CS472 Module 1 Part B - Fundamental Data Structures

Athens State University

2013-11-18 Mon

Outline

Contents

1 Algorithm vs. data structure

Mathematical Foundation

Set A collection of elements that, at minimum, meets the following axioms

- **Existence** There is a set with no elements
- **Extensionality** For every element of a set X is in the set Y, and every element of a set Y is the set X, then $X = Y$

Mathematical Foundation

Function Given two sets A and B, a *function* is a mapping s.t. for each a in A, there is exactly one b in B in the domain of f.

Important terms from discrete math: sequence, injection, surjection, bijection, permutation, inverse

sequence

inverse $f(x) = y \to f^{-1}(y) = x$

injection An injective (or one-to-one) function f is a function f s.t. f^{-1} is also a function.

surjection A surjective (or onto) function f is a function s.t. $\forall y \in Y$ there exists some $x \in X$ where $f(x) = y$.

bijection A bijective function f is both an injection and surjection.

permutation

Moving the foundation to the computer

- We have the sets Z, Q, and R for the integers, rational, and real numbers
- The primitive data types on the computer are physical simulations of these sets: *Integer*, *Float*, *Double*
- Unlike the mathematician, we have to concern ourselves with both values and the operations on those values

What do we mean when say that the mathematician isn't concerned with values? In discrete math, we define functions in terms of set elements. This means that we're thinking in terms of existence rather than construction. The computer scientist has to be concerned with how things get constructed. This means we need to extend how we have to reason about data.

Abstract Data Type

Abstract data type (ADT) A set of sets $< V, O >$ that define a set V of values that make up the data type and a set of functions O called operations that define the *interface* used to manipulate the values of V.

Note the key point: the only way that one can use the ADT is through the operations in the interface

Algorithms operate upon ADTs

ADTs and OOP

- Recall that when one directs the system to allocate space for something like an *int*, we have an *object* in memory
- For more complex data, we define *classes* for those objects, where the class defines the internal structure and methods of an object
- Thought question: Is a class an ADT? Why or why not?

It's certainly possible to think about a "class" as being an abstract data type. In fact, it's an example of a "meta" concept where we describe how we describe things using the tools we use to describe things (yep... it's somewhat circular thinking).

This thought is the theory behind meta-programming concepts such as functor objects in C++ and lambda-functions in languages such as Lisp, C++, Java, Ruby, and many others. It's also what leads into working with purely functional languages such as ML, Haskell, and Scala.

Implementation

- In our programming model, objects are the implementation of an ADT
- We can consider then the possibility of a collection of (abstract) objects
- Most modern programming languages provide a library of classes that we can use rather than having to implement our collections

  C++ Standard Library and the Standard Template Library

  Java The java.system collection classes

  C# The .NET System Framework Collection classes

- For this course, use these class libraries, don't try to make your own.

2 Linear data structures

Stacks and queues

Stacks and queues are ADTs in which the element removed from the set of values by the *Delete* function is pre-specified:

Stack the deleted element is the one most recently added (LIFO)

Queue the deleted element is the one that has been in the set the longest time (FIFO)

In both cases, the interface will contain a function $Empty() : S \rightarrow B$, with $B = (true, false)$ that returns true if $S = 0$

Stacks

We define a *stack* as being an ADT $< S, I >$ where there are three functions in I:

$Empty()$ Is the stack empty?

$Push(x)$ Add x to S

$Pop()$ Return the top element from S

Note the implication of ordering in S.

Queues

An ADT $< S, I >$ with the following operations

- $Empty()$
- $Enqueue()$
- $Dequeue()$

Stacks and Queues : Implementation

- Use an array
- Linked list ... more shortly
- What about C++?

Note that many object frameworks will re-use the $Push(x)$ and $Pop(x)$ notation from Stacks. We see that in C++ Standard Library with the `push_back()`, `pop_back()`, `push_front()`, and `pop_back()` notation.

