Chapter 16

Linked Lists, Stacks, and Queues

Objectives

• To create nodes to store elements in a linked list (§16.2).
• To access the nodes in a linked list via pointers (§16.3).
• To declare a LinkedList class for storing and processing data in a list (§16.4).
• To implement adding an element to the head of a list (§16.4.1).
• To implement adding an element to the end of a list (§16.4.2).
• To implement inserting an element into a list (§16.4.3).
• To implement removing the first element from a list (§16.4.4).
• To implement removing the last element from a list (§16.4.5).
• To implement removing an element at a specified position in a list (§16.4.6).
• To know the variations of linked lists (§16.5).
• To implement the Stack class using a linked list (§16.6).
• To implement the Queue class using a linked list (§16.7).
16.1 Introduction
The preceding chapter introduced a generic Stack class. The elements in the stack are stored in an array. The array size is fixed. If the array is too small, the elements cannot be stored in the stack. If the array is too large, a lot of space will be wasted. A possible solution to fix this problem was proposed in §15.5, “Improving the Stack Class.” Initially, the stack uses a small array. When there is no room to add a new element, the stack creates a new array that doubles the size of the old array, copies the contents from the old array to this new array, and discards the old array. It is time-consuming to copy the array.

<Side Remark: linked list>
This chapter introduces a new data structure, called linked list. A linked list is efficient for storing and managing a varying number of elements. This chapter will also discuss how to implement stacks and queues using linked lists.

16.2 Nodes
In a linked list, each element is contained in a structure, called the node. When a new element is added to the list, a node is created to contain the element. All the nodes are chained through pointers, as shown in Figure 16.1.

![Figure 16.1](image)

A linked list consists of any number of nodes chained together.

Nodes can be defined using structs or classes in C++. Using classes are preferred. The class definition for a node can be as follows:

```cpp
template<typename T>
class Node
{
public:
    T element;  // Element contained in the node
    Node *next; // Pointer to the next node

    Node(); // No-arg constructor
};
```

```cpp
// No-arg constructor

Node() // No-arg constructor
{ 
    next = NULL;
}
```
Node(T element) // Constructor
{
    this->element = element;
    next = NULL;
}

Node is declared as a template class with a type parameter T for specifying the element type.

By convention, pointer variables named head and tail are used to point to the first node and last node in the list, as shown in Figure 16.1. They are declared as

    Node<string>* head, *tail;

If the list is empty, both head and tail should be NULL. NULL is a C++ constant for 0, which indicates that a pointer does not point to any node. The definition of NULL is in a number of standard library including <iostream> and <cstddef>. Here are the examples to add three strings to the list.

***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.***

.Side Remark line 2: create a node
.Side Remark line 7: new tail

```c++
// Create a node to store the first string
head = new Node<string>("Chicago");
tail = head;

// Create a node to store the second string
tail->next = new Node<string>("Denver");
tail = tail->next;

// Create a node to store the second string
tail->next = new Node<string>("Dallas");
tail = tail->next;
```

Each new node is dynamically created using the new operator. The process of creating a new linked list and adding three nodes is shown in Figure 16.2. Initially, the head and tail are NULL. When you add the first node, both head and tail point to this node. When you add the second node, it is pointed by the tail node’s next pointer. Since this new node becomes the last node in the list, tail now points to this new node. head still points to the same node, but tail now points to this new node (line 7).
Figure 16.2
Three nodes are added to a new linked list.

Each node contains the element and a pointer that points to the next element. If the node is the last in the list, its pointer data field next contains the value NULL. You can use this property to detect the last node. For example, you may write the following loop to traverse all the nodes in the list.

***PD: Please add line numbers in the following code***
***Layout: Please layout exactly. Don’t skip the space. This is true for all source code in the book. Thanks, AU.***

```cpp
Node<string> *current = head;
while (current != NULL)
{
    cout << current->element << endl;
    current = current->next;
}
```

The current pointer points to the first node in the list initially (line 1). In the loop, the element of the current node is retrieved (line 4), and then current points to the next node (line 5). The loop continues until the current node is NULL.

