Stacks and Queues

Stacks and queues are closely related to linked lists, so we can go through this material fairly quickly. We'll see that we can pretty easily use a linked list to implement both. However, we don't have to use a linked list – we could use an array or other structures, but a linked list is a natural data structure for both.

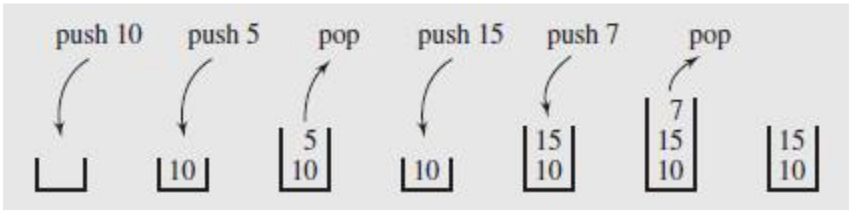
Stack

This is the same type of stack we discuss in 211 or 248. The classic analogy is a stack of trays in a cafeteria; trays are removed (popped) from the top and placed back on the top (pushed). Since the first tray in the pile is the last one to be removed it is known as a LIFO or Last-In-First-Out structure.

The operations we can perform on a stack are:

- clear( ): clears the stack
- isEmpty( ): determines if the stack is empty
- push(el): pushes the data item el onto the top of the stack
- pop( ): removes the top element from the stack
- topEl( ): returns the value of the top element of the stack without removing it

A series of pushes and pops is shown below:



Stacks are especially useful in situations where data has to be stored and processed in reverse order. There are many applications of this:

- Evaluation expressions and parsing syntax
- Balancing delimiters in code
- Converting numbers between bases
- Backtracking algorithms

The book has an example of processing delimiters and adding large numbers.

Here is an implementation of a stack using the STL list class:

File: Stack.h

```
#pragma once
#include <list>
using std::list;

template<class T>
class Stack
```

```
{
public:
	Stack();
	~Stack();
	void clear();
	bool isEmpty();
	T topEl();
	T pop();
	void push(const T& el);
private:
	list<T> lst;
};
```

File: Stack.cpp

```
#include "Stack.h"

template<class T>
Stack<T>::Stack()
{
	lst.clear();
}

template<class T>
Stack<T>::~Stack()
{
}

template<class T>
void Stack<T>::clear()
{
	lst.clear();
}

template<class T>
bool Stack<T>::isEmpty()
{
	return (lst.empty());
}

template<class T>
T Stack<T>::topEl()
{
	if (lst.empty())
	{
		T defaultItem;
		return defaultItem;
	}
	return (lst.front());
}

template<class T>
T Stack<T>::pop()
{
	T item;
	if (!lst.empty())
	{
		item = lst.front();
		lst.pop_front();
	}
	return item;
}

template<class T>
void Stack<T>::push(const T& el)
{
	lst.push_front(el);
}
```

File: source.cpp

```
#include <iostream>
#include "Stack.cpp";
using namespace std;

int main()
{
	Stack<int> stck;

	stck.push(3);
	stck.push(5);
	stck.push(100);
	while (!stck.isEmpty())
	{
		int i = stck.pop();
		cout << i << endl;
	}
	system("pause");
}
```

Let's do an example of writing a calculator that can input a calculation in **reverse polish notation**. This was once popular on older HP calculators. It is much easier to calculate than infix notation on today's standard calculators.

In reverse polish the operator comes after the operands. To add 3 and 4 we write: 3 4 +

It has the advantage that there is no ambiguity; we don't even need parentheses! When we get an operator we must have two previous values to operate on. We can implement this on a stack, where we push each operand on a stack and when we get to an operator we pop off the previous two operands, apply the operator, and push the result back on the stack.

Example computing (3+4) * (2+2) / 2 = 14

In reverse polish this could be calculated as: 3 4 + 2 2 + * 2 /

Here is code in main:

```
#include <iostream>
#include <string>
#include "Stack.cpp";
using namespace std;

void processOperator(Stack<string> &stck)
{
	string item = stck.pop();
	if ((item == "+") || (item == "*") || (item == "/"))
	{
		int num1 = stoi(stck.pop());
		int num2 = stoi(stck.pop());
		string s;
		if (item == "+")
			s = to_string(num1 + num2);
		if (item == "*")
			s = to_string(num1 * num2);
		if (item == "/")
			s = to_string(num2 / num1);
		stck.push(s);
	}
}

int main()
{
	Stack<string> stck;

	stck.push("3");
	stck.push("4");
	stck.push("+");
	processOperator(stck);

	stck.push("2");
	stck.push("2");
	stck.push("+");
	processOperator(stck);

	stck.push("*");
	processOperator(stck);

	stck.push("2");
	stck.push("/");
	processOperator(stck);

	cout << stck.topEl() << endl;
	system("pause");
}
```

