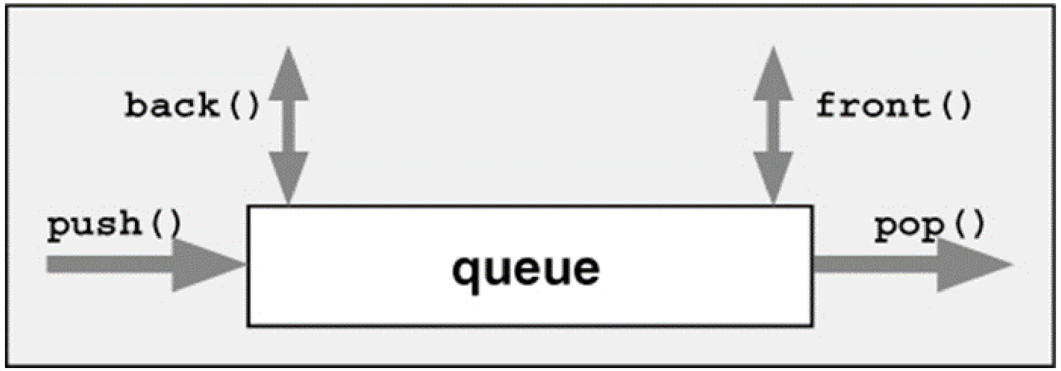


Example: spring-loaded plate dispenser

Queue Concepts

- FIFO – First In, First Out (Queue)
  - push() - insert elements into the stack
  - pop() - remove elements in the same order in which they were inserted (“first in, first out”)
  - back() - view the most recent element inserted (the one at the back)
  - front() - view the least recent element inserted (the one at the front)

zyBook calls this peek()

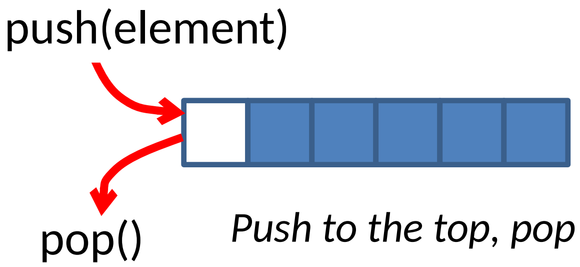


Push to the back, pop from the front

Example: standing in line

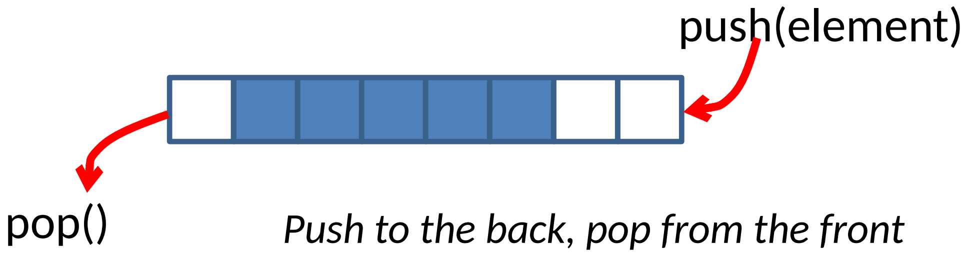
Stack vs Queue

- Stack (Last-In First-Out)



Push to the top, pop from the top

- Queue (First-In First-Out)



Push to the back, pop from the front

Performance

- Performance
  - Let ***n*** be the number of elements in the stack or queue
  - The space used is ***O***(***n***)
  - Each operation runs in time ***O***(1)

Operation	Time	Applicability
void push(element)	O(1)	stack, queue
void pop()	O(1)	stack, queue
element top()	O(1)	stack
element front()	O(1)	queue
element back()	O(1)	queue
size_t size()	O(1)	stack, queue
bool empty()	O(1)	stack, queue

Perspective

Two ways of working with data structures

1. As a class consumer (client) - using a data structure

2. As the class designer - implementing a data structure

As a class consumer (client) - using a data structure

USING A STACK

Example – Using a Stack

Client creates instances of stacks specifying what underlying container to use

```
int main()
{
  try
  {
    // array based Stack, not in zyBook
    // empty stack where stack is implemented over a fixed sized standard
 array
    Stack<Student, std::array<Student, 10>> myStack_3;
    test( myStack_3 );

    // Standard Stack usage with standard containers
    // default standard stack (uses std::deque as underlying container)
    std::stack<Student> myStack_4;
    test( myStack_4 );

    // standard stack with standard doubly linked list as underlying cont
ainer
    std::stack<Student, std::list<Student>> myStack_5;
    test( myStack_5 );

    // standard stack with standard vector as underlying
    std::stack<Student, std::vector<Student>> myStack_6;
    test( myStack_6 );
  }

  catch (const std::exception & ex)
  {
    std::cerr << ex.what() << '\n';
  }
}
```

