CHAPTER 2

C++ Linked Lists, Stacks and Queues

LESSON 2.1

Linked Lists

A linked list is a collection of components, called *nodes*. Every node (except the last node) contains the address of the next node. Thus, every node in a linked list has two components: one to store the relevant information (data) and one to store the address, called the *link*, of the next node in the list. The address of the first node in the list is stored in a separate location, called the *head* or *first*. Figure 2.1.1 is a pictorial representation of a node.

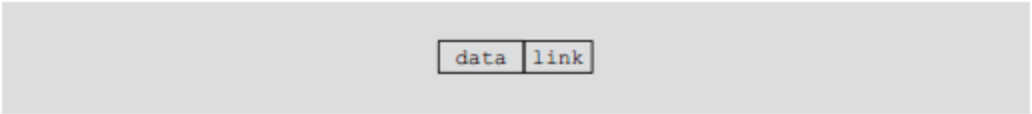


Figure 2.2.1 Structure of a node

Linked list: A list of items, called *nodes*, in which the order of the nodes is determined by the address, called *link*, stored in each node.

The list in Figure 2.1.2 is an example of a linked list.

Figure 2.1.2

A linked list is made up of a series of objects, called the *nodes of the list*. Because list node is distinct object (as opposed to simply a cell in an array), it is good practice to make a separate list node class. An additional benefit to creating a list node class is that it can be reused by the linked implementations for the stack and queue data structures presented later in this chapter.

> **Objects** in the Link class contain an element field to store the element value, and a next field to store a pointer to the next node on the list. The list built from such nodes is called *singly linked list*, or a *one-way list*, because each list node has a single pointer to the next node on the list.
>
> **The Link class** is quite simple. There are two forms for its constructor, one with an initial element value and one without. Because the Link class is also used by the stack and queue implementations presented later, its data members are made public. While technically this is breaking encapsulation, in practice the link class should be implemented as a private class of the linked list (stack or queue) implementation, and thus not visible to the rest of the program.

Figure 2.1.3 shows a graphical depiction for a linked list storing four integers. The value stored in a pointer variable is indicated by an arrow "pointing" to something. C++ uses the special symbol NULL for a pointer value that points nowhere, such as for the last list node's next field. A NULL pointer is indicated graphically

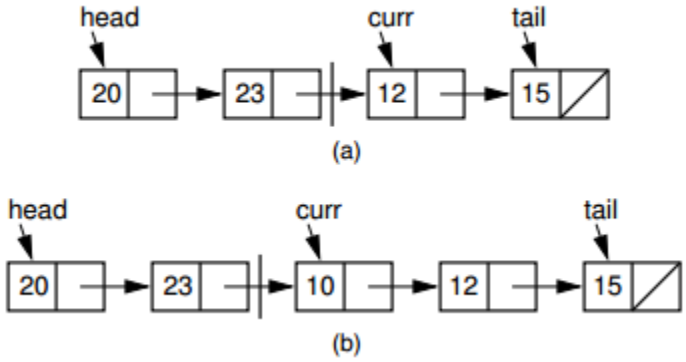


Figure 2.1.3 Illustration of a faulty linked-list implementation where curr points directly to the current node. (a) Linked list prior to inserting element with value 10. (b) Desired effect of inserting element with value 10.

by a diagonal slash through a pointer variable's box. The vertical line between the nodes labeled 23 and 12 in Figure 2.1.3(a) indicates the current position (immediately to the right of this line).

The list's first node is accessed from a pointer named *head*. To speed access to the end of the list, and to allow the append method to be performed in constant time; a pointer named *tail* is also kept to the last link of the list. The position of the current element is indicated by another pointer, named *curr.* Finally, because there is no simple way to compute the length of the list simply from these three pointers, the list length must be stored explicitly, and updated by every operation that modifies the list size. The value cannot store the length of the list.

Class List also includes private helper methods in it and removes all. They are used by List's constructor, destructor, and clear methods.

Note that List's constructor maintains the optional parameter for minimum list size introduced for Class A List. This is done simply to keep the calls to the constructor the same for both variants. Because the linked list class does not

need to declare a fixed-size array when the list is created, this parameter is unnecessary for linked lists. It is ignored by the implementation.

Figure 2.1.3(a) shows the list's curr pointer pointing to the current element. The vertical line between the nodes containing 23 and 12 indicates the logical position of the current element. Consider what happens if we wish to insert a new node with value 10 into the list. The result should be as shown in Figure 4.5(b). However, there is a problem. To "*splice*" the list node containing the new element into the list, the list node storing 23 must have its next pointer changed to point to the new node. Unfortunately, there is no convenient access to the node preceding the one pointed to by curr.

Because each node of a linked list has two components, we need to declare each node as a class or struct. The data type of each node depends on the specific application—that is, what kind of data is being processed. However, the link component of each node is a pointer. The data type of this pointer variable is the node type itself. For the previous linked list, the definition of the node is as follows. (Suppose that the data type is int.)

```
struct nodeType
{
int info;
nodeType *link;
};
```

The variable declaration is as follows:

```
nodeType *head;
```

Linked Lists: Properties

To better understand the concept of a linked list and a node, some important properties of linked lists are described next.

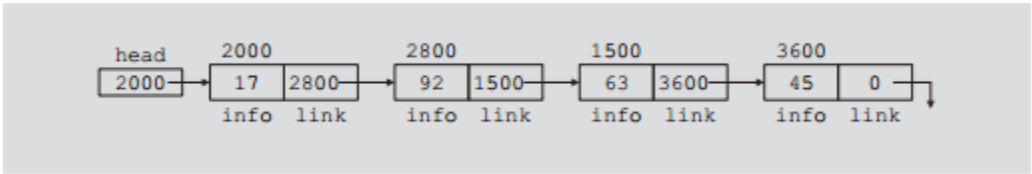


Figure 2.1.4 Linked lists with four nodes

This linked list has four nodes. The address of the first node is stored in the pointer head. Each node has two components: info, to store the info, and link,

to store the address of the next node. For simplicity, we assume that info is of type int.

Suppose that the first node is at location 2000, the second node is at location 2800, the third node is at location 1500, and the fourth node is at location 3600. Table 2.1.1 shows the values of head and some other nodes in the list shown in Figure 2.1.4.

Table 2.1.1 Values of head and some of the nodes of the linked list in Figure 2.1.4.

	Value	Explanation
`head`	2000	
`head->info`	17	Because **head** is **2000** and the **info** of the node at location **2000** is **17**
`head->link`	2800	
`head->link->info`	92	Because **head->link** is **2800** and the **info** of the node at location **2800** is **92**

Suppose that current is a pointer of the same type as the pointer head. Then the statement

current = head;

copies the value of head into current. Now consider the following statement:

current = current->link;

This statement copies the value of current->link, which is 2800, into current. Therefore, after this statement executes current points to the second node in the list. (When working with linked lists, we typically use these types of statements to advance a pointer to the next node in the list.) See Figure 2.1.5.

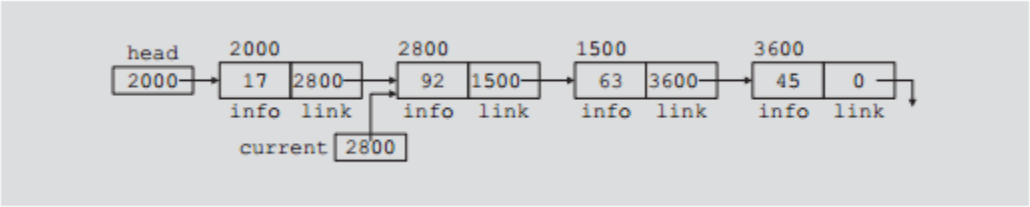


*Figure 2.1.5*List after the statement current = current->link; executes

Table 2.1.2 shows the values of current, head, and some other nodes in Figure 2.1.5.

