第17章

经典抽象数据类型

有些抽象数据类型(ADT)是C程序员不可或缺的工具，这是由于它们的属性决定的。这类ADT有链表、堆栈、队列和树等。第12章已经讨论了链表，本章我们将讨论剩余的ADT。

本章的第1部分描述了这些结构的属性和基本实现方法。在本章的最后，我们将探讨如何提高它们在实现上的灵活性以及由此导致的安全性能的妥协。

17.1 内存分配

所有的ADT都必须确定一件事情——如何获取内存来存储值。有三种可选的方案：静态数组、动态分配的数组和动态分配的链式结构。

静态数组要求结构的长度固定。而且，这个长度必须在编译时确定。但是，这个方案最为简单，而且最不容易出错。

如果你使用动态数组，你可以在运行时才决定数组的长度。而且，如果需要的话，你可以通过分配一个新的、更大的数组，把原来数组的元素复制到新数组中，然后删除原先的数组，从而达到动态改变数组长度的目的。在决定是否采用动态数组时，你需要在由此增加的复杂性和随之产生的灵活性（不需要一个固定的、预定确定的长度）之间作一番权衡。

最后，链式结构提供了最大程度的灵活性。每个元素在需要时才单独进行分配，所以除了不能超过机器的可用内存之外，这种方式对元素的数量几乎没有什么限制。但是，链式结构的链接字段需要消耗一定的内存，在链式结构中访问一个特定元素的效率不如数组。

17.2 堆栈

堆栈（stack）这种数据最鲜明的特点就是其后进先出(Last-In First-Out, LIFO)的方式。参加Party的人们对堆栈是很熟悉的。主人的车道就是一个汽车的堆栈，最后一辆进入车道的汽车必须首先开出，第1辆进入车道的汽车只有等其余所有车辆都开走后才能开出。

bbs.theithome.com

17.2.1 堆栈接口

基本的堆栈操作通常被称为 push 和 pop。push 就是把一个新值压入到堆栈的顶部，pop 就是把堆栈顶部的值移出堆栈并返回这个值。堆栈只提供对它的顶部值的访问。

在传统的堆栈接口中，访问顶部元素的唯一方法就是把它移除。另一类堆栈接口提供三种基本的操作：push、pop 和 top。push 操作和前面描述的一样，pop 只把顶部元素从堆栈中移除，它并不返回这个值。top 返回顶部元素的值，但它并不把顶部元素从堆栈中移除。

提示：

传统的 pop 函数具有一个副作用：它将改变堆栈的状态。它也是访问堆栈顶部元素的唯一方法。top 函数允许你反复访问堆栈顶部元素的值而不必把它保存在一个局部变量中。这个例子再次说明了设计不带副作用的函数的好处。

我们需要两个额外的函数来使用堆栈。一个空堆栈不能执行 pop 操作，所以我们需要一个函数告诉我们堆栈是否为空。在实现堆栈时如果存在最大长度限制，那么我们也需要另一个函数告诉我们堆栈是否已满。

17.2.2 实现堆栈

堆栈是最容易实现的 ADT 之一。它的基本方法是当值被 push 到堆栈时把它们存储于数组中连续的位置上。你必须记住最近一个被 push 的值的下标。如果需要执行 pop 操作，你只需要简单地减少这个下标值就可以了。程序 17.1 的头文件描述了一个堆栈模块的非传统接口。

提示：

注意接口只包含了用户使用堆栈所需要的信息，特别是它并没有展示堆栈的实现方式。事实上，对这个头文件稍作修改（稍后讨论），它可以用于所有三种实现方式。用这种方式定义接口是一种好方法，因为它防止用户以为它依赖于某种特定的实现方式。

```
/*
** 一个堆栈模块的接口
*/

#define STACK_TYPE int/* 堆栈所存储的值的类型 */

/*
**  push
**把一个新值压入到堆栈中。它的参数是需要被压入的值。
*/
voidpush( STACK_TYPE value );

/*
**  pop
**从堆栈弹出一个值，并将其丢弃。
*/
void pop( void );

/*
```


bbs.theithome.com

```
**  top
**    返回堆栈顶部元素的值，但不对堆栈进行修改。
*/
STACK_TYPE top( void );

/*
**  is_empty
**    如果堆栈为空，返回 TRUE，否则返回 FALSE。
*/
int is_empty( void );

/*
**  is_full
**  如果堆栈已满，返回 TRUE，否则返回 FALSE。
*/
int is_full( void );
```

程序 17.1　堆栈接口　　stack.h

提示：

这个接口的一个有趣特性是存储于堆栈中的值的类型的声明方式。在编译这个堆栈模块之前，用户可以修改这个类型以适合自己的需要。

一、数组堆栈

在程序 17.2 中，我们的第 1 种实现方式是使用一个静态数组。堆栈的长度以一个#define 定义的形式出现，在模块被编译之前用户必须对数组长度进行设置。我们后面所讨论的堆栈实现方案就没有这个限制。

提示：

所有不属于外部接口的内容都被声明为 static，这可以防止用户使用预定义接口之外的任何方式访问堆栈中的值。

```
/*
** 用一个静态数组实现的堆栈。数组的长度只能通过修改#define 定义
** 并对模块重新进行编译来实现。
*/

#include "stack.h"
#include <assert.h>

#define   STACK_SIZE     100/* 堆栈中值数量的最大限制 */

/*
** 存储堆栈中值的数组和一个指向堆栈顶部元素的指针。
*/
static    STACK_TYPE     stack[ STACK_SIZE ];
static    int            top_element = -1;

/*
**  push
*/
void
```


bbs.theithome.com

```
push( STACK_TYPE value )
{
      assert( !is_full() );
      top_element += 1;
      stack[ top_element ] = value;
}


/*
** pop
*/
void
pop( void )
{
      assert( !is_empty() );
      top_element -= 1;
}


/*
**  top
*/
STACK_TYPE top( void )
{
      assert( !is_empty() );
      return stack[ top_element ];
}


/*
**    is_empty
*/
int
is_empty( void )
{
      return top_element == -1;
}


/*
**    is_full
*/
int
is_full( void )
{
      return top_element == STACK_SIZE - 1;
}
```

程序 17.2　用静态数组实现堆栈　　a_stack.c

变量 top_element 保存堆栈顶部元素的下标值。它的初始值为-1，提示堆栈为空。push 函数在存储新元素前先增加这个变量的值，这样 top_element 始终包含顶部元素的下标值。如果它的初始值为 0，top_element 将指向数组的下一个可用位置。这种方式当然也可行，但它的效率稍差一些，因此它需要执行一次减法运算才能访问顶部元素。

一种简单明了的传统 pop 函数的写法如下所示：

bbs.theithome.com

```
STACK_TYPE
pop( void )
{
        STACK_TYPE temp;

        assert( !is_empty() );
        temp = stack[ top_element ];
        top_element -= 1;
        return temp;
}
```

这些操作的顺序是很重要的。top_element 在元素被复制出数组之后才减 1，这和 push 相反，后者是在被元素复制到数组之前先加 1。我们可以通过消除这个临时变量以及随之带来的复制操作来提高效率：

```
assert( !is_empty() );
return stack[ top_element-- ];
```

pop 函数不需要从数组中删除元素——只减少顶部指针的值就足矣，因为用户此时已不能再访问这个旧值了。

提示：

这个堆栈模块的一个值得注意的特性是它使用了 assert 来防止非法操作，诸如从一个空堆栈弹出元素或者向一个已满的堆栈压入元素。这个断言调用 is_full 和 is_empty 函数而不是测试 top_element 本身。如果你以后决定以不同的方法来检测空堆栈和满堆栈，使用这种方法显然要容易很多。

对于用户无法消除的错误，使用断言是很合适的。但如果用户希望确保程序不会终止，那么程序向堆栈压入一个新值之前必须检测堆栈是否仍有空间。因此，断言必须只能够对那些用户自己也能进行检查的内容进行检查。

二、动态数组堆栈

接下来的这种实现方式使用了一个动态数组，但我们首先需要在接口中定义两个新函数：

```
/*
** create_stack
** 创建堆栈。参数指定堆栈可以保存多少个元素。
** 注意: 这个函数并不用于静态数组版本的堆栈。
*/
void  create_stack( size_t  size );

/*
** destroy_stack
** 销毁堆栈。它释放堆栈所使用的内存。
** 注意: 这个函数也不用于静态数组版本的堆栈。
*/
void   destroy_stack( void );
```

第 1 个函数用于创建堆栈，用户向它传递一个参数，用于指定数组的长度。第 2 个函数用于删除堆栈，为了避免内存泄漏，这个函数是必需的。

这些声明可以添加到 stack.h 中，尽管前面的堆栈实现中并没有定义这两个函数。注意，用户在使用静态数组类型的堆栈时并不存在错误地调用这两个函数的危险，因为它们在那个模块中并

bbs.theithome.com

不存在。

提示：

一个更好的方法是把不需要的函数在数组模块中以存根的形式实现。如此一来，这两种实现方式的接口将是相同的，因此从其中一个转换到另一个会容易一些。

有趣的是，使用动态分配数组在实现上改动得并不多（见程序 17.3）。数组由一个指针代替，程序引入 stack_size 变量保存堆栈的长度。它们在缺省情况下都初始化为零。

create_stack 函数首先检查堆栈是否已经创建。然后分配所需数量的内存并检查分配是否成功。destroy_stack 在释放内存之后把长度和指针变量重新设置为零，这样它们可以用于创建另一个堆栈。