16.3 The LinkedList Class
A linked list is a popular data structure for storing data in sequential order. For example, a list of students, a list of available rooms, a list of cities, and a list of books can all be stored using lists. The operations listed below are typical of most lists:

- Retrieve an element from a list.
- Insert a new element to a list.
- Delete an element from a list.
- Find how many elements are in a list.
- Find whether an element is in a list.
- Find whether a list is empty.

Figure 16.3 gives the class diagram for LinkedList. LinkedList is a class template with type parameter T that represents the type of the elements stored in the list.
Figure 16.3

LinkedList implements a list using a linked list of nodes.

You can get an element from the list using get(int index). The index is 0-based, i.e., the node at the head of the list has index 0. Assume that the LinkedList class is available in the header file LinkedList.h. Let us begin by writing a test program that uses the LinkedList class in Listing 16.1. The program creates a list using LinkedList (line 18). It uses the add function to add strings to the list and the remove function to remove strings from the list.

Listing 16.1 TestList.cpp (Using LinkedList)

***PD: Please add line numbers (including space lines) in the following code***
<side remark line 3: include class>
<side remark line 6: print list>
<side remark line 8: list size>
<side remark line 10: get element>
<side remark line 18: create list>
<side remark line 21: append element>
<side remark line 23: invoke printList>
<side remark line 25: insert element>
<side remark line 29: append element>
<side remark line 33: append element>
<side remark line 37: insert element>
<side remark line 41: insert element>
<side remark line 45: insert element>
#include <iostream>
#include <string>
#include "LinkedList.h"
using namespace std;

void printList(LinkedList<string> list)
{
    for (int i = 0; i < list.getSize(); i++)
    {
        cout << list.get(i) << " ";
    }
    cout << endl;
}

int main()
{
    // Create a list for strings
    LinkedList< string > list;

    // Add elements to the list
    list.add("America"); // Add it to the list
    cout << "(1) ";
    printList(list);

    list.add(0, "Canada"); // Add it to the beginning of the list
    cout << "(2) ";
    printList(list);

    list.add("Russia"); // Add it to the end of the list
    cout << "(3) ";
    printList(list);

    list.add("France"); // Add it to the end of the list
    cout << "(4) ";
    printList(list);

    list.add(2, "Germany"); // Add it to the list at index 2
    cout << "(5) ";
    printList(list);

    list.add(5, "Norway"); // Add it to the list at index 5
    cout << "(6) ";
    printList(list);

    list.add(0, "Netherlands"); // Same as list.addFirst("Netherlands")
    cout << "(7) ";
    printList(list);

    // Remove elements from the list
    list.removeAt(0); // Same as list.remove("Australia") in this case
cout << "(8) ";
printList(list);

list.removeAt(2); // Remove the element at index 2
cout << "(9) ";
printList(list);

list.removeAt(list.getSize() - 1); // Remove the last element
cout << "(10) ";
printList(list);

return 0;

<Output>
(1) America
(2) Canada America
(3) Canada America Russia
(4) Canada America Russia France
(5) Canada America Germany Russia France
(6) Canada America Germany Russia France Norway
(7) Netherlands Canada America Germany Russia France Norway
(8) Netherlands Canada America Germany Russia France Norway
(9) Netherlands Canada Germany Russia France Norway
(10) Netherlands Canada Germany Russia France
<End Output>

16.4 Implementing LinkedList

Now let us turn our attention to implementing the LinkedList class. Some of functions are easy to implement. For example, the isEmpty() function simply returns head == NULL and the clear() function simply destroys all nodes in the list and set head and tail to NULL. The addLast(T element) function is same as the add(T element) function. The reason that both are defined is for convenience.

16.4.1 Implementing addFirst(T element)

The addFirst(T element) function can be implemented as follows:

***PD: Please add line numbers (including space lines) in the following code***

```cpp
<side remark line 4: create a node>
<side remark line 5: link with head>
<side remark line 6: head to it>
<side remark line 7: increase size>
<side remark line 9: was empty?>

```
The `addFirst(T element)` function creates a new node (line 4) to store the element and insert the node to the beginning of the list (line 5). After the insertion, `head` should point to this new element node (line 6), as shown in Figure 16.4.

![Diagram of a list before and after adding a new node](image)

**Figure 16.4**  
A new element is added to the beginning of the list.

If the list is empty (line 9), both `head` and `tail` will point to this new node (line 10). After the node is created, the size should be increased by 1 (line 7).

