第3章　基本数据结构*

要设计一个有效的算法，必须选择或设计适合该问题的数据结构，使得算法采用这种数据结构时能对数据施行有效的运算，因此构造数据结构是改进算法的基本方法之一。

本章简要复习基本数据结构的基本知识。这里只复习线性表、栈和队列的基本知识，树、图和堆的基本知识在第4章复习，集合的基本知识在第5章复习。

- 线性表
- 栈
- 队列

3.1 线性表

3.1.1 线性表的含义

在计算机程序中经常用到的一种最简单的数据组织形式就是线性表。它由某个集合中的一些元素组成，通常写成$(a_0,a_1,...,a_{n-1})$。这里$n\geq 0$，当$n=0$时是一个空表。插入和删除元素是线性表的两个实质性运算，本节主要讨论这2种运算。

线性表的存储结构一般使用顺序表示（使用一维数组表示）和链式表示（使用链接表表示）这两种方法，现分述如下。

3.1.2 线性表的顺序表示

1. 顺序表

用一个一维数组buf[n]来表示线性表，其中n是允许存入表中元素的最大数目。表中的第0号元素存放在buf[0]中，第1号元素存放在buf[1]中，第i号元素存放在buf[i]中。这种存储结构的线性表通常称为顺序表。

```
template <class T> //
class Vector
{
private:
    T *buf = 0; // 一维数组
    int cap = 64, sz = 0; // 容量，大小
    // ...
};
```

2. 插入元素

将元素v插入到位置pos。如图3-1所示。

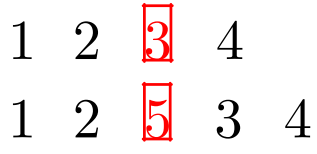


图3-1 插入元素

```
void insert(int pos, const T &v) // 将元素v插入到位置pos
{   if(pos < 0 || pos > sz) return;
    reserve(sz + 1); // 扩大容量
    for(int i = sz; i > pos; --i) buf[i] = buf[i - 1];
    buf[pos] = v, ++sz; // 插入
} // 耗时O(sz)
```

3. 删除元素

删除pos位置的元素。如图3-2所示。

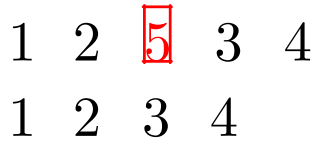


图3-2 删除元素

```
void erase(int pos) // 删除pos位置的元素
{   if(sz <= 0 || pos >= sz) return;
    --sz;
    for(int i = pos; i < sz; ++i) buf[i] = buf[i + 1];
} // 耗时O(sz)
```

3.1.3 顺序表的C++实现

1. 预处理

```
// vector.h
#pragma once
#include <algorithm> // copy_n, etc.
using namespace std;
```

2. 数据成员

```
template <class T> //
class Vector
{   T *buf = 0; // 一维数组
    int cap = 64, sz = 0; // 数组容量，实际元素个数
```

3. 初始化

```
public:
    Vector() // 空顺序表, 使用默认容量
    {   buf = new T[cap]; // 创建数组
    }
```

```
    Vector(int n): sz(n) // 大小为n的顺序表, 容量不小于n
    {   while(cap < n) cap *= 2; // 计算容量
        buf = new T[cap]; // 创建数组
    } // 耗时O(log(sz))
```

```
    Vector(const Vector &X): cap(X.cap), sz(X.sz)
    {   // 使用另一个顺序表初始化
        buf = new T[cap]; // 创建数组
        copy_n(X.buf, sz, buf); // 复制元素
    } // 耗时O(sz)
```

```
    ~Vector()
    {   delete []buf; // 释放数组
    }
```

4. 复制顺序表

```
Vector &operator=(const Vector &X) // 复制顺序表
{   if(this == &X) return *this;
    reserve(X.sz); // 扩大容量
    copy_n(X.buf, X.sz, buf); // 复制元素
    sz = X.sz;
    return *this;
} // 耗时O(sz)
```

