CS-218
DATA STRUCTURE

Dr. Hashim Yasin

National University of Computer and Emerging Sciences,

Faisalabad, Pakistan.

COMMON OPERATIONS IN DATA STRUCTURE

Common Operations in Data Structure

- **Traversing**
  - Accessing each record exactly once so that certain items in the record may be processed
- **Searching**
  - Finding the location of the record with the given key value or finding the location of all records which satisfy one or more conditions
- **Insertion**
  - Adding a new record to the structure

Common Operations in Data Structure

- **Deletion**
  - Removing a record from the structure
- **Sorting**
  - Arrange the records in a logical order
- **Merging**
  - Combining records from two or more files or data structures into one

Selecting a Data Structure

How can we select a data structure?

Answer:

1. Analyse the problem
2. Determine the resource constraints that a solution must meet.
3. Determine the basic operations that must be supported.
   - Quantify the resource constraints for each operation.
4. Select the data structure that best meets these requirements.

Data Structure Philosophy

- **Each data structure has costs and benefits.**
- Rarely is one data structure better than another in all situations.
- A data structure requires:
  - space for each data item it stores,
  - time to perform each basic operation,
  - programming effort.

Data Abstraction

- A separation between computer and our view of data
- Goal is to hide complexity
  - Example: an integer
- More formally we can say that:

*The separation of a data type's **logical properties** from its **implementation** is known as **data abstraction***

Abstract Data Type

- A ***type*** is a collection of values. For example,
  - The Boolean type consists of the values, true and false.
  - The integers also form a type.
- ***Aggregate type***, A bank account record will typically contain several pieces of information such as name, address, account number, and account balance.
- A ***data item*** is a piece of information or a record whose value is drawn from a type.
  - A data item is said to be a ***member of a type***.

Abstract Data Type

- A ***data type*** is a type together with a collection of operations to manipulate the type.
  - For example, an integer variable is a member of the integer data type.
  - Addition is an example of an operation on the integer data type.
- An ***abstract data type (ADT)*** is the specification of a data type within some language, independent of an implementation.
  - The interface for the ADT is defined in terms of a type and a set of operations on that type.

Abstract Data Type

- An ADT does not specify *how the data type is implemented*.
- These implementation details are hidden from the user of the ADT and protected from outside access; a concept referred to as ***encapsulation***.
- **ADT:** A definition for a data type in terms of a *set of values* and a *set of operations* on that data type
  - Each ADT operation is defined by its inputs and outputs

Abstract Data Type

- Example: Flight reservation system
- Basic operations: find an empty seat, reserve a seat, cancel a seat assignment
- Why "abstract?"
- Data, operations, and relations are studied ***independent of implementation***
- **What**, not ***how*** is the focus

Abstract Data Type

- In Object Oriented Programming, data and the operations that manipulate that data are grouped together in classes

- *Abstract Data Types (ADTs)* or *data structures* or *collections* store data and allow various operations on the data to access and change it

Abstract Data Type

- Specify the operations of the data structure and leave implementation details to later
  - in Java use an interface to specify operations
- many, many different ADTs
  - picking the right one for the job is an important step in design
- High level languages often provide builtin ADTs,
  - the C++ Standard Template Library (STL), the Java standard library

Example

- Let us take the following program that demonstrates the vector container (a C++ Standard Template) which is similar to an array
  - But with an exception that it automatically handles its own storage requirements in case it grows.

Vector container

C++ Standard Template Library (STL)

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
vector<int> vec;    int i; // create a vector to store int
// display the original size of vec
cout << "vector size = " << vec.size() << endl;
// push 5 values into the vector
for(i = 0; i < 5; i++) {
        vec.push_back(i);
}
// display extended size of vec
cout << "extended vector size = " << vec.size() << endl;
// access 5 values from the vector
for(i = 0; i < 5; i++) {
        cout << "value of vec [" << i << "] = " << vec[i] << endl;
}
// use iterator to access the values
vector<int>::iterator v = vec.begin();
while( v != vec.end()) {
        cout << "value of v = " << *v << endl;      v++;
}
return 0;
}
```

Vector container

C++ Standard Template Library (STL)

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
vector<int> vec;    int i; // create a vector to store int
// display the original size of vec
cout << "vector size = " << vec.size() << endl;
// push 5 values into the vector
for(i = 0; i < 5; i++) {
        vec.push_back(i);
}
// display extended size of vec
cout << "extended vector size = " << vec.size() << endl;
// access 5 values from the vector
for(i = 0; i < 5; i++) {
        cout << "value of vec [" << i << "] = " << vec[i] << endl;
}
// use iterator to access the values
vector<int>::iterator v = vec.begin();
while( v != vec.end()) {
        cout << "value of v = " << *v << endl;      v++;
}
return 0;
}
```

```
vector size = 0
extended vector size = 5
value of vec [0] = 0
value of vec [1] = 1
value of vec [2] = 2
value of vec [3] = 3
value of vec [4] = 4
value of v = 0
value of v = 1
value of v = 2
value of v = 3
value of v = 4
```

Abstract Data Type

- Solving a problem involves processing data, and an important part of the solution is the careful organization of the data

- In order to do that, we need to identify:
    1. The *collection of data items*
    2. Basic *operation* that must be performed on them

- **Abstract Data Type (ADT):** a collection of data items together with the operations on the data

Abstract Data Type vs Data Structure

- A data structure is the physical implementation of an ADT.
  - Each operation associated with the ADT is implemented by one or more subroutines in the implementation.