### 16.4.2 Implementing `addLast(T element)`

The `addLast(T element)` function creates a node to hold `element` and inserts the node at the end of the list. It can be implemented as follows:

```cpp
***PD: Please add line numbers (including space lines) in the following code***
<side remark line 6: create a node>
<side remark line 9: create a node>
<side remark line 13: increase size>

```}

```cpp
template<typename T>
void LinkedList<T>::addLast(T element) {
  if (tail == NULL) {
    head = tail = new Node<T>(element);
  } else {
    tail->next = new Node<T>(element);
    tail = tail->next;
  }
}
```
Consider two cases: (1) if the list is empty (line 4), both head and tail will point to this new node (line 6); (2), otherwise, insert the node at the end of the list (line 9). After the insertion, tail should refer to this new element node (line 10), as shown in Figure 16.5. In any case, after the node is created, the size should be increased by 1 (line 13).

**Figure 16.5**
A new element is added at the end of the list.

16.4.3 Implementing add(int index, T element)

The add(int index, T element) function adds an element to the list at the specified index. It can be implemented as follows:

***PD: Please add line numbers (including space lines) in the following code***

<side remark line 5: insert first>
<side remark line 7: insert last>
<side remark line 14: create a node>
<side remark line 16: increase size>

```cpp
template<typename T>
void LinkedList<T>::add(int index, T element)
{
    if (index == 0)
        addFirst(element);
    else if (index >= size)
        addLast(element);
    else
    {
        Node<T> *current = head;
        for (int i = 1; i < index; i++)
            current = current->next;
        Node<T> *temp = current->next;
        // Create a new node
        Node<T> *newNode = new Node<T>(element);
        newNode->next = temp;
        current->next = newNode;
        size++;
    }
}
```

Consider three cases: (1) if `index` is 0, invoke `addFirst(element)` (line 5) to insert the element at the beginning of the list; (2) if `index` is greater than or equal to `size`, invoke `addLast(element)` (line 7) to insert the element at the end of the list; (3) otherwise, create a new node to store the new element and locate where to insert it. As shown in Figure 16.6, the new node is to be inserted between the nodes `current` and `temp`. The function assigns the new node to `current.next` and assigns `temp` to the new node’s `next`. The size is now increased by 1 (line 16).

**Figure 16.6**
A new element is inserted in the middle of the list.

### 16.4.4 Implementing `removeFirst()`

The `removeFirst()` function can be implemented as follows:

```cpp
template<typename T>
T LinkedList<T>::removeFirst() throw (runtime_error)
{
    if (size == 0)
        throw runtime_error("No elements in the list");
    else
    {
        current->next = new Node<T>(element);
        (current->next)->next = temp;
        size++;
    }
}
```
Consider three cases: (1) if the list is empty, an exception is thrown (line 5); (2) otherwise, remove the first node from the list by pointing head to the second node, as shown in Figure 16.7. The size is reduced by 1 after the deletion (line 10); (3) If there is one element, after removing the element, tail should be set to NULL (line 11).

**Figure 16.7**
The first node is deleted from the list.

### 16.4.5 Implementing `removeLast()`

The `removeLast()` function can be implemented as follows:

```
**PD: Please add line numbers (including space lines) in the following code**
<side remark line 5: throw exception>
<side remark line 6: size 1?>
<side remark line 9: head and tail NULL>
<side remark line 10: size is 1>
<side remark line 12: destroy the node>
<side remark line 13: return element>
<side remark line 15: size > 1?>
<side remark line 22: move tail>
<side remark line 24: reduce size>
<side remark line 26: destroy the node>
<side remark line 27: return element>
```

```cpp
template<typename T>
T LinkedList<T>::removeLast() throw (runtime_error)
{
    if (size == 0)
        return element;
    head = head->next;
    size--;
    if (head == NULL) tail = NULL;
    T element = temp->element;
    delete temp;
    return element;
    ...
```
throw runtime_error("No elements in the list");
else if (size == 1)
{
    Node<T> *temp = head;
    head = tail = NULL;
    size = 0;
    T element = temp->element;
    delete temp;
    return element;
}
else
{
    Node<T> *current = head;

    for (int i = 0; i < size - 2; i++)
        current = current->next;

    Node<T> *temp = tail;
    tail = current;
    tail->next = NULL;
    size--;
    T element = temp->element;
    delete temp;
    return element;
}

Consider three cases: (1) if the list is empty, an exception is thrown (line 5); (2) if the list contains only one node, this node is destroyed, head and tail both become NULL; (3) otherwise, the last node is destroyed (line 26) and the tail is repositioned to point to the second last node, as shown in Figure 16.8. For the last two cases, the size is reduced by 1 after the deletion (lines 10, 24) and the element value of the deleted node is returned (lines 13, 27).

