Stacks and Queues

Lecture 11

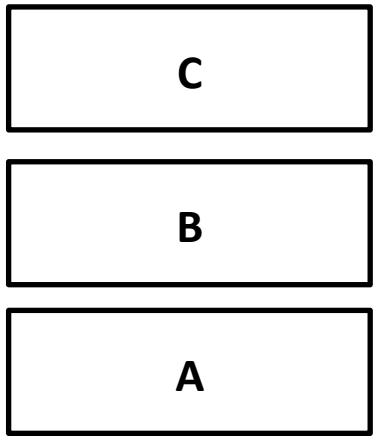
More Data Structures

- In this lecture we will use a linked list to implement two **abstract data types** (ADT)
- An ADT provides the interface, or *what* a data structure does
- We can then use code (e.g. other data structures) to implement the interface (i.e. the *how*)

Stacks

- A collection of items
- Supports two primary operations
  - To **push** an item on [the top]
  - To **pop** an item off [the top]
- Result: **LIFO** (last in, first out)
- Applications
  - Function stack
  - Parsing languages
  - Backtracing (e.g. maze)

```
push( 'A' );
push( 'B' );
push( 'C' );
pop(); // C
pop(); // B
pop(); // A
```



Interface: Stack of Characters

```
class CharStack
{
public:
     // Initializes an empty stack
     CharStack();

     // Determines if the stack is empty
     //
     // Returns: true if the stack is empty
     bool isEmpty();

     // Pushes a character onto the stack
     //
     // PostConditions: the character c is on
     //                 the top of the stack
     void push(char c);

     // Pops the top of the stack
     //
     // PostConditions: the top of the stack is popped
     // Returns: the character at the top of the stack,
     //          null character if the stack was empty
     char pop();
};
```

Example: Reverse

```
#include <iostream>
#include <string>
#include "CharStack.h"
using namespace std;

int main()
{
     CharStack s;

     string word;
     cin >> word;

     for ( int i=0; i<word.length(); i++ )
          s.push( word[i] );

     while ( !s.isEmpty() )
          cout << s.pop();
     cout << endl;

     return 0;
}
```

```
> ./reverse
ward
draw
```

Implementing a Stack

- It turns out that a linked list is very useful to implement the stack interface

- For the next set of slides, we will implement the core member functions of a stack assuming we have a functioning singly linked list

Implementation

```
class CharStack
{
public:
	// Initializes an empty stack
	CharStack();

	// Determines if the stack is empty
	//
	// Returns: true if the stack is empty
	bool isEmpty();

	// Pushes a character onto the stack
	//
	// PostConditions: the character c is on
	//                 the top of the stack
	void push(char c);

	// Pops the top of the stack
	//
	// PostConditions: the top of the stack is popped
	// Returns: the character at the top of the stack,
	//          null character if the stack was empty
	char pop();

private:
	CharSLL l;
};
```

```
class CharSLL
{
public:
	// Initializes an empty list
	CharSLL();

	// Releases all list memory
	~CharSLL();

	// Determines if the list is empty
	//
	// Returns: true if the list is empty
	bool isEmpty();

	// Adds a character to the front of the list
	//
	// PostConditions: c is at the head of the list
	void addToFront(char c);

	// Removes a character from the front of the list
	//
	// PostConditions: the character at the front of
	//                 the list has been removed
	// Returns: character at the front of the list,
	//          null character if the list was empty
	char removeFromFront();
};
```

Exercise

Implement the **`isEmpty`** member function of the **`CharStack`** class.

Answer

```
bool isEmpty()
{
  return l.isEmpty();
}
```

Exercise

Implement the **`push`** member function of the **`CharStack`** class.

Answer

```
void push(char c)
{
  l.addToFront( c );
}
```

Exercise

Implement the **`pop`** member function of the **`CharStack`** class.

Answer

```
char pop()
{
  return l.removeFromFront();
}
```

Benefits of OOP

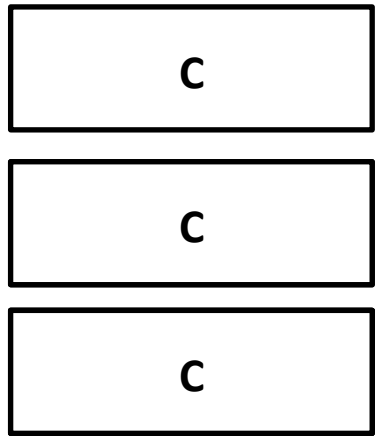
- By implementing a general linked list class, implementing a general stack class is trivial
  - In a moment we will see that a linked list can also support another data structure – a queue!