5. 访问元素

提供使用指针访问和使用下标访问2种方式，其中begin()和end()分别得到首元素地址和结束元素的地址。

```
T *begin() const { return buf; }
T *end() const { return buf + sz; }
T &operator[](int i) const { return buf[i]; }
```

6. 插入元素

```
void insert(int pos, const T &v) // 将元素v插入到位置pos
{   if(pos < 0 || pos > sz) return;
    reserve(sz + 1); // 扩大容量
    for(int i = sz; i > pos; --i) buf[i] = buf[i - 1];
    buf[pos] = v, ++sz; // 插入
} // 耗时O(sz)
```

7. 删除元素

```
void erase(int pos) // 删除pos位置的元素
{   if(sz <= 0 || pos >= sz) return;
    --sz;
    for(int i = pos; i < sz; ++i) buf[i] = buf[i + 1];
} // 耗时O(sz)
void clear()
{   sz = 0;
}
```

8. 大小

```
    void reserve(int n) // 扩大容量, 新容量不小于n
    {   if(cap >= n) return; // 无需扩容
        T *buf0 = buf; // 原数组
        while(cap < n) cap *= 2; // 计算新容量
        buf = new T[cap]; // 创建新数组
        copy_n(buf0, sz, buf); // 复制原有元素
        delete []buf0; // 释放原数组
    } // 耗时O(sz)
    void resize(int n) // 更改大小
    {   reserve(n);
        sz = n;
    } // 耗时O(sz)
    int size() const { return sz; }
    bool empty() const { return sz <= 0; }
};
```

3.1.4 C++ STL中的顺序表

在C++ STL中，使用类模板std::vector表示可变长的数组（顺序表）。vector使用与元素数目成正比例的操作完成元素的插入和删除。1.5.3节举例说明了vector的使用方法。

3.1.5 线性表的链式表示

1. 链接表

链接表是一些结点的集合，每个结点由若干个信息段组成，在这些信息段中可以有一个信息段用来存放数据对象（即前面所说的元素），其余的信息段用来存放数据对象之间的顺序的链接信息。

2. 线性链表

用来表示线性表的最简单的链接表是一种单向链接表，表中每个结点有两个分别叫做val和next的信息段。val信息段用来存放数据对象，next信息段则指向表示下一个数据对象的结点。如果当前结点没有下一个数据对象，则当前结点的next信息段存放一个零。例如，图3-3表示一个含有4个元素的线性链表，其中first表示首结点地址。

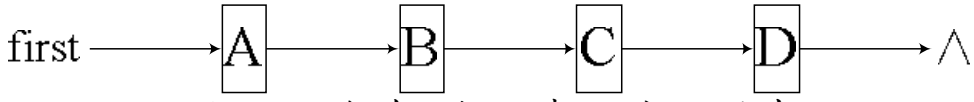


图3-3　含有4个元素的线性链表

```
template<class T> //
class ForwardList
{
    struct LNode // 结点类型
    {   T val; // 元素值
        LNode *next; // 下一结点
    };
    // 数据成员
    LNode *first = 0; // 首结点
```

