C/C++从入门到精通-高级程序员之路

【 数据结构 】

队列及其企业级应用

第 1 节 队列的故事导入

Jack 在一个著名软件外包公司上班，公司每年要接很多的外包项目给各个开发团队开发，现在刚刚接到一个项目，要实现银行排队叫号系统，基本需求是:

取号：

1. 银行用户进入银行大厅办理业务
2. 用户从自动排号机上取号
3. 自动取号机生成号码以及当前等待人数信息给用户

叫号:

1. 当银行业务员(**1 位**或**多位**)处理完业务以后，会指示叫号系统呼叫下一位（按排队次序）正在等待的用户，用户达到窗口以后，银行业务员会将用户的号码从叫号系统中删除。

如果你是 Jack ,你该如何设计这个功能?

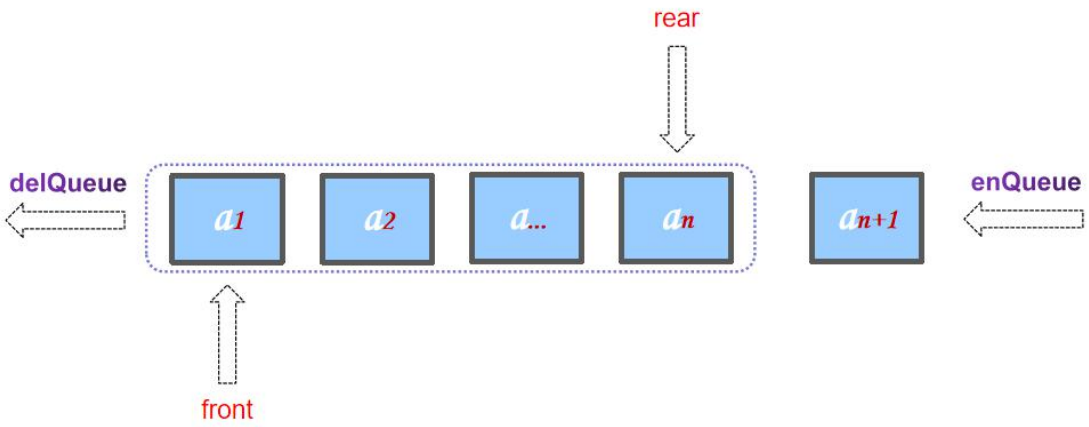
第 2 节 队列的原理精讲

队列是一种受限的线性表，(Queue)，它是一种运算受限的线性表,先进先出(FIFO First In First Out)

