

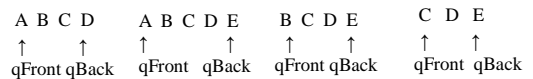
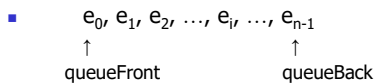
队列(QUEUE)

本章内容

- 9.1 定义和应用
- 9.2 抽象数据类型
- 9.3 数组描述
- 9.4 链表描述
- 9.5 应用

9.1 定义与应用

- 定义:
 - 队列 (queue) 是一个线性表, 其插入和删除操作分别在表的不同端进行。
 - 添加新元素的那一端被称为队尾 (queueBack)。
 - 删除元素的那一端被称为队首 (queueFront)。



- 队列是一个先进先出（first-in-first-out, FIFO）的线性表。

9.2 抽象数据类型

抽象数据类型queue {

实例

元素的有序线性表，一端称为队首，另一端称为队尾；

操作

```
empty(); //队列为空时返回true, 否则返回false;
size(); //返回队列中元素个数
front(); //返回队列头元素;
back(); //返回队列尾元素;
pop(); //删除队列头元素
push(x); //将元素x加入队尾
```

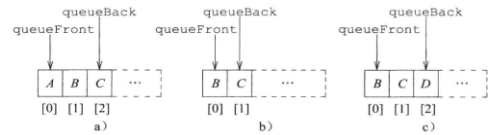
C++抽象类queue

```
template <class T>
class queue{
public:
    virtual ~queue() {}
    virtual bool empty() const = 0;
        //队列为空时返回true，否则返回false
    virtual int size() const = 0;
        //返回队列中元素个数
    virtual T& front() = 0;
        //返回队列头元素；
    virtual T& back() = 0;
        //返回队列尾元素
    virtual void pop() = 0;
        //队列头元素
    virtual void push(const T& theElement) = 0;
        //将元素theElement加入队尾
};
```

队列的描述

- 数组描述
- 链表描述

9.3 数组描述-1



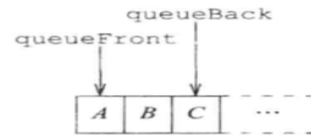
- 映射公式-1:

$$location(i) = i$$
 队列第 i 个元素存储在 $queue[i]$ 中
- 队首元素: $queue[0]$;
- $queueFront$ 表示队首元素的位置, 总是为 0;
- $queueBack$ 始终是最后一个元素 (队尾元素) 的位置

数组描述-1

- 队列的长度: $queueBack + 1$
- 空队列: $queueBack = -1$
- 队首元素出队:
 $queue[0..queueBack-1] \leftarrow queue[1..queueBack];$
 $queueBack = queueBack - 1;$
 $\Theta(n)$
- 元素 x 入队: $queueBack = queueBack + 1;$
 $queue[queueBack] = x;$
 $O(1)$

数组描述-2

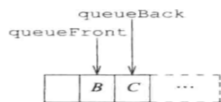


- 映射公式-2:

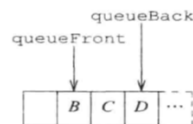
$$location(i) = location(\text{队首元素}) + i$$
- $queueFront = location(\text{队首元素})$
- $queueBack = location(\text{队尾元素})$
- 空队列: $queueBack < queueFront$

数组描述-2

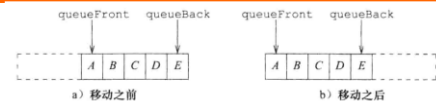
- 删除队首元素
 - $queueFront = queueFront + 1;$
 - $\Theta(1)$



- 元素 D 入队
 $queueBack = queueBack + 1;$
 $queue[queueBack] = x;$
 $\Theta(1)$




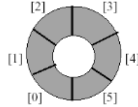
数组描述-2



- 每次删除一个元素都使 $queueFront$ 右移一个位置, 当 $queueBack = arrayLength - 1$ 且 $queueFront > 0$ 时, 队列左端还有放置新元素的空间, 为了可以继续插入元素, 可以把队列所有元素整体平移到队列左端
- 平移队列
 - $queue[0..queueBack-queueFront+1] \leftarrow queue[queueFront..queueBack];$
 - $queueBack = queueBack + 1; queue[queueBack] = x;$

- 时间复杂性: $\Theta(n)$

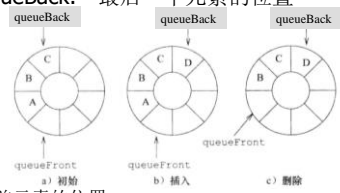
循环队列

- 描述队列的数组 
- 描述队列的数组被视为一个环 
- 公式-3 :