3. 插入元素

如图3-4所示。

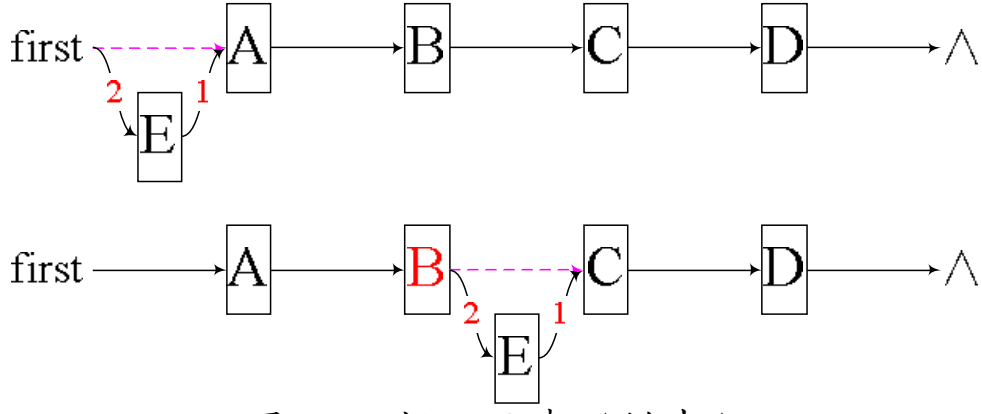


图3-4　插入元素（链表）

```
void push_front(const T &v) // 在首部插入元素v
{   first = new LNode{v, first};
}
```

```
LNode *insert_after(LNode *q, const T &v) // 在结点q后插入v
{   if(q == 0) return 0;
    return q->next = new LNode{v, q->next};
}
```

```
void insert(int i, const T &v) // 在位置i插入v
{   if(i <= 0) push_front(v); // 在首部插入
    else insert_after(next(i - 1), v); // 在位置i - 1后插入
} // 耗时O(i)
```

4. 删除元素

如图3-5所示。

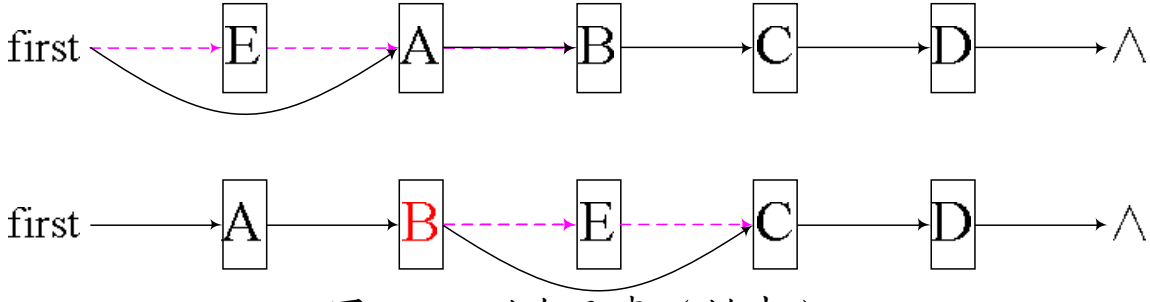
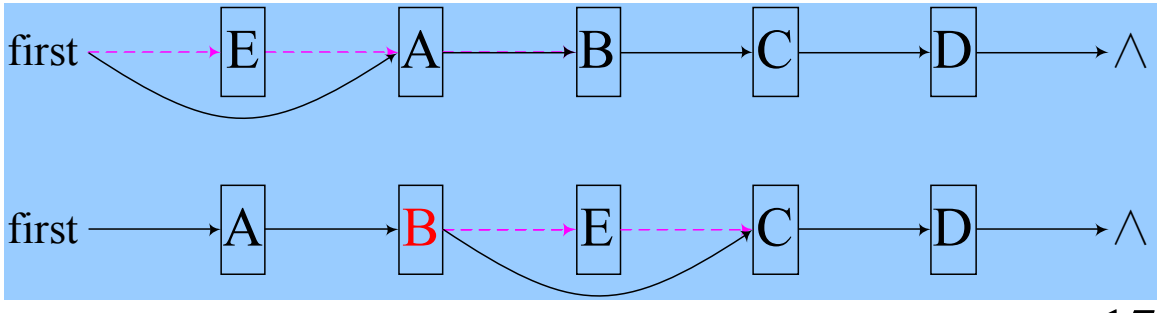


图3-5　删除元素（链表）

```
void pop_front() // 删除首部元素
{   if(first == 0) return;
    LNode *tmp = first;
    first = first->next;
    delete tmp;
}
```

```
LNode *erase_after(LNode *q) // 删除结点q后的第1个元素
{   LNode *tmp = q->next;
    if(q == 0 || tmp == 0) return 0;
    q->next = tmp->next;
    delete tmp;
    return q->next;
}
```

```
void erase(int i) // 删除第i号元素
{   if(i <= 0) pop_front(); // 在首部删除
    else erase_after(next(i - 1)); // 在位置i - 1后删除
} // 耗时O(i)
```