C++ Standard Library (CSL) : Deques

- Defines the concept of "container"

Deque double-ended queue

- Interface for both stack, single-ended queue, and double-ended queue semantics

CSL Deques : Example

```
#include <deque>
#include <iostream>
using namespace std;
int main() {
    deque<float> coll;
    for (int i=1; i <= 6; ++i) {
      coll.push_front(i*1.1);
    }
    for (int i=0; i < coll.size(); ++i) {
      cout << coll[i] << ' ';
    }
    cout << endl;
}
```

CSL Arrays and Lists : Arrays

Array an object of class *array*

- Different than ordinary "C-style array"
- The *Array* container has similar semantics
- Prefer to use the STL. Why?

CSL Arrays and Lists: Vectors

- Arrays are fixed size, need dynamic storage
- The *Vector* data type provides this function
- Supports dynamic access, inserts are slower

CSL Arrays and Lists: Vectors : Example

```
#include <vector>
#include <iostream>
using namespace std;
int main() {
    vector<int> coll;
    for (int i=1; i <= 6; ++i) {
       coll.push_back(i);
    }
    for (int i=0; i < coll.size(); ++ i) {
       cout << coll[i] << ' ';
    }
    cout << endl;
}
```

The List ADT

Linked list sequence of zero or more elements called *nodes*

Node Structure containing some data and one more links called *pointers*

Pointer Link to another node in the linked list

Interface: $search(k)$, $insert(x)$. $delete(x)$

Types of lists

- Singly linked vs. doubly linked lists
- Sorted list
- Circular list

CSL Arrays and Lists: Lists

- The *list* class in the CSL is a doubly linked list
- Lists do not provide random access (no '[]' operator)
- Note that C++11 introduced *forward_list* which is a singly linked list
- If no random access, how do you do *iterate* through a list?

CSL Iterators

Iterator an object that iterate over the elements in a set

- Navigate from element to element
- Can all or a subset of the elements of an container
- Models the semantics of traversing a linked list constructed using pointers

Four operators

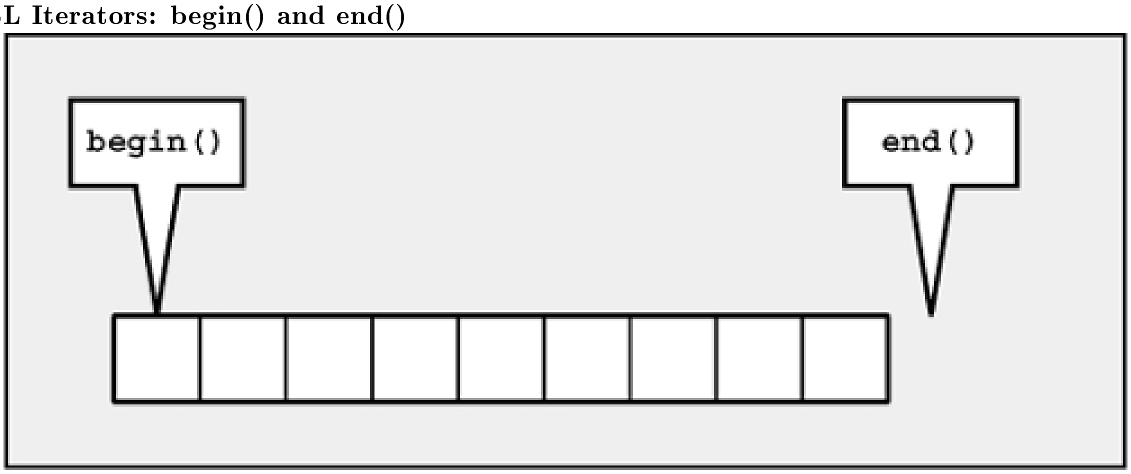
Operator * returns the element at the current position

Operator ++ Increment to next element

Operator `=` **and** `!` Is this operator at the same position as another operator

Operator = assigns an iterator

CSL Iterators: begin() and end()



