Chapter 3

ADT Unsorted List

Outline

• Unsorted List
• Array-based Implementation
• Linked Implementation
• Comparison
Lists

• **Linear relationship**
  – Each element except the first has a unique predecessor, and each element except the last has a unique successor

• **Length**
  – The number of items in a list; the length can vary over time

• **Unsorted list**
  – A list in which data items are placed in no particular order; the only relationships between data elements are the list predecessor and successor relationships

Lists

• **Sorted list**
  – A list that is sorted by the value in the key; there is a semantic relationship among the keys of the items in the list

• **Key**
  – The attributes that are used to determine the logical order of the list

*Name some possible keys*
ADT Unsorted List

• **Logical Level**

• **Abstract Data Type (ADT)**
  – A data type whose properties (domain and operations) are specified independently of any particular implementation

---

Can you think of what operations we should provide for our ADT Unsorted List?

---

ADT Unsorted List

• **Transformers**
  – MakeEmpty
  – InsertItem
  – DeleteItem

• **Observers**
  – IsFull
  – GetLength
  – RetrieveItem

• **Iterators**
  – ResetList
  – GetNextItem

change state

observe state

process all
ADT Unsorted List

• Client is responsible for error checking.
  – Precondition of each function enforces it.
  – Provide clients the tools with which to check for the conditions.
    • Observers

ADT Unsorted List

• **Generic data type**
  – A type for which the operations are defined but the types of the items being manipulated are not defined

  – *How can we make the items on the list generic?*

  – List items are of class **ItemType**, which has a **ComparerTo** function that returns **LESS**, **GREATER**, **EQUAL**
    • **ComparerTo** function compares the keys of items.
Class ItemType

```cpp
//  SPECIFICATION FILE ItemType.h
#ifndef ITEMTYPE_H
#define ITEMTYPE_H

const int MAX_ITEM = 5;
enum RelationType { LESS, EQUAL, GREATER };

class ItemType // declares class data type
{
public:
    ItemType(); // 4 public member functions
    RelationType ComparedTo( ItemType ) const{..}
    void Print() const{..}
    void Initialize( int value ){...}

private:
    int value; // 1 private data member
}; // could be any type
#endif
```

ADT Unsorted List

- Implementation Level

```
Array
[0] x
[1] x
[2] x
[3] x
[4] x
[5] x

Linked List
LIr
```

Two implementations
// SPECIFICATION FILE
#include "ItemType.h"
class UnsortedType // declares a class data type
{
  public:
    UnsortedType();
  
    void MakeEmpty();       // 8 public member functions
    bool IsFull() const;
    bool IsEmpty() const;
    int GetLength() const;  // returns length of list
    void RetrieveItem( ItemType& item, bool& found );
    void InsertItem( ItemType item );
    void DeleteItem( ItemType item );
    void ResetList();
    void GetNextItem( ItemType& item );

  private:
    int length;
    ItemType info[MAX_ITEMS];
    int currentPos;
};

Public declarations are the same for either implementation; only private data changes

What is ItemType?

Array-Based Implementation
Private data members for array-based implementation

private:
  int length;
  ItemType info[MAX_ITEMS];
  int currentPos;
};

Where does MAX_ITEMS come from?
Array-Based Implementation

- Notice the difference between the array and the list stored in an array.

Array-based implementation

The array ends at the slot with index \( \text{MAX\_ITEMS} - 1 \); the list ends in the slot with index \( \text{length} - 1 \).

ADT Unsorted List

- **Constructor**
  - A special member function of a class that is implicitly invoked when a class object is defined

\[
\text{UnsortedType}();
\]

- **What should the constructor do?**

\[
\text{UnsortedType}:\text{UnsortedType}()
\begin{cases}
    \text{length} = 0;
\end{cases}
\]
Array-Based Implementation

• *What is a full list? An empty list?*

//pre: List has been initialized
//post: function value = (list is full)
bool UnsortedType::IsFull() const
{
    return (length == MAX_ITEM);
}

//pre: List has been initialized
//post: function value = (list is empty)
bool UnsortedType::IsEmpty() const
{
    return (length == 0);
}

//pre: List has been initialized
//post:
int UnsortedType::GetLength() const
{
    return length ;
}
Array-Based Implementation

If the list is unsorted, where should the next element go?

insert("Hsing");

That was easy! Can you code it?
Array-Based Implementation

//pre: List has been initialized. List is not full, item is not in list.
// Post: item is in the list.

void UnsortedType::InsertItem(ItemType item)
{
    info[length] = item;
    length++;
}

How would you go about finding an item in the list?

- Cycle through the list looking for the item

What are the two ending cases?

- The item is found
- The item is not in the list

How do we compare items?

We use function ComparedTo in class ItemType (compare keys of items)
Array-Based Implementation

// Pre: Key member(s) of item is initialized.
// Post: If found, item’s key matches an element’s key in the list and a copy of that element has been stored in item; otherwise, item is unchanged.

void UnsortedType::RetrieveItem(ItemType& item, bool& found)
{
    found = false;
    for (int i=0; i<length; i++)
    {
        if (item.ComparedTo(info[i]) == EQUAL)
        {
            item = info[i];
            found = true;
            break;
        }
    }
}

Array-Based Implementation

How do you delete an item?

First you find the item

Yes, but how do you delete it?

• Move those below it up on slot
• Replace it with another item

* Tips: This is an unsorted list.

What other item?

How about the item at info[length-1]?
Array-Based Implementation

// Pre: item's key has been initialized.
// One and only one element in the list has a key that
// matches item's.
// Post: No element in the list has a key that matches item's.
void UnsortedType::DeleteItem ( ItemType item )
{
    int location = 0;

    while (item.ComparedTo (info[location]) != EQUAL)
        location++;

    // move last element into position where item was located
    info [location] = info [length - 1] ;
    length-- ; //the length of the list is decreased
}
Array-Based Implementation

Key Bradley has been matched

Copy of last list element is in the position where the key Bradley was before; length has been decremented
Array-Based Implementation

• What will happen if the matching element is the last element?
• What will happen if there is only one element in the list?
• Do we worry about the empty List?
• What if the elements are pointers and they points to dynamically allocated memory?

```cpp
// Pre: N/A
// Post: the list is empty
bool UnsortedType::MakeEmpty()
{
    length = 0;
}
• We do not have to do anything to the array that holds the list items to make a list empty.
```
Array-Based Implementation

void UnsortedType::ResetList()
{
    currentPos = -1;
}

void UnsortedType::GetNextItem(ItemType& item)
{
    currentPos++;
    item = info[currentPos];
}

• Application Level
  • How to use the List implemented?
    – PrintList(UnsortedType&)
    – CreateFromFile(ifstream &, UnsortedType&)
  • Users do not need to know how the list is implemented.
Testing Array-Based Implementation

// Pre: list has been initialized.
// Post: Each component in list has been written.

void PrintList(UnsortedType & list)
{
    int length;
    ListItem item;

    list.ResetList();
    length = list.GetLength();
    for (int counter = 1; counter <= length; counter++)
    {
        list.GetNextItem(item);
        item.Print();
    }
}

Test Plan

• Testing Strategy
  – Combination of black-box and clear-box.
  – Black-box : test precondition & postcondition.
  – Clear-box : test the code inside the functions, try path testing.

• Values for testing
  – End cases.
Data Encapsulation

- Information internal to the implementation of the Unsorted List ADT.
  - Private data members
    - The array `info[ ]`
    - The index of the iterator `currentPos`
  - Local variable `location`, which contains the array index of the list item being processed.

Reference

- Reproduced by permission of Jones and Bartlett Publishers International.