DATA STRUCTURES AND ALGORITHMS

DR SAMABIA TEHSIN

BS (AI)

Abstract Data Types

Typical operations on data

- Add data to a data collection
- Remove data from a data collection
- Ask questions about the data in a data collection

Abstract Data Types

Data abstraction

- Asks you to think *what* you can do to a collection of data independently of *how* you do it
- Allows you to develop each data structure in relative isolation from the rest of the solution

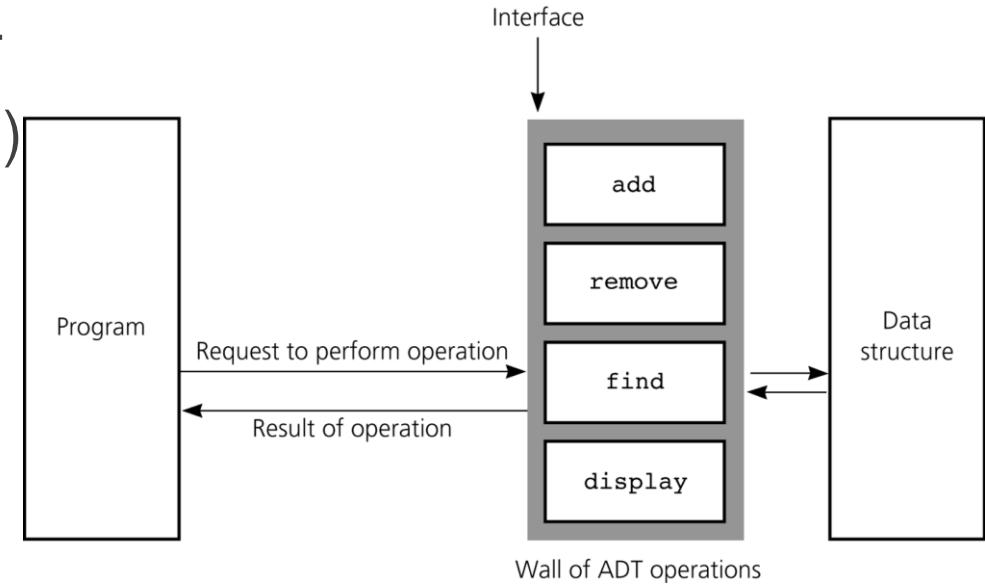
Abstract Data Types

An abstract data type (ADT) is composed of

- A collection of data
- A set of operations on that data

Specifications of an ADT indicate what the ADT operations do (but not how to implement them)

Implementation of an ADT includes choosing a particular data structure



The ADT List

Except for the first and last items, each item has a unique predecessor and a unique successor

Head (or front) does not have a predecessor

Tail (or end) does not have a successor

The ADT List

Items are referenced by their position within the list

Specifications of the ADT operations

- Define the contract for the ADT list
- Do not specify how to store the list or how to perform the operations

ADT operations can be used in an application without the knowledge of how the operations will be implemented

The ADT List

ADT List operations

- Create an empty list
- Determine whether a list is empty
- Determine the number of items in a list
- Add an item at a given position in the list
- Remove the item at a given position in the list
- Remove all the items from the list
- Retrieve (get) item at a given position in the list

Implementing ADTs

Choosing the data structure to represent the ADT's data is a part of implementation

- Choice of a data structure depends on
  - Details of the ADT's operations
  - Context in which the operations will be used

