Data Structures

---

6. List ADT

Roadmap

- List as an ADT
- An array-based implementation of lists

- Introduction to linked lists
- A pointer-based implementation in C++
- Variations of linked lists

Examples of Everyday List

- Everyday usage of the term "list" refers to a linear collection of data items
  - Groceries to be purchased
  - Ingredients of a recipe
  - Job to-do list
  - List of assignments for a course
  - List of courses for summer semester

- Can you name some others??

List (1)

- A collection of items of the same type
- A flexible structure that can grow and shrink on demand

- Elements can be:
  - Inserted
  - Accessed
  - Deleted

  at any position!

List (2)

- A list is a sequence of zero or more elements of a given type
- Represented by a comma-separated sequence of elements

$$a_1, a_2, a_3, \ldots, a_n$$

  - Where $n \geq 0$ and each a_i is of type `element_type`
- Number of elements n determines the length of the list
  - If $n \geq 1$
    - a_1 is the first element
    - a_n is the last element
  - If $n = 0$
    - List has no elements (empty list)

List (3)

- Elements of a list can be linearly ordered according to their position
  - a_i precedes a_{i+1} for `i = 1,2,3…n-1`
  - a_i follows a_{i-1} for `i = 2,3,4…n`
- Element a_i is at position i

Properties of Lists

- Can have a single element
- Can have no elements
- Can be list of lists
- Can be concatenated together
- Can be split into sub-lists

List as an ADT

- We will look at the list as an abstract data type
  - Homogeneous
  - Finite length ??
  - Sequential elements

- Is this information sufficient for defining ADT?

Basic Operations (1)

- Create the list
  - The list is initialized to an empty state
- Determine whether the list is empty
  - Determine whether the list is full
- Find the size of the list
- Destroy, or clear, the list
- Insert an item in the list at the specified location
- Delete an item from the list at the specified location
- Replace an item at the specified location with another item
- Retrieve an item from the list at the specified location
- Search the list for a given item
- Traverse (iterate through) the elements of the list

Basic Operations (2)

- `INSERT(x, p, L)`
  - Insert x at position p in list L
  - If list L has no position p, the result is undefined

- `RETRIEVE(p, L)`
  - Return the element at position p on list L

- `LOCATE(x, L)`
  - Return the position of x on list L

- `DELETE(p, L)`
  - Delete the element at position p on list L

Basic Operations (3)

- `MAKENULL(L)`
  - Causes L to become an empty list and returns position `END(L)`
- `NEXT(p, L)`
  - Return the position following p on list L
- `PREVIOUS(p,L)`
  - Return the position preceding position p on list L
- `FIRST(L)`
  - Returns the first position on the list L

Basic Operations (4)

- `PRINTLIST(L)`
  - Print the elements of L in order of occurrence

- And more …

List as a Data Structure (1)

- We know the ADT of the list, how to implement it?

- Create a List class, containing at least the following function members
  - Constructor
  - `isEmpty()`
  - `insert()`
  - `delete()`
  - `print()`

- What are the other function members?
  - `isFull()`, `listSize()`, `retrieve()`, `replace()`, `search()`, `clearList( )`, …

List as a Data Structure (2)

- Implementation involves
  - Defining data members
  - Defining function members from design phase

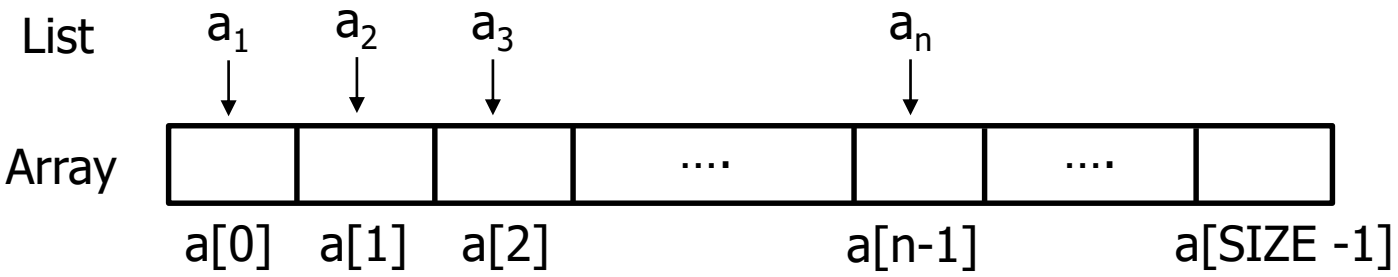
- In terms of implementation, there are two possible approaches
  - Array-based implementation of list
  - Pointers-based implementation of list (called Linked List)