*Table 2.1.2*Values of current, head, and some of the nodes of the linked list in Figure 2.1.5

	Value
`current`	2800
`current->info`	92
`current->link`	1500
`current->link->info`	63
`head->link->link`	1500
`head->link->link->info`	63
`head->link->link->link`	3600
`current->link->link->link`	0 (that is, **NULL**)
`current->link->link->link->info`	Does not exist (run-time error)

From now on, when working with linked lists, we will use only the arrow notation.

Traversing the Linked Lists

The basic operations of a linked list are as follows: Search the list to determine whether a particular item is in the list, insert an item in the list, and delete an item from the list. These operations require the list to be traversed. That is, given a pointer to the first node of the list, we must step through the nodes of the list.

Suppose that the pointer head points to the first node in the list, and the link of the last node is NULL. We cannot use the pointer head to traverse the list because if we use the head to traverse the list, we would lose the nodes of the list. This problem occurs because the links are in only one direction. The pointer

head contains the address of the first node, the first node contains the address of the second node, and the second node contains the address of the third node, and so on. If we move head to the second node, the first node is lost (unless we save a pointer to this node). If we keep advancing head to the next node, we will lose all the nodes of the list (unless we save a pointer to each node before advancing head, which is impractical because it would require additional computer time and memory space to maintain the list).

Therefore, we always want head to point to the first node. It now follows that we must traverse the list using another pointer of the same type. Suppose that current is a pointer of the same type as head. The following code traverses the list:

```
current = head;
while (current != NULL)
{
//Process current
current = current->link;
}
```

For example, suppose that head points to a linked list of numbers. The following code outputs the data stored in each node:

```
current = head;
while (current != NULL)
{
cout << current->info << " ";
current = current->link;
}
```

Item Insertion and Deletion

This section discusses how to insert an item into, and delete an item from, a linked list. Consider the following definition of a node. (For simplicity, we assume that the info type is int. The next section, which discusses linked lists as an abstract data type (ADT) using templates, uses the generic definition of a node.)

```
struct nodeType
{
int info;
nodeType *link;
};
```

We will use the following variable declaration:

nodeType *head, *p, *q, *newNode;

INSERTION

Consider the linked list shown in Figure 2.1.6.

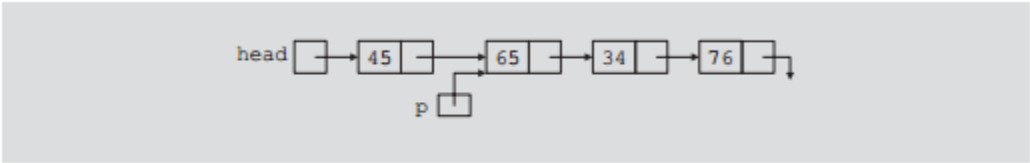


*Figure 2.1.6*Linked list before item insertion

Suppose that p points to the node with info 65, and a new node with info 50 is to be created and inserted after p. Consider the following statements:

```
newNode = new nodeType; //create newNode
newNode->info = 50; //store 50 in the new node
newNode->link = p->link;
p->link = newNode;
```

Table 2.1.3 shows the effect of these statements

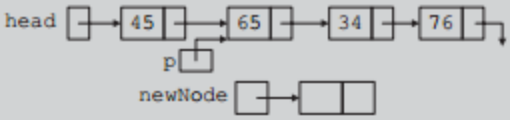
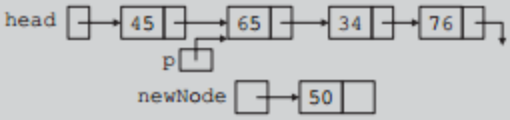
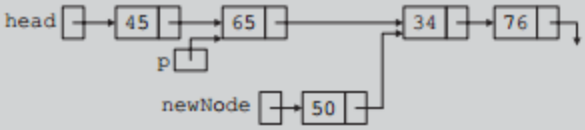
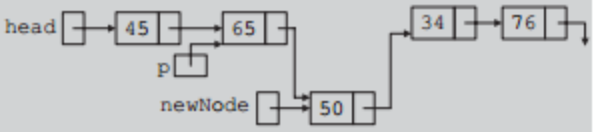
Statement	Effect
`newNode = new nodeType;`	
`newNode->info = 50;`	
`newNode->link = p->link;`	
`p->link = newNode;`	

*Table 2.1.3*Inserting a node in a linked list

Note that the sequence of statements to insert the node, that is,

```
newNode->link = p->link;
```

p->link = newNode;
is very important because to insert newNode in the list we use only one pointer, p, to adjust the links of the nodes of the linked list. Suppose that we reverse the sequence of the statements and execute the statements in the following order:

p->link = newNode;
newNode->link = p->link;

Figure 2.1.7 shows the resulting list after these statements execute.

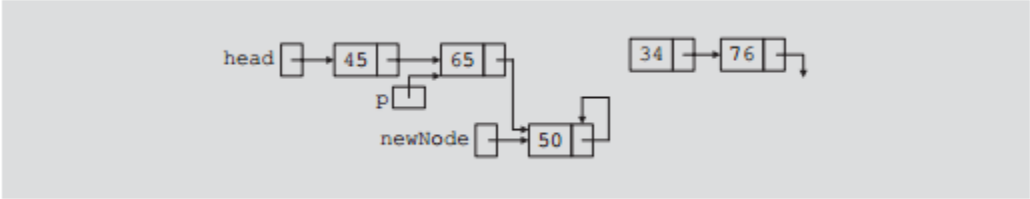


Figure 2.1.7 List after the execution of the statement p->link = newNode; followed by the execution of the statement newNode->link = p->link;

From Figure 2.1.7, it is clear that newNode points back to itself and the remainder of the list is lost.

Using two pointers, we can simplify the insertion code somewhat. Suppose q points to the node with info 34. (See Figure 2.1.8.)

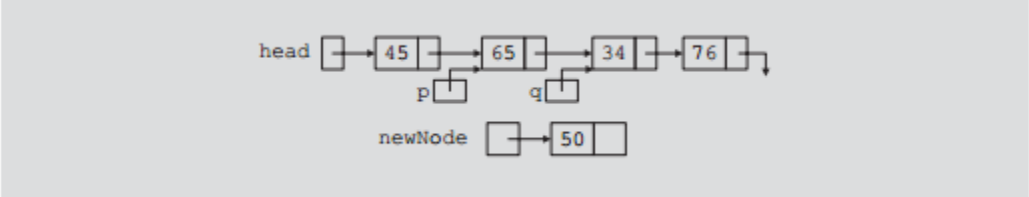


Figure 2.1.8 List with pointers p and q

The following statements insert newNode between p and q:

newNode->link = q;
p->link = newNode;

The order in which these statements execute does not matter. To illustrate this, suppose that we execute the statements in the following order:

p->link = newNode;

newNode->link = q;

Table 2.1.4 shows the effect of these statements.

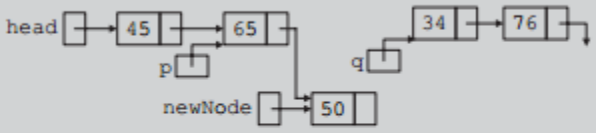
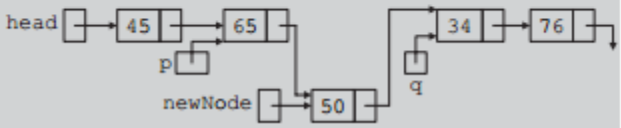
Statement	Effect
`p->link = newNode;`	
`newNode->link = q;`	

Table 2.1.4 Inserting a node in a linked list using two pointers

DELETION

Consider the linked list shown in Figure 2.1.8

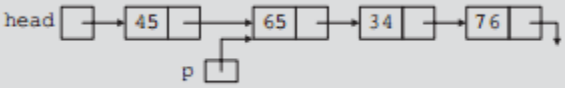


Figure 2.1.8 Node to be deleted is with info 34

Suppose that the node with info 34 is to be deleted from the list. The following statement removes the node from the list:

p->link = p->link->link;

Figure 2.1.10 shows the resulting list after the preceding statement executes.