- A data structure usually refers to an organization of data in main memory.

Abstract Data Type vs Data Structure

- An Abstract Data Type is implementation independent
- A Data Structure is implementation dependent
- A Data Structure is how we implement the data in an abstract data type whose values have component parts
- The <u>operation</u> on an abstract data type are translated into algorithms on the data structure

EXAMPLE

Example

- Consider an example of an **airplane flight with 10 seats** to be assigned
- Tasks
  - List available seats
  - Reserve a seat
- How to store, access data?
  - 10 individual variables
  - An array of variables

Solution 1 (10 individual variables)

Algorithm to List available seats

1. If seat1 == ' ':
   display 1
2. If seat2 == ' ':
   display 2

.

.

.

10. If seat10 == ' ':
    display 10

Algorithm to Reserve a seat

1. Set DONE to false
2. If seat1 == ' ':
   print "do you want seat #1??"
   Get answer
   if answer== 'Y':
      set seat1 to 'X'
      set Done to True
3. If seat2 == ' ':
   print "do you want seat #2??"
   Get answer
   if answer== 'Y':
      set seat2 to 'X'
      set Done to True

Solution 2 **Array**

Algorithm to List available seats

For number ranging from 0 to max_seats-1, do:
    If seat[number] == ' ':
        Display number

Algorithm to Reserve a seat

Reading number of seat to be reserved
If seat[number] is equal to ' ':
    set seat[number] to 'X'
else
    Display a message that the seat with this number is occupied

Example

- This simple example does illustrate the concept of an **Abstract Data Type**

- **ADT consists of**
  - The *collection of data items*
  - The *basic operation* must be performed on them

- **In the example**,
  - The *collection of data* is a list of seats
  - The *basic operations* are
    - (1) Scan the list to determine which seats are occupied
    - (2) change seat's status

LIST

Lists

- The List is among the most generic data structures.
- Real life examples:
  - shopping list,
  - groceries list,
  - list of people to invite to dinner
  - List of students that appear in exam

Lists

- A list is collection of items that are of the same type (grocery items, integers, names)
- The items, or elements of the list, are stored in some particular order
- It is possible to insert new elements into various positions in the list and remove any element of the list

Lists

- List is a set of elements in a linear order.
- **For example,** data values a1, a2, a3, a4 can be arranged in a list: (a3, a1, a2, a4)
- In this list, a3, is the first element, a1 is the second element, and so on.
- The order is important here; this is not just a random collection of elements; it is an *ordered* collection.

Lists

Useful operations

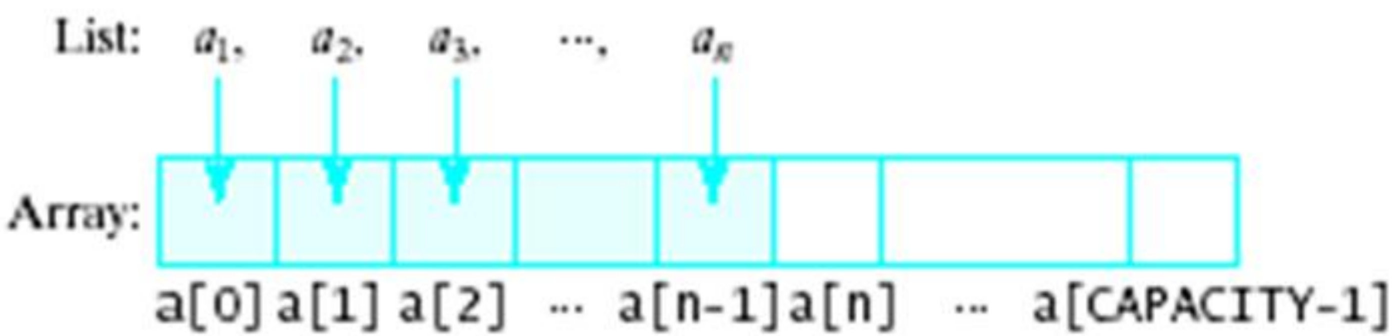
- createList(): create a new list (presumably empty)
- copy(): set one list to be a copy of another
- clear(); clear a list (remove all elements)
- insert(X, ?): Insert element X at a particular position in the list
- remove(?): Remove element at some position in the list

Useful operations

- get(?): Get element at a given position
- update(X, ?): replace the element at a given position with X
- find(X): determine if the element X is in the list
- length(): return the length of the list.