Array-Based Implementation of Lists

Array-Based Implementation

- An array is a viable choice for storing list elements
  - Elements are sequential
  - Array is a commonly available data type
  - Algorithm development is easy

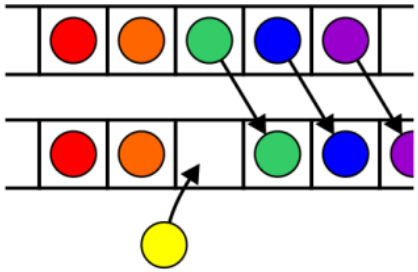
- Normally sequential orderings of list elements match with array indices



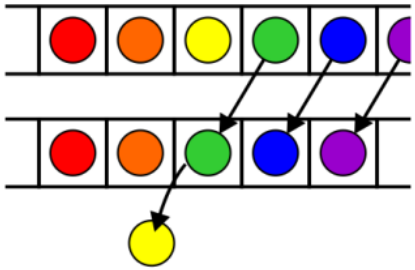
Implementing Operations

- Constructor
  - Static array allocated at compile time
- `isEmpty`
  - Check if `size == 0`
- `traverse/ print`
  - Use a loop from 0th element to `size - 1`
- `insert`
  - Shift elements to right of insertion point
- `delete`
  - Shift elements back

Insertion of a new object



Removal of an object



Inefficiency of Array-Based Implementation

- `insert(), delete()` functions inefficient for dynamic lists
  - Frequent changes
  - Many insertions and deletions

List Class with Static Arrays - Problems

- Stuck with "one size fits all"
  - Could be wasting space
  - Could run out of space

- Better to have instantiation of a list by specifying the capacity (i.e., size)

- Consider creating a List class with dynamically-allocated array

Dynamic Allocation of List Class (1)

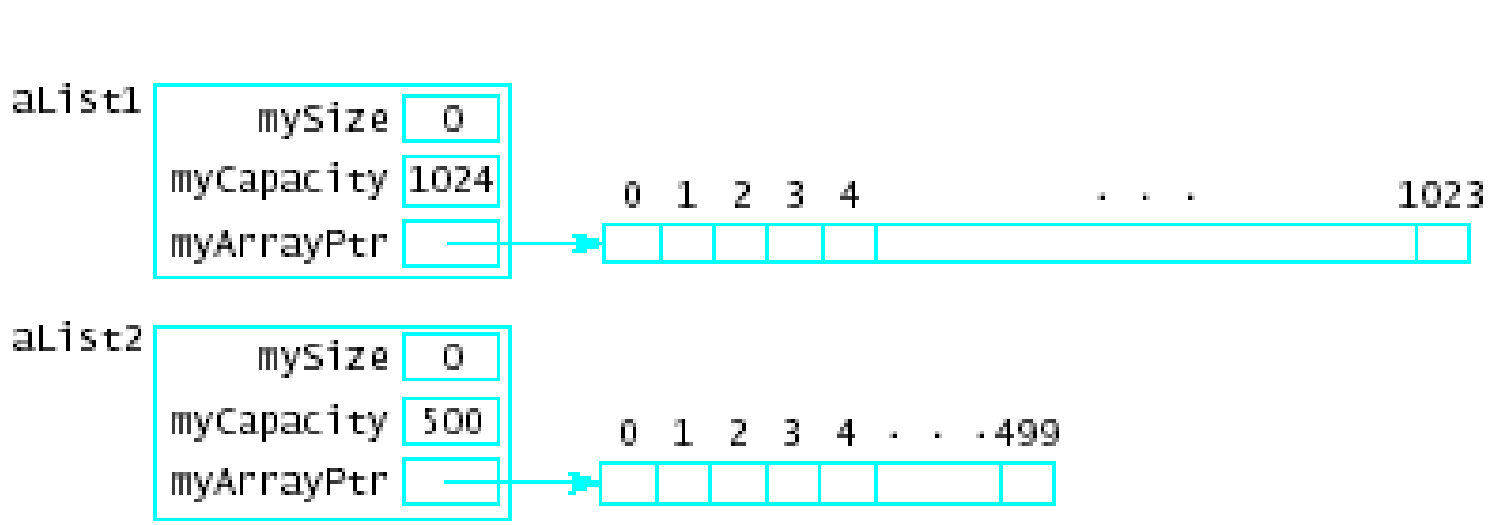
- Changes required in data members
  - Eliminate constant declaration for `CAPACITY/SIZE`
  - Data member to store capacity specified by client program

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

Dynamic Allocation of List Class (2)

- Now possible to specify different sized lists

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



Implementation of List Class

- Problem 1: Array used has fixed capacity
  - 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
  - Create multiple List classes with differing names
  - Use class template

Any Question So Far?