Data Structures with Collections / Java

Alice E. Fischer

Lecture 2 – 2016

Abstract Data Types vs. Data Structures

An abstract data type (ADT) an interface and a set of guarantees.

A data structure is an interface, a set of guarantees and an implementation strategy.

Data structures can be grouped by their nature:

- Linear: Each data item has one predecessor and one successor.
- Tree: Each data item has one predecessor and one or more successors.
- Associative: Each item is a pair of a key and associated data.
- Graph: A data item can have zero or more predecessor and zero or more successors. Circular relationships can exist.
- Composite: Real applications combine the above basic data structures.

Typical Applications

We would use very different data structures for these applications:

- A spelling-checker: data changes occasionally, is searched often, and must be searched using the letters from an input word.
- Simulating the activity of bank customers, in order to optimize the number of tellers: Customers often come and go. Their order of arrival determines the order of processing.
- Sorting a list of employees by zip code. First, we need to sort pointers to Employee objects, not the objects themselves. Second, if the list is large, we must use a representation that supports an efficient $O(n * log(n))$ sorting method.
- A database: We need potentially massive storage with fast retrieval and the ability to index the data in several orders simultaneously.

The Usual Suspects

- Array of values or of pointers, with 1 or more dimensions.
- A dynamic array such as vector in C++, that grows longer when needed.
- Linked List, a linear sequence of objects connected by pointers.
- Stack and Queue: A sequence of data values with restricted access, implemented by a vector or a linked list.
- Priority Queue: A set of values arranged in order of importance. Implemented by a list or a heap.
- Tree: Usually ordered; a branching linked data structure.
- Heap: A binary tree represented within an array or a vector.
- Hash Table: An array of queues, used for rapid data retrieval. Not ordered.
- Graph: a set of data items with arbitrary connections. Implemented as an array of lists.

The Simplest ADT

Guarantees for an Unstructured Set:

- Data can be stored in it, up to some maximum quantity.
- You can find out how much data is stored in it. (N)
- You can find out if there is space for more data.
- Given a data item, its location in the set can be found.
- Given a location, data can be removed from the set.
- The data in the set can be traversed or printed out in some unspecified order.

The Simplest Implementation for this ADT

Use an array. (Call it ar)

- $int\ size()$
  Return n, the number of items stored in the array. }
- $boolean\ isFull()\{$ return $n == ar.length$ }
- $insert(item\ T)$
  Use subscript to store the data at the end of ar; increment n.
- $remove(location\ k)$
  Copy $ar[n-1]$ to $ar[k]$ and decrement n.
- $int\ find(item\ T)$
  Do a sequential search for T; return the subscript.
- $for(int\ k = 0;\ k < n, ++k)$ { process $ar[k]$ };

An Unstructured Set Implementation

This class implements the ADT Unstructured Set .

```
class CharSet {
private:
    const int maxSize = 26;
    char data[26];
    int n=0;  // Number of objects currently in the set.

public:
    int  size() { return n; }
    bool isFull(){ return n >= maxSize; }
    void insert( char ch ) { data[n++] = toupper(ch); }
    void remove( int k ) { data[k] = data[--n]; }
    ...
```

Find and Print

```
    ...
    //-----------------------------------------------
    // Given a data item, find its location.
    int find( char ch ) {
        int k;
        for( k=0; k<n; ++k ) if (data[k] == ch) break;
        return k<n ? k : -1;  // -1 means failure.
    }
    //-----------------------------------------------
    void print( ostream& out ){
        for( int k=0; k<n; ++k )  out << data[k];
        out << endl;
    }
};
```

Homework 1, part 1

Due Tuesday, Feb. 2
The prior two slides contain an array implementation of the Unstructured Set ADT. Answer these questions about the class functions.

1. Which functions operate in constant time: O(constant) ?
2. Which functions operate in logarithmic time: O(log(n)) ?
3. Which functions operate in linear time: O(n)?

Note: The answer may be none, one function, or more than one.

Later in this lecture the OrderedSet ADT is presented. There will be three more questions about that implementation.

ADT Ordered List

The data in an ordered list is sorted at all times.

When a new item is inserted into the list, it is inserted in its proper place.

An ordering implies that we have a key part of the data item and a function to compare two data items according to that part.

The basic operations on an ordered list have the same names as the operations on an unsorted list BUT the actions and running time might be different.

Ordered List Operations

The running time of the first group of functions is the same as for an unstructured set.

- size
- isFull
- traverse or print

The running time of these functions is not like an unstructured set.

- find
- remove (given the location of the item to delete)
- insert

An Ordered List Class

```
class OrderedSet {
private:
    char data[26];
    int n=0;
    const int maxSize = 26;

public:
    int  size() { return n; }
    bool isFull(){ return n >= maxSize; }
    void insert( char ch );
    void remove( int k );
    int  find( char ch );
    void print( ostream& out );
};
```

Insert

```
#include "OrderedSet.hpp"
// ---------------------------------------------
// Keep the data sorted in ascending order.
void OrderedSet::insert( char ch ) {
    int j;
    for (j=n; j>0; --j){
        if (data[j-1] < ch) break;
        else data[j] = data[j-1];
    }
    n++;
    data[j] = ch;
}
```

Find

```
// ---------------------------------------
// Find a specified character in the set.
// Return -1 for failure
int OrderedSet::find( char ch ) {
    int low=0,  high=n;
    for (;;) {
        int mid = (high+low)/2;
        if (data[mid] == ch) return mid;
        if (high == low) return -1;
        if (data[mid] > ch) high = mid;
        else low = mid;
    }
}
```

Remove and Print

```
// ----------------------------------------------------
void OrderedSet::remove( int k ) {
    for( int j=k; j<n; ++j ) data[j] = data[j+1];
    --n;
}

// ----------------------------------------------------
void OrderedSet::print( ostream& out ) {
    for( int k=0; k<n; ++k )  out << data[k];
    out << endl;
}
```

Homework 1, part 2

Due Tuesday, Feb. 2
The prior four slides contain an array implementation of the Ordered List ADT. Answer these questions about the class functions.

4. Which functions operate in constant time: O(constant) ?
5. Which functions operate in logarithmic time: O(log(n)) ?
6. Which functions operate in linear time: O(n)?

Which is Better, Ordered or Unordered?

That depends on the nature of your application.

- If you search frequently through a stable set of data, ordered is better.
- If search is infrequent but items come and go often, unordered is better.
- If additions, deletions, and searches are all frequent, you don't want an array.