![Diagram](image)

(a) Before the node is deleted.

(b) After the last node is deleted.

**Figure 16.8**
The last node is deleted from the list.

### 16.4.6 Implementing `removeAt(int index)`

The `removeAt(int index)` function finds the node at the specified index and then removes it. It can be implemented as follows:
Consider four cases: (1) if index is beyond the range of the list (i.e., index < 0 || index >= size), throw an exception; (2) if index is 0, invoke removeFirst() to remove the first node; (3) if index is size - 1, invoke removeLast() to remove the last node; (4) otherwise, locate the node at the specified index. Let current denote this node and previous denote the node before this node, as shown in Figure 16.9. Assign current.next to previous.next to eliminate the current node.
Figure 16.9
A node is deleted from the list.

Listing 16.2 gives the implementation of LinkedList. The implementation of lastIndexOf(T element), remove(T element), contains(T element), and set(int index, Object o) is omitted and left as an exercise.

Listing 16.2 LinkedList.h (Defining LinkedList)
***PD: Please add line numbers (including space lines) in the following code***
<side remark line 3: runtime error header>
<side remark line 6: class template>
<side remark line 7: class Node>
<side remark line 25: class template>
<side remark line 26: class LinkedList>
<side remark line 55: constructor>
<side remark line 62: addFirst>
<side remark line 74: addLast>
<side remark line 89: getFirst>
<side remark line 98: getLast>
<side remark line 107: removeFirst>
<side remark line 123: removeLast>
<side remark line 154: add>
<side remark line 160: add>
<side remark line 179: clear>
<side remark line 192: get>
<side remark line 205: indexOf>
<side remark line 220: isEmpty>
<side remark line 226: getSize>
<side remark line 232: removeAt>

#ifndef LINKEDLIST_H
#define LINKEDLIST_H
#include <stdexcept>
using namespace std;

template<typename T>
class Node
{...
public:

T element; // Element contained in the node
Node<T> *next; // Pointer to the next node

Node() // No-arg constructor
{
    next = NULL;
}

Node(T element) // Constructor
{
    this->element = element;
    next = NULL;
}

};

template<typename T>
class LinkedList
{
public:
    LinkedList();
    void addFirst(T element);
    void addLast(T element);
    T getFirst();
    T getLast();
    T removeFirst() throw (runtime_error);
    T removeLast();
    void add(T element);
    void add(int index, T element);
    void clear();
    bool contains(T element);
    T get(int index);
    int indexOf(T element);
    bool isEmpty();
    int lastIndexOf(T element);
    bool remove(T element);
    int getSize();
    T removeAt(int index);
    T set(int index, T element);

private:
    Node<T> *head, *tail;
    int size;

};

template<typename T>
LinkedList<T>::LinkedList()
{
    head = tail = NULL;
    size = 0;
}

template<typename T>
void LinkedList<T>::addFirst(T element)
{
Node<T> *newNode = new Node<T>(element);
newNode->next = head;
head = newNode;
size++;

if (tail == NULL)
  tail = head;
}

template<typename T>
void LinkedList<T>::addLast(T element)
{
  if (tail == NULL)
  |
  |
  head = tail = new Node<T>(element);
  |
  else |
  |
  tail->next = new Node<T>(element);
  |
  tail = tail->next;
  |
  size++;

template<typename T>
T LinkedList<T>::getFirst()
{
  if (size == 0)
    throw runtime_error("Index out of range");
  else
    return head->element;
}

template<typename T>
T LinkedList<T>::getLast()
{
  if (size == 0)
    throw runtime_error("Index out of range");
  else
    return tail->element;
}