Stack class that we wrote

Stack class from the STL

```
// A simple test driver to exercise the container.  Sense stacks and
queues have
// (nearly) the same interface, the same test driver is used for both.
The
// container tested is intentionally passed by value (makes a local copy).
template<class Container_Type>
void test( Container_Type myContainer )
{
  // A stack is an Abstract Data Type, usually implemented as a limited
  // interlace over some other data structure Things you can do to a
stack:
  myContainer.push( {"Tom"    } );
  myContainer.push( {"Aaron" } );
  myContainer.push( {"Brenda"} );
  myContainer.pop();
  myContainer.push( {"Katelyn"} );

  // Display the contents.  Stacks and queues do not allow traversal (you
can't
  // see anything but the top (stack and queue) and both (queue only), so
to
  // display the contents we have to inspect each element at the top and
then
  // remove it until the container is
  while( !myContainer.empty() )
  {
    std::cout << myContainer.top();
    myContainer.pop();
  }
  std::cout << '\n';
}
```

Client inserts and removes from the stack

Client views element at the top of the stack

main.cpp

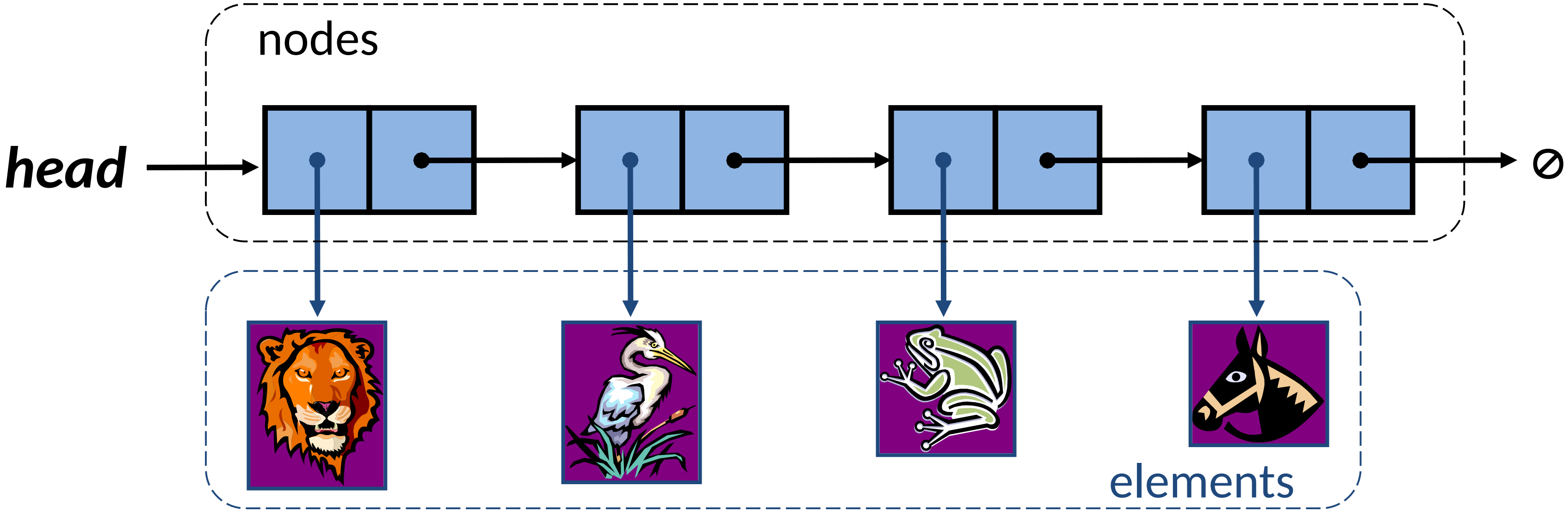
Open file to see full implementation
(taken from our Implantation Examples on TITANium)

As the class designer - implementing a data structure

IMPLEMENTING A STACK WITH A:
SINGLY LINKED LIST
FIXED SIZE ARRAY

Stack as a Linked List

- Stack can be implemented with a singly linked list
- The top element is stored at the first node of the list
- The space used is ***O***(***n***) and each operation of the Stack ADT takes ***O***(1) time



