Chapter 17
Linked Lists
ADT: Abstract Data Types

- A **data structure** is a particular way of organizing data in a computer so that it can be used efficiently
  - A university needs to manage a lot of individual students
  - An RPG needs to manage many different characters
  - Working with individual objects is inefficient
  - Group objects together in a data structure

- What data structures do you know?
  - Arrays
  - STL vectors
ARRAYS

What’s wrong with arrays?
- **Static:** need to know the size before you run the code
- **Dynamic:** can choose the size at runtime

Can you resize an array?
- Not directly
- You can create another array of a different size
- Copy all elements from old array to new array
- Delete the old array

What if you have 1,000,000 entries in your array, and you need to increase the size of array by 1?
What about vectors?

- Vectors are resized automatically
- Bad news: Vectors are not magical
- STL vectors are implemented using dynamic arrays
- Therefore, they are just as inefficient

What if you want to insert or delete an element in the middle of a vector/array?

- Have to shift the remaining elements one position to the right or left
- What if you have 1 000 000 entries in your array, and you insert a new element at position 0?
Inserting an element into an array

18

want to insert
18 at position 2

5 2 7 -5 16 2 ？ ？ ？ ？ ？ ？

© Pearson Education Limited 2015
Inserting an element into an array

Want to insert 18 at position 2

Shift these elements one position right
INSERTING AN ELEMENT INTO AN ARRAY

put 18 to the position 2
DELETING AN ELEMENT FROM AN ARRAY

want to remove element at the position 3

5 2 7 -5 16 2 ?? ?? ?? ?? ??
DELETING AN ELEMENT FROM AN ARRAY

want to remove element at the position 3

shift these elements one position left
DELETING AN ELEMENT FROM AN ARRAY

element has been successfully removed

5 2 7 16 2 ？ ？ ？ ？ ？ ？ ？
Arrays / STL Vectors

To summarise:
- Increasing the size of an array/vector is not very efficient
- Adding/removing elements from the middle of an array/vector is very inefficient

Can we come up with a data structure that does not have these deficiencies?

Yes: Linked Lists
Introduction to the Linked List ADT

- **Linked list**: set of data structures (nodes) that contain references to other data structures
- Every **node** has two components:
  - Data item
  - Pointer to the next node

```
list head
```

```
null
```
Introduction to the Linked List ADT

- A list of linked nodes
- Nodes can be added to or removed from the linked list during execution
- Linked list needs to store the address of the first node only (list head)
- Every node knows how to get to the next node
- Last node points to NULL
Linked Lists vs. Arrays and Vectors

- Linked lists can grow and shrink as needed, unlike arrays, which have a fixed size.
- Linked lists can insert a node between other nodes easily.
- Just update two pointers, and you’re done!
NODE ORGANIZATION

- A node contains:
  - **data**: one or more data fields – may be organized as structure, object, etc.
  - a **pointer** that can point to another node
  - A node is like an advanced array element that knows the address of the next element
Linked List Organization

- Linked list contains 0 or more nodes:

![Diagram of a linked list with a head pointing to the first node and the last node pointing to null](null)

- Has a list **head** to point to **first node**
- Last node must point to **null (address 0)**
**EMPTY LIST**

- If a list currently contains 0 nodes, it is an **empty list**
- In this case the list **head** points to **null**

![Diagram of an empty list with head pointing to null]
DECLARING A NODE

- Declare a node:

```c
struct ListNode
{
    double data; // data item
    ListNode * next; // address of // the next node
};
```

- No memory is allocated at this time
- By the way: what is the difference between a struct and a class in C++?
Defining a Linked List

- Define a pointer for the head of the list:

  ```
  ListNode * head = NULL;
  ```

- Head pointer initialized to NULL indicates an empty list

```
head

null
```
**The Null Pointer**

- Is used to indicate end-of-list
- Should always be tested for before using a pointer:

```c
ListNode * p;
// Store data in p
if (p != NULL) {
   // haven’t reached end of the list yet
   ...
}
```
**Linked List Operations**

- How do you use a linked list?
  - Add data
  - Remove data
  - Iterate through the data in the list

- Basic linked list operations (terminology):
  - append a node to the end of the list
  - insert a node within the list
  - traverse the linked list
  - delete a node
  - delete/destroy the list
**NUMBERLIST: A LINKED LIST OF DOUBLES**

class NumberList
{
  private:
    struct ListNode // node structure
    {
      double value;  // The value in this node
      struct ListNode *next;  // Address of the next node
    }
    ListNode *head;  // List head pointer

  public:
    NumberList() { head = NULL; } // constructor: initialize head to 0
    ~NumberList(); // destructor

    // Linked list operations:
    void appendNode(double); // add node at the end of the list
    void insertNode(double); // insert a node (maintain ascending order)
    void deleteNode(double); // remove a node
    void displayList() const; // print list on the screen
};
CREATE A NEW NODE

- When data is added to the list, a new node is created to contain the data
  
1) Allocate memory for the new node:
   
   ```
   newNode = new ListNode;
   ```

2) Initialize the contents of the node:
   
   ```
   newNode->value = num;
   ```

3) Set the pointer field to NULL:
   
   ```
   newNode->next = NULL;
   ```
Appending a Node

- **Append:** Add a node to the end of the list
- **Basic process:**
  - Create the new node (as already described)
  - Add node to the end of the list:
    - If list is empty, set head pointer to this node
    - Else,
      - Iterate to the end of the list
      - Set pointer of last node to point to new node
APPENDING A NODE

New node created, end of list located
Appending a Node

New node added to end of list
C++ code for Appending a Node

```cpp
11  void NumberList::appendNode(double num)  
12  {
13      ListNode *newNode;       // To point to a new node
14      ListNode *nodePtr;  // To move through the list
15
16      // Allocate a new node and store num there.
17      newNode = new ListNode; // must we use “new”?
18      newNode->value = num;    // set value to num
19      newNode->next = NULL;    // set next to NULL
20
21      // If there are no nodes in the list
22      // make newNode the first node:
23      if (!head)  // “If head is null”
24          head = newNode; // both are pointers
```
else  // Otherwise, insert newNode at end.
{
    // Initialize nodePtr to head of list.
    nodePtr = head;

    // Find the last node in the list.
    while (nodePtr->next)  // "while next != 0"
    
        nodePtr = nodePtr->next;
    
    // Insert newNode as the last node
    nodePtr->next = newNode;
}
// This program demonstrates a simple append
// operation on a linked list.
#include <iostream>
#include "NumberList.h"
using namespace std;

int main()
{
    // Define a NumberList object.
    NumberList list;

    // Append some values to the list.
    list.appendNode(2.5);
    list.appendNode(7.9);
    list.appendNode(12.6);
    return 0;
}

(This program displays no output.)