Data Structures

(Feodor F. Dragan)
Department of Mathematics and Computer Science
Kent State University

Fall, 2000

CHAPTER 7
The Standard Template Library

Contents

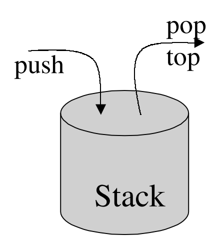
Introduction

- Many algorithms require proper DS
  - How to create a DS,
  - How to insert data ➔ arrays vs. stack vs. queue
  - How to delete data ➔ arrays vs. stack vs. queue
  - How to access data ➔ arrays vs. stack vs. queue
- Data Structure = Data representation + Operations
- Recently adopted C++ Standard allow supporting for Standard Template Library (STL)
- STL contains DS + Algorithms guaranteed available on compilers

Stacks and Queues

Stacks

- Stack is a list in which all insertions and deletions are made at one end, called the *top*.
- When we add an item to a stack, we *push* it into the stack and when we remove an item, we *pop* it from the stack.
- Sometimes the term *push-down list* is used synonymously with stack.
- Operations: Given a stack ***s*** and an item ***i***, the following operations can be performed.
  - ***push(s, i)*** ➔ only if ***s*** is not full. O(1)
  - ***pop(s)*** ➔ only if ***s*** is not empty. O(1)
  - ***empty(s)*** O(1)
  - ***top(s)*** ➔ only if ***s*** is not empty. O(1)



Applications of stacks.

• Subprogram implementation: Suppose A call B, B call C, C call D. Then the subprogram which is recently called will finish first. That is why we call a stack LIFO list.

• Reversing a list

• Checking the correctness of nested parenthesis in an expression (vs. parenthesis count).

A stack can be used to implement this procedure by pushing a "(" into the stack and popping it when we encounter a ")". At the end the stack should remain empty.

m + ((a + b) – (c + e * (f + g)))

0 0 1 2 2 2 2 1 1 2 2 2 2 2 2 3 3 3 3 2 1 0

• This procedure can be generalized to handle "(", "[", "{".

m + { [a + b } – (c + e * (f + g))]

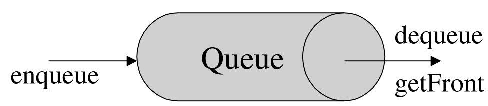
0 0 1 2 2 2 2 1 1 2 2 2 2 2 2 3 3 3 3 2 1 0

*

```
valid = true;
s = the empty stack;
while ( not the end of expression) {
        read the next symbol (symb);
        if (symb == '(' || symb == '[' || symb == '{' )
                push(s, symb);
        if (symb == ')' || symb == ']' || symb == '}' )
                if (empty(s)) valid = false;
                else { i = pop(s);
                        if (i is not the matching opener of symb)
                                valid = false;
                     }
        }
if (!empty(s)) valid = false;
if (valid) cout << "the string is valid";
else cout << "the string is not valid";
```

Queues

- A *queue* is an ordered list in which item may be deleted at one end (*front*) and inserted at the other end (*rear*).
- A queue is called FIFO list.
- Operations on Queue
  - ***insert(Q, x)*** (enqueue) O(1)
  - ***remove(Q)*** (dequeue) O(1)
  - ***x = getFront(Q)*** O(1)
  - ***empty(Q)*** O(1)



Basics of STL: Containers and Iterators

- An *iterator* is an object that is used to traverse a collection of objects
- In STL such a collection is called a *container*
- An iterator maintains a notion of current position in the container
- It provides basic operations

Container

- Represents a group of objects, known as its elements
- STL operations:
  - ***bool empty() const*** returns *true* if the container is empty, else *false*
  - ***iterator begin() const*** returns an iterator that can be used to begin traversing
  - ***iterator end() const*** returns an iterator that represents the *end marker*, a position one past the last element in the container
  - ***int size() const*** returns the number of elements in the container