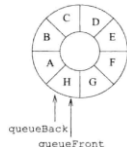
$$location(i) = (location(\text{队首元素}) + i) \% arrayLength$$

$$location(i+1) = (location(i) + 1) \% arrayLength$$
- 循环队列

循环队列

- queueFront**: 指向队列首元素的下一个位置（逆时针方向）。
 - queueBack**: 最后一个元素的位置
- 
- 队列首元素的位置:
 $(queueFront + 1) \% arrayLength$

循环队列



9-6 一个具有 $arrayLength$ 个元素的循环队列

- 空队列: $queueFront = queueBack$
- 队列为满的条件: $queueFront = queueBack$
- 如何区分两种情况: 队列为空和队列为满?
 队列中最多元素个数 $= arrayLength - 1$
- 队列为满的条件:
 $queueFront = (queueBack + 1) \% arrayLength$
- 队列满时, 队列容量可进行加倍处理: 读P210, P211程序9-3

```
template<class T>
class arrayQueue : public queue<T>;
{public:
    arrayQueue (int initialCapacity = 10);
    ~arrayQueue () { delete [] queue; }
    bool empty() const { return queueFront == queueBack; }
    int size() const
    { return (arrayLength + queueBack
        - queueFront) \% arrayLength; }

    T& front() const; //返回队首元素
    T& back() const; //返回队尾元素
    void pop(); //删除队首元素
    void push(const T& theElement); //元素插入到队尾
private:
    int queueFront; //与第一个元素在反时针方向上相差一个位置
    int queueBack; //指向最后一个元素
    int arrayLength; //队列数组容量
    T *queue; //元素数组
};
```

arrayQueue构造函数

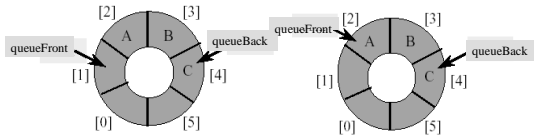
```
template<class T>
arrayQueue <T>::arrayQueue(int initialCapacity = 10);
{ // 构造函数
    if (initialCapacity < 1) .....//输出错误信息, 抛出异常
    arrayLength = initialCapacity;
    queue = new T[arrayLength];
    queueFront = queueBack = 0;
}
```

方法 'front' 'back'

```
template<class T>
T& arrayQueue <T>::front() const
{ //返回队首元素
    if (queueFront == queueBack) throw QueueEmpty();
    return queue[(queueFront + 1) \% arrayLength];
}

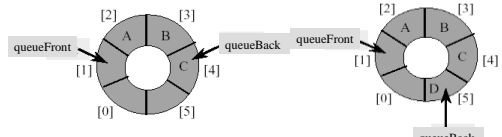
template<class T>
T& arrayQueue <T>::back() const
{ //返回队尾元素
    if (queueFront == queueBack) throw QueueEmpty();
    return queue[queueBack];
}
```

方法 'pop'



```
template<class T>
void arrayQueue<T>::pop()
{// 删除队首元素
if (queueFront == queueBack) throw QueueEmpty();
queueFront = (queueFront + 1) % arrayLength;
queue[queueFront] = T(); //给T析构
}
```

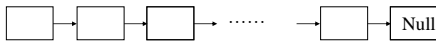
方法 'push'



```
template<class T>
void arrayQueue<T>::push(const T& theElement)
{// 把元素theElement添加到队列的尾部
//如果队列空间满,则加数组长度
if (queueBack + 1 == queueFront)
.....//加数组长度,程序9-3
queueBack = (queueBack + 1) % arrayLength;
queue[queueBack] = theElement;
}
```

9.4 链表描述

- 使用链表来描述一个队列



- 两种选择:

- 1)表头为 queueFront, 表尾为 queueBack
- 2)表头为 queueBack, 表尾为 queueFront

链表描述

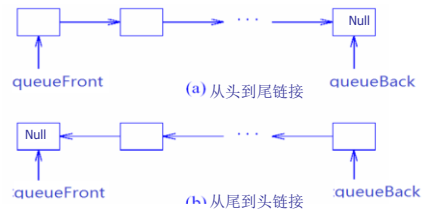


图9-8 链接队列

向链表队列中添加元素



图 9-9 (a) 向图 9-8 (a)中插入元素

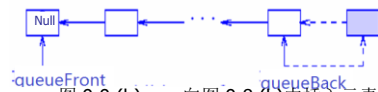


图 9-9 (b) 向图 9-8 (b)中插入元素

- 图 9-9 (a)的时间复杂性?
- 图 9-9 (b) 的时间复杂性?