- 队列是一种受限的线性结构

- 它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作。

生活中队列场景随处可见: 比如在电影院，商场，或者厕所排队。。。。。。

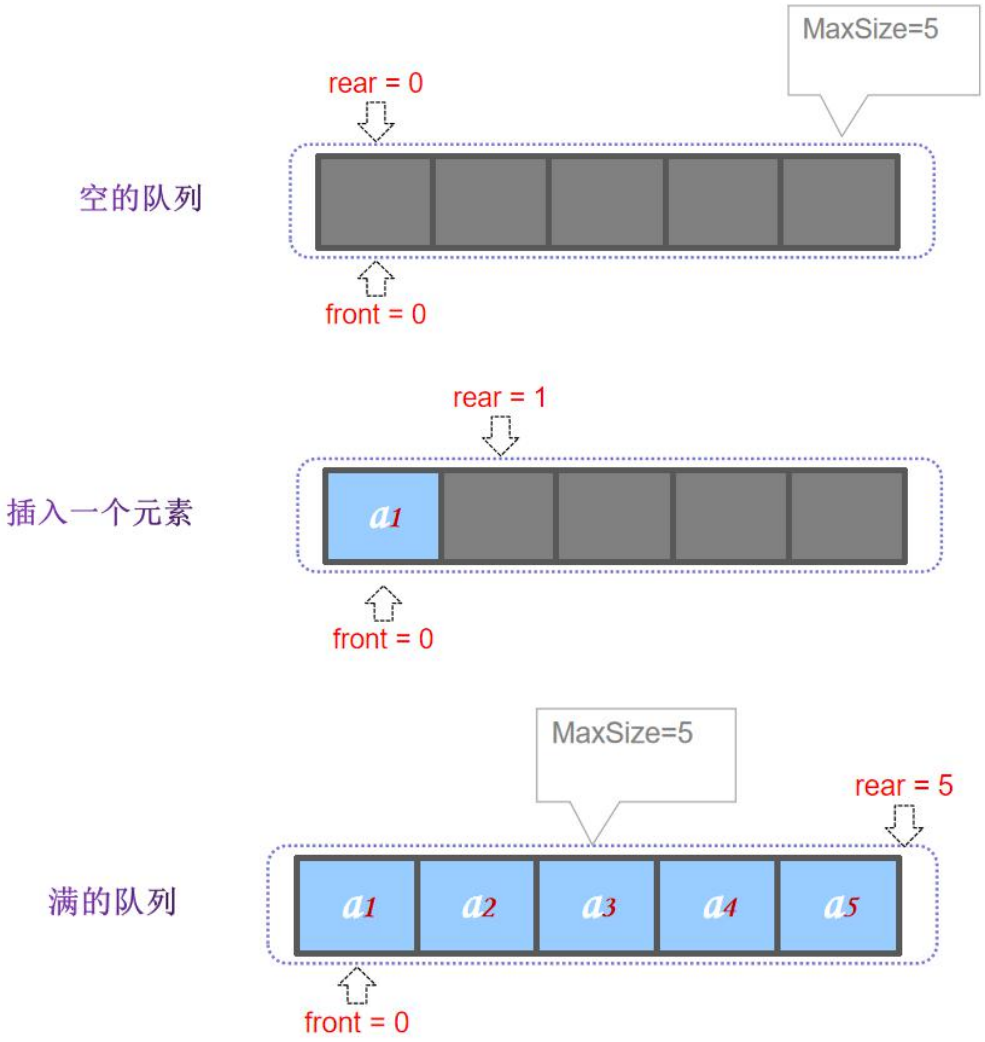


队列图示

第 3 节 队列的算法实现

顺序存储

采用数组来保存队列的元素，设立一个队首指针 front ，一个队尾指针 rear，分别指向队首和队尾元素。则 rear-front 即为存储的元素个数!




```
#define MaxSize 5     //队列的最大容量
typedef int DataType;  //队列中元素类型

typedef struct Queue
{
    DataType queue[MaxSize];
    int front;   //队头指针
    int rear;    //队尾指针
}SeqQueue;
```

队列初始化：

```
//队列初始化，将队列初始化为空队列
void InitQueue(SeqQueue *SQ)
{
    if(!SQ) return ;

    SQ->front = SQ->rear = 0;  //把对头和队尾指针同时置 0
 }
```

队列为空：

```
//判断队列为空
int IsEmpty(SeqQueue *SQ)
{
    if(!SQ) return 0;

    if (SQ->front == SQ->rear)
    {
        return 1;
    }
    return 0;
}
```

队列为满：

```
//判断队列是否为满
int IsFull(SeqQueue *SQ)
{
    if(!SQ) return 0;

    if (SQ->rear == MaxSize)
    {
        return 1;
    }
    return 0;
}
```