template<typename T>
T LinkedList<T>::removeFirst() throw (runtime_error)
{
  if (size == 0)
    throw runtime_error("No elements in the list");
  else
    |
    Node<T> *temp = head;
    |
    head = head->next;
    |
    size--;
    |
    T element = temp->element;
    |
    delete temp;
    |
    return element;
template<typename T>
T LinkedList<T>::removeLast()
{
    if (size == 0)
        throw runtime_error("No elements in the list");
    else if (size == 1)
    {
        Node<T> *temp = head;
        head = tail = NULL;
        size = 0;
        T element = temp->element;
        delete temp;
        return element;
    }
    else
    {
        Node<T> *current = head;

        for (int i = 0; i < size - 2; i++)
            current = current->next;

        Node<T> *temp = tail;
        tail = current;
        tail->next = NULL;
        size--;
        T element = temp->element;
        delete temp;
        return element;
    }
}

template<typename T>
void LinkedList<T>::add(T element)
{
    addLast(element);
}

template<typename T>
void LinkedList<T>::add(int index, T element)
{
    if (index == 0)
        addFirst(element);
    else if (index >= size)
        addLast(element);
    else
    {
        Node<T> *current = head;

        for (int i = 1; i < index; i++)
            current = current->next;

        Node<T> *temp = current->next;
        current->next = new Node<T>(element);
        (current->next)->next = temp;
    }
}
```cpp
template<typename T>
void LinkedList<T>::clear()
{
    while (head != NULL)
    {
        Node<T> *temp = head;
        delete temp;
        head = head->next;
    }

tail = NULL;
}

template<typename T>
T LinkedList<T>::get(int index)
{
    if (index < 0 || index > size - 1)
        throw runtime_error("Index out of range");

    Node<T> *current = head;
    for (int i = 0; i < index; i++)
        current = current->next;

    return current->element;
}

template<typename T>
int LinkedList<T>::indexOf(T element)
{
    // Implement it in this exercise
    Node<T> *current = head;
    for (int i = 0; i < size; i++)
    {
        if (current->element == element)
            return i;
        current = current->next;
    }

    return -1;
}

template<typename T>
bool LinkedList<T>::isEmpty()
{
    return head == NULL;
}

template<typename T>
int LinkedList<T>::getSize()
{
    return size;
}
```
template<typename T>
T LinkedList<T>::removeAt(int index)
{
    if (index < 0 || index >= size)
        throw runtime_error("Index out of range");
    else if (index == 0)
        return removeFirst();
    else if (index == size - 1)
        return removeLast();
    else {
        Node<T> *previous = head;

        for (int i = 1; i < index; i++)
        {
            previous = previous->next;
        }

        Node<T> *current = previous->next;
        previous->next = current->next;
        size--;
        T element = current->element;
        delete current;
        return element;
    }
}

// The functions remove(T element), lastIndexOf(T element),
// contains(T element), and set(int index, T element) are
// left as an exercise

TIP

<side remark: array vs. linked list>
You can use an array or a linked list to store elements. If you don’t know the number of elements in advanced, it is more efficient to use a linked list, because a linked list can grow and shrink dynamically. If your application requires frequent insertion and deletion anywhere, it is more efficient to store elements using a linked list, because inserting an element into an array would require all the elements in the array after the insertion point to be moved. If the number of the elements in an application is fixed and the application does not require random insertion and deletion, it is simple and efficient to use an array to store the elements.

16.5 Variations of Linked Lists
The linked list introduced in the preceding section is known as a singly linked list. It contains a pointer to the list’s first node, and
each node contains a pointer to the next node sequentially. Several variations of the linked list are useful in certain applications.

A circular, singly linked list is like a singly linked list, except that the pointer of the last node points back to the first node.

A doubly linked list contains the nodes with two pointers. One points to the next node and the other points to the previous node. These two pointers are conveniently called a forward pointer and a backward pointer. So, a doubly linked list can be traversed forward and backward.

A circular, doubly linked list is doubly linked list, except that the forward pointer of the last node points to the first node and the backward pointer of the first node points to the last node.

The implementation of these linked lists is left as exercises.

16.6 Implementing Stack Using a LinkedList
The Stack class in the preceding chapter was implemented using an array. It is more efficient to implement the Stack class using a linked list. Listing 16.3 presents new implementation using a linked list.