从链表队列中删除元素



图 9-10 (a) 从图 9-8 (a)中删除元素



图 9-10 (b) 从图 9-8 (b)中删除元素

- 图 9-10 (a)的时间复杂性?
- 图 9-10 (b) 的时间复杂性?

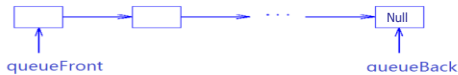
队列的链表描述



- 如何实现队列的链表描述?
 - 定义LinkedQueue为一个基类
 - 把类LinkedQueue 定义为extendedChain类（见程序6-12）的一个派生类

```
template<class T>
class LinkedQueue : public queue <T>;
{ public:
    LinkedQueue(int initialCapacity = 10);
    ~ LinkedQueue();
    bool empty() const {return queueSize == 0; }
    int size() const { return queueSize; }
    T& front();
    T& back();
    void push(const T& theElement);
    void pop();
private:
    chainNode<T>* queueFront; //队列首指针
    chainNode<T>* queueBack; //队列尾指针
    int queueSize; //队列中元素个数
};
```

LinkedQueue类



```
template<class T>
T& LinkedQueue<T>::front()
{ // 返回队列首元素
    if (queueSize == 0) throw QueueEmpty();
    return queueFront ->element;
}
```

```
template<class T>
T& LinkedQueue<T>::back()
{ // 返回队列尾元素
    if (queueSize == 0) throw QueueEmpty();
    return queueBack ->element;
}
```



```
template<class T>
Void LinkedQueue<T>::push(const T& theElement)
{// 把元素theElement加入 到队尾

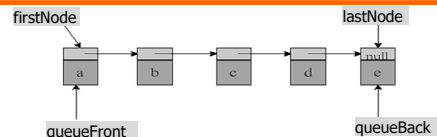
    // 为新元素申请节点
    chainNode<T>* newNode =
        new chainNode<T>(theElement,NULL);
    // 在队列尾部添加新节点
    if (queueSize != 0)
        queueBack->next = newNode; //队尾不为空
    else queueFront = newNode;      // 队列为空
    queueBack = newNode;
    queueSize++;
}
```



```
template<class T>
LinkedQueue<T>& LinkedQueue<T>::pop()
{// 删除队首元素
    if (queueFront == NULL) throw QueueEmpty();

    chainNode<T>* nextNode = queueFront->next;
    delete queueFront; // 删除第一个节点
    queueFront = nextNode;
    queueSize--;
}
```

从extendedChain 派生

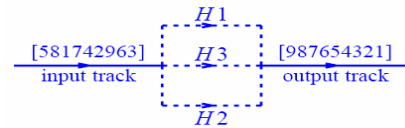


- linkedQueue.empty() → extendedChain.empty()
- linkedQueue.front() → extendedChain.get(0)
- linkedQueue.back() → extendedChain.get(size()-1)
- linkedQueue.push(x) →
 - extendedChain.push_back(x)
- linkedQueue.pop() → extendedChain.erase(0)

9.5 应用

- 9.5.1 列车车厢重排
- 9.5.2 电路布线
- 9.5.3 图元识别
- 9.5.4 工厂仿真

9.5.1 列车车厢重排



- 缓冲铁轨位于入轨和出轨之间
- 禁止：
 - 将车厢从缓冲铁轨移动至入轨
 - 从出轨移动车厢至缓冲铁轨
- 第k条轨道Hk 为可直接将车厢从入轨移动到出轨的通道，其余k-1条轨道用来缓存不能直接进入出轨的车厢

车厢移动到缓冲铁轨的原则

- 车厢c应移动到这样的缓冲铁轨中：
- 该缓冲铁轨中现有各车厢的编号均小于c；如果有多个缓冲铁轨都满足这一条件，
 - 则选择一个左端车厢编号最大的缓冲铁轨；
 - 否则选择一个空的缓冲铁轨（如果有的话）。

列车车厢重排思想

```
int NowOut=1; // NowOut: 下一次要输出的车厢号
for (int i=1; i<=n; i++) // 从前至后依次检查的所有车厢
{1. 车厢 p[i] 从入轨上移出
2. If (p[i] == NowOut) // NowOut: 下一次要输出的车厢号
    ① 使用缓冲铁轨Hk把p[i]放到出轨上去; NowOut++;
    ② while (minH(当前缓冲铁轨中编号最小的车厢) == NowOut)
        {把minH放到出轨上去;
        更新 minH, minQ (minH所在的缓冲铁轨);
        NowOut++;}
    else 按照分配规则将车厢p[i]送入某个缓冲铁轨 }
```

● 读程序 9-6 9-7

- 作业 P211 练习5