Array-based Implementation

- An array is a feasible choice for storing list elements
  - Element are sequential
  - It is a commonly available data type
  - Algorithm development is easy
- Normally sequential orderings of list elements match with array elements



Array-based Implementation

- Constructor
  - Static array allocated at compile time
- isEmpty
  - Check if size == 0
- Traverse
  - Use a loop from 0^{th} element to `size - 1`

Array-based Implementation

- Insert
  - Shift elements to right of insertion point

Current position

Size Becomes **size+1**

- Delete
  - Shift elements back

Size Becomes **size-1**

List Class with Static Array - Problems

- Stuck with "one size fits all"
  - Could be wasting space
  - Could run out of space
- Better to have instantiation of specific list specify what the capacity should be
- Thus, we may consider creating a `List` class with *dynamically-allocated array*

Dynamic Allocation for List Class

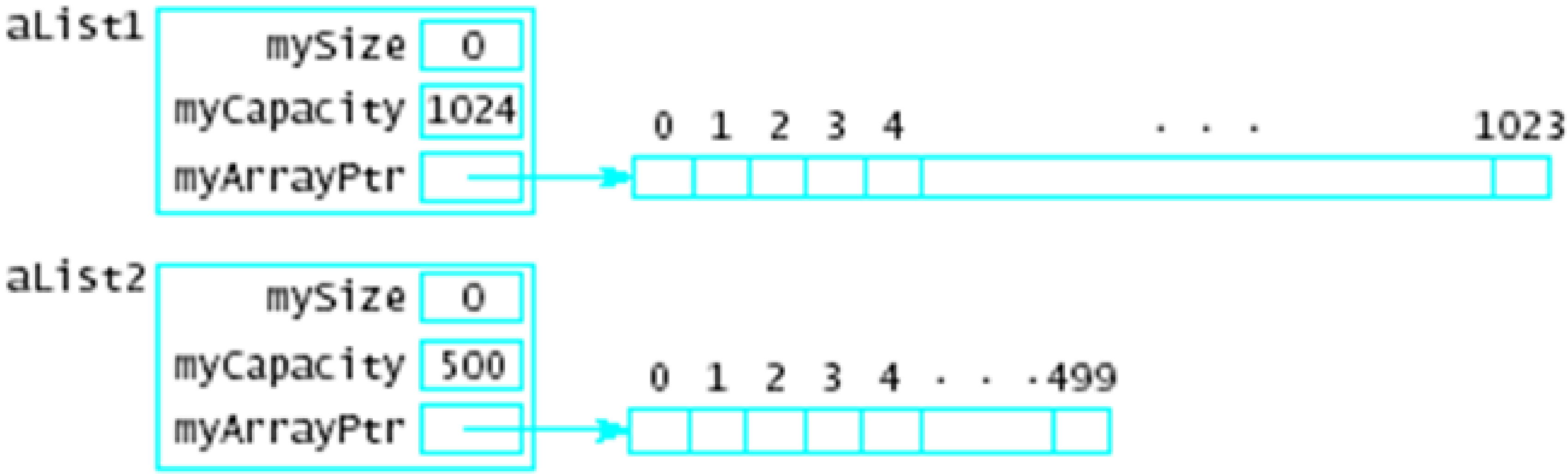
- Changes required in data members
  - Eliminate constant declaration for CAPACITY
  - Add variable data member to store capacity specified by client program

- Little or no changes required for
  - `isEmpty()`
  - `display()`
  - `delete()`
  - `insert()`

Dynamic Allocation for List Class

- Now possible to specify different sized lists

```
cin >> maxListSize;
List aList1 (maxListSize);
List aList2 (500);
```



Improvements in List Class

- Problem 1: Array used has fixed capacity
  Solution:
  - If larger array needed during program execution, allocate copy smaller array to the new one

- Problem 2: Class bound to one type at a time
  Solution:
  - Create multiple `List` classes with differing names
  - Use class template

Inefficiency in Array-based List

- `Insert()`, `erase()` functions inefficient for dynamic lists
  - Those that change frequently
  - Those with many insertions and deletions

So ...

We look for an alternative implementation.

Homework

List ADT Implementation (via dynamic array)

- Implement the following operations of List ADT by using array class (as discussed in the lecture)
- Constructors (default, parameterize, copy) & destructor
- void printList ( ), int searchElement (int X), void insertElement (int X), void insertElementAt (int X, int pos), bool deleteElement (int X), bool isFull ( ), bool isEmpty ( ), int length ( ), void reverseList ( ), void emptyList ( ), void copyList (...)
- Also write a driver (main) program to test your code (provide menu for all operations).

Reading Materials

- Nell Dale: Chapter # 2 (Section 2.1), Chapter # 3
- Schaum's Outlines: Chapter # 1
- Mark A. Weiss: Chapter # 3 (Section – 3.1, 3.2)
- Articles: (abstraction vs. encapsulation)