Data Structure and Algorithms

Affefah Qureshi

Department of Computer Science

Iqra University, Islamabad Campus.

List

- 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

- A list is a sequence of zero or more elements of a given type

- Represented by a comma-separated sequence of elements

  a1, a2, a3, ..., an

  - Where n ≥ 0 and each ai is of type element_type

- Number of elements n determines the length of the list
  - If n ≥ 1
    - a1 is the first element
    - an is the last element
  - If n = 0
    - List has no elements (empty list)

List

- 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

Basic Operations

- 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

- 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

- 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
- PRINTLIST(L)
  - – Print the elements of L in order of occurrence
- And more …

List as a Data Structure

- 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 other function members?
  - isFull(), listSize(), retrieve(), replace(), search(), clearList( ), ...

List as a Data Structure

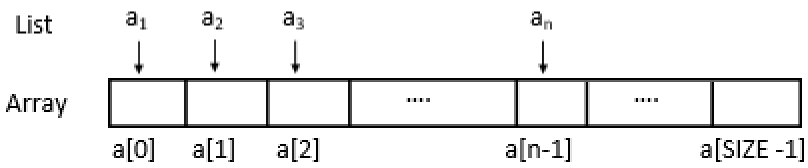
- Implementation involves
  - Defining data members
  - Defining function members from design phase

- In terms of implementation, there are two possible approaches
  - Array-based Implementation of lists
  - Linked list using pointers-based implementation of lists

Array-Based Implementation of Lists

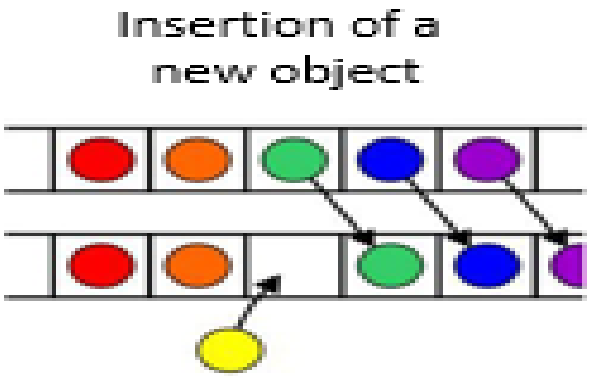
- An array is a viable choice for storing list elements
  - Elements are sequential
  - Array is a commonly available data type
  - Algorithm development is easy

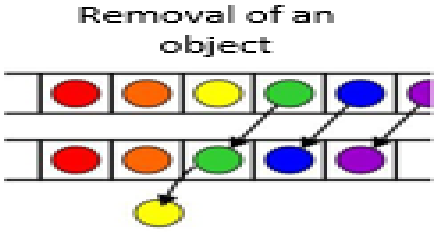
- Normally sequential orderings of list elements match with array indices



Implementing Operations

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





Operations Code in C++

```
#include <iostream>
using namespace std;

class List {
private:
   int items[10];       // Static array to store list elements
   int currentSize;     // Current number of items in the list
   const int maxSize;   // Maximum capacity of the list (fixed size)

public:
List() : maxSize(10), currentSize(0) {}

void clearList() {
       currentSize = 0;
   }
```

Empty and Full Operation

```
bool isEmpty() const {
    return currentSize == 0;
}

// Checks if the list is full
bool isFull() const {
    return currentSize == maxSize;
}
```

Inserts an item at the end of the list

```
void insert(int item) {
    if (isFull()) {
cout << "List is full. Cannot insert item." << endl;
        return;
    }
    items[currentSize++] = item;
  }
```

Deletes the first occurrence of an item from the list

```
void deleteItem(int item) {
      int index = -1;
      for (int i = 0; i < currentSize; ++i) {
         if (items[i] == item) {
            index = i;
            break;
         }       }
      if (index == -1) {
         cout << "Item '" << item << "' not found in list." << endl;
         return;
      }
// Shift elements to the left to fill the gap
      for (int i = index; i < currentSize - 1; ++i) {
         items[i] = items[i + 1];
      }
      --currentSize; } }
```