入队：将新元素插入 rear 所指的位置，然后 rear 加 1。

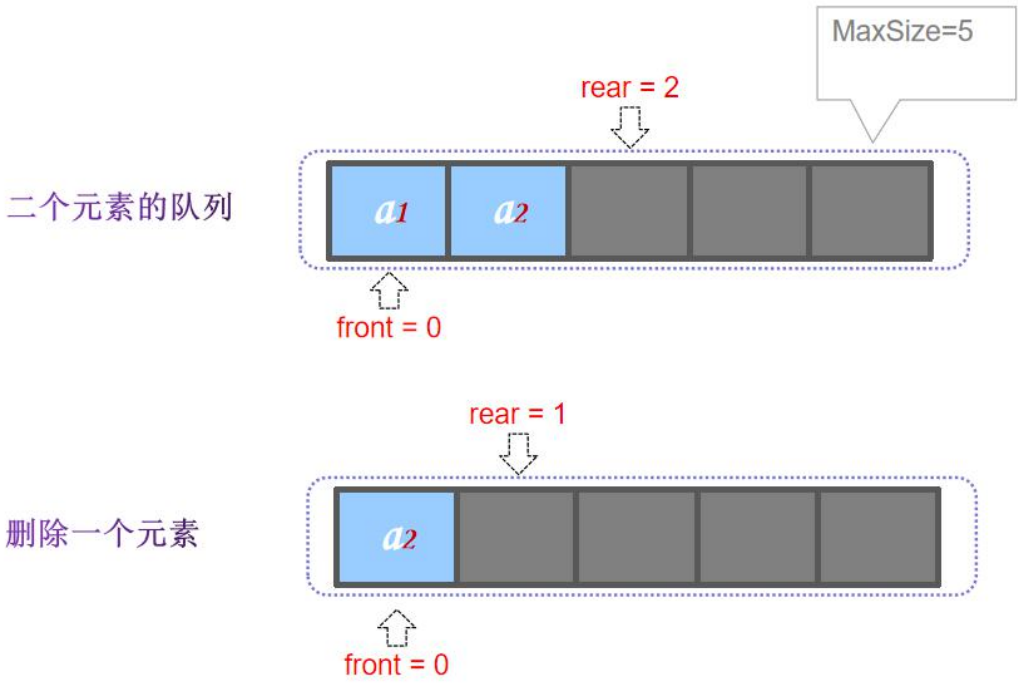
```
//入队,将元素 data 插入到队列 SQ 中
int EnterQueue( SeqQueue *SQ,DataType data){
    if(!SQ) return 0;

    if(IsFull(SQ)){
        cout<<"无法插入元素 "<<data<<", 队列已满!"<<endl;
        return 0;
    }

    SQ->queue[SQ->rear] = data; //在队尾插入元素 data
    SQ->rear++; //队尾指针后移一位
    return 1;
}
```

打印队列中的元素

```
//打印队列中的各元素
void PrintQueue(SeqQueue* SQ)
{
    if(!SQ) return ;

    int i = SQ->front;
    while(i<SQ->rear)
    {
        cout<<setw(4)<<SQ->queue[i];
        i++;
    }
    cout<<endl;
}
```

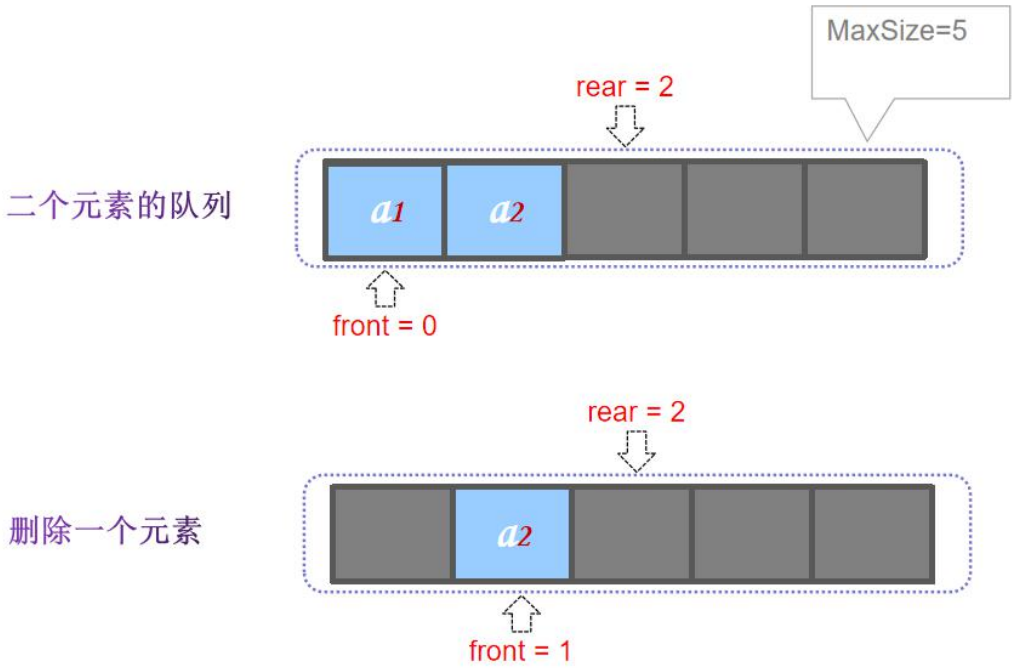
方式 1 - 删除 front 所指的元素，后面所有元素前移 1 并返回被删除元素.

```
//出队，将队列中队头的元素 data 出队，后面的元素向前移动
int DeleteQueue(SeqQueue* SQ, DataType *data){

    if(!SQ || IsEmpty(SQ)){
        cout<<"队列为空！"<<endl;
         return 0;
    }

    if(!data) return 0;

    *data = SQ->queue[SQ->front];

    for(int i=SQ->front+1; i<SQ->rear; i++){//移动后面的元素
        SQ->queue[i-1]=SQ->queue[i];
    }

    SQ->rear--;//队尾指针前移一位
    return 1;
}
```