模块剩余部分的唯一改变是在 is_full 函数中与 stack_size 变量进行比较而不是与常量 STACK_SIZE 进行比较，并且在 is_full 和 is_empty 函数中都增加了一条断言。这条断言可以防止任何堆栈函数在堆栈被创建前就被调用。其余的堆栈函数并不需要这条断言，因为它们都调用了这两个函数之一。

```
/*
** 一个用动态分配数组实现的堆栈
** 堆栈的长度在创建堆栈的函数被调用时给出，该函数必须在任何其他操作堆栈的函数被调用之前调用。
*/
#include "stack.h"
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <assert.h>
/*
** 用于存储堆栈元素的数组和指向堆栈顶部元素的指针。
*/
static    STACK_TYPE    *stack;
static    size_t        stack_size;
static    int           top_element = -1;

/*
**  create_stack
*/
void
create_stack( size_t size )
{
      assert( stack_size == 0 );
      stack_size = size;
      stack = malloc( stack_size * sizeof( STACK_TYPE ) );
      assert( stack != NULL );
}

/*
**  destroy_stack
*/
void
destroy_stack( void )
{
      assert( stack_size > 0 );
      stack_size = 0;
      free( stack );
```


bbs.theithome.com

```
        stack = NULL;
}

/*
**  push
*/
void
push( STACK_TYPE value )
{
        assert( !is_full() );
        top_element += 1;
        stack[ top_element ] = value;
}

/*
**    pop
*/
void
pop( void )
{
        assert( !is_empty() );
        top_element -= 1;
}

/*
**    top
*/
STACK_TYPE top( void )
{
        assert( !is_empty() );
        return stack[ top_element ];
}

/*
**    is_empty
*/
int
is_empty( void )
{
        assert( stack_size > 0 );
        return top_element == -1;
}

/*
**    is_full
*/
int
is_full( void )
{
        assert( stack_size > 0 );
        return top_element == stack_size - 1;
}
```

程序 17.3　用动态数组实现堆栈　　d_stack.c

警告：

在内存有限的环境中，使用 assert 检查内存分配是否成功并不合适，因此它很可能导致程序终

bbs.theithome.com

止，这未必是你希望的结果。一种替代策略是从 create_stack 函数返回一个值，提示内存分配是否成功。当这个函数失败时，用户程序可以用一个较小的长度再试一次。

三、链式堆栈

由于只有堆栈的顶部元素才可以被访问，所以使用单链表就可以很好地实现链式堆栈。把一个新元素压入到堆栈是通过在链表的起始位置添加一个元素实现的。从堆栈中弹出一个元素是通过从链表中移除第 1 个元素实现的。位于链表头部的元素总是很容易被访问。

在程序 17.4 所示的实现中，不再需要 create_stack 函数，但可以实现 destroy_stack 函数用于清空堆栈。由于用于存储元素的内存是动态分配的，它必须予以释放以避免内存泄漏。

```
/*
**  一个用链表实现的堆栈。这个堆栈没有长度限制。
*/
#include "stack.h"
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <assert.h>

#define   FALSE 0

/*
**  定义一个结构以存储堆栈元素，其中 link 字段将指向堆栈的下一个元素。
*/
typedef   struct   STACK_NODE {
          STACK_TYPE    value;
          struct STACK_NODE *next;
} StackNode;

/*
**  指向堆栈中第一个节点的指针。
*/
static    StackNode*stack;

/*
**    create_stack
*/
void
create_stack( size_t size )
{
}

/*
**    destroy_stack
*/
void
destroy_stack( void )
{
     while( !is_empty() )
          pop();
```


bbs.theithome.com

```
}

/*
**    push
*/
void
push( STACK_TYPE value )
{
      StackNode     *new_node;

      new_node = malloc( sizeof( StackNode ) );
      assert( new_node != NULL );
      new_node->value = value;
      new_node->next = stack;
      stack = new_node;
}

/*
**    pop
*/
void
pop( void )
{
      StackNode*first_node;

      assert( !is_empty() );
      first_node = stack;
      stack = first_node->next;
      free( first_node );
}

/*
**    top
*/
STACK_TYPE top( void )
{
      assert( !is_empty() );
      return stack->value;
}

/*
**    is_empty
*/
int
is_empty( void )
{
      return stack == NULL;
}

/*
**    is_full
*/
int
is_full( void )
{
```


bbs.theithome.com

```
return FALSE;
}
```

程序 17.4　用链表实现堆栈　　l_stack.c

STACK_NODE 结构用于把一个值和一个指针捆绑在一起，而 stack 变量是一个指向这些结构变量之一的指针。当 stack 指针为 NULL 时，堆栈为空，也就是初始时的状态。

提示：

destroy_stack 函数连续从堆栈中弹出元素，直到堆栈为空。同样，注意这个函数使用了现存的 is_empty 和 pop 函数而不是重复那些用于实际操作的代码。

create_stack 是一个空函数，由于链式堆栈不会填满，所以 is_full 函数始终返回假。

17.3　队列

队列和堆栈的顺序不同：队列是一种先进先出(First-IN First-OUT, FIFO)的结构。排队就是一种典型的队列。首先轮到的是排在队伍最前面的人，新入队的人总是排在队伍的最后。

17.3.1　队列接口

和堆栈不同，在队列中，用于执行元素的插入和删除的函数并没有被普遍接受的名字，所以我们将使用 insert 和 delete 这两个名字。同样，对于插入应该在队列的头部还是尾部也没有完全一致的意见。从原则上说，你在队列的哪一端插入并没有区别。但是，在队列的尾部插入以及在头部删除更容易记忆一些，因为它准确地描述了人们在排队时的实际体验。

在传统的接口中，delete 函数从队列的头部删除一个元素并将其返回。在另一种接口中，delete 函数从队列的头部删除一个元素，但并不返回它。first 函数返回队列第 1 个元素的值但并不将它从队列中删除。

程序 17.5 的头文件定义了后面那种接口。它包括链式和动态分配实现的队列需要使用的 create_queue 和 destroy_queue 函数的原型。

```
/*
**  一个队列模块的接口
*/

#include <stdlib.h>

#define   QUEUE_TYPE     int/* 队列元素的类型 */

/*
**  create_queue
**    创建一个队列，参数指定队列可以存储的元素的最大数量。
**  注意: 这个函数只适用于使用动态分配数组的队列。
*/
void create_queue( size_t size );

/*
**  destroy_queue
**    销毁一个队列。注意: 这个函数只适用于链式和动态分配数组的队列。
```


bbs.theithome.com

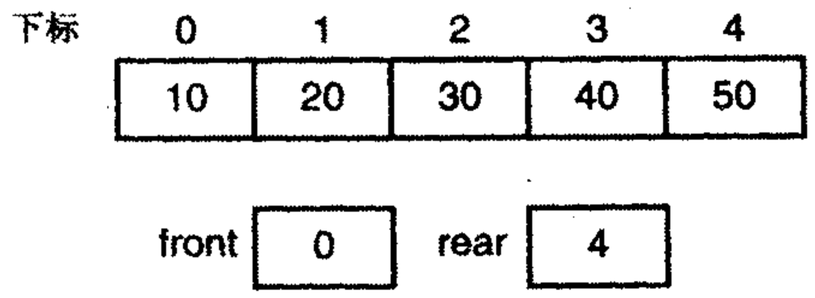
```
*/
void destroy_queue( void );

/*
** insert
**    向队列添加一个新元素，参数就是需要添加的元素。
*/
void insert( QUEUE_TYPE value );

/*
** delete
**    从队列中移除一个元素并将其丢弃。
*/
void delete( void );

/*
** first
**    返回队列中第一个元素的值，但不修改队列本身。
*/
QUEUE_TYPE first( void );

/*
** is_empty
**    如果队列为空，返回 TRUE，否则返回 FALSE。
*/
int   is_empty( void );

/*
** is_full
**    如果队列已满，返回 TRUE，否则返回 FALSE。
*/
int   is_full( void );
```

程序 17.5　队列接口　　queue.h

17.3.2　实现队列

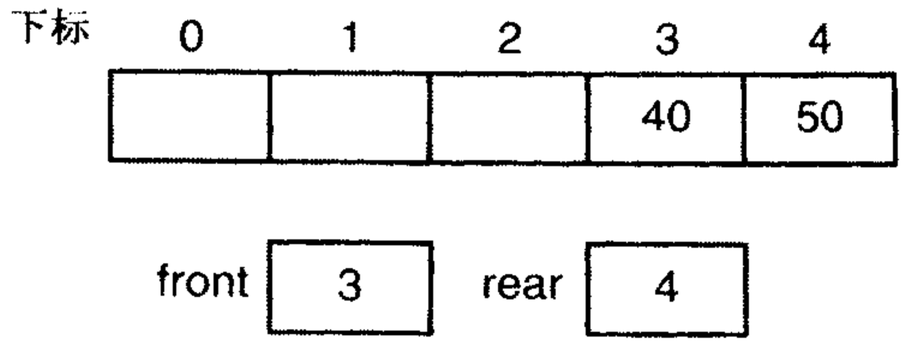
队列的实现比堆栈要难得多。它需要两个指针——一个指向队头，一个指向队尾。同时，数组并不像适合堆栈那样适合队列的实现，这是由于队列使用内存的方式决定的。