- By *encapsulating* the code into classes, we can develop general, tested code and then use these as building blocks for more complex systems

Queues

- A collection of items
- Supports two primary operations
  - To **enqueue** an item
  - To **dequeue** an item
- Result: **FIFO** (first in, first out)
- Applications
  - Prioritization
  - Parallelization

```
enqueue( 'A' );
enqueue( 'B' );
enqueue( 'C' );
dequeue(); // A
dequeue(); // B
dequeue(); // C
```



Interface: Queue of Characters

```
class CharQueue
{
public:
     // Initializes an empty queue
     CharQueue();

     // Determines if the queue is empty
     //
     // Returns: true if the queue is empty
     bool isEmpty();

     // Enqueue's a character
     //
     // PostConditions: the character c is at
     //                 the back of the queue
     void enqueue(char c);

     // Dequeue's a character
     //
     // PostConditions: the front of the queue is removed
     // Returns: the character at the front of the queue,
     //          null character if the queue was empty
     char dequeue();
};
```

Implementing a Queue

- It turns out that a doubly linked list is very useful to implement the queue interface
  - Need to be able to add to the back!

- For the next set of slides, we will implement the core member functions of a queue assuming we have a functioning doubly linked list

Implementation

```
class CharQueue
{
public:
        // Initializes an empty queue
        CharQueue();

        // Determines if the queue is empty
        //
        // Returns: true if the queue is empty
        bool isEmpty();

        // Enqueue's a character
        //
        // PostConditions: the character c is at
        //                 the back of the queue
        void enqueue(char c);

        // Dequeue's a character
        //
        // PostConditions: the front of the queue is removed
        // Returns: the character at the front of the queue,
        //          null character if the queue was empty
        char dequeue();

private:
        CharDLL l;
};
```

```
class CharDLL
{
public:
        // Initializes an empty list
        CharSLL();

        // Releases all list memory
        ~CharSLL();

        // Determines if the list is empty
        //
        // Returns: true if the list is empty
        bool isEmpty();

        // Adds a character to the front of the list
        //
        // PostConditions: c is at the head of the list
        void addToFront(char c);

        // Adds a character to the tail of the list
        //
        // PostConditions: c is at the tail of the list
        void addToBack(char c);

        // Removes a character from the front of the list
        //
        // PostConditions: the character at the front of
        //                 the list has been removed
        // Returns: character at the front of the list,
        //          null character if the list was empty
        char removeFromFront();
};
```

Exercise

Implement the **`isEmpty`** member function of the **`CharQueue`** class.

Answer

```
bool isEmpty()
{
  return l.isEmpty();
}
```

Exercise

Implement the **`enqueue`** member function of the **`CharQueue`** class.

Answer

```
void enqueue(char c)
{
  l.addToBack( c );
}
```

Exercise

Implement the **`dequeue`** member function of the **`CharQueue`** class.

Answer

```
char dequeue()
{
  return l.removeFromFront();
}
```

Exercise

Implement the **`CharQueue`** class using the generic **`LinkedList`** class API from HW6.

Answer

CharQueue.h

```
class CharQueue
{
public:
	// Determines if the queue is empty
	//
	// Returns: true if the queue is empty
	bool isEmpty();

	// Enqueue's a character
	//
	// PostConditions: the character c is at
	//                 the back of the queue
	void enqueue(char c);

	// Dequeue's a character
	//
	// PostConditions: the front of the queueu is removed
	// Returns: the character at the front of the queue,
	//          null character if the queue was empty
	char dequeue();

private:
	LinkedList<char> l;
};
```

CharQueue.cpp

```
bool isEmpty()
{
	return l.empty();
}

void enqueue(char c)
{
	l.addToBack( c );
}

char dequeue()
{
	char return_char = '\0';

	if ( !isEmpty() )
	{
		Node<char>* n = l.first();
		return_char = n->getData();
		l.remove( n );
	}

	return return_char;
}
```

Wrap Up

- An abstract data type (ADT) provides the interface for a data structure
- A stack is a LIFO collection, which supports pushing to, and popping from, the front
- A queue is a FIFO collection, which supports enqueue to add to the back, and dequeue to remove from the front
- Taking advantage of OOP, you can easily implement stacks and queues using linked lists
  - Though other implementations are possible (e.g. arrays)