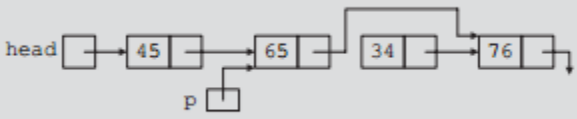


Figure 2.1.10 List after the statement p->link = p->link->link; executes

From Figure 2.1.10, it is clear that the node with info 34 is removed from the list. However, the memory is still occupied by this node and

this memory is inaccessible; that is, this node is dangling. To deallocate the memory, we need a pointer to this node. The following statements delete the node from the list and deallocate the memory occupied by this node:

```
q = p->link;
p->link = q->link;
delete q;
```

Table 2.1.5 shows the effect of these statements.

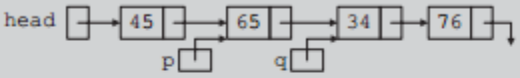
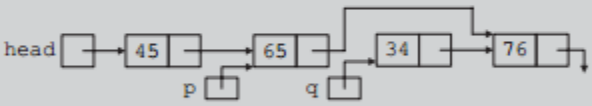
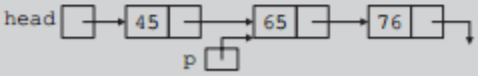
Statement	Effect
`q = p->link;`	
`p->link = q->link;`	
`delete q;`	

Table 2.1.5 Deleting a node from a linked list

Building a Linked Lists

Suppose that the nodes are in the usual info-link form and info is of type int. Let us assume that we process the following data:

2 15 8 24 34

We need three pointers to build the list: one to point to the first node in the list, which cannot be moved, one to point to the last node in the list, and one to create the new node. Consider the following variable declaration:

```
nodeType *first, *last, *newNode;
int num;
```

Suppose that first points to the first node in the list. Initially, the list is empty, so both first and last are NULL. Thus, we must have the statements

```
first = NULL;
```

last = NULL;

to initialize first and last to NULL.

Next, consider the following statements:

```
1   cin >> num;              //read and store a number in num
2   newNode = new nodeType;  //allocate memory of type nodeType
                             //and store the address of the
                             //allocated memory in newNode
3   newNode->info = num;     //copy the value of num into the
                             //info field of newNode
4   newNode->link = NULL;    //initialize the link field of
                             //newNode to NULL
5   if (first == NULL)       //if first is NULL, the list is empty;
                             //make first and last point to newNode
    {
5a     first = newNode;
5b     last = newNode;
    }
6   else                     //list is not empty
    {
6a     last->link = newNode; //insert newNode at the end of the list
6b     last = newNode;        //set last so that it points to the
                              //actual last node in the list
    }
```

Let us now execute these statements. Initially, both first and last are NULL. Therefore, we have the list as shown in Figure 2.1.11.

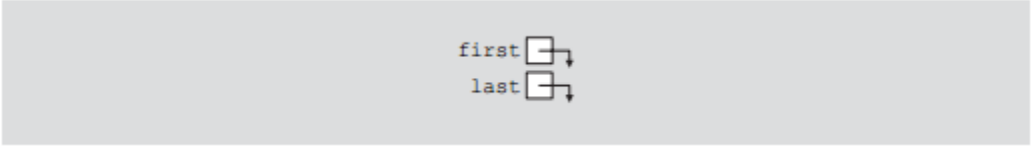


Figure 2.1.11 Empty list

After statement 1 executes, num is 2. Statement 2 creates a node and stores the address of that node in newNode. Statement 3 stores 2 in the info field of newNode, and statement 4 stores NULL in the link field of newNode. (See Figure 2.1.12.)

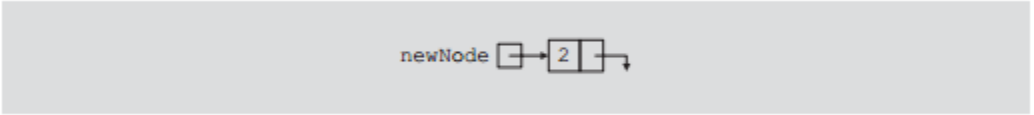


Figure 2.1.12 newNode with info 2

Because first is NULL, we execute statements 5a and 5b. Figure 2.1.13 shows the resulting list.

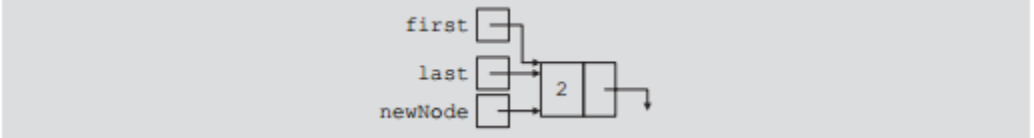


Figure 2.1.13 List after inserting newNode in it

We now repeat statements 1 through 6b. After statement 1 executes, num is 15. Statement 2 creates a node and stores the address of this node in newNode. Statement 3 stores 15 in the info field of newNode, and statement 4 stores NULL in the link field of newNode. (See Figure 2.1.14.)

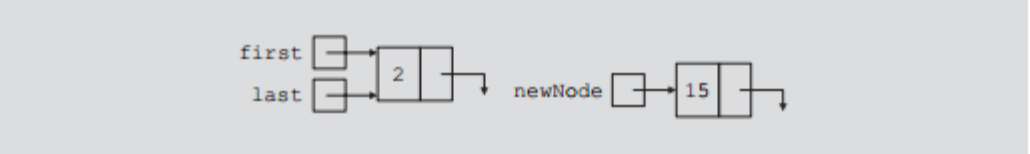


*Figure 2.1.14*List and newNode with info 15

Because first is not NULL, we execute statements 6a and 6b. Figure 2.1.15 shows the resulting list

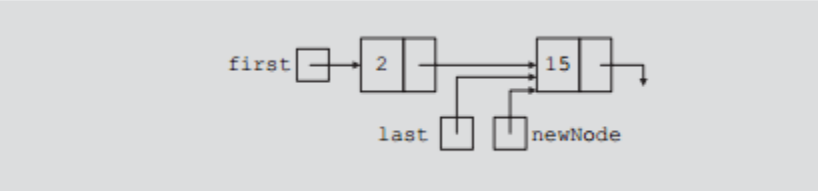


Figure 2.1.15 List after inserting newNode at the end

We now repeat statements 1 through 6b three more times. Figure 2.1.16 shows the resulting list.

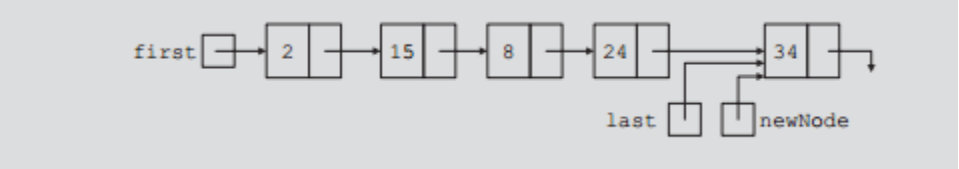


Figure 2.1.16 List after inserting 8, 24, and 34

We can put the previous statements in a loop, and execute the loop until certain conditions are met, to build the linked list. We can, in fact, write a C++ function to build a linked list. Suppose that we read a list of integers ending with -999. The following function, buildListForward, builds a linked list (in a forward manner) and returns the pointer of the built list:

```
nodeType* buildListForward()
{
    nodeType *first, *newNode, *last;
    int num;

    cout << "Enter a list of integers ending with -999."
         << endl;
    cin >> num;
    first = NULL;

    while (num != -999)
    {
        newNode = new nodeType;
        newNode->info = num;
        newNode->link = NULL;

        if (first == NULL)
        {
            first = newNode;
            last = newNode;
        }
        else
        {
            last->link = newNode;
            last = newNode;
        }
        cin >> num;
    } //end while

    return first;
} //end buildListForward
```

Linked Lists as ADT

The previous sections taught you the basic properties of linked lists and how to construct and manipulate linked lists. Because a linked list is a very important data structure, rather than discuss specific lists such as a list of integers or a list of strings, this section discusses linked lists as an abstract data type (ADT). Using templates, this section gives a generic definition of linked lists, which is then used in the next section and later in this book. The programming example at the end of this chapter also uses this generic definition of linked lists.