堆栈总是扎根于数据的一端。但是，当队列的元素插入和删除时，它所使用的是数组中的不同元素。考虑一个用 5 个元素的数组实现的队列。下面的图是 10、20、30、40 和 50 这几个值插入队列以后队列的样子。



经过三次删除之后，队列的样子如下所示：

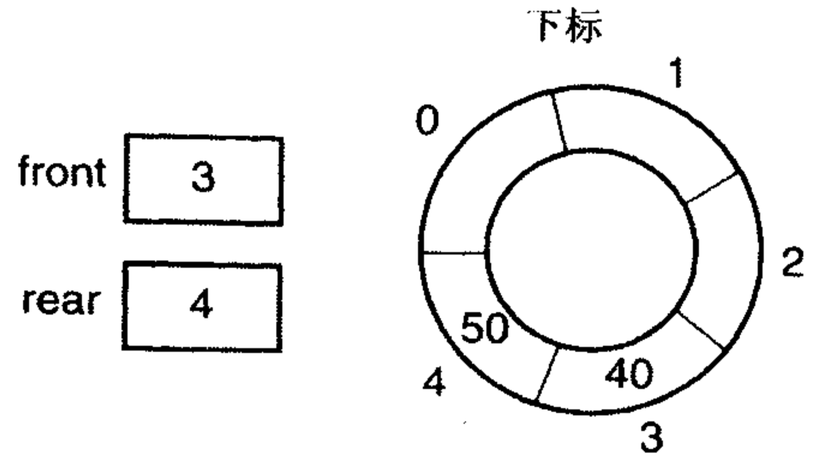
bbs.theithome.com



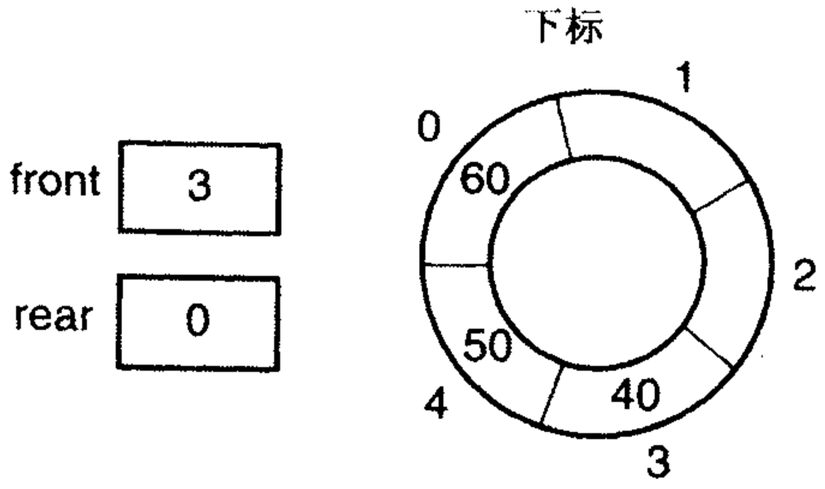
数组并未满，但它的尾部已经没有空间，无法再插入新的元素。

这个问题的一种解决方法是当一个元素被删除之后，队列中的其余元素朝数组起始位置方向移动一个位置。由于复制元素所需的开销，这种方法几乎不可行，尤其是那些较大的队列。

一个好一点的方案是让队列的尾部“环绕”到数组的头部，这样新元素就可以存储到以前删除元素所留出来的空间中。这个方法常常被称为循**环数组**(circular array)。下图说明了这个概念。



插入另一个元素之后的结果如下：



循环数组很容易实现——当尾部下标移出数组尾部时，把它设置为 0。用下面的代码便可以实现。

```
rear += 1;
if( rear >= QUEUE_SIZE )
        rear = 0;
```

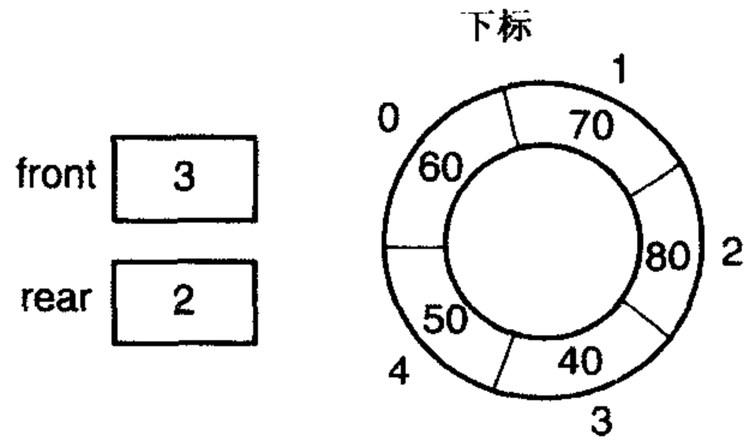
下面的方法具有相同的结果。

```
rear = ( rear + 1 ) % QUEUE_SIZE;
```

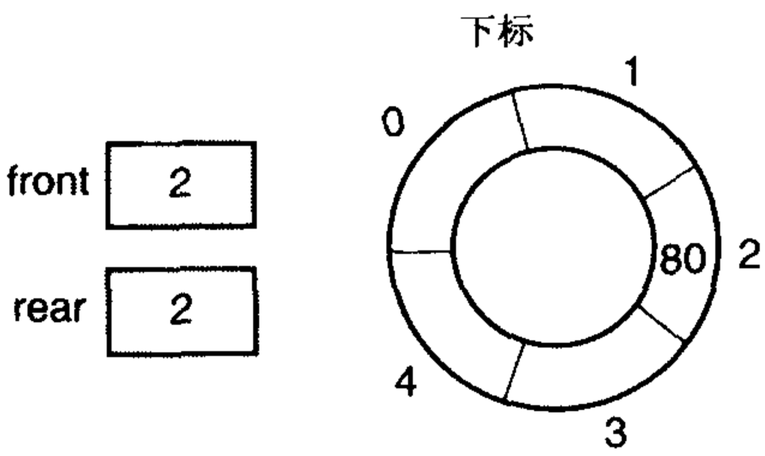
在对 front 增值时也必须使用同一个技巧。

但是，循环数组自身也引入了一个问题。它使得判断一个循环数组是否为空或者已满更为困难。假定队列已满，如下图所示：

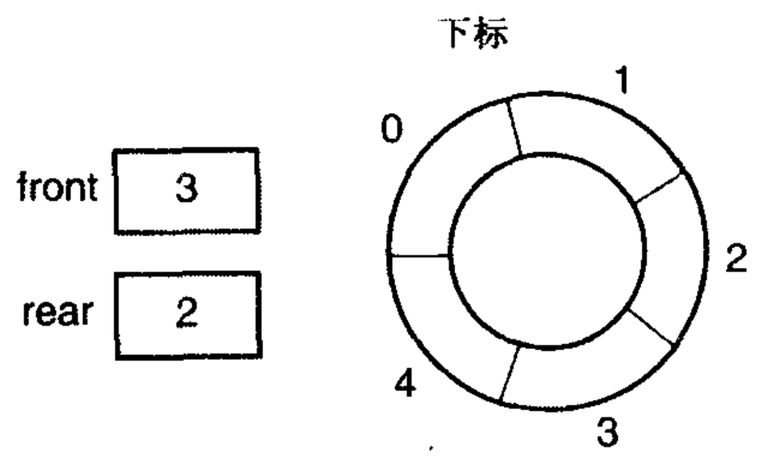
bbs.theithome.com



注意 front 和 rear 的值：分别是 3 和 2。如果有 4 个元素从队列中删除，front 将增值 4 次，队列中的情况如下图所示：



当最后一个元素被删除时，队列中的情况如下图所示：



问题是现在 front 和 rear 的值是相同的，这和队列已满时的情况是一样的。当队列空或者满时对 front 和 rear 进行比较，其结果都是真。所以，我们无法通过比较 front 和 rear 来测试队列是否为空。

有两种方法可以解决这个问题。第 1 种是引入一个新变量，用于记录队列中的元素数量。它在每次插入元素时加 1，在每次删除元素时减 1。对这个变量的值进行测试就可以很容易分清队列空间为空还是已满。

第 2 种方法是重新定义“满”的含义。如果使数组中的一个元素始终保留不用，这样当队列“满”时 front 和 rear 的值便不相同，可以和队列为空时的情况区分开来。通过不允许数组完全填满，问题便得以避免。

不过还是留下一个小问题：当队列为空时，front 和 rear 的值应该是什么？当队列只有一个元素时，我们需要使 front 和 rear 都指向这个元素。一次插入操作将增加 rear 的值，所以为了使 rear 在第 1 次插入后指向这个插入的元素，当队列为空时 rear 的值必须比 front 小 1。幸运的是，从队列中删除最后一个元素后的状态也是如此，因此删除最后一个元素并不会造成一种特殊情况。