Example - Implementing a stack with a Singly Linked List

Stack ADT takes two template arguments, the type of data to store, and the underlying container, usually defaulted to a recommended choice.

```
template<typename T, class UnderlyingContainer = SLinkedList<T>>
class Stack{
  public:
    void        push( const T & element );
    T           pop();
    T &         top();    // peek() in zyBook
    bool        empty();  // isEmpty() in zyBook
    std::size_t size();   // getLength() in zyBook

  private:
    UnderlyingContainer collection;
};
```

Class designer creates a Stack ADT interface

The underlying container is a private attribute of the Stack ADT.

Stack.hpp

Open file to see full interface definition
(taken from our Implantation Examples on TITANium)

```
void push( const T & element )
{ collection.prepend( element ); }
```

Error checking is performed, but delegated

```
T pop()
{
  // Note: zyBook returns the value popped, the C++ standard template library
  //       does not. popping an element from an empty stack error handling is
  //       handled by the underlying c
  auto element = collection.front();
  collection.removeFront();
  return element;
}
```

The real work is delegated to the underlying container, called collection in this example

```
T & top()
{
  // Note: viewing an element from an empty stack error handling is handled by
  // the underlying container
  return collection.front();
}
```

```
bool empty()
{ return collection.empty(); }
```

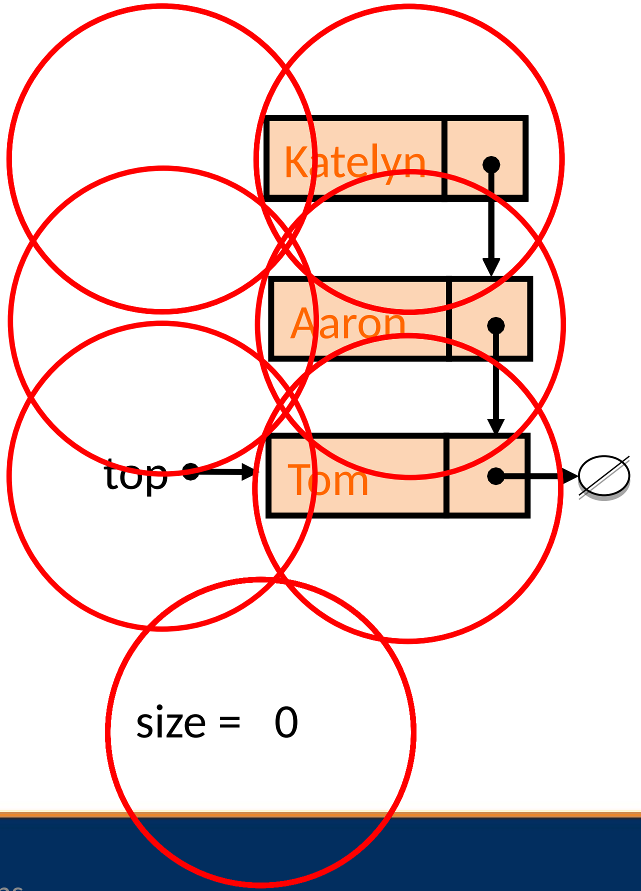
Stack.hxx

Open file to see full implementation
(taken from our Implantation Examples on TITANium)

```
std::size_t size()
{ return collection.size(); }
```

Example - Sketching a stack with a Singly Linked List

```
Stack<Student,
SLinkedList<Student>> myContainer;

myContainer.push( {"Tom"    } );
myContainer.push( {"Aaron" } );
myContainer.push( {"Brenda"} );
myContainer.pop();
myContainer.push( {"Katelyn"} );

while( !myContainer.empty() )
{
  std::cout << myContainer.top();
  myContainer.pop();
}
```



An empty stack

As the class designer - implementing a data structure

IMPLEMENTING A STACK WITH A :