The basic operations on linked lists are as follows:

1. Initialize the list.
2. Determine whether the list is empty.
3. Print the list.
4. Find the length of the list.

5. Destroy the list.
6. Retrieve the info contained in the first node.
7. Retrieve the info contained in the last node.
8. Search the list for a given item.
9. Insert an item in the list.
10. Delete an item from the list.
11. Make a copy of the linked list

In general, there are two types of linked lists—sorted lists, whose elements are arranged according to some criteria, and unsorted lists, whose elements are in no particular order. The algorithms to implement the operations search, insert, and remove slightly differ for sorted and unsorted lists. Therefore, we will define the class linked List Type to implement the basic operations on a linked list as an abstract class. Using the principal of inheritance, we derive two classes— unordered Linked List and ordered Linked List—from the class linked List Type.

Objects of the class unordered Linked List would arrange list elements in no particular order that is, these lists may not be sorted. On the other hand, objects of the class ordered Linked List would arrange elements according to some comparison criteria, usually less than or equal to. That is, these lists will be in ascending order. Moreover, after inserting an element into or removing an element from an ordered list, the resulting list will be ordered.

If a linked list is unordered, we can insert a new item at either the end or the beginning. Furthermore, you can build such a list in either a forward manner or a backward manner. The function build List Forward inserts the new item at the end, whereas the function build List Backward inserts the new item at the beginning. To accommodate both operations, we will write two functions: insert First to insert the new item at the beginning of the list and insert Last to insert the new item at the end of the list. Also, to make the algorithms more efficient, we will use two pointers in the list: first, which points to the first node in the list, and last, which points to the last node in the list.

LESSON 2.2

Stacks

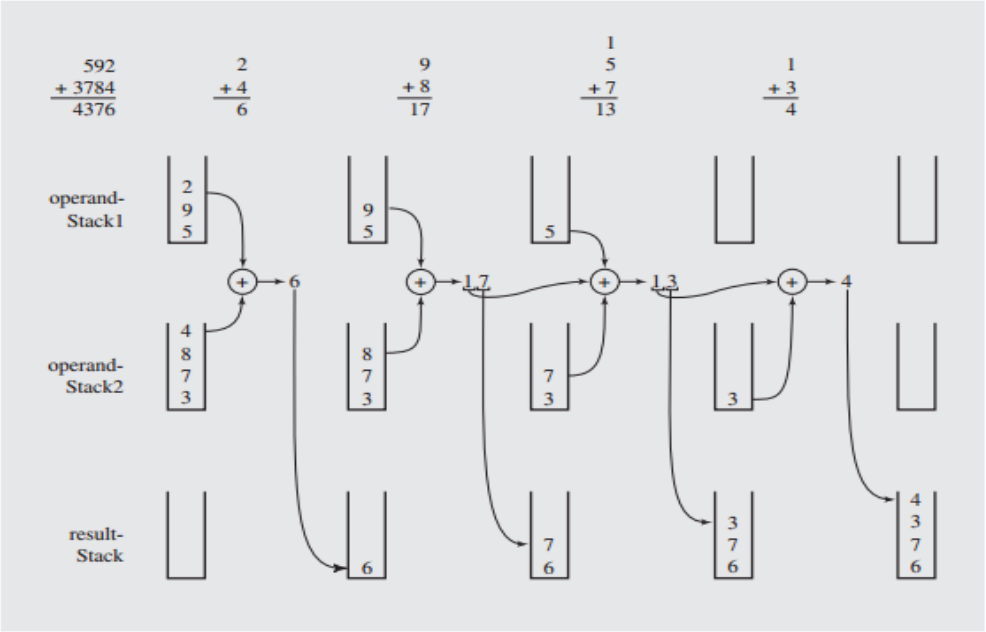
A stack is a linear data structure that can be accessed only at one of its ends for storing and retrieving data. Such a stack resembles a stack of trays in a cafeteria: new trays are put on the top of the stack and taken off the top. The last tray put on the stack is the first tray removed from the stack. For this reason, a stack is called an *LIFO structure: last in/first out*.

A tray can be taken only if there is at least one tray on the stack, and a tray can be added to the stack only if there is enough room; that is, if the stack is not too high. Therefore, a stack is defined in terms of operations that change its status and operations that check this status. The operations are as follows:

- clear()—Clear the stack.
- isEmpty()—Check to see if the stack is empty.
- push(el)—Put the element el on the top of the stack.
- pop()—Take the topmost element from the stack.
- topEl()—Return the topmost element in the stack without removing it.

The stack is a list-like structure in which elements may be inserted or removed from only one end. While this restriction makes stacks less flexible than lists, it also makes stacks both efficient (for those operations they can do) and easy to implement. Many applications require only the limited form of insert and remove operations that stacks provide. In such cases, it is more efficient to use the simpler stack data structure rather than the generic list.

Example of Stack



Despite their restrictions, stacks have many uses. Thus, a special vocabulary for stacks has developed. Accountants used stacks long before the invention of the computer. They called the stack a “LIFO” list, which stands for “Last-In, First-Out.” Note that one implication of the LIFO policy is that stacks remove elementsin reverse order of their arrival.

```
// Stack abtract class
template <typename E> class Stack {
private:
  void operator =(const Stack&) {}     // Protect assignment
  Stack(const Stack&) {}          // Protect copy constructor

public:
  Stack() {}                          // Default constructor
  virtual ~Stack() {}                 // Base destructor

  // Reinitialize the stack.  The user is responsible for
  // reclaiming the storage used by the stack elements.
  virtual void clear() = 0;

  // Push an element onto the top of the stack.
  // it: The element being pushed onto the stack.
  virtual void push(const E& it) = 0;

  // Remove the element at the top of the stack.
  // Return: The element at the top of the stack.
  virtual E pop() = 0;

  // Return: A copy of the top element.
  virtual const E& topValue() const = 0;

  // Return: The number of elements in the stack.
  virtual int length() const = 0;
};
```

Figure 2.2.1 The stack ADT

The accessible element of the stack is called the *top element*. Elements are not said to be inserted, they are pushed onto the stack. When removed, an element is said to be popped from the stack. Figure 2.2.1 shows a sample stack ADT.

As with lists, there are many variations on stack implementation. The two approaches presented here are array-based and linked stacks, which are analogous to array-based and linked lists, respectively.

Array Based Stacks

Figure 2.2.2 shows a complete implementation for the array-based stack class. As with the array-based list implementation, list Array must be declared of fixed size when the stack is created. In the stack constructor, size serves to indicate this size. Method top acts somewhat like a current position value (because the "current" position is always at the top of the stack), as well as indicating the number of elements currently in the stack.

```
// Array-based stack implementation
template <typename E> class AStack: public Stack<E> {
private:
  int maxSize;               // Maximum size of stack
  int top;                   // Index for top element
  E *listArray;            // Array holding stack elements

public:
  AStack(int size =defaultSize)   // Constructor
    { maxSize = size; top = 0; listArray = new E[size]; }

  ~AStack() { delete [] listArray; }  // Destructor

  void clear() { top = 0; }           // Reinitialize

  void push(const E& it) {          // Put "it" on stack
    Assert(top != maxSize, "Stack is full");
    listArray[top++] = it;
  }

  E pop() {                 // Pop top element
    Assert(top != 0, "Stack is empty");
    return listArray[--top];
  }

  const E& topValue() const {     // Return top element
    Assert(top != 0, "Stack is empty");
    return listArray[top-1];
  }

  int length() const { return top; }  // Return length
};
```

Figure 2.2.2 Array-based stack class implementation.

