Abstract Data Type

- A set of data values and associated operations that are precisely specified independent of any particular implementation.
- A data type having
  - a logical representation of the data
  - operations over its data
- A logical description
- may be implemented in various ways
  - implementation-independent

*from http://xlinux.nist.gov/dads/

Data Structures again

- The term “data structures” is often extended to include both concrete AND logical descriptions of complicated data types.
- A list of data structures could include ADTs
  - arrays
  - linked lists
  - stacks
  - queues
  - vectors or lists

Which are concrete?
Which are abstract?
Commonly used ADTs

- The purpose of many commonly used ADTs is to:
  - store a collection of objects
  - potentially organize the objects in a specific way
  - provide potentially limited access to the objects

- These ADTs are often called:
  - containers
  - collections
  - container classes

Examples:
- List (or sequence or vector)
- Set
- Multi-set (or bag)
- Stack and Queue
- Tree
- Map (or dictionary)

Stacks, Queues, and Trees will be covered later in the semester

A List ADT

- **Values**: ordered (1st, 2nd, etc) set of objects
- **Operations** often include:
  - constructor: creates an empty list
  - isEmpty: is the list empty
  - size: returns the number of elements
  - add(i,e): inserts an element e at position i
  - remove(i): removes the element at position i
  - get(i): returns the element at position i
  - set(i,e) changes the element at position i to value e

A Set ADT

- **Values**: unordered collection of unique objects
- **Operations** often include:
  - constructor: creates an empty set
  - isEmpty: is the set empty
  - size: returns the number of elements
  - add(e): adds an element to the set (if not there)
  - remove(e): removes an element from the set (if it is there)
  - contains(x): true if x is in the set
  - addAll(s): adds all elements from set s to this one (union)
A Bag (multi-set) ADT

- **Values**: unordered collection of objects (may include duplicates)
- **Operations** may include:
  - constructor: creates an empty bag
  - isEmpty: is the bag empty
  - size: returns the number of elements
  - add(e): adds an element e to the bag
  - remove(e): removes one copy of an element from the bag (if it has any)
  - removeAll(e): removes all copies of e from the bag
  - occurrences(x): how many times x is in the bag

A Map ADT

- **Values**: a collection of unique keys and a collection of values where each key is associated with a single value. Keys have one type, values another.
- **Operations** may include:
  - constructor: creates an empty map
  - isEmpty: returns true if map has no key-value pairs
  - size: returns the number of key-value pairs in the map
  - get(k): returns value associated with key k (if any)
  - put(k,v): associates value v with key k (adds a pair)
  - keySet: returns a set of all the keys in the map

Implementing an ADT

- **Interface (*.h)**:
  - class declaration
  - prototypes for the operations (interface)
  - data members for the actual (concrete) representation
- **Implementation (*.cpp)**
  - function definitions for the operations
  - depends on representation of data members (their concrete implementation)

Example ADT: bag version 1

```cpp
class Bag {
public:
    Bag();
    void add(int element);
    void remove(int element);
    int occurrences(int element) const;
    bool isEmpty() const;
    int size() const;
static const int CAPACITY = 20;
private:
    int data[CAPACITY];
    int count;
};
```

What is the difference between count and CAPACITY?
Example ADT: bag version 1

```cpp
#include "bag.h"
#include <cassert>
using namespace std;

Bag::Bag () {
    count = 0;
}

void Bag::add(int element) {
    assert (count < CAPACITY);
    data[count] = element;
    count++;
}

void Bag::remove(int element) {
    int index = -1;  //change to position if found
    for (int i=0; i<count && index== -1; i++) {
        if (data[i]==element) {
            index = i;
        }
    }
    if (index!=-1) {  //found, replace w/ last elem
        data[index] = data[count-1];
        count--;
    }

    //continued...
```

```cpp
Example ADT: bag version 1

```cpp
```cpp
int  Bag::occurrences(int element) const {
    int occurrences=0;
    for (int i=0; i<count; i++) {
        if (data[i]==element) {
            occurrences++;
        }
    }
    return occurrences;
}

bool Bag::isEmpty() const {
    return (count==0);
}

int  Bag::size() const {
    return count;
}
```

```cpp
bag "driver"

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

int main () {
    Bag b;
    b.add(4);
    b.add(8);
    b.add(4);
    cout << "size " << b.size() << endl;
    cout << "how many 4's: " << b.occurrences(4) << endl << endl;
    b.remove(4);
    cout << "removed a 4" << endl;
    cout << "size " << b.size() << endl;
    cout << "how many 4's: " << b.occurrences(4) << endl << endl;
    b.add(10);
    cout << "added 10 to b" << endl;
    cout << "size " << b.size() << endl;
    cout << "how many 4's: " << b.occurrences(4) << endl;

    bagTest.cpp

```cpp
```cpp
Bag c(b);

cout << "copied to c" << endl;
    cout << "size " << c.size() << endl;
    cout << "how many 4's: " << c.occurrences(4) << endl;
    b.add(10);
    cout << "added 10 to b" << endl;
    cout << "size " << b.size() << endl;
    cout << "how many 4's: " << b.occurrences(4) << endl;

    for (int i=0; i<20; i++)
        b.add(33);
    cout << "added 20 more items to b" << endl;
    return 0;
};
```
Bag “driver”: output

```
size 3
how many 4's: 2
removed a 4
size 2
how many 4's: 1
copied to c
size 2
how many 4's: 1
added 10 to b
b.size 3
c.size 2
starting to add 20 items
Assertion failed: (count < CAPACITY), function add, file bag.cpp, line 12.
Abort trap: 6
```

Bag version 1 summary

- Implemented using a fixed size array
- When adding more elements than fit in the bag, the program exits.
- Solution:
  - use a dynamically allocated array
  - when its capacity is reached, allocate a new, bigger array.

Bag version 2

```
class Bag
{
public:
  Bag();
  Bag(const Bag &); //copy constructor
  ~Bag();
  void operator=(const Bag &);
  void add(int element);
  void remove(int element);
  int occurrences(int element) const;
  bool isEmpty() const;
  int size() const;

private:
  int *data; //pointer to bag array
  int capacity; //size of the array
  int count; //number of elements currently in array
};
```

```
Bag::Bag () {
  count = 0;
  capacity = INCREMENT;
  data = new int[capacity];
}
```

```
//copy constructor
Bag::Bag(const Bag &rhs) {
  data = new int[rhs.capacity]; //allocate new array
  capacity = rhs.capacity;
  count = rhs.count;
  for (int i=0; i<count; i++) {
    data[i] = rhs.data[i];
  }
}
```

```
//destructor
Bag::~Bag() {
  delete [] data;
}
```
void Bag::operator=(const Bag &rhs) {
    if (data) delete [] data;      //delete old array
    data = new int[rhs.capacity];  //allocate new array
    capacity = rhs.capacity;       //copy values
    count = rhs.count;
    for (int i=0; i<count; i++) {
        data[i] = rhs.data[i];
    }
}

void Bag::add(int element) {
    //if count is at the capacity, resize
    if (count==capacity) {
        capacity += INCREMENT;
        int *newData = new int[capacity];  //new array
        for (int i=0; i<count; i++) {      //copy values
            newData[i] = data[i];
        }
        delete [] data;                 //delete old array
        data = newData;                 //make data point to new
    }
    data[count] = element;           //add new element
    count++;
}