Prints all items in the list

```
void print() const {
    for (int i = 0; i < currentSize; ++i) {
        cout << items[i] << " ";
    }
    cout << endl;
}

// Returns the current size of the list
int listSize() const {
    return currentSize;
}
```

Retrieves the item at a specific index

```
int retrieve(int index) const {
    if (index >= 0 && index < currentSize) {
        return items[index];
    } else {
        cerr << "Index out of range." << endl;
        return -1; // Could also throw an exception
    }
}
```

// cerr is an output stream in C++ that's used to output **error messages**.

Searches for an item and returns its index

```
int search(int item) const {
    for (int i = 0; i < currentSize; ++i) {
        if (items[i] == item) {
            return i;
        }
    }
    return -1;
}
```

Replaces the item at a specific index with a new item

```
void replace(int index, int newItem) {
        if (index >= 0 && index < currentSize) {
            items[index] = newItem;
        } else {
            cerr << "Index out of range." << endl;
        }
    }

};
```

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

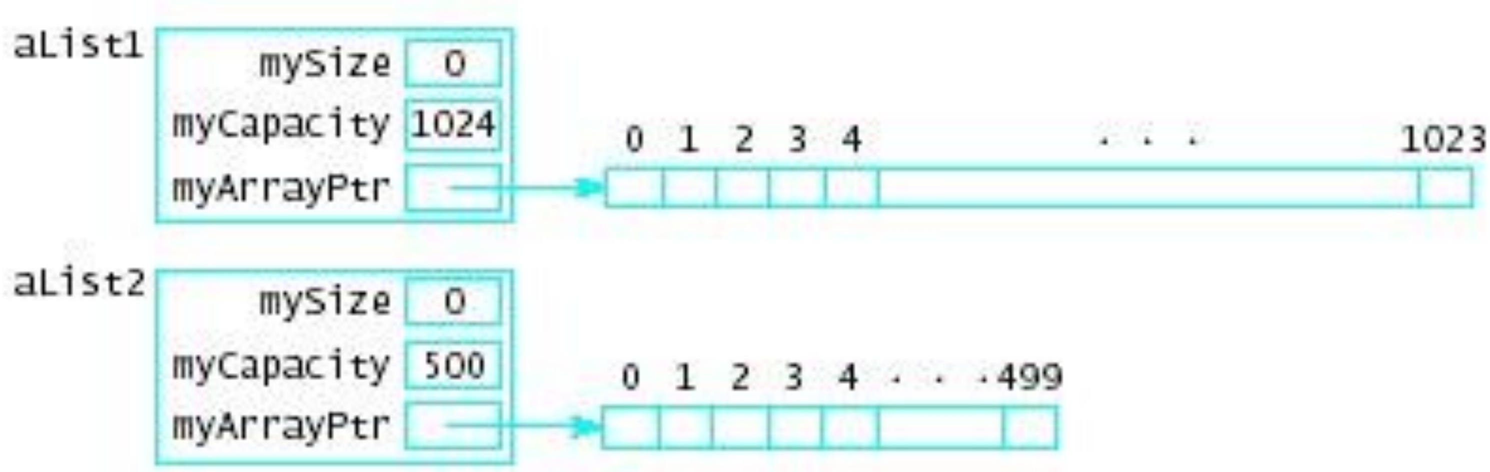
Dynamic Allocation of List Class

- 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

Now possible to specify different sized lists

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



Dynamic Allocation of List Class

```
class List {
private:
   int* items;         // Pointer to dynamically allocated array
   int currentSize;     // Current number of items in the list
   int maxSize;        // Maximum capacity of the list

public:
   // Constructor with a default size
   List(int size = 10) : maxSize(size), currentSize(0) {
      items = new int[maxSize];  // Dynamically allocate array of given size
   }

   // Destructor to free dynamically allocated memory
   ~List() {
      delete[] items;
   }
```

Any Question So Far?