Implementation details should be hidden behind a wall of ADT operations

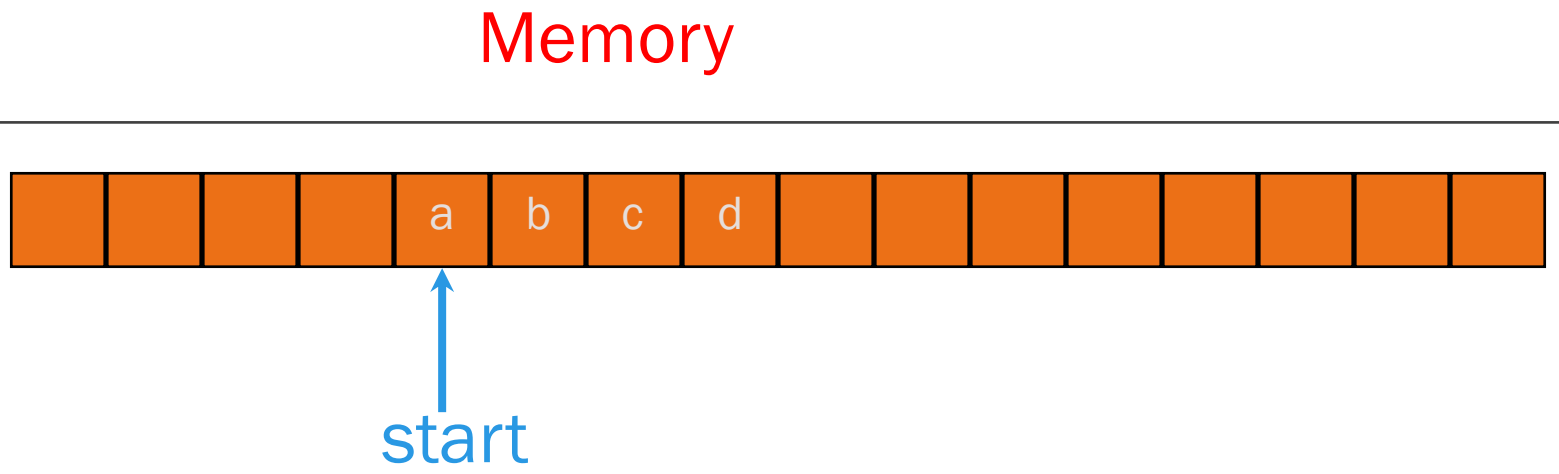
- A program would only be able to access the data structure using the ADT operations

An Array-Based ADT List

A list's items are stored in an array `items`

Both an array and a list identify their items by number

Array



Storing data in a sequential memory locations

Access each element using integer index

Very basic, popular, and simple

int a[10]; int *a = new int(10);

An Array-Based ADT List

A list's k^{th} item will be stored in `items[k-1]`

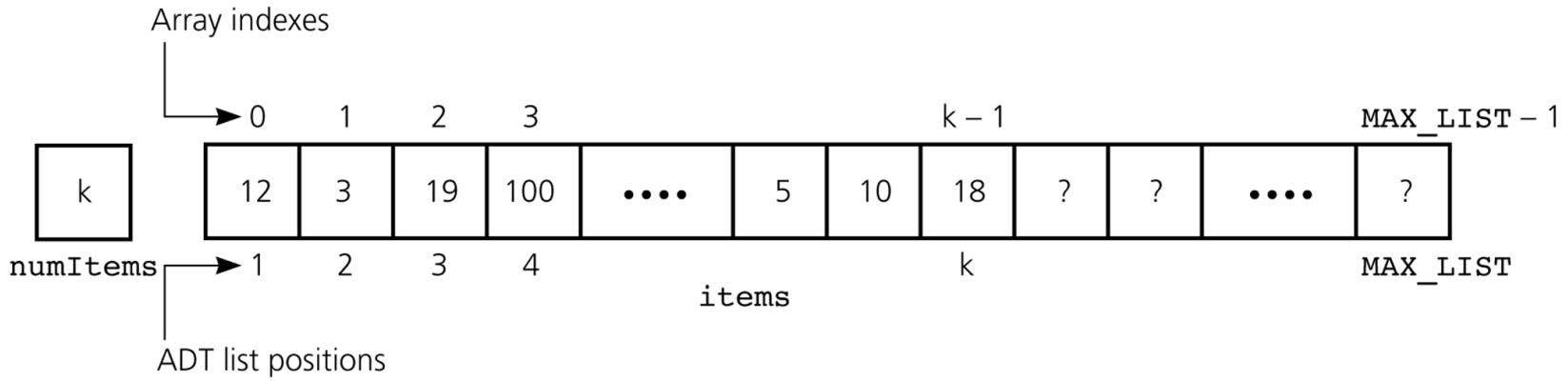


Figure 3.11 An array-based implementation of the ADT list

Array-Based ADT List Implementation

```
// ************************************
// Header file ListA.h for the ADT list
// Array-based implementation
// ************************************

const int MAX_LIST = maximum-size-of-list;
typedef desired-type-of-list-item ListItemType;


class List{

public:
   List();  // default constructor
            // destructor is supplied by
            // compiler
```