The array-based stack implementation is essentially a simplified version of the array-based list. The only important design decision to be made is which end of the array should represent the top of the stack. One choice is to make the top be at position 0 in the array. In terms of list functions, all insert and remove operations would then be on the element in position 0. This implementation is inefficient, because now every push or pop operation will require that all elements currently in the stack be shifted one position in the array, for a cost of $\Theta(n)$ if there are n elements. The other choice is have the top element be at

position n − 1 when there are n elements in the stack. In other words, as elements are pushed onto the stack, they are appended to the tail of the list. Method pop removes the tail element. In this case, the cost for each push or pop operation is only Θ(1).

For the implementation of Figure 2.2.2, top is defined to be the array index of the first free position in the stack. Thus, an empty stack has top set to 0, the first available free position in the array. (Alternatively, top could have been defined to be the index for the top element in the stack, rather than the first free position. If this had been done, the empty list would initialize top as −1.) Methods push and pop simply place an element into, or remove an element from, the array position indicated by top. Because top is assumed to be at the first free position, push first inserts its value into the top position and then increments top, while pop first decrements top and then removes the top element.

```
// Linked stack implementation
template <typename E> class LStack: public Stack<E> {
private:
  Link<E>* top;             // Pointer to first element
  int size;                    // Number of elements

public:
  LStack(int sz =defaultSize) // Constructor
    { top = NULL; size = 0; }

  ~LStack() { clear(); }           // Destructor

  void clear() {                   // Reinitialize
    while (top != NULL) {          // Delete link nodes
      Link<E>* temp = top;
      top = top->next;
      delete temp;
    }
    size = 0;
  }

E pop() {                  // Remove "it" from stack
  Assert(top != NULL, "Stack is empty");
  E it = top->element;
  Link<E>* ltemp = top->next;
  delete top;
  top = ltemp;
  size--;
  return it;
}

const E& topValue() const { // Return top value
  Assert(top != 0, "Stack is empty");
  return top->element;
}

int length() const { return size; } // Return length
};
```

*Figure 2.2.3*Linked stack class implementation.

Limited Stacks

The linked stack implementation is quite simple. Elements are inserted and removed only from the head of the list. A header node is not used because no special-case code is required for lists of zero or one elements. Figure 2.2.4 shows the complete linked stack implementation.

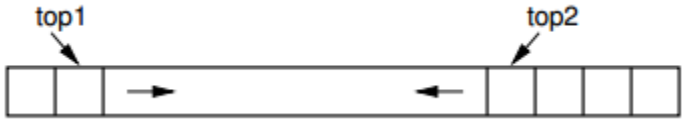


Figure 2.2.4 Two stacks implemented within in a single array, both growing toward the middle

The only data member is top, a pointer to the first (top) link node of the stack. Method push first modifies the next field of the newly created link node to point to the top of the stack and then sets top to point to the new link node. Method pop is also quite simple. Variable temp stores the top nodes' value, while ltemp links to the top node as it is removed from the stack. The stack is updated by setting top to point to the next link in the stack. The old top node is then returned to free store (or free list), and the element value is returned.

Comparison of Array Based and Linked Lists

All operations for the array-based and linked stack implementations take constant time, so from a time efficiency perspective, neither have a significant advantage. Another basis for comparison is the total space required. The analysis is similar to that done for list implementations. The array-based stack must declare a fixed-size array initially, and some of that space is wasted whenever the stack is not full. The linked stack can shrink and grow but requires the overhead of a link field for every element.

When multiple stacks are to be implemented, it is possible to take advantage of the one-way growth of the array-based stack. This can be done by using a single array to store two stacks. One stack grows inward from each end as illustrated by Figure 2.2.4, hopefully leading to less wasted space. However, this only works well when the space requirements of the two stacks are inversely correlated. In other words, ideally when one stack grows, the other will shrink. This is particularly effective when elements are taken from one stack and given to the other. If instead both stacks grow at the same time, then the free space in the middle of the array will be exhausted quickly.

```
//*********************  genStack.h  *************************
//      generic class for vector implementation of stack

#ifndef STACK
#define STACK

#include <vector>

template<class T, int capacity = 30>
class Stack {
public:
    Stack() {
        pool.reserve(capacity);
    }
    void clear() {
        pool.clear();
    }
    bool isEmpty() const {
        return pool.empty();
    }
    T& topEl() {

        return pool.back();
    }
    T pop() {
        T el = pool.back();
        pool.pop_back();
        return el;
    }
    void push(const T& el) {
        pool.push_back(el);
    }
private:
    vector<T> pool;
};

#endif
```

*Figure 2.2.5*A vector implementation of a stack.

```
//*********************  genListStack.h  *************************
//     generic stack defined as a doubly linked list

#ifndef LL_STACK
#define LL_STACK

#include <list>

template<class T>
class LLStack {
public:
    LLStack() {
    }
    void clear() {
        lst.clear();
    }
    bool isEmpty() const {
        return lst.empty();


    }
    T& topEl() {
        return lst.back();
    }
    T pop() {
        T el = lst.back();
        lst.pop_back();
        return el;
    }
    void push(const T& el) {
        lst.push_back(el);
    }
private:
    list<T> lst;
};

#endif
```

*Figure 2.2.6*Implementing a stack as a linked list.

The linked list implementation matches the abstract stack more closely in that it includes only the elements that are on the stack because the number of nodes in the list is the same as the number of stack elements. In the vector implementation, the capacity of the stack can often surpass its size.

The vector implementation, like the linked list implementation, does not force the programmer to make a commitment at the beginning of the program concerning the size of the stack. If the size can be reasonably assessed in advance, then the predicted size can be used as a parameter for the stack

constructor to create in advance a vector of the specified capacity. In this way, an overhead is avoided to copy the vector elements to a new larger location when pushing a new element to the stack for which size equals capacity.

It is easy to see that in the vector and linked list implementations; popping and pushing are executed in constant time O(1). However, in the vector implementation, pushing an element onto a full stack requires allocating more memory and copies the elements from the existing vector to a new vector. Therefore, in the worst case, pushing takes O(n) time to finish.

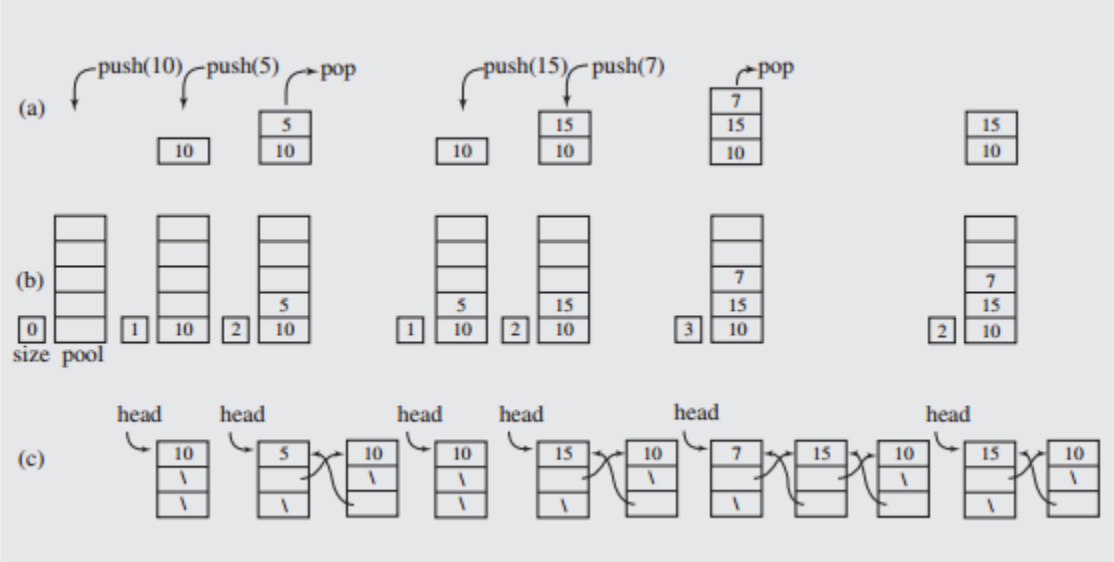


Figure 2.2.7 A series of operations executed on (a) an abstract stack and the stack implemented (b) with a vector and (c) with a linked list.