出队： **方式 2** - 删除 front 所指的元素，然后加 1 并返回被删元素。

```
//出队，将队列中队头的元素 data 出队，出队后队头指针 front 后移一位
int DeleteQueue2(SeqQueue* SQ,DataType* data)
{
    if (!SQ || IsEmpty(SQ))
    {
        cout<<"队列为空！"<<endl;
        return 0;
    }

   if(SQ->front>=MaxSize){
       cout<<"队列已到尽头!"<<endl;
       return 0;
   }

    *data = SQ->queue[SQ->front];    //出队元素值
    SQ->front = (SQ->front)+1;       //队首指针后移一位
    return 1;
}
```



取队首元素：返回 front 指向的元素值

```
//获取队首元素
int GetHead(SeqQueue* SQ,DataType* data)
{
    if (!SQ || IsEmpty(SQ))
    {
        cout<<"队列为空!"<<endl;
    }
    return *data = SQ->queue[SQ->front];
}
```

清空队列

```
//清空队列
void ClearQueue(SeqQueue* SQ)
{
    SQ->front = SQ->rear = 0;
}
```

完整代码实现：

```
#include <stdio.h>
#include <assert.h>
#include <Windows.h>
#include <iostream>
#include <iomanip>

using namespace std;

#define MaxSize 5  //队列的最大容量

typedef int DataType;  //队列中元素类型

typedef struct Queue
{
    DataType queue[MaxSize];
    int front;   //队头指针
    int rear;    //队尾指针
}SeqQueue;
```

```
//队列初始化，将队列初始化为空队列
void InitQueue(SeqQueue *SQ)
{
    if(!SQ) return ;

    SQ->front = SQ->rear = 0;  //把对头和队尾指针同时置 0
 }

//判断队列为空
int IsEmpty(SeqQueue *SQ)
{
    if(!SQ) return 0;

    if (SQ->front == SQ->rear)
    {
        return 1;
    }
    return 0;
}


//判断队列是否为满
int IsFull(SeqQueue *SQ)
{
    if(!SQ) return 0;

    if (SQ->rear == MaxSize)
    {
        return 1;
    }
    return 0;
}

//入队,将元素 data 插入到队列 SQ 中
int EnterQueue( SeqQueue *SQ,DataType data){
    if(!SQ) return 0;

    if(IsFull(SQ)){
        cout<<"无法插入元素 "<<data<<", 队列已满!"<<endl;
        return 0;
    }

    SQ->queue[SQ->rear] = data; //在队尾插入元素 data
    SQ->rear++; //队尾指针后移一位
    return 1;
```