3.1.6 线性链表的C++实现

1. 结点类型

```
// ForwardList.h
#pragma once
template<class T> //
class ForwardList
{   struct LNode // 结点类型
    {   T val; // 元素值
        LNode *next; // 下一结点
    };
```

2. 数据成员

```
    // 数据成员
    LNode *first = 0; // 首结点
```

3. 复制线性链表

```
void _copy(const ForwardList &X)
{   // 复制另一线性链表, 要求*this是空的
   if(X.empty()) return; // 无需复制
    push_front(X.first->val); // 插入首元素
    LNode *p = first, *q = X.first;
    for(q = q->next; q != 0; q = q->next) // 插入其余元素
        p = insert_after(p, q->val);
} // 耗时O(n)
```

4. 初始化与赋值运算

```
public:
    ForwardList() {}
```

```
    ForwardList(const ForwardList &X)
    {   _copy(X);
    } // 耗时O(n)
```

```
    ~ForwardList()
    {   clear(); // 删除所有结点
    } // 耗时O(n)
```

```
    ForwardList &operator=(const ForwardList &X)
    {   if(this == &X) return *this; // 无需复制
        clear(); // 删除所有结点
        _copy(X); // 复制
        return *this;
    } // 耗时O(n)
```

5. 访问元素

```
LNode *begin() const // 首元素所在结点, 可读写
{   return first;
}
```

```
LNode *next(int i) const // 第i号元素所在结点
{   LNode *q = first;
    while(i > 0 && q != 0) q = q->next, --i;
    return q;
} // 耗时O(i)
```

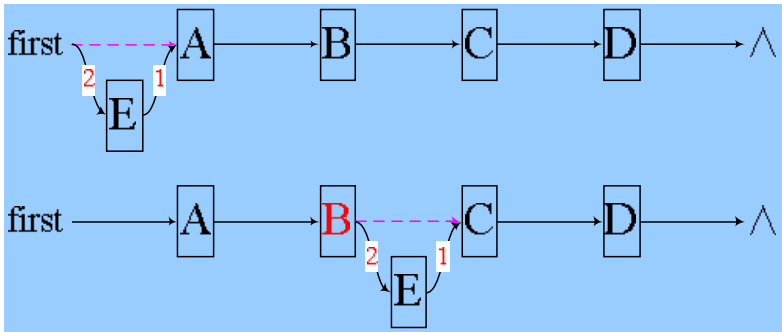
```
T &operator[](int i) const
{   return next(i)->val;
} // 耗时O(i)
```

6. 插入元素

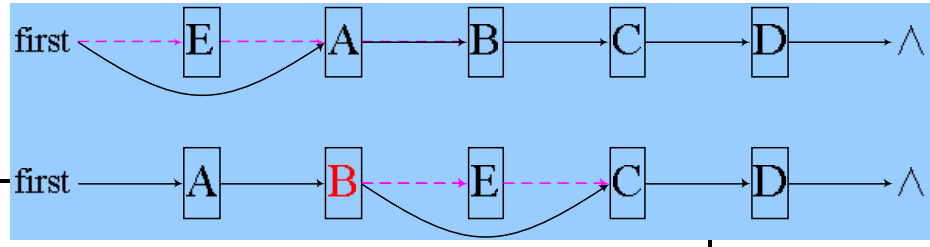
```
void push_front(const T &v) // 在首部插入元素v
{   first = new LNode{v, first};
}
```

```
LNode *insert_after(LNode *q, const T &v) // 在结点q后插入v
{   if(q == 0) return 0;
    return q->next = new LNode{v, q->next};
}
```

```
void insert(int i, const T &v) // 在位置i插入v
{   if(i <= 0) push_front(v); // 在首部插入
    else insert_after(next(i - 1), v); // 在位置i - 1后插入
} // 耗时O(i)
```