当满足下面的条件时，队列为空：

bbs.theithome.com

```
( rear + 1 ) % QUEUE_SIZE == front
```

由于在 front 和 rear 正好满足这个关系之前，我们必须停止插入元素，所以当满足下列条件时，队列必须认为已“满”。

```
( rear + 2 ) % QUEUE_SIZE == front
```

一、数组队列

程序 17.6 用一个静态数组实现了一个队列。它使用“不完全填满数组”的技巧来区分空队列和满队列。

```
/*
**  一个用静态数组实现的队列。数组的长度只能通过修改#define 定义并重新编译模块来调整。
*/

#include "queue.h"
#include <stdio.h>
#include <assert.h>

#define   QUEUE_SIZE     100/* 队列中元素的最大数量 */
#define   ARRAY_SIZE     ( QUEUE_SIZE + 1 )/* 数组的长度 */

/*
**  用于存储队列元素的数组和指向队列头和尾的指针。
*/
static    QUEUE_TYPE     queue[ ARRAY_SIZE ];
static    size_t         front = 1;
static    size_t         rear = 0;

/*
**    insert
*/
void
insert( QUEUE_TYPE value )
{
      assert( !is_full() );
      rear = ( rear + 1 ) % ARRAY_SIZE;
      queue[ rear ] = value;
}

/*
**    delete
*/
void
delete( void )
{
      assert( !is_empty() );
      front = ( front + 1 ) % ARRAY_SIZE;
}

/*
**    first
*/
QUEUE_TYPE first( void )
```


bbs.theithome.com

```
{
      assert( !is_empty() );
      return queue[ front ];
}

/*
**    is_empty
*/
int
is_empty( void )
{
      return ( rear + 1 ) % ARRAY_SIZE == front;
}

/*
**    is_full
*/
int
is_full( void )
{
      return ( rear + 2 ) % ARRAY_SIZE == front;
}
```

程序 17.6　用静态数组实现队列　　a_queue.c

QUEUE_SIZE 常量设置为用户希望队列可以容纳的元素的最大数量。由于这种实现方式永远不会真正填满队列，ARRAY_SIZE 的值被定义为比 QUEUE_SIZE 大 1。这些函数是我们所讨论的那些技巧的简单明了的实现。

我们可以使用任何值初始化 front 和 rear，只要 rear 比 front 小 1。程序 17.6 所使用的初始值使数组的第 1 个元素保留不用，直到 rear 第 1 次“环绕”至数组头部，猜猜接下来会怎样？

二、动态数组队列和链式队列

用动态数组实现队列所需要的修改和堆栈的情况类似。因此，它的实现留作练习。

链式队列在几个方面比数组形式的队列简单。它不使用数组，所以不存在循环数组的问题。测试队列是否为空只是简单测试链表是否为空就可以了。测试队列是否已满的结果总是假，链式队列的实现也留作练习。

17.4　树

对树的所有种类作完整的描述超出了本书的范围。但是，通过描述一种非常有用的树：**二叉搜索树**(binary search tree)，可以很好地说明实现树的技巧。

树是一种数据结构，它要么为空，要么具有一个值并具有零个或多个**孩子**(child)，每个孩子本身也是树。这个递归的定义正确地提示了一棵树的高度并没有内在的限制。**二叉树**(binary tree)是树的一种特殊形式，它的每个节点至多具有两个孩子，分别称为**左孩子**(left)和**右孩子**(right)。二叉搜索树具有一个额外的属性：每个节点的值比它的左子树的所有节点的值都要大，但比它的右子树的所有节点的值都要小。注意这个定义排除了树中存在值相同的节点的可能性。这些属性使二叉搜索树成为一种用关键值快速查找数据的优秀工具。图 17.1 是二叉搜索树的一个例子。这棵树的每个节

bbs.theithome.com

点都正好具有一个双亲节点（它的上层节点），零个、一个或两个孩子（直接在它下面的节点）。唯一的例外是最上面的那个节点，称为树根，它没有双亲节点。没有孩子的节点被称为**叶节点**(leaf node)或**叶子**(leaf)。在绘制树时，根位于顶端，叶子位于底部[1]。

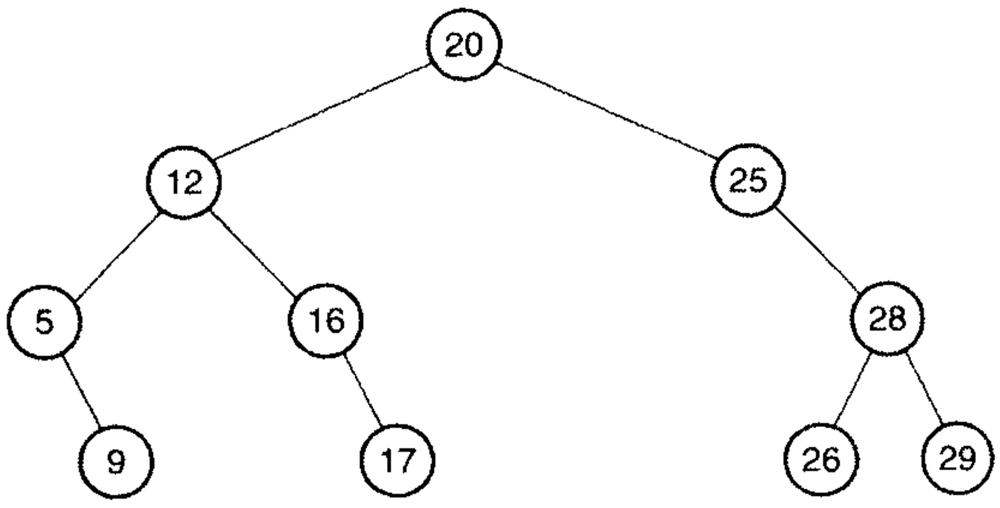


图 17.1　二叉搜索树

17.4.1　在二叉搜索树中插入

当一个新值添加到一棵二叉搜索树时，它必须被放在合适的位置，继续保持二叉搜索树的属性。幸运的是，这个任务是很简单的。其基本算法如下所示：

```
如果树为空:
    把新值作为根节点插入
否则:
    如果新值小于当前节点的值:
        把新值插入到当前节点的左子树
    否则:
        把新值插入到当前节点的右子树
```

这个算法的递归表达正是树的递归定义的直接结果。

为了把 15 插入到图 17.1 的树，把 15 和 20 比较。15 更小，所以它被插入到左子树。左子树的根为 12，因此重复上述过程：把 15 和 12 比较。这次 15 更大，所以它被插入到 12 的右子树。现在我们把 15 和 16 比较。15 更小，所以插入到节点 16 的左子树。但这个子树是空的，所以包含 15 的节点便成为节点 16 的新左子树的根节点。

提示：

由于递归在算法的尾部出现（尾部递归），所以我们可以使用迭代更有效地实现这个算法。

17.4.2　从二叉搜索树删除节点

从树中删除一个值比从堆栈或队列删除一个值更为困难。从一棵树的中部删除一个节点将导致它的子树和树的其余部分断开——我们必须重新连接它们，否则它们将会丢失。

我们必须处理三种情况：删除没有孩子的节点；删除只有一个孩子的节点；删除有两个孩子的节点。第 1 个情况很简单，删除一个叶节点不会导致任何子树断开，所以不存在重新连接的问题。

[1] 注意这和自然世界中根在底叶在上的树实际上是颠倒的。

bbs.theithome.com

删除只有一个孩子的节点几乎同样容易：把这个节点的双亲节点和它的孩子连接起来就可以了。这个解决方法防止了子树的断开，而且仍能维持二叉搜索树的次序。

最后一种情况要困难得多。如果一个节点有两个孩子，它的双亲不能连接到它的两个孩子。解决这个问题的一种策略是不删除这个节点，而是删除它的左子树中值最大的那个节点，并用这个值代替原先应被删除的那个节点的值。删除函数的实现留作练习。

17.4.3　在二叉搜索树中查找

由于二叉搜索树的有序性，所以在树中查找一个特定的值是非常容易的。下面是它的算法：

```
如果树为空:
    这个值不存在于树中
否则:
    如果这个值和根节点的值相等:
        成功找到这个值
    否则:
        如果这个值小于根节点的值:
            查找左子树
        否则:
            查找右子树
```

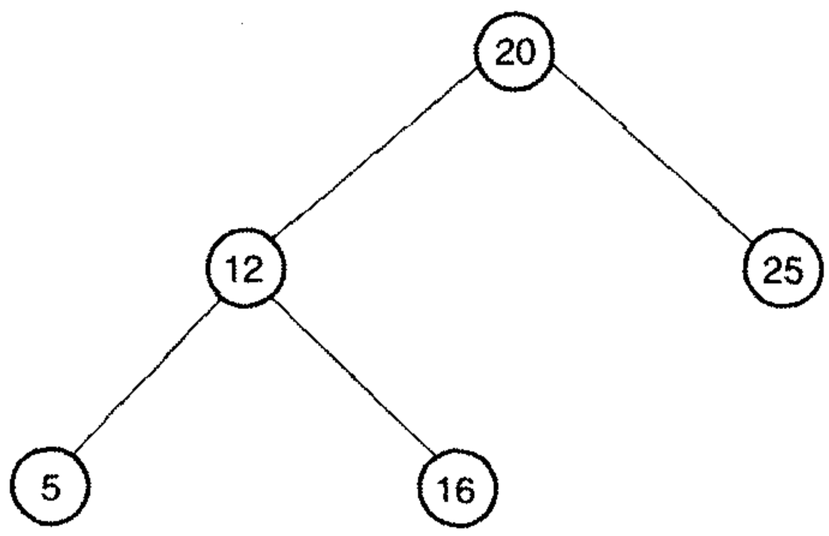
这个递归算法也属于尾部递归，所以采用迭代方案来实现效率更高。

当值被找到时你该做些什么呢？这取决于用户的需要。有时，用户只需要确定这个值是否存在于树中。这时，返回一个真/假值就足够了。如果数据是一个由一个关键值字段标识的结构，用户需要访问这个查找到的结构的非关键值成员，这就要求函数返回一个指向该结构的指针。