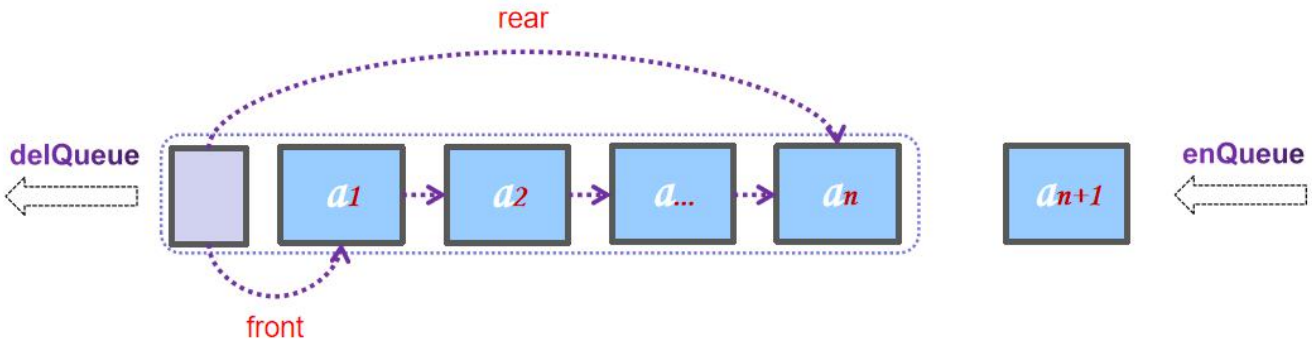
```
}


//出队，将队列中队头的元素 data 出队，后面的元素向前移动
int DeleteQueue(SeqQueue* SQ, DataType *data){

    if(!SQ || IsEmpty(SQ)){
        cout<<"队列为空！"<<endl;
         return 0;
    }

    if(!data) return 0;

    *data = SQ->queue[SQ->front];

    for(int i=SQ->front+1; i<SQ->rear; i++){//移动后面的元素
        SQ->queue[i-1]=SQ->queue[i];
    }

    SQ->rear--;//队尾指针前移一位
    return 1;
}

//出队，将队列中队头的元素 data 出队，出队后队头指针 front 后移一位
int DeleteQueue2(SeqQueue* SQ,DataType* data)
{
    if (!SQ || IsEmpty(SQ))
    {
        cout<<"队列为空！"<<endl;
        return 0;
    }

    if(SQ->front>=MaxSize){
        cout<<"队列已到尽头!"<<endl;
        return 0;
    }

    *data = SQ->queue[SQ->front];    //出队元素值
    SQ->front = (SQ->front)+1;       //队首指针后移一位
    return 1;
}

//打印队列中的各元素
void PrintQueue(SeqQueue* SQ)
{
    if(!SQ) return ;
```

```
    int i = SQ->front;
    while(i<SQ->rear)
    {
        cout<<setw(4)<<SQ->queue[i];
        i++;
    }
    cout<<endl;
}

//获取队首元素，不出队
int GetHead(SeqQueue* SQ,DataType* data)
{
    if (!SQ || IsEmpty(SQ))
    {
        cout<<"队列为空!"<<endl;
    }
    return *data = SQ->queue[SQ->front];
}

//清空队列
void ClearQueue(SeqQueue* SQ)
{
    if(!SQ) return ;
    SQ->front = SQ->rear = 0;
}

//获取队列中元素的个数
int getLength(SeqQueue* SQ){
    if(!SQ) return 0;

    return SQ->rear-SQ->front;
}

int main()
{
    SeqQueue *SQ = new SeqQueue;
    DataType data = -1;

    //初始化队列
    InitQueue(SQ);

    //入队
    for(int i=0; i<7; i++){
        EnterQueue(SQ, i);
    }

    //打印队列中的元素
```

```
    printf("队列中的元素(总共%d 个)： ", getLength(SQ));
    PrintQueue(SQ);
    cout<<endl;

    //出队
    //for(int i=0; i<10; i++){
        if(DeleteQueue2(SQ, &data)){
            cout<<"出队的元素是： "<<data<<endl;
        }else {
            cout<<"出队失败！ "<<endl;
        }
    //}

    //打印队列中的元素
    printf("出队一个元素后，队列中剩下的元素： ");
    PrintQueue(SQ);
    cout<<endl;


    system("pause");
    return 0;
}
```