7. 删除元素




```
void pop_front() // 删除首部元素
{   if(first == 0) return;
    LNode *tmp = first;
    first = first->next;
    delete tmp;
}
```

```
LNode *erase_after(LNode *q) // 删除结点q后的第1个元素
{   LNode *tmp = q->next;
    if(q == 0 || tmp == 0) return 0;
    q->next = tmp->next;
    delete tmp;
    return q->next;
}
```

```
void erase(int i) // 删除第i号元素
{   if(i <= 0) pop_front(); // 在首部删除
    else erase_after(next(i - 1)); // 在位置i - 1后删除
} // 耗时O(i)
```

8. 清空

```
    void clear() // 清空, 删除所有结点
    {   while(!empty()) pop_front();
    } // 耗时O(n)
    bool empty() const
    {   return first == 0;
    }
};
```

3.1.7 C++ STL中的线性链表

在C++ STL中，使用类模板std::forward_list表示线性链表。forward_list使用与元素数目无关的操作完成元素的插入和删除。下面举例说明forward_list的使用方法。

```
#include <forward_list>
#include <iostream>
using namespace std;
```

```
template<class T>
ostream &operator<<(ostream &out, const forward_list<T> &X)
{   for(const T &e : X)
        out << e << ' ';
    return out;
}
```

```
int main()
{   forward_list<int> X;
    X.push_front(0), X.push_front(9); // 首部插入
    auto I = X.begin();
    for(int v = 1; v < 9; ++v)
        I = X.insert_after(I, -v); // 中间插入
    cout << X << endl;
    for(I = X.begin(); I != X.end();) // 隔元素删除
        I = X.erase_after(I);
    X.pop_front(); // 删除首部元素
    cout << X << endl; // 只剩下负偶数
    X.clear(); // 清空
    cout << boolalpha << X.empty() << endl;
}
```

```
9 -1 -2 -3 -4 -5 -6 -7 -8 0
-2 -4 -6 -8
true
```

3.2 栈

3.2.1 栈的含义

栈（stack）是一种特殊的线性表。栈的插入和删除都在称为栈顶（top）的一端进行。栈的运算意味着如果依次将元素1, 2, 3, 4插入栈，那末从栈中移出的第一个元素必定是4，即最后进入栈的元素将首先移出，因此栈又称为后进先出（LIFO）表。如图3-6所示。

图3-6 栈的例子

3.2.2 栈的顺序表示

用一个一维数组buf[n]来表示栈，其中n是允许存入栈中元素的最大数目。栈中的第0号元素存放在buf[0]，第1号元素存放在buf[1]，第i号元素存放在buf[i]。此外，还需要一个变量sz表示栈的实际大小。可以使用sz-1表示栈顶位置。可以使用“sz<=0”测试栈是否为空。若非空，栈顶元素为buf[sz-1]。

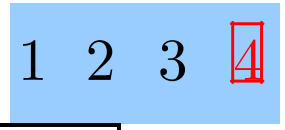
```
class Stack
{   T *buf = 0; // 一维数组
    int cap = 64, sz = 0; // 容量, 大小
    // ...
};
```

3.2.3 栈的操作

插入和删除元素是栈的两个实质性运算。

1. 访问元素

只能直接访问栈顶元素。如图3-6所示。




```
T &top() const
{   return buf[sz - 1];
}
```

2. 插入元素

只能栈顶一端插入元素。如图3-7所示。

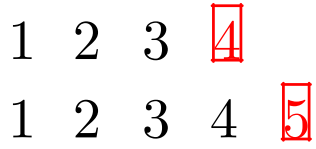


图3-7 入栈

```
void push(const T &x)
{   reserve(sz + 1); // 扩大容量
    buf[sz] = x, ++sz; // 插入元素
}
```