17.4.4　树的遍历

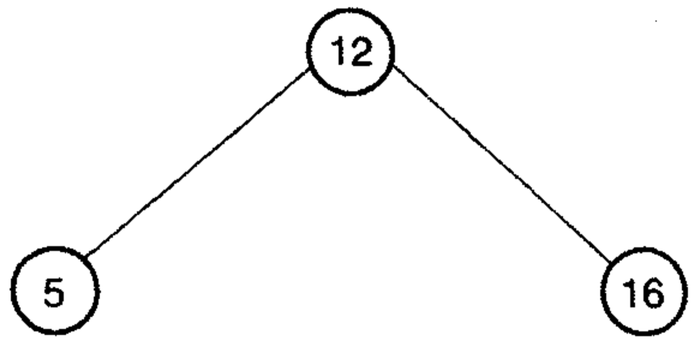
和堆栈和队列不同，树并未限制你只能访问一个值。因此树具有另一个基本操作——**遍历**(traversal)。当你检查一棵树的所有节点时，你就在遍历这棵树。遍历树的节点有几种不同的次序，最常用的是**前序**(pre-order)、**中序**(in-order)、**后序**(post-order)和**层次遍历**(breadth-first)。所有类型的遍历都是从树的根节点或你希望开始遍历的子树的根节点开始。

前序遍历检查节点的值，然后递归地遍历左子树和右子树。例如，下面这棵树的前序遍历



将从处理 20 这个值开始。然后我们再遍历它的左子树：

bbs.theithome.com



在处理完 12 这个值之后，我们继续遍历它的左子树

并处理 5 这个值。它的左右子树皆为空，所以我们就完成了这棵子树的遍历。
在完成节点 12 的左子树遍历之后，我们继续遍历它的右子树：

并处理 16 这个值。它的左右子树皆为空，这意味着我们已经完成了根为 16 的子树和根为 12 的子树的遍历。

在完成了节点 20 的左子树遍历之后，下一个步骤就是处理它的右子树：

处理完 25 这个值以后便完成了整棵树的遍历。

对于一个较大的例子，考虑图 17.1 的二叉搜索树。当检查每个节点时打印出它的值，那么它的前序遍历的输出结果将是：20，12，5，9，16，17，25，28，26，29。

中序遍历首先遍历左子树，然后检查当前节点的值，最后遍历右子树。图 17.1 的树的中序遍历结果将是：5，9，12，16，17，20，25，26，28，29。

后序遍历首先遍历左右子树，然后检查当前节点的值。图 17.1 的树的后序遍历结果将是：9，5，17，16，12，26，29，28，25，20。

最后，层次遍历逐层检查树的节点。首先处理根节点，接着是它的孩子，再接着是它的孙子，依次类推。用这种方法遍历图 17.1 的树的次序是：20，12，25，5，16，28，9，17，26，29。虽然前三种遍历方法可以很容易地使用递归来实现，但最后这种层次遍历要采用一种使用队列的迭代算法。本章的练习对它有更详细的描述。

17.4.5 二叉搜索树接口

程序 17.7 的接口提供了用于把值插入到一棵二叉搜索树的函数的原型。它同时包含了一个 find 函数用于查找树中某个特定的值，它的返回值是一个指向找到的值的指针。它只定义了一个遍历函数，因为其余遍历函数的接口只是名字不同而已。

```
/*
**  二叉搜索树模块的接口
*/
```


bbs.theithome.com

```
#define   TREE_TYPE int     /* 树的值类型 */

/*
** insert
**    向树添加一个新值。参数是需要被添加的值，它必须原先并不存在于树中。
*/
void insert( TREE_TYPE value );

/*
** find
**    查找一个特定的值，这个值作为第 1 个参数传递给函数。
*/
TREE_TYPE *find( TREE_TYPE value );

/*
** pre order traverse
**执行树的前序遍历。它的参数是一个回调函数指针，它所指向的函数将在树中处理每
**个节点被调用，节点的值作为参数传递给这个函数。
*/
void pre_order_traverse(void (*callback)( TREE_TYPE value ));
```

程序 17.7　二叉搜索树接口　　tree.h

17.4.6　实现二叉搜索树

尽管树的链式实现是最为常见的，但将二叉搜索树存储于数组中也是完全可能的。当然，数组的固定长度限制了你可以插入到树中的元素的数量，但如果你使用动态数组，当原先的数组溢出时，你可以创建一个更大的空间并把值复制给它。

一、数组形式的二叉搜索树

用数组表示树的关键是使用下标来寻找某个特定值的双亲和孩子。规则很简单：

节点 N 的双亲是节点 N/2。
节点 N 的左孩子是节点 2N。
节点 N 的右孩子是节点 2N+1。

双亲节点的公式是成立的，因为整除操作符将截去小数部分。

警告：

唉！这里有个小问题。这些规则假定树的根节点是第 1 个节点，但 C 的数组下标从 0 开始。最容易的解决方案是忽略数组的第 1 个元素。如果元素非常大，这种方法将浪费很多空间，如果这样你可以使用基于零下标数组的另一套规则：

节点 N 的双亲节点是节点(N + 1)/2-1。
节点 N 的左孩子节点是节点 2N+1。
节点 N 的右孩子节点是节点 2N+2。

程序 17.8 是一个用静态数组实现的二叉搜索树。这个实现方法有几个有趣之处。它使用第 1 种更简单的规则来确定孩子节点，这样数组声明的长度比宣称的长度大 1，它的第 1 个元素被忽略。它定义了一些函数访问一个节点的左右孩子。尽管计算很简单，这些函数名还是让使用这些函数的代码看上去更清晰。这些函数同时简化了修改模块以便使用其他规则的任务。

bbs.theithome.com

这种实现方法使用 0 这个值提示一个节点未被使用。如果 0 是一个合法的数据值，那就必须另外挑选一个不同的值，而且数组元素必须进行动态初始化。另一个技巧是使用一个比较数组，它的元素是布尔型值，用于提示哪个节点被使用。

数组形式的树的问题在于数组空间常常利用得不够充分。空间之所以被浪费是由于新值必须插入到树中特定的位置，无法随便放置到数组中的空位置。

为了说明这种情况，假定我们使用一个拥有 100 个元素的数组来容纳一棵树。如果值 1，2，3，4，5，6 和 7 以这个次序插入，它们将分别存储在数组中 1，2，4，8，16，32 和 64 的位置。但现在值 8 不能被插入，因为 7 的右孩子将存储于位置 128，数组的长度没有那么长。这个问题会不会实际发生取决于值插入的顺序。如果相同的值以这样的顺序插入：4，2，1，3，6，5 和 7，它们将占据数组 1 至 7 的位置，这样插入 8 这个值便毫无困难。

使用动态分配的数组，当需要更多空间时我们可以对数组进行重新分配。但是，对于一棵不平衡的树，这个技巧并不是一个好的解决方案，因为每次新插入都将导致数组的大小扩大一倍，这样可用于动态分配的内存很快便会耗尽。一个更好的方法是使用链式二叉树而不是数组。

```
/*
** 一个使用静态数组实现的二叉搜索树。数组的长度只能通过修改#define 定义
** 并对模块进行重新编译来实现。
*/
#include "tree.h"
#include <assert.h>
#include <stdio.h>

#define   TREE_SIZE 100 /* Max # of values in the tree */
#define   ARRAY_SIZE    ( TREE_SIZE + 1 )

/*
**  用于存储树的所有节点的数组。
*/
static    TREE_TYPE tree[ ARRAY_SIZE ];

/*
** left_child
** 计算一个节点左孩子的下标。
*/
static int
left_child( int current )
{
     return current * 2;
}

/*
** right_child
**    计算一个节点右孩子的下标。
*/
static int
right_child( int current )
{
     return current * 2 + 1;
}
```


bbs.theithome.com

```
/*
** insert
*/
void
insert( TREE_TYPE value )
{
      int  current;

      /*
      ** 确保值为非零，因为零用于提示一个未使用的节点。
      */
      assert( value != 0 );

      /*
      **  从根节点开始。
      */
      current = 1;

      /*
      **  从合适的子树开始，直到到达一个叶节点。
      */
      while( tree[ current ] != 0 ){
      /*
      ** 根据情况，进入叶节点或右子树（确信未出现重复的值）。
      */
           if( value < tree[ current ] )
                current = left_child( current );
           else {
                assert( value != tree[ current ] );
                current = right_child( current );
           }
           assert( current < ARRAY_SIZE );
      }

      tree[ current ] = value;
}

/*
** find
*/
TREE_TYPE *
find( TREE_TYPE value )
{
      int  current;

      /*
      ** 从根节点开始。直到找到那个值，进入合适的子树。
      */
      current = 1;

      while( current < ARRAY_SIZE && tree[ current ] != value ){
      /*
      ** 根据情况，进入左子树或右子树。
      */
         if( value < tree[ current ] )
              current = left_child( current );
```


bbs.theithome.com

```
            else
                current = right_child( current );
        }
        if( current < ARRAY_SIZE )
            return tree + current;
        else
            return 0;
}

/*
**  do_pre_order_traverse
**    执行一层前序遍历，这个帮助函数用于保存我们当前正在处理的节点的信息，
**    它并不是用户接口的一部分。
*/
static void
do_pre_order_traverse( int current,
    void (*callback)( TREE_TYPE value ) )
{
        if( current < ARRAY_SIZE && tree[ current ] != 0 ){
            callback( tree[ current ] );
            do_pre_order_traverse( left_child( current ),
                callback );
            do_pre_order_traverse( right_child( current ),
                callback );
        }
}

/*
** pre_order_traverse
*/
void
pre_order_traverse( void (*callback)( TREE_TYPE value ) )
{
    do_pre_order_traverse( 1, callback );
}
```

