CIT237
Chapter 19: Stacks and Queues

November 25, 2019

Reminders

- Quiz 7 will be held at the start of class on

  Wednesday, December 11.

  The material covered on Quiz 7 will be announced as the date gets closer.

- Project 3:

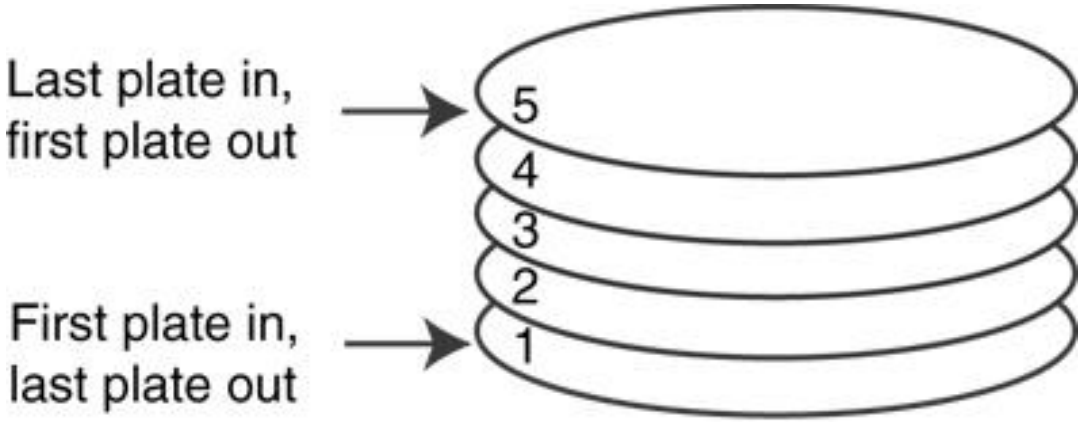
  Due date is December 2.

Introduction to the Stack ADT

- Stack: a LIFO (last in, first out) data structure
- Examples:
  - plates in a cafeteria
  - return addresses for function calls
- Implementation:
  - static: fixed size,
    implemented as array
    Example: IntStack class, pages 1168-1170
  - dynamic: variable size,
    implemented as a linked list
    Example: DynIntstack class, pages 1183-1185

A "Last-in, First-Out" Data Structure

- Last-in, First-out (LIFO):
  - The most recent item placed on the stack will be the next item to be removed.
- Think of a stack of plates in a cafeteria:



Stack Operations and Functions

- Operations:

  push: add a value onto the top of the stack

  pop: remove a value from the top of the stack

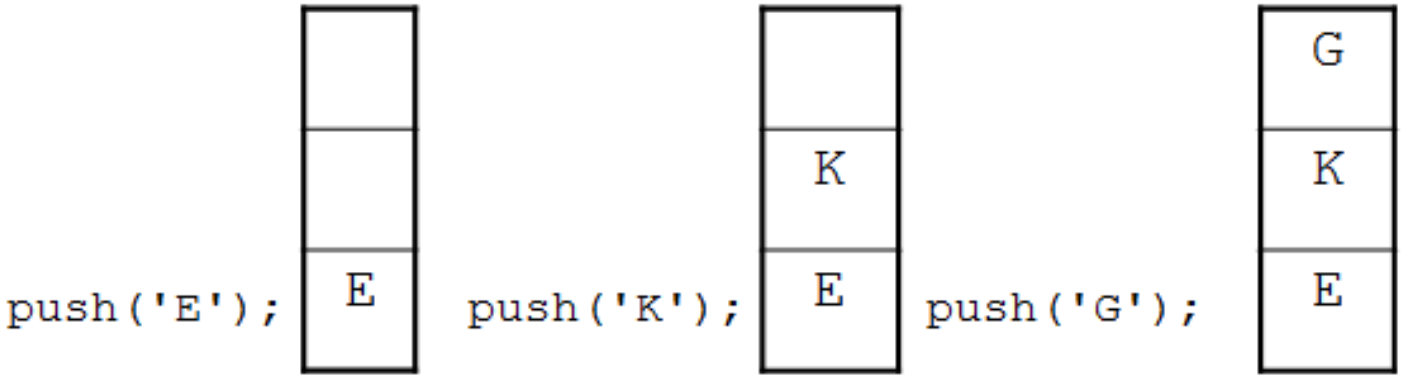
- Functions:

  `isFull`: `true` if the stack is currently full, *i.e.*, has no more space to hold additional elements

  `isEmpty`: `true` if the stack currently contains no elements

Stack Operations – Push Example

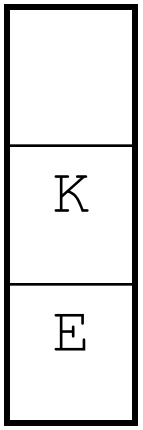
- A stack that can hold `char` values:



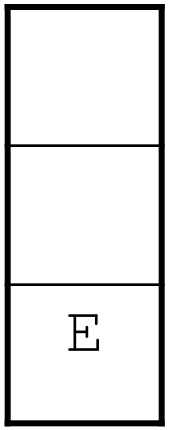
Stack Operations – Pop Example

- A stack that can hold `char` values:

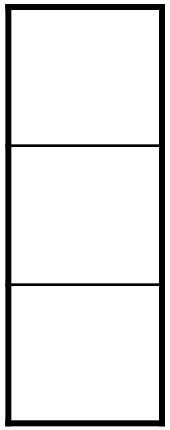
`pop();`
(remove `G`)



`pop();`
(remove `K`)



`pop();`
(remove `E`)



Dynamic Stacks

- Grow and shrink as necessary
- Can never be full as long as memory is available
- Implemented as a linked list

Implementing a Stack

- Programmers can write their own routines to implement stack functions
  - **`IntStack`** class: Textbook section 19.1
  - **`DynIntStack`** class: Textbook section 19.2

A Stack Template

- The DynamicStack class template supports creating a stack of *any* datatype.
- Available on Moodle:

  Ch19_DynamicStackTemplate_sample_code

Sample Code: Dynamic Int Stack

- Program 19-4 uses the **`DynIntStack`** class, which is provided in section 19.2.
- Also present on Moodle.

The STL `stack` container

- Stack template can be implemented as a `vector`, a linked list, or a `deque`
- Implements `push`, `pop`, and `empty` member functions
- Also implements other member functions:
  - `size`: number of elements on the stack
  - `top`: reference to element on top of the stack

Defining a `stack`

- Defining a stack of `chars`, named `cstack`, implemented using a `vector`:

```
stack< char, vector<char> > cstack;
```

- implemented using a list:

```
stack< char, list<char> > cstack;
```

- implemented using a `deque`:

```
stack< char > cstack;
```

- Spaces are required between consecutive `> >`, `< <` symbols (to avoid confusion with “>>” or “<<“ operators)

STL stack: Member Functions

empty(): returns **`true`** if the stack is empty

pop(): removes element from top of stack

push(**`x`**): pushes element **`x`** onto the stack

size(): number of elements in the stack

top(): returns a reference to the element at the top of the stack.

NOTE: the **pop()** function does not retrieve the value from the top of the stack, it only removes it. To retrieve the value, you must call the **top()** function first.

Program 19-6: STL stack Demo(1)

```
#include <iostream>
#include <string>
#include <stack>  // STL stack
using namespace std;

int main()  // (see Program 19-6, page 1194)
{
   const int MAX = 8;
   int count;      // Loop counter
   stack< int, vector<int> > iStack;
   for (count = 2; count < MAX; count += 2)
   {
      cout << "Pushing " << count << endl;
      iStack.push(count);
   }
```

Program 19-6: STL stack Demo(2)

```
   // Display the size of the stack.
   cout << "The size of the stack is ";
   cout << iStack.size() << endl;

   // Pop the values of the stack.
   for (count = 2; count < MAX; count += 2)
   {
      cout << "Popping " << iStack.top() << endl;
      iStack.pop();
   }
   system("pause");
   return 0;
} // (end main)
```

Output from Program 19-6

```
Pushing 2
Pushing 4
Pushing 6
The size of the stack is 3
Popping 6
Popping 4
Popping 2
```

Introduction to the Queue ADT

- Queue: a FIFO (first in, first out) data structure.
- Examples:
  - people in line at the theatre box office
  - print jobs sent to a printer
- Implementation:
  - static: fixed size, implemented as array
  - dynamic: variable size, implemented as linked list

Queue Locations and Operations

- Locations:

  rear: position where elements are added

  front: position from which elements are removed

- Operations:

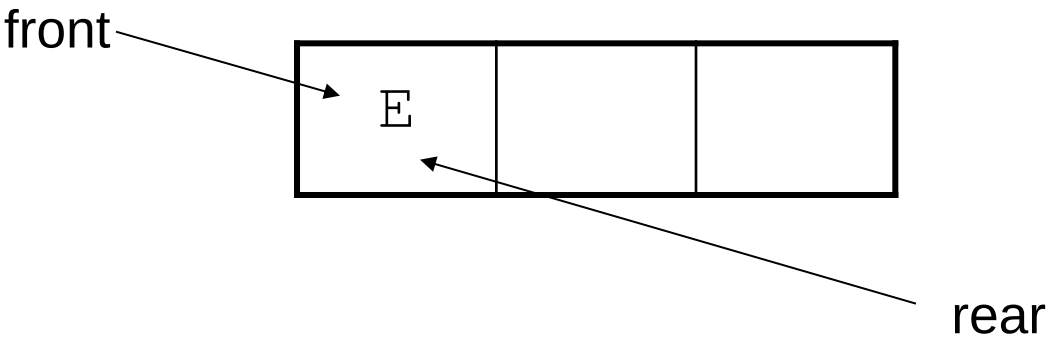
  enqueue: add an element to the rear of the queue

  dequeue: (pronounced “Dee-Q”) remove an element from the front of a queue

Queue Operations Example(1)

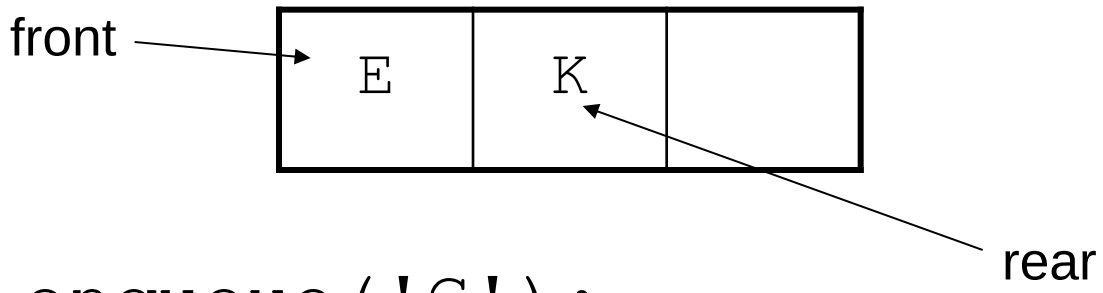
- A currently empty queue that can hold `char` values:

- `enqueue('E');`

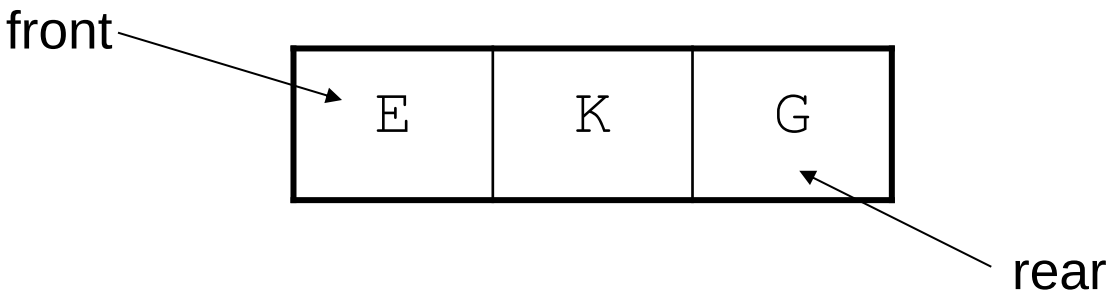


Queue Operations Example(2)