LESSON 2.3

Queues

A queue is simply a waiting line that grows by adding elements to its end and shrinks by taking elements from its front. Unlike a stack, a queue is a structure in which both ends are used: one for adding new elements and one for removing them. Therefore, the last element has to wait until all elements preceding it on the queue are removed.

A queue is an FIFO structure: first in/first out.

Queue operations are similar to stack operations. The following operations are needed to properly manage a queue:

- clear()—Clear the queue.
- isEmpty()—Check to see if the queue is empty.
- enqueue(el)—Put the element el at the end of the queue.
- dequeue()—Take the first element from the queue.
- firstEl()—Return the first element in the queue without removing it.

A series of enqueue and dequeue operations is shown in Figure 2.3.1. This time—unlike for stacks—the changes have to be monitored both at the beginning of the queue and at the end. The elements are enqueued on one end and dequeued from the other. For example, after enqueuing 10 and then 5, the dequeue operation removes 10 from the queue (Figure 2.3.1).

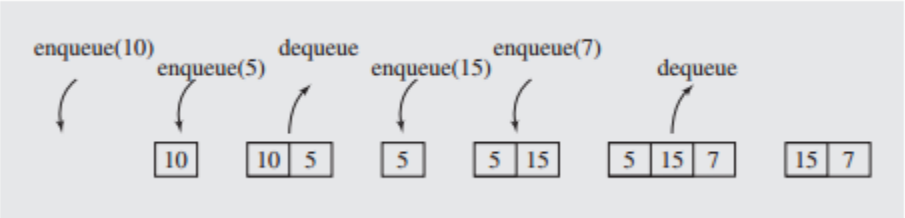


*Figure 2.3.1*A series of operations executed on a queue.

For an application of a queue, consider the following poem written by Lewis Carroll:

Round the wondrous globe I wander wild,
Up and down-hill—Age succeeds to youth—
Toiling all in vain to find a child
Half so loving, half so dear as Ruth.

The poem is dedicated to Ruth Dymes, which is indicated not only by the last word of the poem, but also by reading in sequence the first letters of each line, which also spells Ruth. This type of poem is called an acrostic, and it is characterized by initial letters that form a word or phrase when taken in order. To see whether a poem is an acrostic, we devise a simple algorithm that reads a poem, echo prints it, retrieves and stores the first letter from each line on a queue, and after the poem is processed; all the stored first letters are printed in order. Here is an algorithm:

```
acrosticIndicator()
 while not finished
        read a line of poem;
        enqueue the first letter of the line;
        output the line;
 while queue is not empty
        dequeue and print a letter;
```

There is a more significant example to follow, but first consider the problem of implementation.

One possible queue implementation is an array, although this may not be the best choice. Elements are added to the end of the queue, but they may be removed from its beginning, thereby releasing array cells. These cells should not be wasted. Therefore, they are utilized to enqueue new elements, whereby the end of the queue may occur at the beginning of the array. This situation is better pictured as a circular array, as Figure 2.3.2c illustrates. The queue is full if the first element immediately precedes the last element in the counterclockwise direction. However, because a circular array is implemented with a “normal” array, the queue is full if either the first element is in the first cell and the last element is in the last cell (Figure 2.3.2a) or if the first element is right after the last (Figure 2.3.2b).

Similarly, enqueue() and dequeue() have to consider the possibility of wrapping around the array when adding or removing elements. For example, enqueue() can be viewed as operating on a circular array (Figure 2.3.2c), but in reality, it is operating on a one-dimensional array. Therefore, if the last element is in the last cell and if any cells are available at the beginning of the array, a new element is placed there (Figure 2.3.2d). If the last element is in any other position, then the new element is put after the last, space permitting (Figure 2.3.2e). These two situations must be distinguished when implementing a queue viewed as a circular array (Figure 2.3.2f).

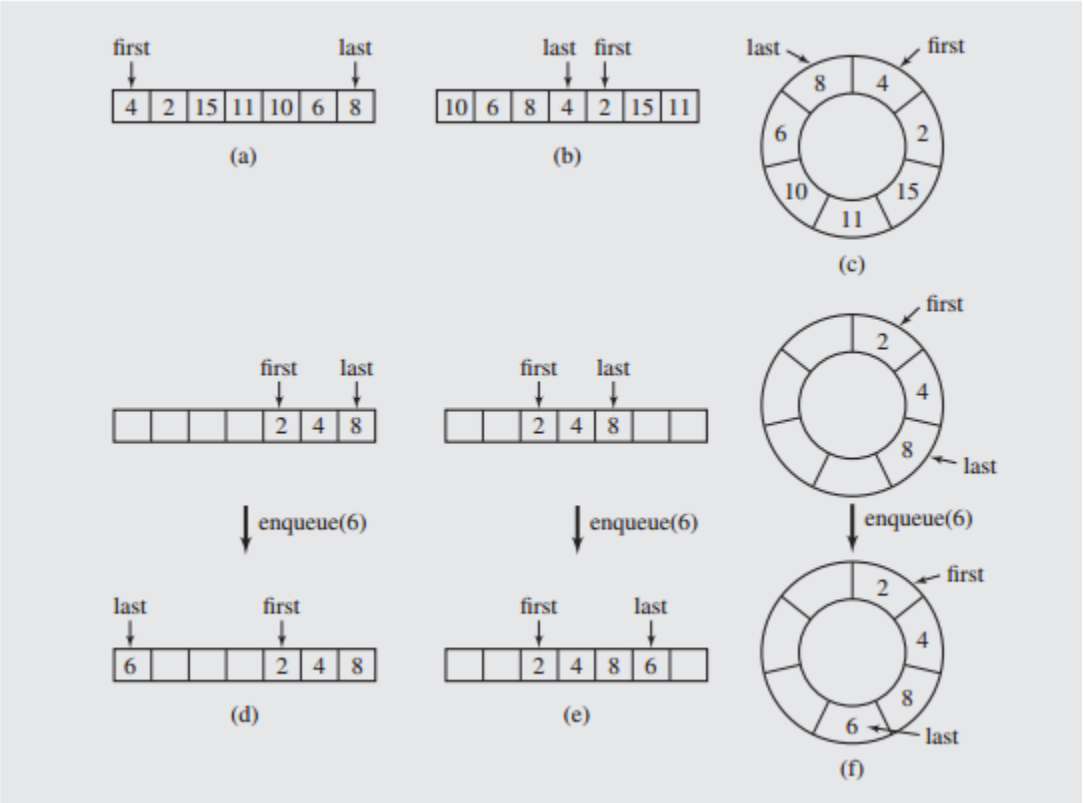


Figure 2.3.2 (a–b) Two possible configurations in an array implementation of a queue when the queue is full. (c) The same queue viewed as a circular array. (f) Enqueuing number6 to a queue storing 2, 4, and 8. (d–e) The same queue seen as a one-dimensional array with the last element (d) at the end of the array and (e) in the middle.

Figure 2.3.3 contains possible implementations of member functions that operate on queues.

A more natural queue implementation is a doubly linked list, as offered in the previous chapter and also in STL's list (Figure 2.3.4).