Listing 16.3 StackWithLinkedList.h (Implementing Stack Using LinkedList)

```cpp
#ifndef STACKWITHLINKEDLIST_H
#define STACKWITHLINKEDLIST_H
#include "LinkedList.h"

namespace Stack
{
    template<typename T>
    class Stack
    {
    public:
        Stack();
        bool empty();
        T peek();
        T push(T value);
        T pop();
        int getSize();
    private:
        LinkedList<T> list;
    };

    #endif

    #define STACKWITHLINKEDLIST_H
    #include "LinkedList.h"
}
```

```cpp
```
The interface (e.g., public functions and constructors) for Stack class remains unchanged, as shown in Figure 15.1(b). Rather than implementing a stack using an array, a linked list is used (line 17). The implementation is rather simple. To test whether a stack is empty, return list.isEmpty() (line 28). To peek an element on the stack, return list.getLast() (line 34). To push an element to the stack, invoke list.addLast(value) (line 40) and return this value. To delete an element from the stack, return list.removeLast() (line 47). To find the size of the stack, return list.getSize() (line 53).

16.7 Queues

A queue represents a waiting list. A queue can be viewed as a special type of list whose elements are inserted into the end (tail) of the queue, and are accessed and deleted from the beginning (head) of the queue, as shown in Figure 16.10.
Figure 16.10
A queue holds objects in a first-in first-out fashion.

There are two ways to design the queue class:

<Side Remark: inheritance>
• Using inheritance: You can declare a queue class by extending the linked list class.

<Side Remark: composition>
• Using composition: You can declare a linked list as a data field in the queue class.

Both designs are fine, but using composition is better because it enables you to declare a completely new queue class without inheriting the unnecessary and inappropriate functions from the linked list. Figure 16.11 shows the UML class diagram for the queue. Its implementation is shown in Listing 16.4.

<PD: UML Class Diagram>

<table>
<thead>
<tr>
<th>Queue&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>-list: LinkedList&lt;T&gt;</td>
</tr>
<tr>
<td>+enqueue(element: T): void</td>
</tr>
<tr>
<td>+dequeue(): T</td>
</tr>
<tr>
<td>+getSize(): int</td>
</tr>
</tbody>
</table>

Figure 16.11
Queue uses a linked list to provide a first-in/first-out data structure.

Listing 16.4 Queue.h (Implementing Queue)

***PD: Please add line numbers in the following code***

<side remark line 3: include LinkedList>
<side remark line 5: class template>
<side remark line 15: linked list>
<side remark line 19: constructor>
<side remark line 24: enqueue>
<side remark line 30: dequeue>
<side remark line 36: getSize>

#ifndef QUEUE_H
#define QUEUE_H
#include "LinkedList.h"

//ifndef QUEUE_H
//define QUEUE_H
#include "LinkedList.h"

592
#include <stdexcept>
using namespace std;

template<typename T>
class Queue
{
public:
    Queue();
    void enqueue(T element);
    T dequeue() throw (runtime_error);
    int getSize();

private:
    LinkedList<T> list;
};

template<typename T>
Queue<T>::Queue()
{
}

template<typename T>
void Queue<T>::enqueue(T element)
{
    list.addLast(element);
}

template<typename T>
T Queue<T>::dequeue() throw (runtime_error)
{
    return list.removeFirst();
}

template<typename T>
int Queue<T>::getSize()
{
    return list.getSize();
}

#endif

A linked list is created to store the elements in a queue (line 15). The enqueue(T element) function (lines 23-27) adds element into the tail of the queue. The dequeue() function (lines 29-32) removes an element from the head of the queue and returns the removed element. The getSize() function (lines 35-39) returns the number of elements in the queue.

Listing 16.7 gives an example that creates a queue for int values (line 9) and a queue for strings (line 17) using the Queue class. It uses the enqueue function to add elements to the queues (lines 11, 18-20) and the dequeue function to remove int values and strings from the queue.