The Iterator

- An object that allows **access to** and **iteration through** all elements in the container
- Basic operations:
  - *itr++* advances the *iterator* (*++itr*)
  - **itr* returns a reference to the object stored at *itr*'s location
  - *itr1==itr2* returns *true* if *itr1* and *itr2* point to the same location, otherwise *false*
- Bidirectional iterator
  - Forward *iterator* via *operator++*
  - Backward *iterator* via *operator--*
- Random access *iterator* : *itr+100*

Example: Printing the contents of any container

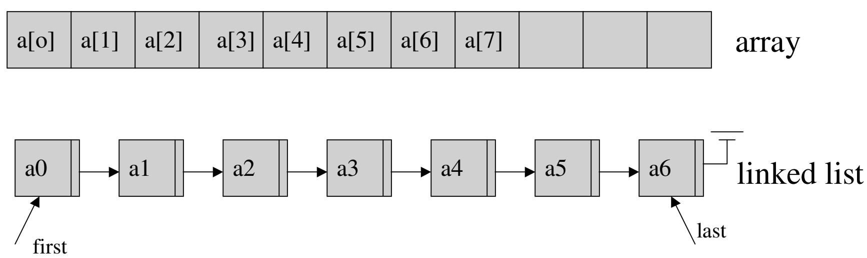
```
// print the contents of container c
template <class Container>
void printCollection( const Container& c )
{
        typename Container::const_iterator itr;
        for ( itr = c.begin(); itr != c.end(); ++itr )
        cout << *itr << '\n';
}
```

Sequences and Linked Lists

- *Sequences* vs *stacks* and *queues* (allow insertions and deletions in the middle of the container)
- Access is performed by *iterator* class
- Representation

Sequences → vectors, arrays

Sequences → linked lists (class <list>)

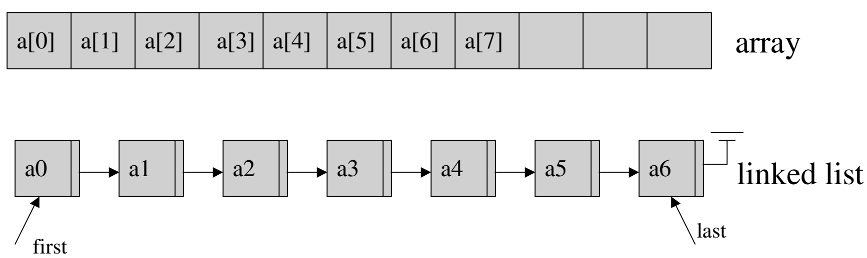


Operations on sequences include

- ***void push_back(const Object & element)*** appends an element at the end
- ***void push_front(const Object & element)*** prepends an element to the front (not for vectors)
- ***Object & front( )*** returns the first element
- ***Object & back( )*** returns the last element
- ***void pop_front( )*** removes the first element (not for vector)
- ***void pop_back( )*** removes the last element
- ***iterator insert(iterator pos, const Object & obj)*** inserts ***obj*** prior to the element in the position referred to by ***pos***, returns the position of the newly inserted element (time for list vs vector)
- ***iterator erase(iterator pos)*** removes the object at the position referred to by ***pos***, returns the position of the element that followed ***pos*** prior to the call to ***erase*** (time for list vs vector)

Linked lists vs vectors

- Random access iterators are available only for vectors
- Insertion and removal from the middle of a container is efficiently supported for lists only
- Linked lists are used to avoid large amount of data movement



Sets

- The ***set*** is an ordered container that allows no duplicates (multiset allows duplicate)
- In additional to the usual operations on containers *begin*, *end*, *size*, *empty*, the ***set*** provides the following operations

pair<iterator, bool> insert(const Object & element)

adds ***element*** to the set if it is not already there

bool component of the return is *true* if the set did not already contain ***element***; otherwise, *false*

the ***iterator*** component of the return is the location of ***element*** in the set

iterator find(const Object & element) const

returns an ***iterator*** containing the location of ***element*** in the set, or ***end( )*** if ***element*** is not in the set

int erase(const Object & element)

removes ***element*** from the set if it is present; returns the number of elements removed (either 0 or 1)