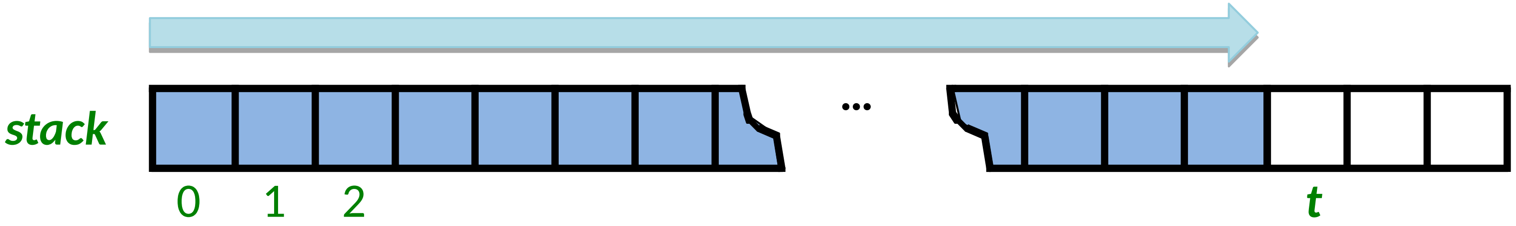
SINGLY LINKED LIST

FIXED SIZE ARRAY

Array-based Stack – Size & Pop

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the next available slot to fill

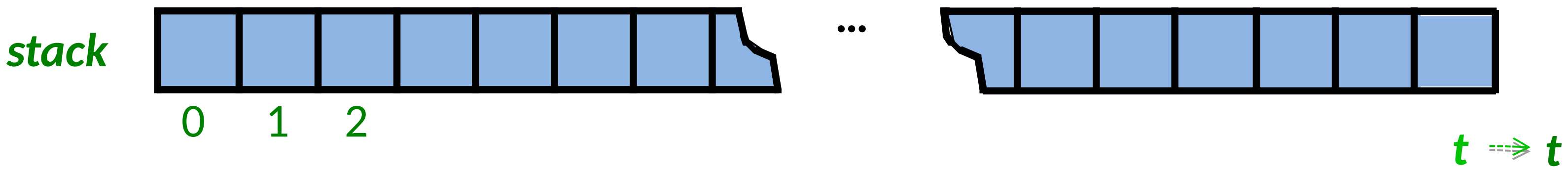
```
Algorithm getlength()
    return t    // slots 0..(t-1) are occupied

Algorithm pop()
    if isempty() then
        throw exception
    else
        $t \leftarrow t - 1$
```



Array-based Stack - Push

```
Algorithm push(e)
        if t == capacity then
                throw exception
         else  {
                stack[t] ← e
        t ← t + 1
}
```

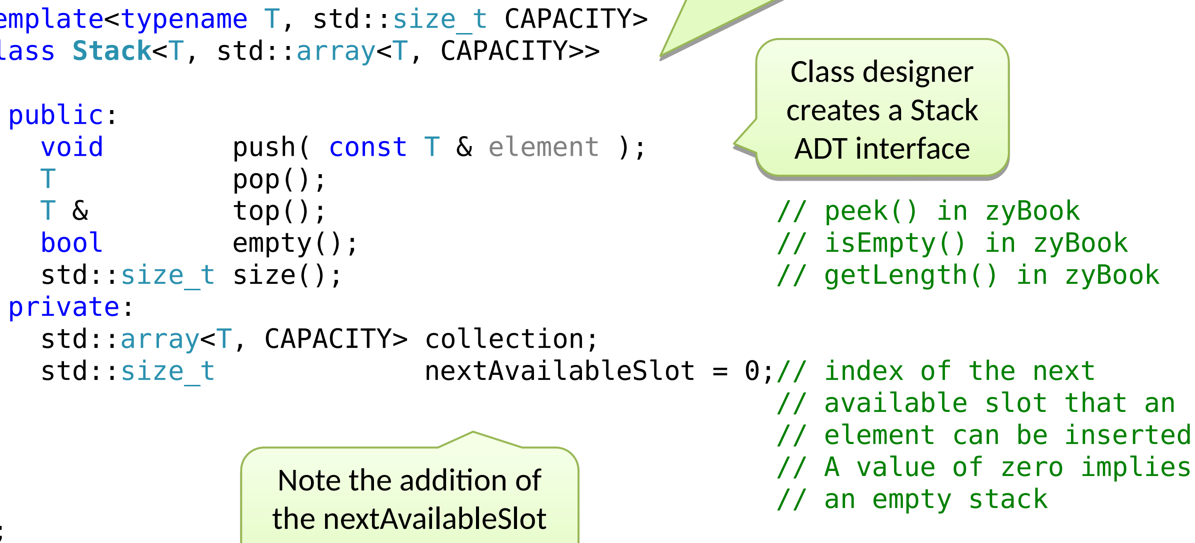
- ❑ The array storing the stack elements may become full
- ❑ A push operation will then throw an exception
  - Limitation of the fixed size array implementation
  - Not limitation of the Stack ADT