程序 17.8　用静态数组实现二叉搜索树　　a_tree.c

二、链式二叉搜索树

队列的链式实现消除了数组空间利用不充分的问题，这是通过为每个新值动态分配内存并把这些结构链接到树中实现的。因此，不存在不使用的内存。

程序 17.9 是二叉搜索树的链式实现方法。请将它和程序 17.8 的数组实现方法进行比较。由于树中的每个节点必须指向它的左右孩子，所以节点用一个结构来容纳值和两个指针。数组由一个指向树根节点的指针代替。这个指针最初为 NULL，表示此时为一棵空树。

insert 函数使用两个指针[1]。第 1 个用于检查树中的节点，寻找新值插入的合适位置。第 2 个指针指向另一个节点，后者的 link 字段指向当前正在检查的节点。当到达一个叶节点时，这个指针必须进行修改以插入新节点。这个函数自上而下，根据新值和当前节点值的比较结果选择进入左子树或右子树，直到到达叶节点。然后，创建一个新节点并链接到树中。这个迭代算法在插入第 1 个节

[1] 我们使用了和第 12 章的函数中把值插入到一个有序的单链表的相同技巧。如果你沿着从根到叶的路径观察插入发生的位置，你就会发现它本质上就是一个单链表。

bbs.theithome.com

点时也能正确处理，不会造成特殊情况。

三、树接口的变型

find 函数只用于验证值是否存在于树中。返回一个指向找到元素的指针并无大用，因为调用程序已经知道这个值：它就是传递给函数的参数嘛！

假定树中的元素实际上是一个结构，它包括一个关键值和一些数据。现在我们可以修改 find 函数，使它更加实用。通过它的关键值查找一个特定的节点并返回一个指向该结构的指针可以向用户提供更多的信息——与这个关键值相关联的数据。但是，为了取得这个结果，find 函数必须设法只比较每个节点元素的关键值部分。解决办法是编写一个函数执行这个比较，并把一个指向该函数的指针传递给 find 函数，就像我们在 qsort 函数中所采取的方法一样。

有时候用户可能要求自己遍历整棵树，例如，计算每个节点的孩子数量。因此，TreeNode 结构和指向树根节点的指针都必须声明为公用，以便用户遍历该树。最安全的方法是通过函数向用户提供根指针，这样可以防止用户自行修改根指针，从而导致丢失整棵树。

```
/*
** 一个使用动态分配的链式结构实现的二叉搜索树。
*/
#include "tree.h"
#include <assert.h>
#include <stdio.h>
#include <malloc.h>

/*
**  TreeNode 结构包含了值和两个指向某个树节点的指针。
*/
typedef struct TREE_NODE {
        TREE_TYPE   value;
        struct TREE_NODE *left;
        struct TREE_NODE *right;
} TreeNode;

/*
**  指向树根节点的指针。
*/
static    TreeNode  *tree;

/*
**    insert
*/
void
insert( TREE_TYPE value )
{
      TreeNode *current;
      TreeNode **link;

      /*
      **  从根节点开始。
      */
      link = &tree;
```


bbs.theithome.com

```
        /*
        ** 持续查找值，进入合适的子树。
        */
        while( (current = *link) != NULL ){
        /*
        **  根据情况，进入左子树或右子树（确认没有出现重复的值）
        */
            if( value < current->value )
                link = &current->left;
            else {
                assert( value != current->value );
                link = &current->right;
            }
}

/*
** 分配一个新节点，使适当节点的 link 字段指向它。
*/
current = malloc( sizeof( TreeNode ) );
assert( current != NULL );
current->value = value;
current->left = NULL;
current->right = NULL;
*link = current;
}

/*
**    find
*/
TREE_TYPE *
find( TREE_TYPE value )
{
    TreeNode    *current;

    /*
    **  从根节点开始，直到找到这个值，进入合适的子树。
    */
    current = tree;

while( current != NULL && current->value != value ){
/*
**  根据情况，进入左子树或右子树。
*/
    if( value < current->value )
            current = current->left;
    else
            current = current->right;
  }

  if( current != NULL )
            return &current->value;
  else
       return NULL;
}

/*
```


bbs.theithome.com

```
** do_pre_order_traverse
**  执行一层前序遍历。这个帮助函数用于保存我们当前正在处理的节点的信息。
**  这个函数并不是用户接口的一部分。
*/
static void
do_pre_order_traverse( TreeNode *current,
    void (*callback)( TREE_TYPE value ) )
{
    if( current != NULL ){
        callback( current->value );
        do_pre_order_traverse( current->left, callback );
        do_pre_order_traverse( current->right, callback );
    }
}

/*
**    pre_order_traverse
*/
void
pre_order_traverse( void (*callback)( TREE_TYPE value ) )
{
    do_pre_order_traverse( tree, callback );
}
```

程序 17.9　链式二叉搜索树　　l_tree.c

让每个树节点拥有一个指向它的双亲节点的指针常常是很有用的。用户可以利用这个双亲节点指针在树中上下移动。这种更为开放的树的 find 函数可以返回一个指向这个树节点的指针而不是节点值，这就允许用户利用这个指针执行其他形式的遍历。

程序的最后一个可供改进之处是用一个 destroy_tree 函数释放所有分配给这棵树的内存。这个函数的实现留作练习。

17.5　实现的改进

本章的实现方法说明了不同的 ADT 是如何工作的。但是，当它们用于现实的程序时，它们在好几个方面是不够充分的。本节的目的是找出这些问题并建议如何解决它们。我们使用数组形式的堆栈作为例子，但这里所讨论的技巧适用于其他所有 ADT。

17.5.1　拥有超过一个的堆栈

到目前为止的实现中，最主要的一个问题是它们把用于保存结构的内存和那些用于操纵它们的函数都封装在一起了。这样一来，一个程序便不能拥有超过一个的堆栈！

这个限制很容易解决，只要从堆栈的实现模块中去除数组和 top_element 的声明，并把它们放入用户代码即可。然后，它们通过参数被堆栈函数访问，这些函数便不再固定于某个数组。用户可以创建任意数量的数组，并通过调用堆栈函数将它们作为堆栈使用。

警告：

这个方法的危险之处在于它失去了封装性。如果用户拥有数据，他可以直接访问它。非法的访问，例如在一个错误的位置向数组增加一个新值或者增加一个新值但并不调整 top_element，都有可

bbs.theithome.com

能导致数据丢失或者非法数据或者导致堆栈函数运行失败。

一个相关的问题是当每个堆栈函数被调用时，用户应该确保向它传递正确的堆栈和 top_element 参数。如果这些参数发生混淆，其结果就是垃圾。我们可以通过把堆栈数组和它的 top_element 值捆绑在一个结构里来减少这种情况发生的可能性。

当堆栈模块包含数据时，就不存在出现上述两种问题的危险性。本章的练习部分描述了一个修改方案，允许堆栈模块管理超过一个的堆栈。

17.5.2 拥有超过一种的类型

即使前面的问题得以解决，存储于堆栈的值的类型在编译时便已固定，它就是 stack.h 头文件中所定义的类型。如果你需要一个整数堆栈和一个浮点数堆栈，你就没那么幸运了。

解决这个问题最简单的方法是另外编写一份堆栈函数的拷贝，用于处理不同的数据类型。这种方法可以达到目的，但它涉及大量重复代码，这就使得程序的维护工作变得更为困难。

一种更为优雅的方法是把整个堆栈模块实现为一个#define 宏，把目标类型作为参数。这个定义然后便可以用于创建每种目标类型的堆栈函数。但是，为了使这种解决方案得以运作，我们必须找到一种方法为不同类型的堆栈函数产生独一无二的函数名，这样它们相互之间就不会冲突。同时，你必须小心在意，对于每种类型只能创建一组函数，不管你实际需要多少个这种类型的堆栈。这种方法的一个例子在第 17.5.4 节描述。