3. 删除元素

只能栈顶一端删除元素。如图3-8所示。

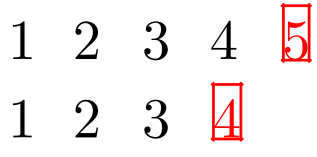


图3-8 出栈

```
void pop()
{   if(sz <= 0) return;
    --sz; // 更改大小
}
```

3.2.4 栈的C++实现

1. 预处理

```
// stack.h
#pragma once
#include <algorithm> // copy_n, etc.
using namespace std;
```

2. 数据成员

```
template<class T> //
class Stack
{   T *buf = 0; // 一维数组
    int cap = 64, sz = 0; // 容量，大小
```

3. 初始化

```
public:
    Stack &operator=(const Stack &X) = delete; // 禁止赋值
```

```
    Stack() // 空栈, 使用默认容量
    {   buf = new T[cap]; // 创建数组
    }
```

```
    Stack(const Stack &X): cap(X.cap), sz(X.sz) // 使用另一个栈初始化
    {   buf = new T[cap]; // 创建数组
        copy_n(X.buf, sz, buf); // 复制元素
    } // 耗时O(sz)
```

```
    ~Stack()
    {   delete []buf; // 释放数组
    }
```

4. 访问元素

```
T &top() const
{   return buf[sz - 1];
}
```

5. 插入元素

```
void push(const T &x)
{   reserve(sz + 1); // 扩大容量
    buf[sz] = x, ++sz; // 插入元素
}
```

6. 删除元素

```
void pop()
{   if(sz <= 0) return;
    --sz; // 更改大小
}
```

7. 大小

```
    void reserve(int n) // 扩大容量, 新容量不小于n
    {   if(cap >= n) return; // 无需扩容
        T *buf0 = buf; // 原数组
        while(cap < n) cap *= 2; // 计算新容量
        buf = new T[cap]; // 创建新数组
        copy_n(buf0, sz, buf); // 复制原有元素
        delete []buf0; // 释放原数组
    } // 耗时O(sz)
    int size() const
    {   return sz;
    }
    bool empty() const
    {   return sz <= 0;
    }
};
```

3.2.5 C++ STL中的栈

在C++ STL中，使用类模板std::stack表示栈。stack使用常数时间的操作完成元素的插入和删除。1.5.4节举例说明了stack的使用方法。

3.3 队列

3.3.1 队列的含义

队列（queue）也是一种特殊的线性表。队列的插入只能在称为尾部的一端进行，删除则只能在称为前部的另一端进行。队列的运算要求第一个插入队中的元素也第一个移出，因此队列是一个先进先出（FIFO）表。如图3-9所示。

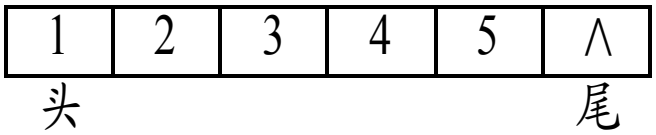


图3-9 队列

3.3.2 队列的顺序表示

用一个一维数组buf[n]来表示栈，其中n是允许存入栈中元素的最大数目。此时，我们可以把它当成一个环形来看待。first表示队首元素的位置，last表示队尾元素的下一个位置（插入位置）。当队尾元素位于n-1时，则last=0。最开始时first=last=0。图3-10描述了一个n=8的数组中含有元素1, 2, 3, 4的环形队列的两种可能存放方式。

