Standard Version of
Starting Out with C++, 4th Edition

Linked Lists
Topics

• Introduction to the Linked List ADT
• Linked List Operations
• A Linked List Template
• Variations of the Linked List
• The STL list Container
Introduction to the Linked List ADT

- **Linked list**: set of data structures (nodes) that contain references to other data structures

![Diagram of a linked list with null head]

Chapter 17 slide 3
Introduction to the Linked List ADT

- References may be addresses or array indices
- Data structures can be added to or removed from the linked list during execution
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

![Linked List Diagram]
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
Linked List Organization

- Linked list contains 0 or more nodes:
  
  ![Diagram of a linked list with a list head pointing to the first node and the last node pointing to NULL]

- Has a list head to point to first node
- Last node points to \texttt{NULL}
Empty List

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

```
list head

  NULL
```

Chapter 17 slide 8
Declaring a Node

• Declare a node:

```c
struct ListNode
{
    int data;
    ListNode *next;
};
```

• No memory is allocated at this time
Defining a Linked List

• Define a pointer for the head of the list:
  
  ```c
  ListNode *head = NULL;
  ```

• Head pointer initialized to `NULL` to indicate an empty list
**NULL Pointer**

- Is used to indicate end-of-list
- Should always be tested for before using a pointer:
  ```c
  ListNode *p;
  while (p != NULL) ...  
  ```
- Can also test the pointer itself:
  ```c
  while (!p) ... // same meaning
  // as above
  ```
Linked List Operations

• Basic operations:
  – 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
Create a New Node

• Allocate memory for the new node:
  
  ```
  newNode = new ListNode;
  ```

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

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

• 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,
      – traverse the list to the end
      – 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
Inserting a Node into a Linked List

• Used to maintain a linked list in order
• Requires two pointers to traverse the list:
  – pointer to locate the node with data value greater than that of node to be inserted
  – pointer to 'trail behind' one node, to point to node before point of insertion
• New node is inserted between the nodes pointed at by these pointers
Inserting a Node into a Linked List

New node created, correct position located
Inserting a Node into a Linked List

New node inserted in order in the linked list
Traversing a Linked List

• Visit each node in a linked list: display contents, validate data, etc.

• Basic process:
  – set a pointer to the contents of the head pointer
  – while pointer is not NULL
    • process data
    • go to the next node by setting the pointer to the pointer field of the current node in the list
  – end while
Traversing a Linked List

nodePtr points to the node containing 5, then the node containing 13, then the node containing 19, then points to NULL, and the list traversal stops.
Deleting a Node

- Used to remove a node from a linked list
- If list uses dynamic memory, then delete node from memory
- Requires two pointers: one to locate the node to be deleted, one to point to the node before the node to be deleted
Deleting a Node

previousNode nodePtr

list head

Locating the node containing 13

Chapter 17 slide 23
Deleting a Node

Adjusting pointer around the node to be deleted
Deleting a Node

Linked list after deleting the node containing 13

prevoiusNode nodePtr

list head

Chapter 17 slide 25
Destroying a Linked List

• Must remove all nodes used in the list
• To do this, use list traversal to visit each node
• For each node,
  – Unlink the node from the list
  – If the list uses dynamic memory, then free the node’s memory
• Set the list head to NULL
A Linked List Template

• When declaring a linked list, must specify the type of data to be held in each node

• Using templates, can declare a linked list that can hold data type determined at list definition time
Variations of the Linked List

• Other linked list organizations:

  – doubly-linked list: each node contains two pointers: one to the next node in the list, one to the previous node in the list
Variations of the Linked List

• Other linked list organizations:
  – circular linked list: the last node in the list points back to the first node in the list, not to \texttt{NULL}

```
list head
```

```
\begin{tikzpicture}
  \node (5) at (0,0) {5};
  \node (13) at (1,0) {13};
  \node (19) at (2,0) {19};
  \node (head) at (-0.5,0) {list head};
  \draw[->] (head) -- (5);
  \draw[->] (5) -- (13);
  \draw[->] (13) -- (19);
  \draw[->] (19) -- (head);
\end{tikzpicture}
```
The STL `list` Container

- Template for a doubly linked list
- Member functions for
  - locating beginning, end of list: `front`, `back`, `end`
  - adding elements to the list: `insert`, `merge`, `push_back`, `push_front`
  - removing elements from the list: `erase`, `pop_back`, `pop_front`, `unique`