链式存储

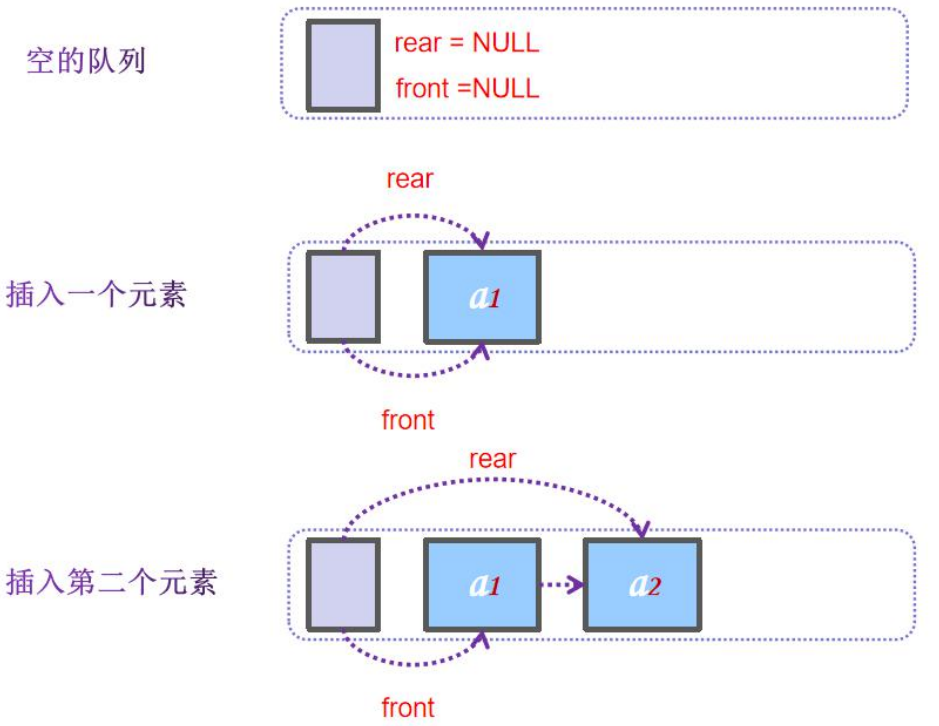
队列的链式存储结构，其实就是线性表的单链表，只不过它只是尾进头出而已，我们把它简称为链队列。为了操作上的方便，我们将队头指针指向链队列的头结点，而队尾指针指向终端节点