Semantics

begin() return an iterator that is the start of the container

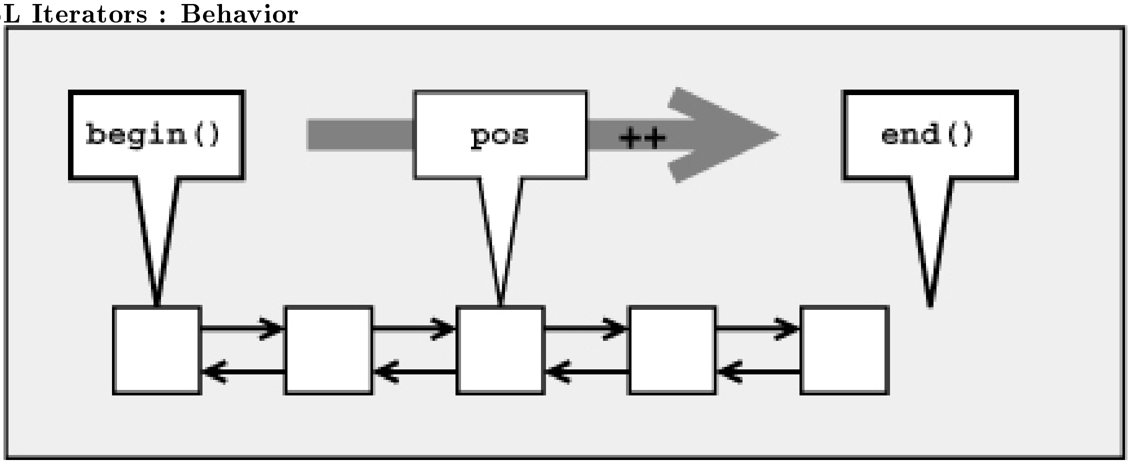
end() returns the end of the last operator (note: half-open)

Makes for-loop semantics work

CSL Iterators: Example

```
#include <list>
#include <iostream>
using namespace std;
int main() {
    list<char> coll;
    for (char c='a'; c <= 'z'; ++c) {
      coll.push_back(c);
    }
    list<char>::const_iterator pos;
    for (pos = coll.begin(); pos != coll.end(); ++pos)
    {
      cout << *pos << ' ';
    }
    cout << endl;
}
```

CSL Iterators : Behavior



CSL Iterators : the increment operator

- When we're first taught how to use the '++' operator, we're usually told to use the prefix form rather than postfix form
- Iterators are the reason why this is so
  - Postincrement has to create a temporary object because it has to return the old position of the iterator
  - This can be a performance hit if the object is large

CSL Iterators: C++11 range-based for loops

```
for (type elem : coll {
   ...
}
// This is equivalent to traditional iterator for loop
for (auto pos=coll.begin(), end = coll.end();
      pos !=end;
      ++pos)
{
```

```
    type elem = *pos;
    ...
}
```

CSL Iterators: C++11 range-based for loops

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v = {0, 1, 2, 3, 4, 5};
    // access by const reference
    for (const int &i : v) {
        cout << i << ' ';
    }
    cout << '\n';
    // access by value, the type of i is int
    for (auto i : v) {
        cout << i << ' ';
    }
    cout << '\n';
}
```

CSL Iterators: C++11 range-based for loops

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v = {0, 1, 2, 3, 4, 5};
    // access by reference, the type of i is int&
    for (auto & i : v) {
       cout << i << ' ';
    }
    cout << '\n';
    // the initializer may be a braced-init-list
    for(int n : {0,1,2,3,4,5}) {
       cout << n << ' ';
    }
    cout << '\n';
}
```

CSL Associative Containers

Set A collection in which elements are stored according to their own value

Multiset Same as set, but duplicates are allowed

Map Contains elements defined as *key-value* pairs

- Sorted by the key value
- Keys can only occur once

- Maps can be used as an associative array

Multimap Same as map, but duplicate keys are allowed

- Multiple elements with the same key
- Multimaps can be used as a dictionary

CSL Associative Containers: Example - Multiset

```
#include <set>
#include <string>
#include <iostream>
using namespace std;
int main() {
    multiset<string> cities {
       "Bruanschweig", "Hanover", "Frankfurt", "New York"
       "Chicago", "Tornoto", "Paris", "Frankfurt" };
    for (const auto& elem : cities) {
       cout << elem << " ";
    }
    // continued on next slide
```

