Chapter 3

ADT Unsorted List

Outline

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

Be sure you understand the differences among location, *location, and location->info.
struct NodeType

#include "ItemType.h"
struct NodeType{
    ItemType info;
    NodeType * next;
};

#include "ItemType.h"
struct NodeType{
    ItemType info;
    NodeType * next;
};

class UnsortedType // declares a class data type
{
public:
    UnsortedType();
    ~UnsortedType(); // 8 public member functions
    void MakeEmpty();
    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 );
Linked Implementation

• Private data for the Unsorted List ADT, linked-list implementation

```cpp
private:
    NodeType *listData;
    int length;
    NodeType *currentPos;
};
```

How do you know that a linked list is empty?

- `listData` is NULL

What should the constructor do?

- Set `length` to 0
- Set `listData` to NULL

What about `currentPos`?

- We let `ResetList()` take care of initializing
  `currentPos`

You write the constructor
Linked Implementation

What about the observers IsFull() and GetLength()?
GetLength() just returns length

Can a linked list ever be full?
Yes, if you run out of memory
Ask for a new node within a try/catch

```cpp
bool Unsortedtype::IsFull() const
{
    NodeType* location;
    try
    {
        location = new NodeType;
        delete location;
        return false;
    }
    catch (std::bad_alloc e)
    {
        return true;
    }
}
```
Linked Implementation

**Initialize location to position of first item**
Set found to false
Set moreToSearch to *(have not examined Info(last))*
while moreToSearch AND NOT found
  if item.ComparedTo(Info(location)) is EQUAL :
    Set found to true
    Set item to Info(location)
  else :
    Set location to Next(location)
    Set moreToSearch to *(have not examined Info(last))*

*RetrieveItem()*

Linked Implementation

*Remember the design notation?*

Node(location)  *location*
Info(location)  location->info
Next(location)  location->next

*How do you set location to Next(location)??*
*How do you set Info(location) to value??*
Linked Implementation

// Pre: Key member 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)
{
    bool moreToSearch;
    NodeType* location = listData;
    found = false;
    moreToSearch = (location != NULL);
    while (moreToSearch && !found)
    {
        if (item.ComparedTo(location->info) == EQUAL) // match here
        {
            found = true;
            item = location->info;
        }
        else // advance pointer
        {
            location = location->next; moreToSearch = (location != NULL);
        }
    }
}
Linked Implementation

How do we go about building a linked list?

Create a list of one item

1. Get a node using new
   ```
   location = new NodeType;
   ```
2. Put value into info portion of node
   ```
   location->info = item;
   ```
3. Put pointer to node in external pointer
   ```
   listData = location;
   ```

We forgot something: We must put NULL in the Next position of the first node

```
Location->next=NULL;
```
Linked Implementation

How do we add a node to our list?

1. Get a node using `new`
   ```
   location = new NodeType;
   ```
2. Put value into info portion
   ```
   location->info = Item;
   ```
3. Put node into list …

Where? Where should the node go? Does it matter?

Now we must put the two parts together--carefully!
Linked Implementation

```
// Pre: list is not full and item is not in list.
// Post: item is in the list; length has been incremented.
void UnsortedType::InsertItem ( ItemType item )
{
    NodeType * location;
    // obtain and fill a node
    location = new NodeType;
    location->info = item;
    location->next = listData;
    listData = location;
    length++;
}
```

Where is the item inserted? Head or End? Why?
**Linked Implementation**

*How do you delete an item from the list?*

1. Find the item
2. Remove the item

```c
NodeType* location = listData;
NodeType* tempLocation;
// Find the item
if (item.ComparedTo(listData->info) == EQUAL)
    // item in first location
    tempLocation = location;
    listData = listData->next; //move the head of the list
else{
    while (item.ComparedTo((location->next)->info) != EQUAL)
        location = location->next;
    tempLocation = location->next;
    location->next = (location->next)->next;
    //remove the middle one, connect the two ends
}
delete tempLocation;
length--;
```

Pre: the item is in the list!!
Linked Implementation

```cpp
void UnsortedType::MakeEmpty()
{
    NodeType *tempPtr;
    while (listData != NULL)
    {
        tempPtr = listData;
        listData = listData->next;
        delete tempPtr;
    }
    length = 0;
}
```

Why can we just set `listData` to `NULL`?