**PD: Please add line numbers in the following code**

Listing 16.5 TestQueue.cpp (Using Queue)

***PD: Please add line numbers in the following code***

<sidenote line 2: include Queue>
#include <iostream>
#include "Queue.h"
#include <string>
using namespace std;

template<typename T>
void printQueue(Queue<T> &queue)
{
    while (queue.getSize() > 0)
    {
        cout << queue.dequeue() << " ";
    }
    cout << endl;
}

int main()
{
    // Queue of int values
    Queue<int> intQueue;
    for (int i = 0; i < 10; i++)
        intQueue.enqueue(i);
    printQueue(intQueue);

    // Queue of strings
    Queue<string> stringQueue;
    stringQueue.enqueue("New York");
    stringQueue.enqueue("Boston");
    stringQueue.enqueue("Denver");
    printQueue(stringQueue);

    return 0;
}

<Output>
0 1 2 3 4 5 6 7 8 9
New York Boston Denver
<End Output>

Key Terms

***PD: Please place terms in two columns same as in intro5e.

- circular doubly linked list 47
- circular singly linked list 47
- dequeue 47
- doubly linked list 47
- enqueue 47
- linked list 47
- peek
Chapter Summary

- A linked list grows and shrinks dynamically. Nodes in a linked list are dynamically created using the `new` operator and they are destroyed using the `delete` operator.
- You can implement a stack using an array or a linked list. The public functions of the Stack class remain unchanged.
- A queue represents a waiting list. A queue can be viewed as a special type of list whose elements are inserted into the end (tail) of the queue, and are accessed and deleted from the beginning (head) of the queue.
- If you don’t know the number of elements in advanced, it is more efficient to use a linked list, because a linked list can grow and shrink dynamically.
- If your application requires frequent insertion and deletion anywhere, it is more efficient to store elements using a linked list, because inserting an element into an array would require all the elements in the array after the insertion point to be moved.
- If the elements need to be processed in a last-in first-out fashion, use a stack. If the elements need to be processed in a first-in first-out fashion, use a queue.

Review Questions
Section 16.2 Nodes

16.1
Are the following class declarations correct?

(a)

```cpp
class A
{
    public:
        A()
        {
        }
    private:
        A *a;
        int i;
};
```

(b)

```cpp
class A
{
    public:
        A()
        {
        }
    private:
        A a;
        int i;
};
```

16.2
What is `NULL` for?

16.3
When a node is created using the `Node` class, is the `next` pointer of this new node `NULL`?

**Section 16.3 The `LinkedList` Class**

16.4
Which of the following statements are to insert a string `s` to the head of the list? Which of the following statements are to append a string `s` to the end of the list?

```java
list.addFirst(s);
list.add(s);
list.add(0, s);
list.add(1, s);
```

16.5
Which of the following statements are to remove the first element from the list? Which of the following statements are to remove the last element from the list?

```java
list.removeFirst(s);
list.removeLast(s);
list.removeFirst();
list.removeLast();
list.remove(0);
list.removeAt(0);
list.removeAt(list.getSize() - 1);
list.removeAt(list.getSize());
```

16.6
Suppose the `removeAt` function is renamed as `remove` so there are two overloaded functions `remove(T element)` and `remove(int index)`. This is incorrect. Explain the reason.

**Section 16.4 Implementing `LinkedList`**

16.7
If a linked list does not contain any nodes, what are the values in `head` and `tail`?

16.8
If a linked list has only one node, is `head == tail` true?

16.9
When a new node is inserted to the head of a linked list, will the `head` pointer and the `tail` pointer be changed?

16.10
When a new node is inserted to the end of a linked list, will the `head` pointer and the `tail` pointer be changed?

16.11
When a node is removed from a linked list, what would happen if you don’t explicitly use the delete operator to release the node?

16.12
Under what circumstances, the functions removeFirst, removeLast, and removeAt would throw an exception?

16.13
Discuss the pros and cons of using arrays and linked lists.

16.14
If the number of elements in the program is fixed, what data structure should you use? If the number of elements in the program changes, what data structure should you use?

