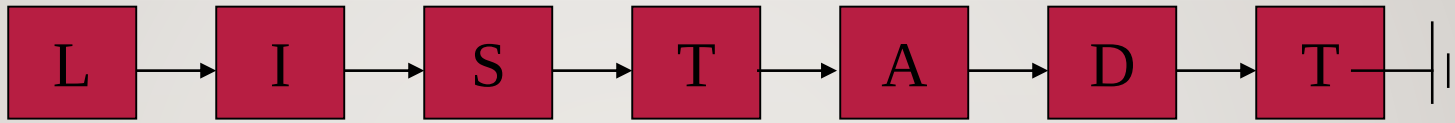


LIST ADT & LINKED LISTS

NATIONAL UNIVERSITY OF TECHNOLOGY (NUTECH)

DR. SAMAN RIAZ

LECTURE # 7

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

CONSIDER EVERYDAY LISTS

- Groceries to be purchased
- Job to-do list
- List of assignments for a course

- Can you name some others??

LIST

A ***Flexible*** structure, because 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 (say element_type)

Represented by a comma-separated sequence of elements:

$a_1, a_2, \ldots a_n$

Where,

n >= 0 and each a_i is of type element_type.

List

if n>= 1,

a_1 is the ***first*** element

a_n is the ***last*** element

if n = 0,

we have an empty list

List

The elements of a list can be ***linearly*** ordered.

$\Rightarrow a_i$ ***precedes*** a_{i+1} for i = 1,2,3…n-1

a_i ***follows*** a_{i-1} for i = 2,3,4…n

The 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

- Construct an empty list
- Determine whether or not empty
- Insert an element into the list
- Delete an element from the list
- Traverse (iterate through) the list to
  - Modify
  - Output
  - Search for a specific value
  - Copy or save
  - Rearrange

Basic Operations

1. **INSERT(*x,p,L*):** Insert x at position p in list L. If list L has no position p, the result is undefined.
2. **DELETE(*p,L*):** Delete the element at position p on list L.
3. **MAKENULL(*L*):** Causes L to become an empty list and returns position END(L).
4. **PRINTLIST(*L*):** Print the elements of L in order of occurrence.
5. **LOCATE(*x,L*):** Return the position of x on list L.
6. **RETRIEVE(*p,L*):** Return the element at position p on list L.

Basic Operations

6. **NEXT(*p,L*):** Return the position following p on list L.
7. **PREVIOUS(*p,L*):** Return the position preceding position p on list L.
8. **FIRST(*L*):** Returns the first position on the list L.

LIST AS A DATA STRUCTURE

- We know the ADT of the list, how to implement it?
- Consider a List class, it should contain at least the following function members
  - Constructor
  - `isEmpty()`
  - `insert()`
  - `delete()`
  - `display()`
- Implementation involves
  - Defining data members
  - Defining function members from design phase

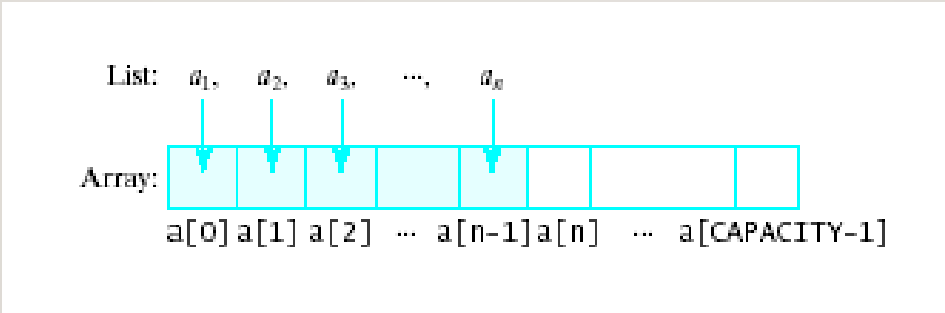
LIST AS A DATA STRUCTURE

- In terms of implementation, there are two basic approaches
  - Array-Based Implementation of Lists
  - Linked-List using Pointers based implementation of Lists

ARRAY-BASED IMPLEMENTATION OF LISTS

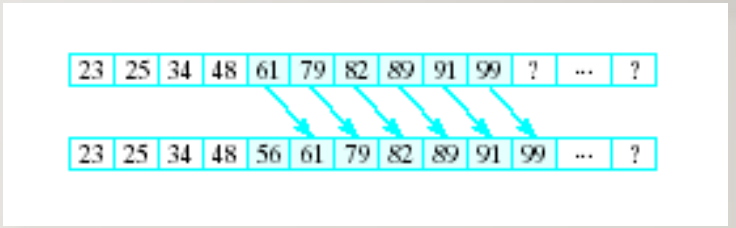
ARRAY-BASED IMPLEMENTATION OF LISTS

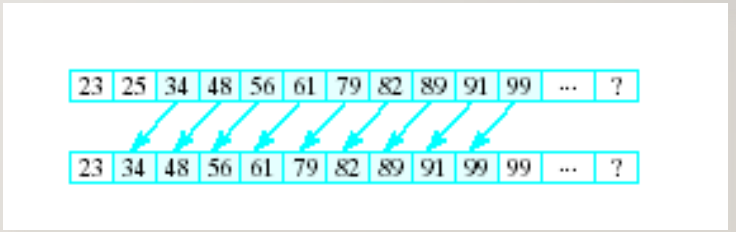
- An array is a viable choice for storing list elements
  - Element are sequential
  - It is a commonly available data type
  - Algorithm development is easy
- Normally sequential orderings of list elements match with array indices



IMPLEMENTING OPERATIONS

- Constructor
  - Static array allocated at compile time
- isEmpty
  - Check if size == 0
- Traverse
  - Use a loop from 0th element to `size - 1`
- Insert
  - Shift elements to right of insertion point
- Delete
  - Shift elements back





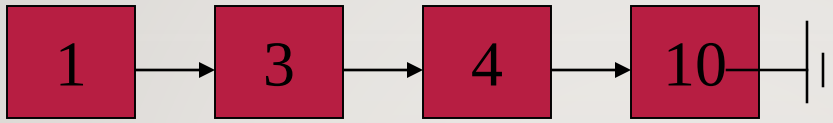
INEFFICIENCY OF ARRAY-IMPLEMENTED LIST

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

So …

We look for an alternative implementation.

AN ALTERNATIVE IMPLEMENTATION



Need a start link.

start

I	data[I]	next[I]
0	3	6
1	*	*
2	1	0
3	10	-1
4	*	*
5	*	*
6	4	3

end

How to insert, delete, and append?

LIST CLASS WITH STATIC ARRAY - PROBLEMS

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