Example - Implementing an Array-Based Stack

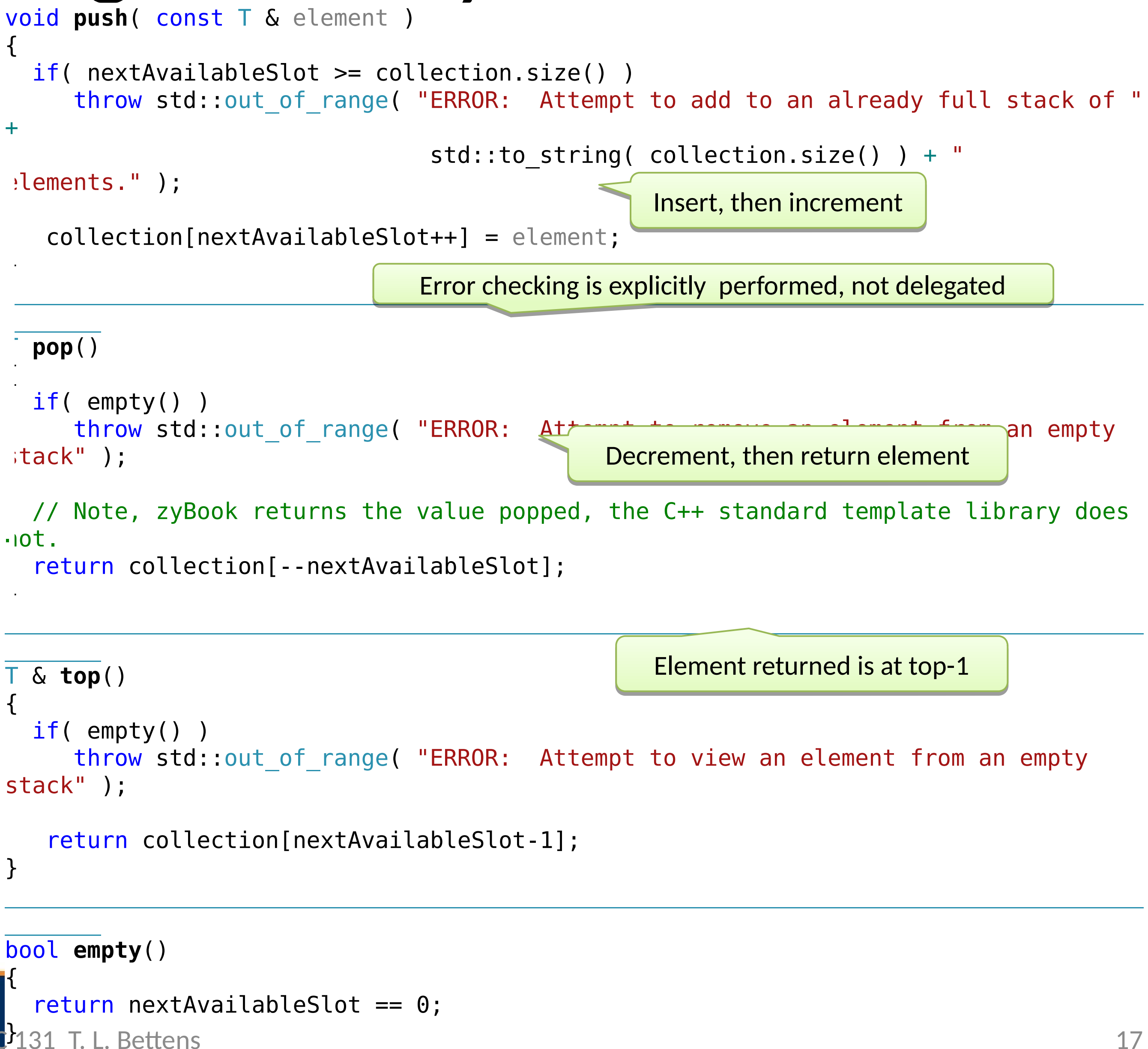
Note the fixed sized array as the underlying container

Class designer creates a Stack ADT interface



```
emplate<typename T, std::size_t CAPACITY>
lass Stack<T, std::array<T, CAPACITY>>

 public:
   void        push( const T & element );
   T           pop();
   T &         top();                          // peek() in zyBook
   bool        empty();                        // isEmpty() in zyBook
   std::size_t size();                         // getLength() in zyBook
 private:
   std::array<T, CAPACITY> collection;
   std::size_t             nextAvailableSlot = 0;// index of the next
                                               // available slot that an
                                               // element can be inserted.
                                               // A value of zero implies
                                               // an empty stack
;
```