In both suggested implementations enqueuing and dequeuing can be executed in constant time O(1), provided a doubly linked list is used in the list implementation.

In the singly linked list implementation, dequeuing requires O(n) operations primarily to scan the list and stop at the next to last node.

```
//*********************  genArrayQueue.h  ***********************
//              generic queue implemented as an array

#ifndef ARRAY_QUEUE
#define ARRAY_QUEUE

template<class T, int size = 100>
class ArrayQueue {
public:
    ArrayQueue() {
        first = last = -1;
    }
    void enqueue(T);
    T dequeue();
    bool isFull()  {
        return first == 0 && last == size-1 || first == last + 1;
    }
    bool isEmpty() {
        return first == -1;
    }
private:
    int first, last;
    T storage[size];
};

template<class T, int size>
void ArrayQueue<T,size>::enqueue(T el) {
   if (!isFull())
        if (last == size-1 || last == -1) {
            storage[0] = el;
            last = 0;
            if (first == -1)
                first = 0;
        }
        else storage[++last] = el;
   else cout << "Full queue.\n";
}

template<class T, int size>
T ArrayQueue<T,size>::dequeue() {

    T tmp;
    tmp = storage[first];
    if (first == last)
         last = first = -1;
    else if (first == size-1)
         first = 0;
    else first++;
    return tmp;
}

#endif
```

Figure 2.3.3 Array implementation of a queue

```
//***********************  genQueue.h  ***************************
//     generic queue implemented with doubly linked list

#ifndef DLL_QUEUE
#define DLL_QUEUE

#include <list>

template<class T>
class Queue {
public:
    Queue() {
    }
    void clear() {
        lst.clear();
    }
    bool isEmpty() const {
        return lst.empty();
    }
    T& front() {
        return lst.front();
    }


    T dequeue() {
        T el = lst.front();
        lst.pop_front();
        return el;
    }
    void enqueue(const T& el) {
        lst.push_back(el);
    }
private:
    list<T> lst;
};

#endif
```

Figure 2.3.4 Linked list implementation of a queue.

Figure 2.3.5 shows the same sequence of enqueue and dequeue operations as Figure 2.3.1, and indicates the changes in the queue implemented as an array(Figure 2.3.5b) and as a linked list (Figure 2.3.5c). The linked list keeps only the numbers that the logic of the queue operations indicated by Figure 2.3.5a requires. The array includes all the numbers until it fills up, after which new numbers are included starting from the beginning of the array.

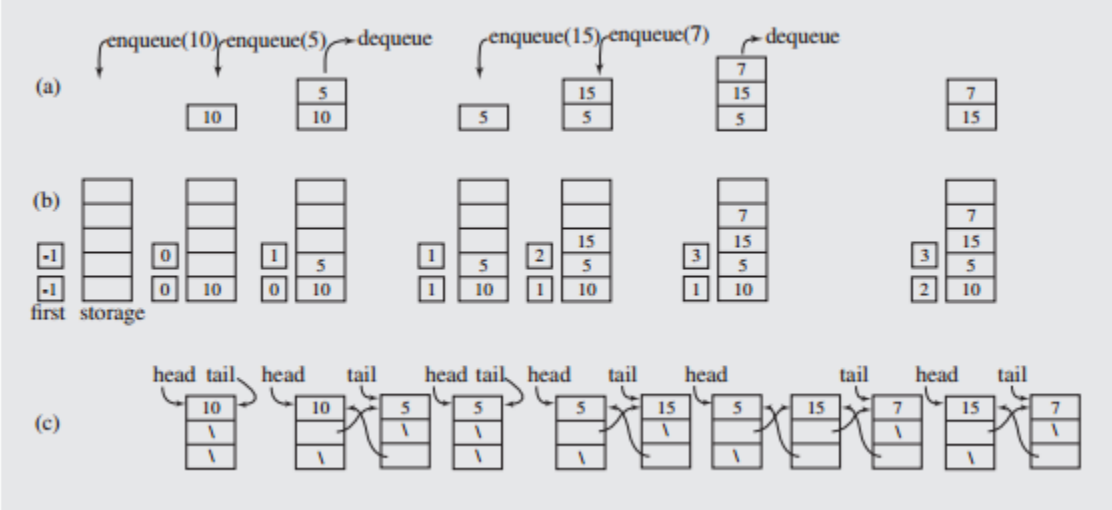


*Figure 2.3.5*A series of operations executed on (a) an abstract queue and the queue implemented (b) with an array and (c) with a linked list.

Queues are frequently used in simulations to the extent that a well-developed and mathematically sophisticated theory of queues exists, called *queuing theory,* in which various scenarios are analyzed and models are built that use queues. In queuing processes there are a number of customers coming to servers to receive service. The throughput of the server may be limited. Therefore, customers have to wait in queues before they are served, and they spend some amount of time while they are being served. By customers, we mean not only people, but also objects. For example, parts on an assembly line in the process of being assembled into a machine, truck waiting for service at a weighing station on an interstate, or barges waiting for a sluic to be opened so they can pass through a channel also wait in queues. The most familiar examples are lines in stores, post offices, or banks. The types of problems posed in simulations are: How many servers are needed to avoid long queues? How large must the waiting space be to put the entire queue in it? Is it cheaper to increase this space or to open one more server?

As an example, consider Bank One which, over a period of three months, recorded the number of customers coming to the bank and the amount of time needed to serve them. The table in Figure 2.3.6a shows the number of customers who arrive during one-minute intervals throughout the day. For 15% of such intervals, no customers arrived, for 20%, only one arrived, and so on. Six clerks were employed, no lines were ever observed, and the bank management wanted to know whether six clerks were too many. Would five suffice? Four? Maybe even three? Can lines be expected at any time? To answer these questions, a simulation program was written that applied the recorded data and checked different scenarios.

Number of Customers Per Minute	Percentage of One-Minute Intervals	Range
0	15	1–15
1	20	16–35
2	25	36–60
3	10	61–70
4	30	71–100

(a)

Amount of Time Needed for Service in Seconds	Percentage of Customers	Range
0	0	—
10	0	—
20	0	—
30	10	1–10
40	5	11–15
50	10	16–25
60	10	26–35
70	0	—
80	15	36–50
90	25	51–75
100	10	76–85
110	15	86–100

(b)

Figure 2.3.6 Bank One example: (a) data for number of arrived customers per one-minute interval and (b) transaction time in seconds per customer.

The number of customers depends on the value of a randomly generated number between 1 and 100. The table in Figure 2.3.6a identifies five ranges of numbers from 1 to 100, based on the percentages of one-minute intervals that had 0, 1, 2, 3, or4 customers. If the random number is 21, then the number of

customers is 1; if the random number is 90, then the number of customers is 4. This method simulates the rate of customers arriving at Bank One.

In addition, analysis of the recorded observations indicates that no customer required10-second or 20-second transactions, 10% required 30 seconds, and so on, as indicated in Figure 2.3.6b. The table in 2.3.6b includes ranges for random numbers to generate the length of a transaction in seconds.