- using the *binary search tree (Chapter 19)* and its variants, the ***find***, ***erase*** and ***insert*** can be implemented to run in *O(logN)* time (direct implementations give *O(N)* time bound)

Illustration of ***set***

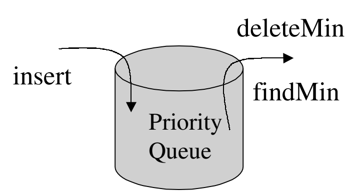
```
#include <iostream>
#include <set>
#include <functional>
#include <string>
using namespace std;

int main( )
{
        set<string, greater<string> >  s; // use reverse order
        s.insert("joe" );
        s.insert("bob" );
        printCollection( s ): // previous lecture
        return 0;
}
```


Priority Queues

- The ***priority queue*** supports access to the minimum item only
- Useful for multi-user OS environment
- Cannot be implemented be ***set, queues*** allows duplicates
- We can implement it as ***multiset,*** but ***multiset*** is overkill, it provides more operations than needed
- the ***priority queue*** supports the following operations

  - ***insert(PQ, x)***
  - ***deleteMin(PQ)***
  - ***findMin(PQ)***
  - ***empty(PQ)***



- using the *binary heap (Chapter 21)*, the ***findMin***, ***deleteMin*** and ***insert*** can be implemented to run in *O(logN)* worst case time (***insert*** in constant average time)

The STL Priority Queue

- uses ***push***, ***top***, and ***pop*** and accesses the maximum, rather than the minimum, item
- Operations from STL

template <class Object, class Container, class Less>
class priority_queue

requires the specification of the type of object being stored, the type of container storing it, and the class that defines the function object for comparison

void push(const Object & element)

adds element to the container

const Object & top( ) const

returns the largest item in the container

void pop( )

removes the largest item in the container

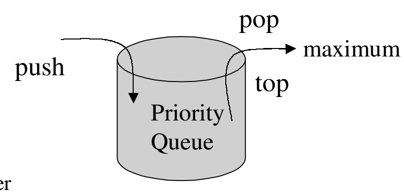


Illustration of ***priority queue***

```
#include <iostream>
#include <vector>
#include <queue>
#include <functional>
#include <string>
using namespace std;
// Empty the priority queue and print its contents.
template <class PriorityQueue>
void dumpContents( const string & msg, PriorityQueue & pq )
{       cout << msg << ":" << endl;
        while( !pq.empty( ) )
        {       cout << pq.top( ) << endl;
                pq.pop( );
        }
}
// Do some inserts and removes (done in dumpContents).
int main( )
{       priority_queue<int,vector<int>,greater<int>> minPQ;
        priority_queue<int,vector<int>,less <int>    > maxPQ;
        minPQ.push( 4 ); minPQ.push( 3 ); minPQ.push( 5 );
        maxPQ.push( 4 ); maxPQ.push( 3 ); maxPQ.push( 5 );
        dumpContents( "minPQ", minPQ );      // 3 4 5
        dumpContents( "maxPQ", maxPQ );      // 5 4 3
        return 0;
}
```

Data Structure	Access	Comments
Stack	Most resent only, ***pop***, *O(1)*	Very very fast
Queue	Least resent only, ***dequeue***, *O(1)*	Very very fast
Linked list	Any item	*O(N)*
Ordered set	Any item *O(logN)*	by binary search tree
Priority queue	***findMin***, *O(1)* ***deleteMin***, *O(logN)*	***insert*** is *O(1)* in average, *O(logN) worst case*

Homework

1. Problems 7.9, 7.10 page 260 of the text
2. Problem 7.13(a,b), page 261 of the text
3. Problem 7.15(a,b,c), page 261-262