Note the addition of the nextAvailableSlot attribute



```
void push( const T & element )
{
  if( nextAvailableSlot >= collection.size() )
     throw std::out_of_range( "ERROR:  Attempt to add to an already full stack of "
+
                             std::to_string( collection.size() ) + "
lements." );

   collection[nextAvailableSlot++] = element;
}

 pop()

  if( empty() )
     throw std::out_of_range( "ERROR:  Attempt to remove an element from an empty
tack" );

  // Note, zyBook returns the value popped, the C++ standard template library does
ot.
  return collection[--nextAvailableSlot];


T & top()
{
  if( empty() )
     throw std::out_of_range( "ERROR:  Attempt to view an element from an empty
stack" );

   return collection[nextAvailableSlot-1];
}

bool empty()
{
  return nextAvailableSlot == 0;
}
```

Insert, then increment

Error checking is explicitly performed, not delegated

Decrement, then return element

Element returned is at top-1

Open file to see full interface definition (taken from our Implantation Examples on TITANium)

Stack.hpp

Stack.hxx

Open file to see full implementation (taken from our Implantation Examples on TITANium)

As a class consumer (client) - using a data structure

USING A QUEUE

Example – Using a Queue

```
int main()
{
  try
  {
    // array based Queue, not in zyBook
    // empty queue where queue is implemented over a doubly linked list (the
default)
    Queue<Student> myQueue_1;
    test( myQueue_1 );

    // empty queue where queue is implemented over a fixed sized standard array
    Queue<Student, std::array<Student, 3>> myQueue_
    test( myQueue_3 );

    // Standard Queue usage with standard containers
    // default standard queue (uses std::deque as underlying container)
    std::queue<Student> myQueue_4;
    test( myQueue_4 );

    // standard queue with standard doubly linked list as underlying container
    std::queue<Student, std::list<Student>> myQueue_5;
    test( myQueue_5 );
  }

  catch (const std::exception & ex)
  {
    std::cerr << ex.what() << '\n';
  }
}
```

Client creates instances of queue specifying what underlying container to use

Queue class that we wrote

Array-based queue

Queue class from the STL

```
// A simple test driver to exercise the container.  Sense stacks and
queues have
// (nearly) the same interface, the same test driver is used for both.
The
// container tested is intentionally passed by value (makes a local copy).
template<class Container_Type>
void test( Container_Type myContainer )
{
  // A queue is an Abstract Data Type, usually implemented as a limited
  // interlace over some other data structure Things you can do to a
queue:
  myContainer.push( {"Tom"    } );
  myContainer.push( {"Aaron" } );
  myContainer.push( {"Brenda"} );
  myContainer.pop();
  myContainer.push( {"Katelyn"} );

  // Display the contents.  Stacks and queues do not allow traversal (you
can’t
  // see anything but the top (stack and queue) and both (queue only), so
to
  // display the contents we have to inspect each element at the top and
then
  // remove it until the container is
  while( !myContainer.empty() )
  {
    std::cout << myContainer.front();
    myContainer.pop();
  }
  std::cout << '\n’;
}
```

Client inserts and removes from the queue

Client views element at the top of the queue

main.cpp

Open file to see full implementation
(taken from our Implantation Examples on TITANium)

As the class designer - implementing a data structure

IMPLEMENTING A QUEUE WITH A :

DOUBLY LINKED LIST

FIXED SIZE ARRAY

Example- Implementing a queue with Doubly Linked List

Queue ADT takes two template arguments, the type of data to store, and the underlying container, usually defaulted to a recommended choice.

```
template<typename T, class UnderlyingContainer = DLinkedList<T>>
class Queue
{
  public:
    void        push( const T & element );
    T           pop();
    T &         top();    // peek() in zyBook
    bool        empty();  // isEmpty() in zyBook
    std::size_t size();   // getLength() in zyBook

  private:
    UnderlyingContainer collection;
};
```

Class designer creates a Queue ADT interface

The underlying container is a private attribute of the Stack ADT.