Array-Based ADT List Implementation

```
// list operations:
```

```
bool isEmpty() const;
// Determines whether a list is empty.
// Precondition: None.
// Postcondition: Returns true if the
// list is empty; otherwise returns
// false.
```

Array-Based ADT List Implementation

```
int getLength() const;
// Determines the length of a list.
// Precondition: None.
// Postcondition: Returns the number of
// items that are currently in the list.
```

Array-Based ADT List Implementation

```
bool retrieve(int index,
              ListItemType& dataItem) const;
// Retrieves a list item by position.
// Precondition: index is the number of
// the item to be retrieved.
// Postcondition: If 1 <= index <=
// getLength(), dataItem is the value of
// the desired item and the function
// returns true. Otherwise it returns
// false.
```

Array-Based ADT List Implementation

```
bool insert(int index,
            ListItemType newItem);
// Inserts an item into the list at
// position index.
// Precondition: index indicates the
// position at which the item should be
// inserted in the list.
// Postcondition: If insertion is
// successful, newItem is at position
// index in the list, and other items are
// renumbered accordingly, and the
// function returns true; otherwise  it
// returns false.
// Note: Insertion will not be successful
// if index < 1 or index > getLength()+1.
```

Array-Based ADT List Implementation

```
bool remove(int index);
// Deletes an item from the list at a
// given position.
// Precondition: index indicates where
// the deletion should occur.
// Postcondition: If 1 <= index <=
// getLength(), the item at position
// index in the list is deleted, other
// items are renumbered accordingly, and
// the function returns true; otherwise
// it returns false.
```

Array-Based ADT List Implementation

```
private:
   ListItemType items[MAX_LIST];
   // array of list items

   int size;
   // number of items in list


   int translate(int index) const;
   // Converts the position of an item in a
   // list to the correct index within its
   // array representation.

}; // end List class
```

Array-Based ADT List Implementation

```
// ************************************
// Implementation file ListA.cpp for the ADT
// list
// Array-based implementation
// ************************************

#include "ListA.h"  //header file


List::List() {
Size=0;

} // end default constructor
```

Array-Based ADT List Implementation

```
bool List::isEmpty() const{

   return size == 0;

} // end isEmpty
```

Array-Based ADT List Implementation

```
int List::getLength() const{

   return size;

} // end getLength
```

Array-Based ADT List Implementation

```
int List::translate(int index) const{

    return index - 1;

} // end translate
```

Array-Based ADT List Implementation

```
bool List::retrieve(int index,
                    ListItemType& dataItem) const{

   bool success = (index >= 1) && (index <= size);

   if (success)
      dataItem = items[translate(index)];

   return success;

} // end retrieve
```

Array-Based ADT List Implementation

```
bool List::insert(int index, ListItemType newItem){

   bool success = (index >= 1) &&
                  (index <= size + 1) &&
                  (size < MAX_LIST);

   if (success){
      // make room for new item by shifting all
      // items at positions >= index toward the end
      // of the list (no shift if index == size+1)
      for (int pos = size; pos >= index; --pos)
         items[translate(pos+1)] = items[translate(pos)];

      // insert new item
      items[translate(index)] = newItem;
      ++size;  // increase the size of the list by one
   }
   return success;
} // end insert
```

Array-Based ADT List Implementation

```
bool List::remove(int index){

   bool success = (index >= 1) && (index <= size) ;

   if (success){
      // delete item by shifting all items at
      // positions > index toward the beginning
      // of the list (no shift if index == size)
      for (int fromPosition = index+1;
               fromPosition <= size;
               ++fromPosition)
         items[translate(fromPosition-1)] =
                  items[translate(fromPosition)];

      --size;  // decrease the size of the list by one
   }

   return success;

}  // end remove
```

Array: Problems

New insertion and deletion: difficult

- Need to shift to make space for insertion
- Need to fill empty positions after deletion

Why don’t we connect all elements just “logically” not “physically”?

- Linked List

Credits and Acknowledgements

Lectures by Prof. Yung Yi, KAIST, South Korea.

Lectures by **Selim Aksoy**, Bilkent University, Ankara, Turkey