Linked Implementation

**ResetList()** and **GetNextItem()**

*What was `currentPos` in the array-based implementation?*

*What would be the equivalent in a linked implementation?*
Linked Implementation

```cpp
void UnsortedType::ResetList()
{
    currentPos = NULL;
}

void UnsortedType::GetNextItem(ItemType& item)
{
    if (currentPos == NULL)
        currentPos = listData;
    else
        currentPos = currentPos->next;

    item = currentPos->info;
}
```

What is the Precondition & Postcondition on GetNextItem?

Class Destructors

- Recall: listData is deallocated when it goes out of scope but what listData points to is not deallocated
- **Class Destructor**
  - A function that is implicitly called when class object goes out of scope
```cpp
~UnsortedType(); // Destructor
```

What must this function do? Have we done a similar job?
Which One Is Better?

- Array-based Or Linked List?
- How to Compare them?

*how much our implementation costs us in terms of *time, space (memory) !*

Algorithm Growth Rates

- An algorithm’s time requirements can be measured as a function of the problem size N
  - Number of nodes in a linked list
  - Size of an array
- Algorithm’s growth rate: enables comparison of one algorithm with another
- Algorithm efficiency is typically a concern for large problems only
The Story of Big O

Big O (order of magnitude) is a measure of the “worst-case” scenario for a given algorithm taking in input of size N.

Mathematically:

Given an input of size N (where N is really really really big), what is the asymptotic upper bounds of the algorithm in terms of time/space (how long will it take to execute/how much memory will it require)?

Algorithm Growth Rates

- Algorithm A requires time proportional to $n^2$
- Algorithm B requires time proportional to $n$

Figure 9-1: Time requirements as a function of the problem size $n$
O(?)

There are a few common notations for Big-O. These are categories into which almost all algorithms fit into (once we throw away all the garbage and reduce the algorithm to its purest form). These are denoted as follows:

- $O(1)$
- $O(\log_2 N)$
- $O(N)$
- $O(N^M)$
- $O(M^N)$
- Or some combination of these.

O(1): What A Dream!

- $O(1)$ doesn’t mean
  - we finish everything in one second or one CPU cycle
  - use one-element array or linked list
- We just wish that it did
O(1): What It Is About

- **O(1)** means *constant time or space*. No matter what we throw at a particular algorithm, we know EXACTLY how long/how much it will take to execute the program:
  - Given an array of size \( N \) (where \( N \) is unknown), access a particular value.
  - Given a list of size \( P \) (where \( P \) is unknown), retrieve the first element of the list.

O(N): Linear time/space

- This one’s pretty easy to see.
- If we have a list containing \( N \) elements, and we are searching for a particular value that isn’t even contained in the list, how many elements do we have to search through?
  
  \[ \text{N elements!} \]
Space

• how much memory space is needed:
  – Array-based: max. size of the array. O(N_{Max})
  – Linked-List: number of elements actually in the list at run time. O(n)
• However, each node element of linked list is larger than array element because the pointer to next member is stored as well.

Time

Which array-based operations are O(1)?
Which linked operations are O(1)?
Which array-based operations are O(N)?
Which linked operations are O(N)?
Can you say which implementation is better?
Time - O(1)

- Array-based
  - IsFull()
  - GetLength()
  - isEmpty()
  - InsertItem()
  - MakeEmpty()
  - ResetList()
  - GetNextItem()

- Linked
  - IsFull()
  - GetLength()
  - isEmpty()
  - InsertItem()
  - ResetList()
  - GetNextItem()

Time - O(N)

- Array-based
  - RetrieveItem()
  - DeleteItem()

- Linked
  - RetrieveItem()
  - DeleteItem()
  - MakeEmpty()
Reference


• Reproduced by permission of Jones and Bartlett Publishers International.