Queue.hpp

Open file to see full interface definition
(taken from our Implantation Examples on TITANium)

```
void push( const T & element )
{
  collection.prepend( element );      // inse
}
```

Error checking is performed, but delegated

```
T pop()
{
   // Note: zyBook returns the value popped, the C++ standard template library
   //       does not. popping an element from an empty queue error handling is
   //       handled by the underlying container
  auto element = collection.back();
  collection.removeBack();
  return element;
}
```

The real work is delegated to the underlying container, called collection in this example

```
T & top()
{
  // Note: viewing an element from an empty queue error handling is handled
  // by the underlying container
  return collection.back();
}
```

```
bool empty()
{
  return collection.empty();
}
```

```
std::size_t size()
{
  return collection.size();
}
```

Queue.hxx

Open file to see full implementation
(taken from our Implantation Examples on TITANium)

As the class designer - implementing a data structure

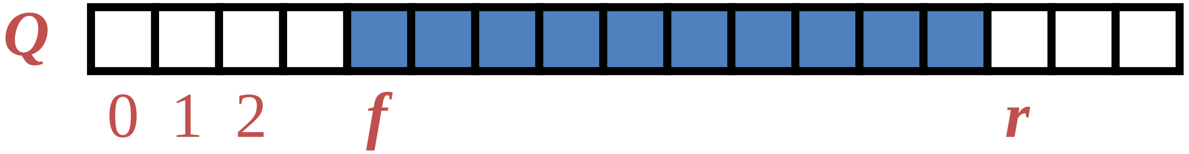
IMPLEMENTING A QUEUE WITH A :

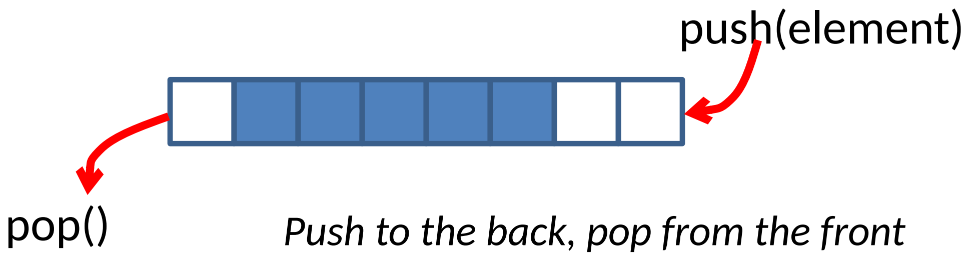
DOUBLY LINKED LIST

FIXED SIZE ARRAY

Array-based Queue – Attributes

- Use *two* integers to keep track of front and rear of the queue
  - f: index of the front element
  - r: index of the empty location where the next element will enter (the rear of the queue)
- Use *two* more to keep track of the size and capacity of the queue
  - n: number of elements in the queue
  - c: capacity of the fixed sized array



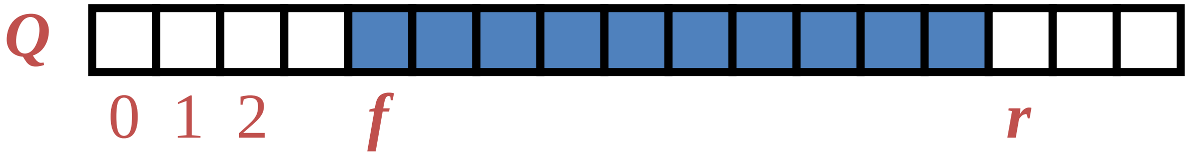


Push to the back, pop from the front

Array-based Queue – Size & Empty

- Use *n* to determine size and emptiness

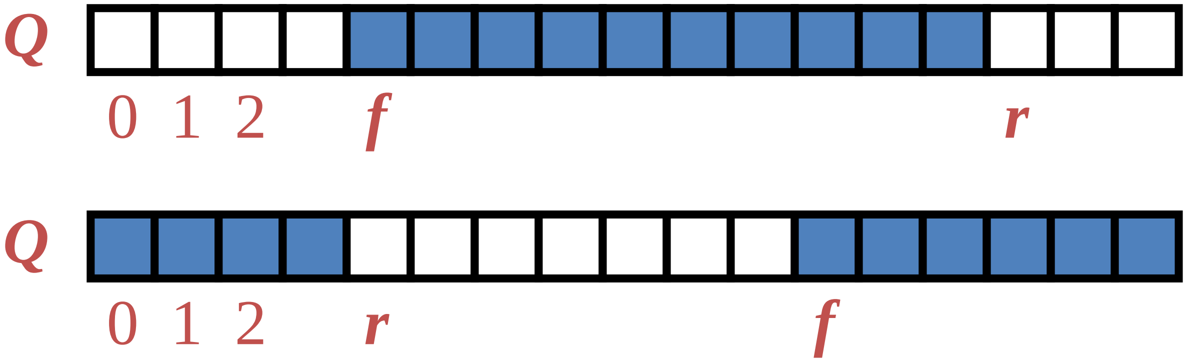
```
Algorithm size()
    return n

Algorithm empty()
    return (n == 0)
```