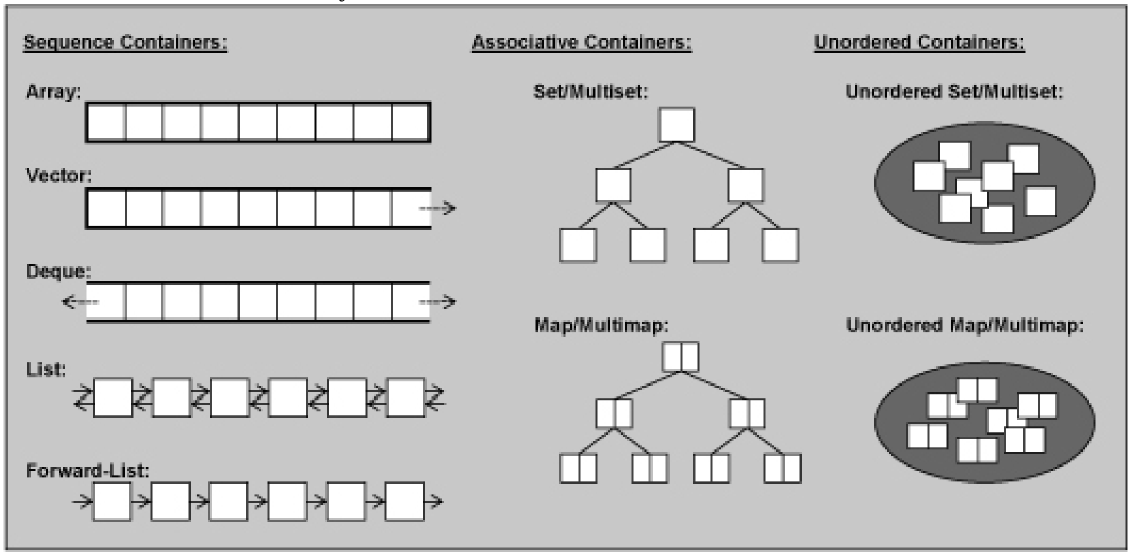
CSL Associative Containers: Example - Multiset

```
    // continued from previous slide
    cout << endl;
    cities.insert({"London", "Munich", "Hanover", "Braunschweig"});
    for (const auto& elem : cities) {
        cout << elem << " ";
    }
    cout << endl;
}
```

CSL Associative Containers: Example - Multimap

```
#include <map>
#include <string>
#include <iostream>
using namespace std;
int main() {
    multimap<int,string> coll;
    // assign some elements in arbitrary order
    // - a value with key 1 gets inserted twice
    coll = { {5,"tagged"}, {2,"a"},
       {1,"this"}, {4,"of"},
       {6,"strings"}, {1,"is"}, {3,"multimap"} };
    // - element member second is the value
    for (auto elem : coll)  {
       cout << elem.second  << ' ';
    }
    cout << endl;
}
```

CSL Containers: A summary



CSL Performance

Op	C array	vector	deque	list
Insert/erase at start	n/a	linear	constant	constant
Insert/erase at end	n/a	constant	constant	constant
Insert/erase in middle	n/a	linear	linear	constant
Access first element	constant	constant	constant	constant
Access last element	constant	constant	constant	constant
Access middle element	constant	constant	constant	linear
Overhead	none	low	medium	high

3 Key Points

Key Points

- Abstract data type: mathematical construct
- Algorithms work on abstract data types
- Relationship between class, object, and ADT
- Most modern object frameworks provide implementations of common data structures
  - Assignments in this course can be done using any modern language with a decent class library (C++ w/ CSL or Boost, Java, C#, Python, . . .)
  - Lectures, exams and sample code will use the C++ Standard Library