第 3 种方法是使堆栈与类型无关，方法是让它存储 void *类型的值。将整数和其他数据都按照一个指针的空间进行存储，使用强制类型转换把参数的类型转换为 void *后再执行 push 函数，top 函数返回的值再转换回原先的类型。为了使堆栈也适用于较大的数据（例如结构），你可以在堆栈中存储指向数据的指针。

警告：

这种方法的问题是它绕过了类型检查。我们没有办法证实传递给 push 函数的值正是堆栈所使用的正确类型。如果一个整数意外地压入到一个元素类型为指针的堆栈中，其结果几乎肯定是一场灾难。

使树模块与类型无关更为困难一些，因为树函数必须比较树节点的值。但是，我们可以向每个树函数传递一个指向由用户编写的比较函数的指针。同样，传递一个错误的指针也会造成灾难性的后果。

17.5.3 名字冲突

堆栈和队列模块都拥有 is_full 和 is_empty 函数，队列和树模块拥有 insert 函数。如果你需要向树模块增加一个 delete 函数，它就会与原先存在于队列模块的 delete 函数发生冲突。

为了使它们共存于程序中，所有这些函数的名字都必须是独一无二的。但是，人们有一种强烈的愿望，在尽可能的情况下，让那些和每个数据结构关联的函数都保持“标准”名字。这个问题的解决方法是一种妥协方案：选择一种命名约定，使它既可以为人们所接受又能保证唯一性。例如，is_queue_empty 和 is_stack_empty 名字就解决了这个问题。它们的不利之处在于这些长名字使用起来不太方便，它们并未传递任何附加信息。

bbs.theithome.com

17.5.4　标准函数库的 ADT

计算机科学虽然不是一门古老的学科，但我们对它的研究显然已经花费了相当长的时间，对堆栈和队列的行为的方方面面已经研究得相当透彻了。那么，为什么每个人还需要自己编写堆栈和队列函数呢？为什么这些 ADT 不是标准函数库的一部分呢？

其原因正是我们刚刚讨论过的三个问题。名字冲突问题很容易解决，但是，类型安全性的缺乏以及让用户直接操纵数据的危险性使得用一种通用而又安全的方式编写实现堆栈的库函数变得极不可行。

解决这个问题就要求实现泛型(genericity)，它是一种编写一组函数，但数据的类型暂时可以不确定的能力。这组函数随后用用户需要的不同类型进行实例化(instantiated)或创建。C 语言并未提供这种能力，但我们可以使用#define 定义近似地模拟这种机制。

程序 17.10a 包含了一个#define 宏，它的宏体是一个数组堆栈的完整实现。这个#define 宏的参数是需要存储的值的类型、一个后缀以及需要使用的数组长度。后缀用于粘贴到由实现定义的每个函数名的后面，用于避免名字冲突。

程序 17.10b 使用程序 10.7a 的声明创建两个堆栈，一个可以容纳 10 个整数，另一个可以容纳 5 个浮点数。当每个#define 宏被扩展时，一组新的堆栈函数被创建，用于操作适当类型的数据。但是，如果需要两个整数堆栈，这种方法将会创建两组相同的函数。

我们将程序 17.10a 进行改写，把它分成三个独立的宏：一个用于声明接口，一个用于创建操纵数据的函数，一个用于创建数据。当我们需要第 1 个整数堆栈时，所有三个宏均被使用。当我们还需要另外的整数堆栈时，通过重复调用最后一个宏来实现。堆栈的接口也应该进行修改。函数必须接受一个附加的参数用于指定进行操作的堆栈。这些修改都留作练习。

这个技巧使得创建泛型抽象数据类型库成为可能。但是，这种灵活性是要付出代价的。用户需要承担几个新的责任。现在，他必须：

1．采用一种命名约定，避免不同类型间堆栈的名字冲突。

2．必须保证为每种不同类型的堆栈只创建一组堆栈函数。

3．在访问堆栈时，必须保证使用适当的名字（例如，push_int 或 push_float 等）。

4．确保向函数传递正确的堆栈数据结构。

毫不吃惊的是，用 C 语言实现泛型是相当困难的，因为它的设计远早于泛型这个概念被提出之时。泛型是面向对象编程语言处理得比较完美的问题之一。

```
/*
** 用静态数组实现一个泛型的堆栈。数组的长度当堆栈实例化时作为参数给出。
*/
#include <assert.h>

#define   GENERIC_STACK( STACK_TYPE, SUFFIX, STACK_SIZE )       \

          static    STACK_TYPE     stack##SUFFIX[ STACK_SIZE ]; \
          static    int            top_element##SUFFIX = -1;    \
                                                                \
          int                                                   \
          is_empty##SUFFIX( void )                              \
          {                                                     \
```


bbs.theithome.com

```
        return top_element##SUFFIX == -1;                    \
    }                                                        \
                                                             \
    int                                                      \
    is_full##SUFFIX( void )                                  \
    {                                                        \
        return top_element##SUFFIX == STACK_SIZE - 1;        \
    }                                                        \
                                                             \
    void                                                     \
    push##SUFFIX( STACK_TYPE value )                         \
    {                                                        \
        assert( !is_full##SUFFIX() );                        \
        top_element##SUFFIX += 1;                            \
        stack##SUFFIX[ top_element##SUFFIX ] = value;        \
    }                                                        \
                                                             \
    void                                                     \
    pop##SUFFIX( void )                                      \
    {                                                        \
        assert( !is_empty##SUFFIX() );                       \
        top_element##SUFFIX -= 1;                            \
    }                                                        \
                                                             \
    STACK_TYPE top##SUFFIX( void )                           \
    {                                                        \
        assert( !is_empty##SUFFIX() );                       \
        return stack##SUFFIX[ top_element##SUFFIX ];         \
    }
```

程序 17.10a　泛型数组堆栈　　g_stack.h

```
/*
**  一个使用泛型堆栈模块创建两个容纳不同类型数据的堆栈的用户程序。
*/
#include <stdlib.h>
#include <stdio.h>
#include "g_stack.h"

/*
**  创建两个堆栈，一个用于容纳整数，另一个用于容纳浮点数。
*/
GENERIC_STACK( int, _int, 10 )
GENERIC_STACK( float, _float, 5 )

int
main()
{
    /*
    **  往每个堆栈压入几个值。
    */
    push_int( 5 );
    push_int( 22 );
    push_int( 15 );
    push_float( 25.3 );
    push_float( -40.5 );
```


bbs.theithome.com

```
/*
** 清空整数堆栈并打印这些值。
*/
while( !is_empty_int() ){
     printf( "Popping %d\n", top_int() );
     pop_int();
}

/*
 ** 清空浮点数堆栈并打印这些值。
 */
while( !is_empty_float() ){
     printf( "Popping %f\n", top_float() );
     pop_float();
}

     return EXIT_SUCCESS;
}
```

程序 17.10b　使用泛型数组堆栈　　g_client.c

17.6　总结

为 ADT 分配内存有三种技巧：静态数组、动态分配的数组和动态分配的链式结构。静态数组对结构施加了预先确定固定长度这个限制。动态数组的长度可以在运行时计算，如果需要数组也可以进行重新分配。链式结构对值的最大数量并未施加任何限制。

堆栈是一种后进先出的结构。它的接口提供了把新值压入堆栈的函数和从堆栈弹出值的函数。另一类接口提供了第 3 个函数，它返回栈顶元素的值但并不将其中堆栈中弹出。堆栈很容易使用数组来实现，我们可以使用一个变量，初始化为-1，用它记住栈顶元素的下标。为了把一个新值压入到堆栈中，这个变量先进行增值，然后这个值被存储到数组中。当弹出一个值时，在访问栈顶元素之后，这个变量进行减值。我们需要两个额外的函数来使用动态分配的数组。一个用于创建指定长度的堆栈，另一个用于销毁它。单链表也能很好地实现堆栈。通过在链表的头部插入，可以实现堆栈的压入。通过删除第 1 个元素，可以实现堆栈的弹出。

队列是一种先进先出的结构。它的接口提供了插入一个新值和删除一个现有值的函数。由于队列对它的元素所施加的次序限制，用循环数组来实现队列要比使用普通数组合适得多。当一个变量被当作循环数组的下标使用时，如果它处于数组的末尾再增值时，它的值就“环绕”到零。为了判断数组是否已满，你可以使用一个用于计数已经插入到队列中的元素数量的变量。为了使用队列的 front 和 rear 指针来检测这种情况，数组应始终至少保留一个空元素。

二叉搜索树(BST)是一种数据结构，它或者为空，或者具有一个值并拥有零个、一个或两个孩子（分别称为左孩子和右孩子），它的孩子本身也是一棵 BST。BST 树节点的值大于它的左孩子所有节点的值，但小于它的右孩子所有节点的值。由于这种次序关系的存在，在 BST 中查找一个值是非常高效的——如果节点并未包含需要查找的值，你总是可以知道接下来应该查找它的哪棵子树。为了向 BST 插入一个值，你首先进行查找。如果值未找到，就把它插入到查找失败的位置。当你从 BST 删除一个节点时，必须小心防止把它的子树同树的其他部分断开。树的遍历就是以某种次序处理它

bbs.theithome.com

的所有节点。有 4 种常见的遍历次序。前序遍历先处理节点，然后遍历它的左子树和右子树。中序遍历先遍历节点的左子树，然后处理该节点，最后遍历节点的右子树。后序遍历先遍历节点的左子树和右子树，最后处理该节点。层次遍历从根到叶逐层从左向右处理每个节点。数组可以用于实现 BST，但如果树不平衡，这种方法会浪费很多内存空间。链式 BST 可以避免这种浪费。