- `enqueue('K');`

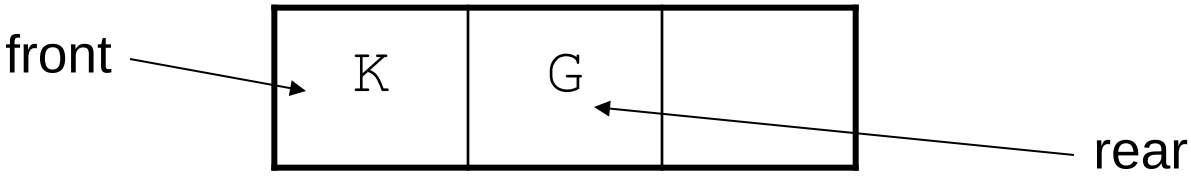


- `enqueue('G');`

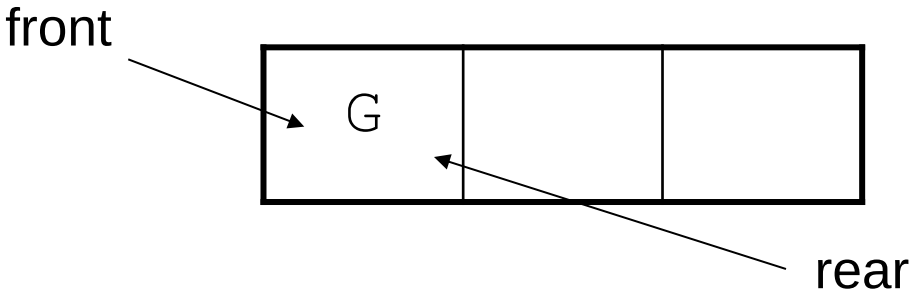


Queue Operations Example(3)

- `dequeue(); // remove E`



- `dequeue(); // remove K`

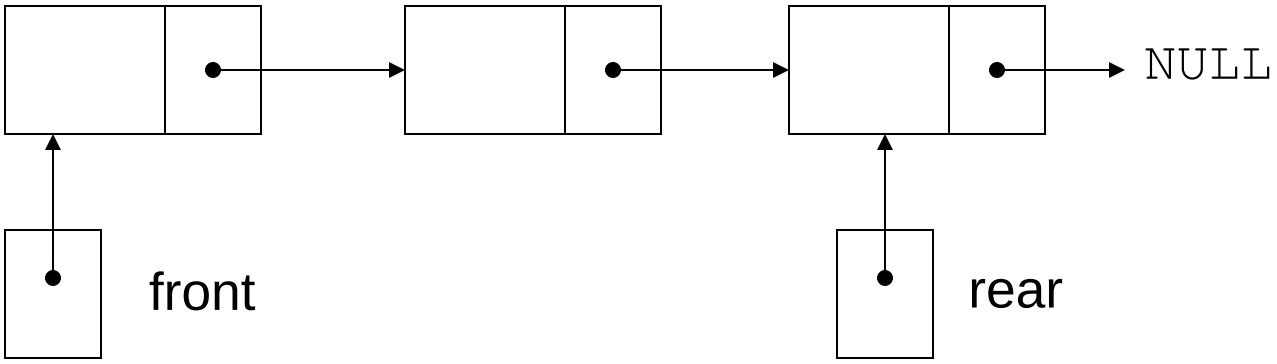


`dequeue` Issue, Solutions

- When removing an element from a queue, remaining elements must effectively shift to front.
- Solutions:
  - Let front index move as elements are removed (works as long as rear index is not at end of array)
  - Use above solution, and also let rear index "wrap around" to front of array, treating array as circular instead of linear (more complex enqueue, dequeue code)

Dynamic Queues

- Like a stack, a queue can be implemented using a linked list
- Allows dynamic sizing, avoids issue of shifting elements or wrapping indices



Implementing a Queue

- Programmers can write their own routines to implement queue operations
- See the **`DynIntQue`** class in the book for an example of a dynamic queue
- Can also use the implementation of queue and dequeue available in the STL

A QueueTemplate

- The DynamicQueue class template supports creating a stack of *any* datatype.
- Available on Moodle:

  Ch19_DynamicQueueTemplate_sample_code

The STL `deque` and `queue` Containers

- **`deque`**: a double-ended queue (pronounced "Deck"). Has member functions to enqueue (**`push_back`**) and dequeue (**`pop_front`**)
- **`queue`**: container ADT that can be used to provide a queue implemented using a **`vector`**, **`list`**, or **`deque`**.
- The **`queue`** container has member functions to enqueue (**`push`**) and dequeue (**`pop`**).

Defining a `queue`

- Defining a queue of **`char`**s, named **`cQueue`**, implemented using a deque:

```
deque<char> cQueue;
```

- implemented using a queue:

```
queue<char> cQueue;
```

- implemented using a `list`:

```
queue< char, list<char> > cQueue;
```

- Spaces are required between consecutive > and < symbols: **< <** or **> >**