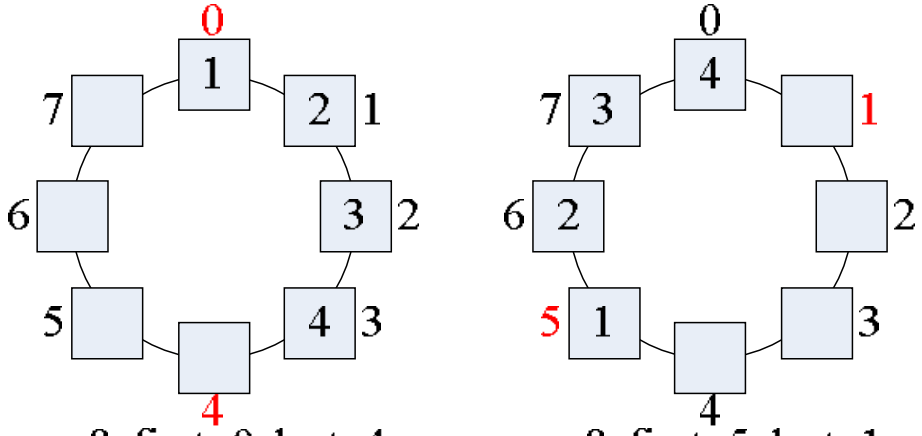


图3-10 两个4元素队列

用上述方法表示队列存在一个问题，就是不好区分队列是满的还是空的，因为在这两种情况下都有first = last。为了区分队列是满的还是空的，我们改用变量first和sz代替变量first和last，其中，sz表示队列的实际大小。此时last = (first + sz) mod n，且sz <= 0表示队列是空的，sz >= n表示队列是满的。

```
template<class T> //
class Queue
{   T *buf = 0; // 缓冲区
    int cap = 64, first = 0, sz = 0; // 容量, 队首, 大小
    // ...
};
```

3.3.3 队列的操作

插入和删除元素是队列的两个实质性运算。

1. 访问元素

只能直接访问队首元素。如图3-11所示。

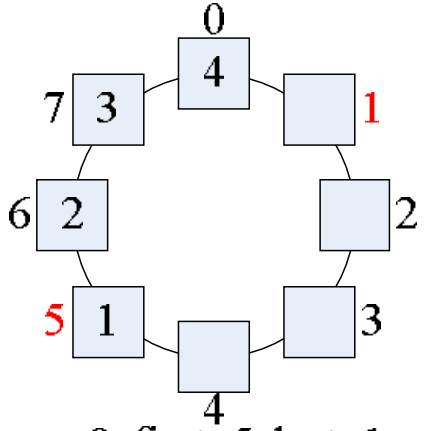


图3-11 访问队列元素

```
T &front() const
{   return buf[first];
}
```

2. 插入元素

只能在队尾一端插入元素。如图3-12所示。

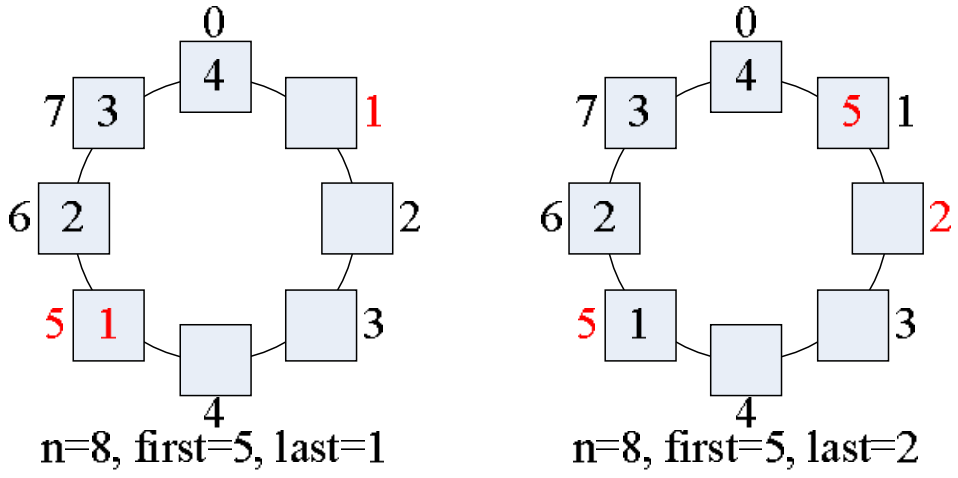


图3-12　入队

```
void push(const T &x)
{   reserve(sz + 1); // 扩大容量
    buf[(first + sz) % cap] = x; // buf[last] = x
    ++sz;
}
```