Array-based Queue – Push

- Operation Push() throws an exception if the array is full

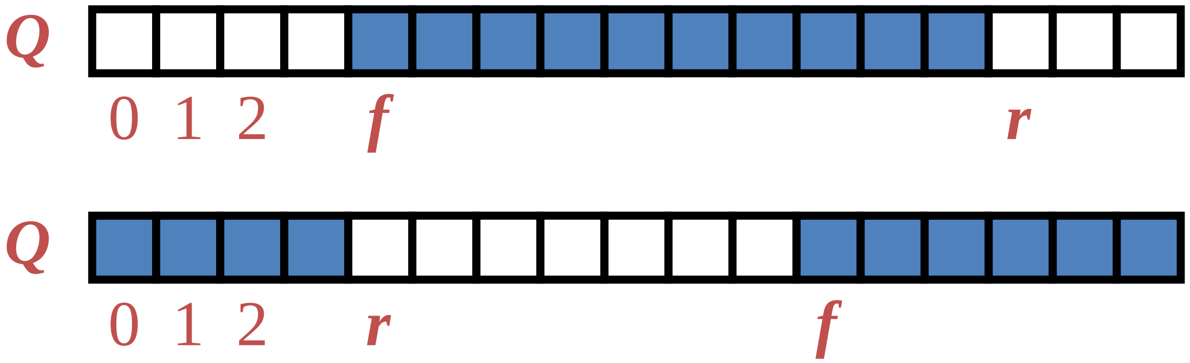
```
Algorithm Push(value)
    if $n == c$ then        // size == capacity
        throw QueueFull
    else
        $Q[r] \leftarrow value$
        $r \leftarrow (r + 1) \bmod N$
        $n \leftarrow n + 1$
```



Array-based Queue – Pop

- Operation Pop() throws an exception if the queue is empty
- This exception is specified in the queue ADT

```
Algorithm Pop()
    if IsEmpty() then
        throw QueueEmpty
    else
        $f \leftarrow (f + 1) \bmod N$
        $n \leftarrow n - 1$
```



Example - Implementing an Array-Based Queue

Note the fixed sized array as the underlying container

```
template<typename T, std::size_t CAPACITY>
class Queue<T, std::array<T, CAPACITY>>
{
  public:
    void        push( const T & element );
    T           pop();
    T &         top();       // peek() in zyBook
    bool        empty();     // isEmpty() in zyBook
    std::size_t size();      // getLength() in zyBook

  private:
    std::array<T, CAPACITY> collection;

    std::size_t _front = 0;  // index of the front element
    std::size_t _rear  = 0;  // index of an empty location where
                             // the next element will enter
    std::size_t _size  = 0;  // number of elements in the queue,
                             // 0 indicates an empty queue};
```

Class designer creates a Queue ADT interface

Note the addition of the front, rear, and size attributes. Capacity is a template constant. See *Using a Queue*

Open file to see full interface definition (taken from our Implantation Examples on TITANium)

Queue.hpp

Queue.hxx

Open file to see full implementation (taken from our Implantation Examples on TITANium)

```
void push( const T & element )
{
  if( _size >= CAPACITY )
     throw std::out_of_range( "ERROR:  Attempt to add to an already full queue of " +
                              std::to_string( CAPACITY ) + " elements." );
  collection[_rear] = element;
  _rear = ( _rear + 1 ) % CAPACITY;
  ++_size;
}

T pop()
{
  if( empty() )
     throw std::out_of_range( "ERROR:  Attempt to remove an element from an empty queue"

  auto temp = std::move( collection[_front] );
  _front = ( _front + 1 ) % CAPACITY;
  --_size;
  // Note, zyBook returns the value popped, the C++ standard template library does not.
  return temp;
}

T & top()
{
  if( empty() )
     throw std::out_of_range( "ERROR:  Attempt to view an element from an empty queue" )
  return collection[_front];
}

ool empty()
  return _size == 0; }

td::size_t size()
{  return _size; }
```

Insert, then increment the rear with modulo arithmetic

Error checking is explicitly performed, not delegated

Remove, then increment the front with modulo arithmetic

Element returned is at front of queue