这些 ADT 的简单实现方法带来了三个问题。首先，它们只允许拥有一个堆栈、一个队列或一棵树。这个问题可以通过把为这些结构分配内存的操作从操纵这些结构的函数中分离出来。但这样做导致封装性的损失，增加了出错机会。第 2 个问题是无法声明不同类型的堆栈、队列和树。为每种类型单独创建一份 ADT 函数使代码的维护变得更为困难。一个更好的办法是用#define 宏实现代码，然后用目标类型对它进行扩展。不过，使用这种方法，你必须小心选择一种命名约定。另一种方法是通过把需要存储到 ADT 的值强制转换为 void *。这种策略的一个缺点是它绕过了类型检查。第 3 个问题是避免不同 ADT 之间以及同种 ADT 用于处理不同类型数据的各个版本之间避免名字冲突。我们可以创建 ADT 的泛型实现，但为了正确使用它们，用户必须承担更多的责任。

17.7 警告的总结

1. 使用断言检查内存是否分配成功是危险的。
2. 数组形式的二叉树节点位置计算公式假定数组的下标从 1 开始。
3. 把数据封装于对它进行操纵的模块可以防止用户不正确地访问数据。
4. 与类型无关的函数没有类型检查，所以应该小心，确保传递正确类型的数据。

17.8 编程提示的总结

1. 避免使用具有副作用的函数可以使程序更容易理解。
2. 一个模块的接口应该避免暴露它的实现细节。
3. 将数据类型参数化，使它更容易修改。
4. 只有模块对外公布的接口才应该是公用的。
5. 使用断言来防止非法操作。
6. 几个不同的实现使用同一个通用接口使模块具有更强的可互换性。
7. 复用现存的代码而不是对它进行改写。
8. 迭代比尾部递归效率更高。

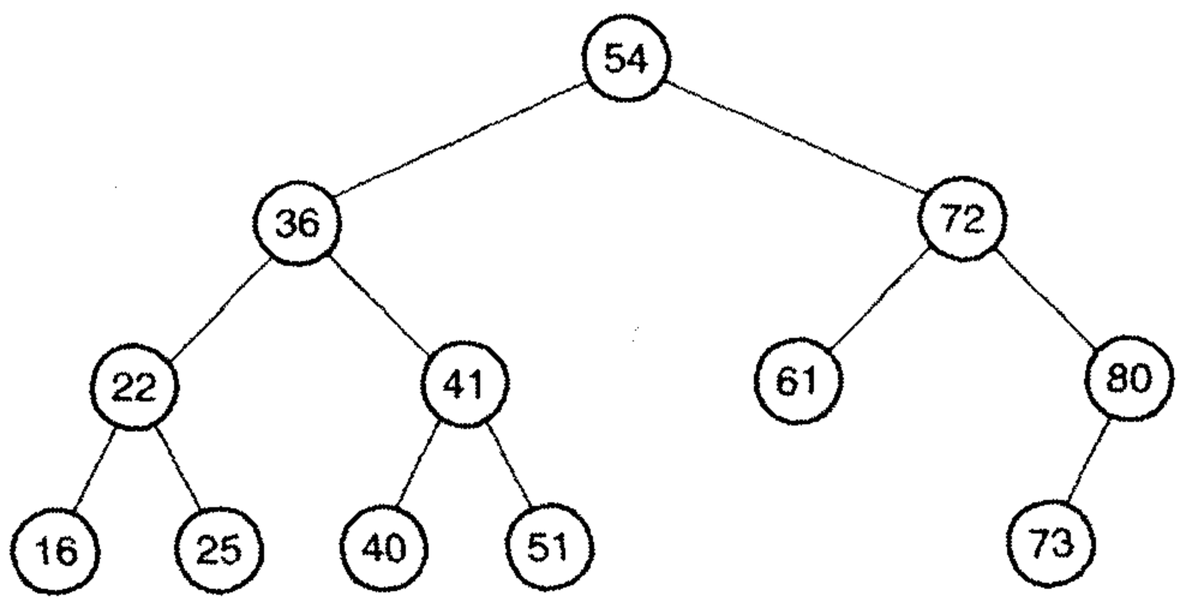
17.9 问题

1. 假定你有一个程序，它读取一系列名字，但必须以反序将它们打印出来。哪种 ADT 更适合完成这个任务？
2. 在超级市场的货架上摆放牛奶时，使用哪种 ADT 更为合适？你即需要考虑顾客购买牛奶，也需要考虑超级市场新到货一批牛奶的情况。
3. 在堆栈的传统接口中，pop 函数返回它从堆栈中删除的那个元素的值。在一个模块中提供两种接口是不是有可能？
4. 如果堆栈模块具有一个 empty 函数，用于删除堆栈中所有的值，你觉得模块的功能

bbs.theithome.com

是不是变得明显更为强大？

5. 在 push 函数中，top_element 在存储值之前先增值。但在 pop 函数中，它却在返回栈顶值后再减值。如果这两个次序弄反，会产生什么后果？
6. 如果在一个使用静态数组的堆栈模块中删除所有的断言，会产生什么后果？
7. 在堆栈的链式实现中，为什么 destroy_stack 函数从堆栈中逐个弹出每个元素。
8. 链式堆栈实现的 pop 函数声明了一个局部变量称为 first_node。这个变量可不可以省略？
9. 当一个循环数组已满时，front 和 rear 值之间的关系和堆栈为空时一样。但是，满和空是两种不同的状态。从概念上说，为什么会出现这种情况？
10. 有两种方法可用于检测一个已满的循环数组：(1)始终保留一个数组元素不使用。(2)另外增加一个变量，记录数组中元素的个数。哪种方法更好一些？
11. 编写语句，根据 front 和 rear 的值计算队列中元素的数量。
12. 实现队列可以使用单链表，也可以使用双链表，哪个更适合？
13. 画一棵树，它是根据下面的顺序把这些值依次插入到一棵二叉搜索树而形成的：20，15，18，32，5，91，-4，76，33，41，34，21，90。
14. 按照升序或降序把一些值插入到一棵二叉搜索树将导致树不平衡。在这样一棵树中查找一个值的效率如何？
15. 使用前序遍历，下面这棵树各节点的访问次序是怎么样的？中序遍历呢？后序遍历呢？层次遍历呢？



16. 改写 do_pre_order_traversal 函数，用于执行树的中序遍历。
17. 改写 do_pre_order_traversal 函数，用于执行树的后序遍历。
18. 二叉搜索树的哪种遍历方法可以以升序依次访问树中所有的节点？哪种遍历方法可以以降序依次访问树中所有的节点？
19. destroy_tree 函数通过释放所有分配给树中节点的内存来删除这棵树，这意味着所有的树节点必须以某个特定的次序进行处理。哪种类型的遍历最适合这个任务？

17.10　编程练习

★ 1. 在动态分配数组的堆栈模块中增加一个 resize_stack 函数。这个函数接受一个参数：

bbs.theithome.com

堆栈的新长度。

★★ 2. 把队列模块转换为使用动态分配的数组形式，并增加一个 resize_queue 函数（类似于第 1 题）。

★★★ 3. 把队列模块转换为使用链表实现。

★★★ 4. 堆栈、队列和树模块如果可以处理超过一个的堆栈、队列和树，它们会更加实用。修改动态数组堆栈模块，使它最多可以处理 10 个不同的堆栈。你将不得不对堆栈函数的接口进行修改，使它们接受另一个参数——需要使用的堆栈的索引。

★★ 5. 编写一个函数，计算一棵二叉搜索树的节点数量。你可以选择任何一种你喜欢的二叉搜索树实现形式。

★★★ 6. 编写一个函数，执行数组形式的二叉搜索树的层次遍历。使用下面的算法：

```
向一个队列添加根节点。
while 队列非空时：
    从队列中移除第 1 个节点并对它进行处理。
    把这个节点所有的孩子添加到队列中。
```

★★★★ 7. 编写一个函数，检查一棵树是不是二叉搜索树。你可以选择任何一种你喜欢的树实现形式。

★★★★★ 8. 为数组形式的树模块编写一个函数，用于从树中删除一个值。如果需要删除的值并未在树中找到，函数可以终止程序。

★★ 9. 为链式实现的二叉搜索树编写一个 destroy_tree 函数。函数应该释放树使用的所有内存。

★★★★★ 10. 为链式实现的树模块编写一个函数，用于从树中删除一个值。如果需要删除的值并未在树中找到，函数可以终止程序。

★★★★ 11. 修改程序 17.10a 的#define 定义，让它拥有三个单独的定义。

a. 一个用于声明堆栈接口

b. 一个用于创建堆栈函数的实现

c. 一个用于创建堆栈使用的数据

你必须修改堆栈的接口，把堆栈数据作为显式的参数传递给函数（把堆栈数据包装于一个结构中会更方便）。这些修改将允许一组堆栈函数操纵任意个对应类型的堆栈。

bbs.theithome.com