Figure 2.3.7 contains the program simulating customer arrival and transaction time at Bank One. The program uses three arrays. arrivals[] records the percentages of one-minute intervals depending on the number of the arrived customers. The array service[] is used to store the distribution of time needed for service. The amount of time is obtained by multiplying the index of a given array cell by 10. For example, service[3] is equal to 10, which means that 10% of the time a customer required 3 · 10 seconds for service. The array clerks[] records the length of transaction time in seconds.

```
#include <iostream>
#include <cstdlib>

using namespace std;

#include "genQueue.h"
int option(int percents[]) {
    register int i = 0, choice = rand()%100+1, perc;
    for (perc = percents[0]; perc < choice; perc += percents[i+1], i++);
    return i;
}

int main() {
    int arrivals[] = {15,20,25,10,30};
    int service[] = {0,0,0,10,5,10,10,0,15,25,10,15};
    int clerks[] = {0,0,0,0}, numOfClerks = sizeof(clerks)/sizeof(int);
    int customers, t, i, numOfMinutes = 100, x;
    double maxWait = 0.0, currWait = 0.0, thereIsLine = 0.0;
    Queue<int> simulQ;
    cout.precision(2);
    for (t = 1; t <= numOfMinutes; t++) {
```

```
        cout << " t = " << t;
        for (i = 0; i < numOfClerks; i++)// after each minute subtract
            if (clerks[i] < 60)          // at most 60 seconds from time
                clerks[i] = 0;           // left to service the current
            else clerks[i] -= 60;        // customer by clerk i;
        customers = option(arrivals);
        for (i = 0; i < customers; i++) {// enqueue all new customers
            x = option(service)*10;      // (or rather service time
            simulQ.enqueue(x);           // they require);
            currWait += x;
        }
        // dequeue customers when clerks are available:
        for (i = 0; i < numOfClerks && !simulQ.isEmpty(); )
           if (clerks[i] < 60)        {
                x = simulQ.dequeue();  // assign more than one customer
                clerks[i] += x;        // to a clerk if service time
                currWait  -= x;        // is still below 60 sec;
           }
           else i++;
        if (!simulQ.isEmpty()) {
             thereIsLine++;
             cout << " wait = " << currWait/60.0;
             if (maxWait < currWait)
                 maxWait = currWait;
        }
        else cout << " wait = 0;";
        }
        cout << "\nFor " << numOfClerks << " clerks, there was a line "
             << thereIsLine/numOfMinutes*100.0 << "% of the time;\n"
             << "maximum wait time was " << maxWait/60.0 << " min.";
        return 0;
}
```

Figure 2.3.7 Bank One example: implementation code.

For each minute (represented by the variable t), the number of arriving customers are randomly chosen, and for each customer, the transaction time is also randomly determined. The function option() generates a random number, finds the range into which it falls, and then outputs the position, which is either the number of customers or a tenth of the number of seconds.

Executions of this program indicate that six and five clerks are too many. With four clerks, service is performed smoothly; 25% of the time there is a short line of waiting customers. However, three clerks are always busy and there is always a long line of customers waiting. Bank management would certainly decide to employ four clerks.

Priority Queues

In many situations, simple queues are inadequate, because first in/first out scheduling has to be overruled using some priority criteria. In a post office example, a handicapped person may have priority over others. Therefore, when a clerk is available, a handicapped person is served instead of someone from the front of the queue. On roads with tollbooths, some vehicles may be put through immediately, even without paying (police cars, ambulances, fire engines, and the like). In a sequence of processes, process P2 may need to be executed before process P1 for the proper functioning of a system, even though P1 was put on the queue of waiting processes before P2. In situations like these, a modified queue, or priority queue, is needed. In priority queues, elements are dequeued according to their priority and their current queue position.

The problem with a priority queue is in finding an efficient implementation that allows relatively fast enqueuing and dequeuing. Because elements may arrive randomly to the queue, there is no guarantee that the front elements will be the most likely to be dequeued and that the elements put at the end will be the last candidates for dequeuing. The situation is complicated because a wide spectrum of possible priority criteria can be used in different cases such as frequency of use, birthday, salary, position, status, and others. It can also be the time of scheduled execution on the queue of processes, which explains the convention used in priority queue discussions in which higher priorities are associated with lower numbers indicating priority.

Member Function	Operation
`bool empty() const`	Return `true` if the stack includes no element and `false` otherwise.
`void pop()`	Remove the top element of the stack.
`void push(const T& el)`	Insert `el` at the top of the stack.
`size_type size() const`	Return the number of elements on the stack.
`stack()`	Create an empty stack.
`T& top()`	Return the top element on the stack.
`const T& top() const`	Return the top element on the stack.

Figure 2.3.8 A list of stack member functions

Member Function	Operation
`T& back()`	Return the last element in the queue.
`const T& back() const`	Return the last element in the queue.
`bool empty() const`	Return `true` if the queue includes no element and `false` otherwise.
`T& front()`	Return the first element in the queue.
`const T& front() const`	Return the first element in the queue.
`void pop()`	Remove the first element in the queue.
`void push(const T& el)`	Insert `el` at the end of the queue.
`queue()`	Create an empty queue.
`size_type size() const`	Return the number of elements in the queue.

Figure 2.3.9 A list of queue member functions.

```
#include <iostream>
#include <queue>
#include <list>

using namespace std;

int main() {
    queue<int> q1;
    queue<int,list<int> > q2; //leave space between angle brackets > >
    q1.push(1); q1.push(2); q1.push(3);
    q2.push(4); q2.push(5); q2.push(6);
    q1.push(q2.back());
    while (!q1.empty()) {
        cout << q1.front() << ' ';    // 1 2 3 6
        q1.pop();
    }
    while (!q2.empty()) {
        cout << q2.front() << ' ';    // 4 5 6
        q2.pop();
    }
    return 0;
}
```

Figure 2.3.10 An example application of queue's member functions.

Member Function	Operation
`bool empty() const`	Return `true` if the queue includes no element and `false` otherwise.
`void pop()`	Remove an element in the queue with the highest priority.
`void push(const T& el)`	Insert `el` in a proper location on the priority queue.
`priority_queue(comp f())`	Create an empty priority queue that uses a two-argument Boolean function `f` to order elements on the queue.
`priority_queue(iterator first, iterator last, comp f())`	Create a priority queue that uses a two-argument Boolean function `f` to order elements on the queue; initialize the queue with elements from the range indicated by iterators `first` and `last`.
`size_type size() const`	Return the number of elements in the priority queue.
`T& top()`	Return the element in the priority queue with the highest priority.
`const T& top() const`	Return the element in the priority queue with the highest priority.

Figure 2.3.11 A list of priority_queue member functions.

```
#include <iostream>
#include <queue>
#include <functional>

using namespace std;

int main() {
    priority_queue<int> pq1; // plus vector<int> and less<int>
    priority_queue<int,vector<int>,greater<int> > pq2;
    pq1.push(3); pq1.push(1); pq1.push(2);
    pq2.push(3); pq2.push(1); pq2.push(2);
    int a[] = {4,6,5};
    priority_queue<int> pq3(a,a+3);
    while (!pq1.empty()) {
        cout << pq1.top() << ' ';    // 3 2 1
        pq1.pop();
    }
    while (!pq2.empty()) {
        cout << pq2.top() << ' ';    // 1 2 3
        pq2.pop();
    }
    while (!pq3.empty()) {
        cout << pq3.top() << ' ';    // 6 5 4
        pq3.pop();
    }
    return 0;
}
```

Figure 2.3.12 A program that uses member functions of the container priority_queue

References:

"Data Structures and Algorithm in C++" 2nd Edition , Michael T. Goodrich, Roberto Tamassia and David Mount

"Fundamentals of Data Structures in C++", E. Horowits, S. Sahni and D. Mehta

"Principles of Data Structures using C and C++", New Age, Vinu V Das, MES College of Engineering, Kuttipuran, Kerala India

"Data Structures Using C++", 2nd Edition, D.S Malik

"Data Structures and Algorithm Analysis", Edition 3.2 (C++ Version), Clifford A Shaffer, Department of Computer Science, Virginia Tech Blacksburg, VA 24061

"Practical Introduction to Data Structures and Algorithm Analysis", C++ Edition, 2nd Edition by Cli!ord A. Sha!er. Prentice Hall, 2000

"Foundations of Multidimensional and Metric Data Structures", by Hanan Samet Morgan Kaufmann, 2006.

"Introduction to Data Structures", Tim French, CITS2200

"Handouts on Fundamenta Data Structures", IS103 Computational Thinking, SMU Singapore Management University

"C++ Language Tutorial", Juan Soulie, http://www.cplusplus.com/doc/tutorial

"Open Data Structures in C++", Edition 0.1G, Pat Morin