ECE 250 - Fall 2018

Aditya Arora

6th September 2018

Week 2 Notes

1 Introduction

Data Structures: Way to represent/encode data. Example: array, linked list vs `int`, `char` i.e. "basic types"
Algorithm: Set of instructions to make a data structure useful. For example

```
SEARCH(A[0,...n-1], i)
    for j from 0 to n-1
        if A[j] = i then return true
    return false
```

Algorithms have two efficiencies related to them, time efficiency and space efficiency Interesting note: space efficiency `j` is not constant space since `j` has to be of data type to reach size `n`, which means that its space efficiency is $log_2 n$

"The Trade-Off"

```
MEDIAN(A[0,...n-1]) // n is odd
    SORT(A)
    return A[(n-1)/2]
```

Another one

```
MEDIAN(A[0,...n-1]) // n is odd
    while true
        pick i uniformly at random from 0,...,n-1:
        check if A[i] is median
        if true, return A[i]
```

The median can be checked by:

```
CHECK_MEDIAN(A[0,...n-1]) // n is odd
    c <- 0
    for j from 0 to n-1
    if A[j] < A[i]:
        c <- c+1
    if c = (n-1)/2:
        return true
```

2 Containers, Relations and Abstract Data Types

2.1 Abstract Data Types

The term Abstract Data Type (or ADT) is used to model the storage, access, manipulation, and removal of related data. These operations will be divided into two general categories:

1. Queries that determine the properties of the objects that are stored
2. Data manipulations that modify the objects being stored

2.1.1 Containers

The most general Abstract Data Type (ADT): A container describes structure that store and give access to objects The queries and operations of interest may be defined on:
- The container as an entity
- The objects stored within a container

Manipulation	**Standard Template Library Equivalent**
Create a new container	`Container()`
Copy an existing container and its contents	`Container( Container const & )`
Destroy a container	`~Container()`
Empty a container	`void clear()`
Take the union of (or merge) two containers	`void insert( Container const & )`
Find the intersection of two containers	no equivalence

Query	**Standard Template Library Equivalent**
Is the container empty?	`bool empty() const`
How many objects are in the container?	`int size() const`
What is the maximum capacity of the container?	`int max_size() const`

Query	**Standard Template Library Equivalent**
Insert an object into the container	`void insert( Type const & )`
Remove an object from the container	`void erase( Type const & )`
Access or modify an object in the container	`iterator find( Type const & ) const`
Determine the number of copies of an object that are in a container	`int count( Type const & ) const`
Iterate (or step) through the objects currently in the container	`iterator begin() const`

2.1.2 Simple and Associative Containers

Simple Containers	**Associative Containers**
Containers that store individual objects	Containers that store keys as well as the information associated with those keys
Eg: Temperature Readings in an Array	*Eg:* Quest Server with Student Records with Student ID key

2.1.3 Unique or Duplicate Objects

Design Requirement might be to either:

1. Store duplicate identical records
2. Require that all objects be unique

We assume uniqueness unless stated specifically, more often than not this requirement only requires subtle code changes

2.1.4 The Standard Template Library (STL)

The STL has four containers related to our discussions above:

	Unique Objects/Keys	Duplicate Objects/Keys
Simple Container of Objects	\codeword{set<T>}	\codeword{multiset<T>}
Associative Containers of key-object pairs	\codeword{map<K, T>}	\codeword{multimap<K, T>}

```
#include <iostream>
#include <set>

int main() {
    std::set<int> ints;
    // Inserts 101 values 10000, ..., 9, 4, 1, 0, 1, 4, 9, ..., 10000
    // in that order
    for ( int i = -100; i <= 100; ++i ) {
        ints.insert( i*i ); // Ignores duplicates: (-3)*(-3) == 3*3
    }
    std::cout << "Size of 'is': " << ints.size() << std::endl;  // Prints 101
    ints.erase( 50 );                       // Does nothing
    ints.erase( 9 );                        // Removes 9
    std::cout << "Size of 'is': " << ints.size() << std::endl;  // Prints 100
    return 0;
}
```

2.1.5 Operations

We must account for all current and future design requirements, moreover in some cases design requirements may be constrained to improve memory or time performance

2.1.6 Relationships

In general, we may not only want to store data, but we may need to also store relationships between the stored data, for example:

1. A genealogical database must not only store people but must also store family relations
2. `maps.google.ca` must not only store roads but how those roads are connected (they actually store intersections and for each intersections, which other intersections are adjacent to it

2.1.6.1 No Relationships Required