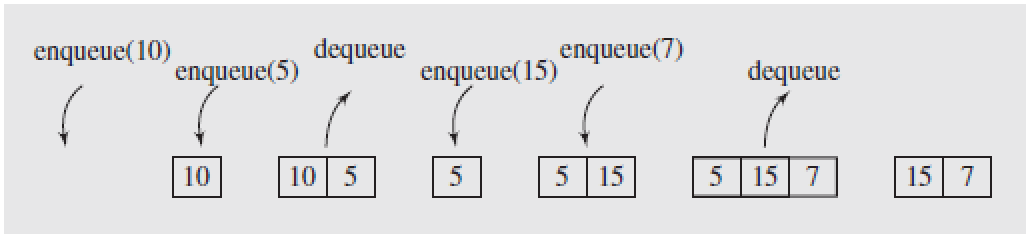
Queues

A queue, like a stack, is a restricted access linear data structure. Unlike a stack, both ends are involved, with additions restricted to one end (the rear) and deletions to the other (the front). Since an item added to the queue must migrate from the rear to the front before it is removed, items are removed in the order they are added. For this reason, queues are also known as **first-in first-out (FIFO)** structures.

Queue functions:

- clear( ): clears the queue
- isEmpty( ): determines if the queue is empty
- enqueue(el): adds the data item el to the end of the queue
- dequeue( ): removes the element from the front of the queue
- firstEl( ): returns the value of the first element of the queue without removing it

A sequence of enqueues and dequeues is shown below:



One method of implementing a queue is in an array, where we keep track of the first and last elements as if they circle around:

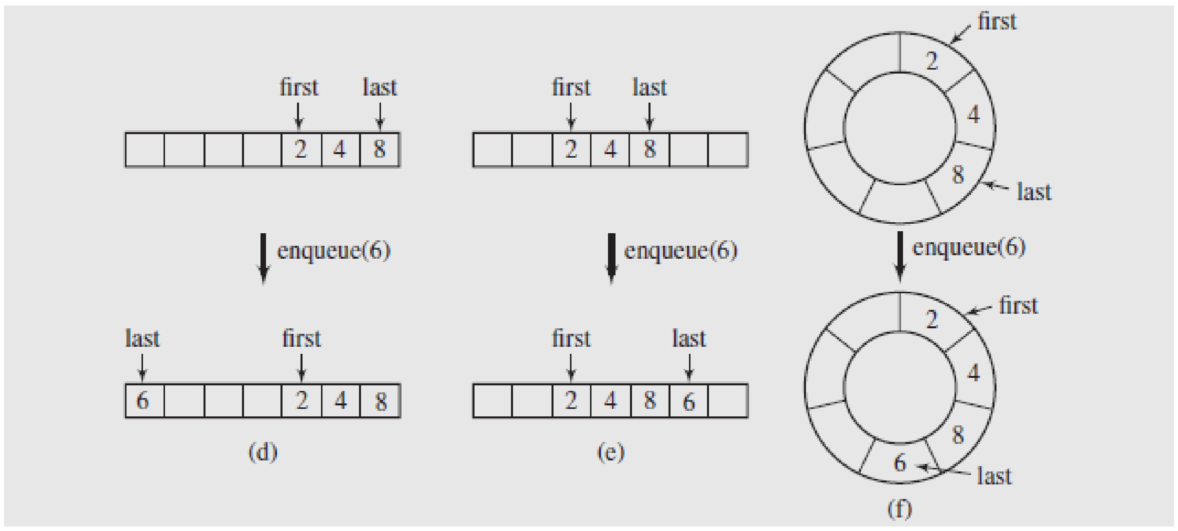


Fig. 4.8 (continued) (f) Enqueuing number 6 to a queue storing 2, 4, and 8; (d–e) the same queue seen as a one-dimensional array with the last element (d) at the end of the array and (e) in the middle

A more flexible implementation uses a linked list. We can pretty easily modify our Stack implementation to turn it into a Queue:

File: Queue.h

```
#pragma once
#include <list>
using std::list;

template<class T>
```

```
class Queue
{
public:
	Queue();
	~Queue();
	void clear();
	bool isEmpty();
	T front();
	T dequeue();
	void enqueue(const T& el);
private:
	list<T> lst;
};
```

File: Queue.cpp

```
#include "Queue.h"

template<class T>
Queue<T>::Queue()
{
	lst.clear();
}

template<class T>
Queue<T>::~Queue()
{
}

template<class T>
void Queue<T>::clear()
{
	lst.clear();
}

template<class T>
bool Queue<T>::isEmpty()
{
	return (lst.empty());
}

template<class T>
T Queue<T>::front()
{
	if (lst.empty())
	{
		T defaultItem;
		return defaultItem;
	}
	return (lst.front());
}

template<class T>
T Queue<T>::dequeue()
{
	T item;
	if (!lst.empty())
	{
```

```
              item = lst.front();
              lst.pop_front();
       }
       return item;
}

template<class T>
void Queue<T>::enqueue(const T& el)
{
       lst.push_back(el);
}
```

In main:

```
       Queue<string> q;

       q.enqueue("Bill");
       q.enqueue("Aaron");
       q.enqueue("Zorro");
       q.enqueue("Gobu");
       while (!q.isEmpty())
       {
              string s = q.dequeue();
              cout << s << endl;
       }
```

Queues are used in a wide variety of applications, especially in studies of service simulations.

This has been analyzed to such an extent that a very advanced body of mathematical theory, called queuing theory, has been developed to deal with it.

The Standard Template Library

You will probably not be surprised to learn that the STL includes a stack and queue class.

The stack actually uses a deque as its underlying implementation. A deque (doubly-ended queue) is like a list except its implementation is like a vector (array-based instead of list-based, which allows random access).

Functions in the stack class are:

- empty() – Returns whether the stack is empty
- size() – Returns the size of the stack
- top() – Returns a reference to the top most element of the stack
- push(g) – Adds the element ‘g’ at the top of the stack
- pop() – Deletes the top most element of the stack

Here is a short sample:

```
	#include<stack>


	stack<int> st;
	st.push(10);
	st.push(20);
	st.push(30);
	while (!st.empty())
	{
		cout << st.top() << endl;
		st.pop();
	}
```

Similarly, there is a queue class (you can look it up online with any reputable C++ reference).

Priority Queues

In some circumstances, the normal FIFO operation of a queue may need to be overridden. This may occur due to priorities that are associated the elements of the queue that affect the order of processing. In cases such as these, a priority queue is used, where the elements are removed based on priority and position. For example, priority queues are used in operating systems to assign priorities to running processes. The priorities may change based on what the user wants to get done at the moment.

The difficulty in implementing such a structure is trying to accommodate the priorities while still maintaining efficient enqueuing and dequeuing. Elements typically arrive randomly, so their order typically reflects no specific priority.

There are many ways to represent priority queues:

- Linked lists that insert new elements based on priority
- A short ordered list and a larger unordered list; can be efficient if we don't need to go to the larger unordered list very often
- Structures like heaps, which we'll see later

The STL includes a priority queue class. To maintain priority you have to override the < operator so it knows what has priority over another object. It is implemented using a heap in an array (we'll cover a heap later).

Here is an example of a priority queue of a Person class.

File: Person.h

```
#pragma once
#include <string>
using std::string;


class Person
{
public:
	Person();
	Person(string name, int priority);
	~Person();
```

```cpp
	string name;
	int priority;
};

// Prototype
bool operator<(const Person &lhs, const Person &rhs);
```

File: Person.cpp

```cpp
#include "Person.h"


Person::Person() : name(""), priority(100)
{
}

Person::Person(string name, int priority) : name(name), priority(priority)
{
}


Person::~Person()
{
}

bool operator<(const Person &lhs, const Person &rhs)
{
	return lhs.priority < rhs.priority;
}
```

Main.cpp:

```cpp
#include <iostream>
#include <string>
#include <queue>
using namespace std;

int main()
{
	priority_queue<Person> pq;
	pq.push(Person("Intern", 2));
	pq.push(Person("Developer", 8));
	pq.push(Person("CEO", 1000));
	pq.push(Person("Waterboy", 3));
	pq.push(Person("Manager", 12));
	pq.push(Person("Vice President", 100));
	while (!pq.empty())
	{
		cout << pq.top().name << " " << pq.top().priority << endl;
		pq.pop();
	}
	system("pause");
}
```