16.15
If you have to add or delete the elements anywhere in a list, should you use an array or a linked list?

16.16
What would happen when you run the following code?

***PD: Please add line numbers (including space lines) in the following code***

```cpp
#include <iostream>
#include <string>
#include "LinkedList.h"
using namespace std;

int main()
{
    LinkedList<string> list;
    list.add("abc");
    cout << list.removeLast() << endl;
    cout << list.removeLast() << endl;
    return 0;
}
```

16.17
Show the output of the following code.

***PD: Please add line numbers (including space lines) in the following code***

```cpp
#include <iostream>
#include <string>
#include "LinkedList.h"
using namespace std;

int main()
{
```

597
```cpp
    LinkedList<string> list;
    list.add("abc");

    try
    {
        cout << list.removeLast() << endl;
        cout << list.removeLast() << endl;
    }
    catch (runtime_error ex)
    {
        cout << "The list size is " << list.getSize() << endl;
    }
    return 0;
}
```

Section 16.5 Variations of Linked Lists

16.18 What is a circular, singly linked list? What is a doubly linked list? What is a circular, doubly linked list?

Sections 16.6-16.7

16.19 You can use inheritance or composition to design the data structures for stacks and queues. Discuss the pros and cons of these two approaches.

16.20 Show the output of the following code.

***PD: Please add line numbers (including space lines) in the following code***

```cpp
#include <iostream>
#include <string>
#include "StackWithLinkedList.h"
#include "Queue.h"
using namespace std;

int main()
{
    Stack<string> stack;
    Queue<int> queue;

    stack.push("Georgia");
    stack.push("Indiana");
    stack.push("Oklahoma");

    cout << stack.pop() << endl;
    cout << "Stack's size is " << stack.getSize() << endl;

    queue.enqueue(1);
    queue.enqueue(2);
    queue.enqueue(3);
```
```cpp
    cout << queue.dequeue() << endl;
    cout << "Queue's size is " << queue.getSize() << endl;
    return 0;
```

### Programming Exercises

#### Sections 16.2-16.4

**16.1**
(Implementing remove(T element)) The implementation of remove(T element) is omitted in the text. Implement it.

**16.2**
(Implementing lastIndexOf(T element)) The implementation of lastIndexOf(T element) is omitted in the text. Implement it.

**16.3**
(Implementing contains(T element)) The implementation of contains(T element) is omitted in the text. Implement it.

**16.4**
(Implementing set(int index, T element)) The implementation of contains(T element) is omitted in the text. Implement it.

**16.5**
(Adding set operations in LinkedList) Add and implement the following functions in LinkedList:

```cpp
    /* Add the elements in otherList to this list.
     * Returns true if this list changed as a result of the call */
    bool addAll(LinkedList<T> otherList)

    /* Remove all the elements in otherList from this list
     * Returns true if this list changed as a result of the call */
    bool removeAll(LinkedList otherList)

    /* Retain the elements in this list if they are also in otherList
     * Returns true if this list changed as a result of the call */
    bool retainAll(LinkedList otherList)
```

Write a test program that creates two LinkedLists, list1 and list2, with the initial values {"Beijing", "Shanghai", "Nanjing", "Wuhan", "Hongkong"} and {"Beijing", "Shanghai", "Xian", "Harbin", "Xiamen"}, then invokes list1.addAll(list2), list1.removeAll(list2), and list1.retainAll(list2), and displays the resulting new list1.
Section 16.5 Variations of Linked Lists
16.5*
(Creating a doubly linked list) The LinkedList class in the text is a singly linked list that enables one-way traversal of the list. Modify the Node class to add the new field name previous to refer to the previous node in the list, as follows:

```cpp
template<typename T>
class Node
{
  public:
    T element; // Element contained in the node
    Node<T> *previous; // Pointer to the previous node
    Node<T> *next; // Pointer to the next node

  Node(); // No-arg constructor
  // (previous = NULL);
  // next = NULL;

  Node(T element) // Constructor
  { // (this->element = element;
    // previous = NULL;
    // next = NULL;
  }

  // simplifying the implementation of add(T element, int index) and removeAt(int index) functions to take advantage of the doubly linked list.

Sections 16.6-16.7
16.6
(Using the Stack class) Write a program that displays the first fifty prime numbers in descending order. Use a stack to store prime numbers.

16.7
(Implementing Stack using inheritance) In §16.6, “Implementing Stack Using a LinkedList,” Stack is implemented using composition. Create a new stack class that extends LinkedList.

16.8
(Implementing Queue using inheritance) In §16.7, “Queues,” Queue is implemented using composition. Create a new queue class that extends LinkedList.