3. 删除元素

只能在队首一端删除元素。如图3-13所示。

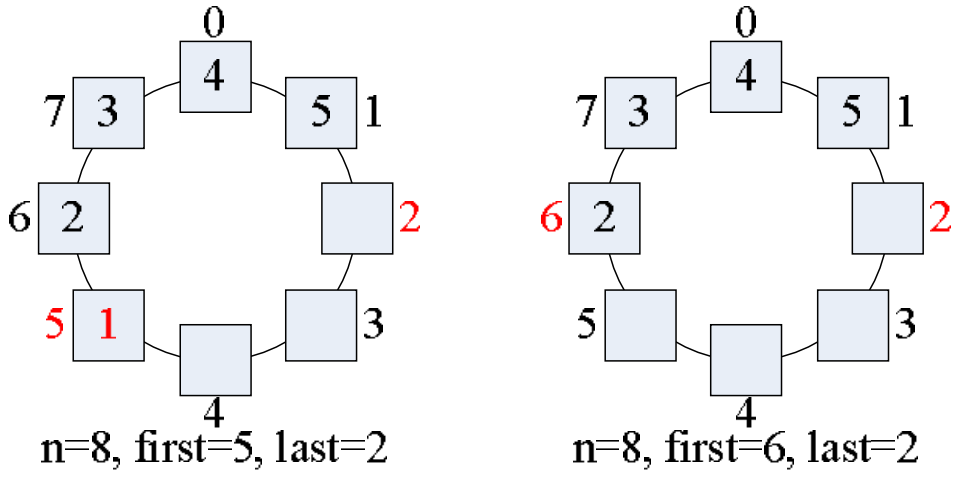


图3-13　出队

```
void pop()
{   if(sz <= 0) return;
    (++first) %= cap; // ++first
    --sz;
}
```

3.3.4 队列的C++实现

1. 数据成员

```
// queue.h
#pragma once
template<class T> //
class Queue
{   T *buf = 0; // 缓冲区
    int cap = 64, first = 0, sz = 0; // 容量，队首，大小
```

2. 初始化

```
public:
    Queue &operator=(const Queue &X) = delete; // 禁止赋值
```

```
    Queue() // 空队列, 使用默认容量
    {   buf = new T[cap]; // 创建数组
    }
```

```
    Queue(const Queue &X): cap(X.cap), first(X.first), sz(X.sz)
    {   // 使用另一个队列初始化
        buf = new T[cap]; // 创建数组
        int last = (first + sz) % cap;
        for(int i = first; i != last; (++i) %= cap) // 复制元素
            buf[i] = X.buf[i];
    }
```

```
    ~Queue()
    {   delete[]buf; // 释放数组
    } //
```

3. 访问元素

```
T &front() const
{   return buf[first];
}
```

4. 插入元素

```
void push(const T &x)
{   reserve(sz + 1); // 扩大容量
    buf[(first + sz) % cap] = x; // buf[last] = x
    ++sz;
}
```

5. 删除元素

```
void pop()
{   if(sz <= 0) return;
    (++first) %= cap; // ++first
    --sz;
}
```

6. 大小

```
    void reserve(int n) // 扩大容量, 新容量不小于n
    {   if(cap >= n) return; // 无需扩容
        T *buf0 = buf; // 原数组
        int cap0 = cap; // 原容量
        while(cap < n) cap *= 2; // 计算新容量
        buf = new T[cap]; // 新数组
        int last = first + sz; // 新last
        for(int i = first; i != last; ++i) // 复制原有元素
            buf[i] = buf0[i % cap0];
        delete []buf0; // 释放原数组
    } // 耗时O(sz)
    int size() const
    {   return sz;
    }
    bool empty() const
    {   return sz <= 0;
    }
};
```

3.3.5 C++ STL中的队列

在C++ STL中，使用类模板std::queue表示队列。queue使用常数时间的操作完成元素的插入和删除。1.5.5节举例说明了queue的使用方法。