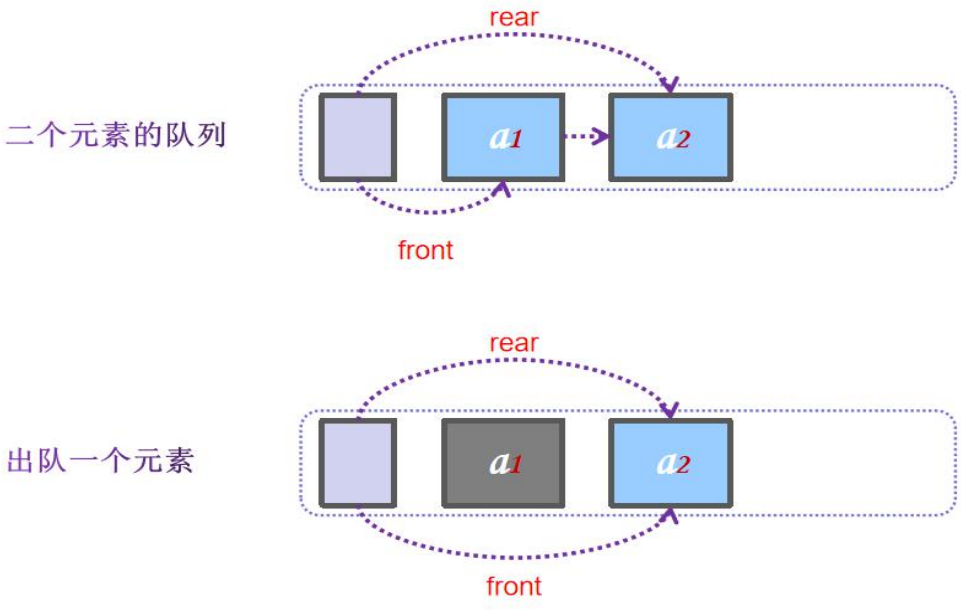

```
typedef int DataType;     //队列中元素类型

typedef struct _QNode {  //结点结构
    DataType data;
    struct _QNode *next;
}QNode;

typedef QNode * QueuePtr;

typedef struct Queue
{
    int      length;  //队列的长度
    QueuePtr front;   //队头指针
    QueuePtr rear;    //队尾指针
}LinkQueue;
```

链式队列操作图示

空队列时，front 和 rear 都指向空。



删除节点



完整代码实现：

```
#include <stdio.h>
#include <assert.h>
#include <Windows.h>
#include <iostream>
#include <iomanip>

using namespace std;

#define MaxSize 5          //队列的最大容量

typedef int DataType;     //队列中元素类型

typedef struct _QNode {  //结点结构
    DataType data;
    struct _QNode *next;
}QNode;

typedef QNode * QueuePtr;

typedef struct Queue
{
    int       length;  //队列的长度
    QueuePtr front;   //队头指针
    QueuePtr rear;    //队尾指针
}LinkQueue;


//队列初始化，将队列初始化为空队列
void InitQueue(LinkQueue *LQ)
{
    if(!LQ) return ;

    LQ->length = 0;
    LQ->front = LQ->rear = NULL;  //把对头和队尾指针同时置 0
 }

//判断队列为空
int IsEmpty(LinkQueue *LQ)
{
    if(!LQ) return 0;

    if (LQ->front == NULL)
```

```
    {
        return 1;
    }
    return 0;
}


//判断队列是否为满
int IsFull(LinkQueue *LQ)
{
    if(!LQ) return 0;

    if (LQ->length == MaxSize)
    {
        return 1;
    }
    return 0;
}

//入队,将元素 data 插入到队列 LQ 中
int EnterQueue( LinkQueue *LQ,DataType data){
    if(!LQ) return 0;

    if(IsFull(LQ)){
        cout<<"无法插入元素 "<<data<<", 队列已满!"<<endl;
        return 0;
    }

    QNode *qNode = new QNode;
    qNode->data = data;
    qNode->next = NULL;

    if(IsEmpty(LQ)){//空队列
        LQ->front = LQ->rear = qNode;
    }else {
        LQ->rear->next =qNode;//在队尾插入节点 qNode
        LQ->rear = qNode;     //队尾指向新插入的节点
    }
    LQ->length++;

    return 1;
}


//出队，将队列中队头的元素出队，其后的第一个元素成为新的队首
int DeleteQueue(LinkQueue *LQ, DataType *data){
    QNode * tmp = NULL;
```

```
    if(!LQ || IsEmpty(LQ)){
        cout<<"队列为空！"<<endl;
         return 0;
    }

    if(!data) return 0;
    tmp = LQ->front;

    LQ->front = tmp->next;
    if(!LQ->front) LQ->rear=NULL;//如果对头出列后不存在其他元素，则
rear 节点也要置空

    *data = tmp->data;
    LQ->length--;

    delete tmp;

    return 1;
}



//打印队列中的各元素
void PrintQueue(LinkQueue *LQ)
{   QueuePtr tmp;

     if(!LQ) return ;

    if(LQ->front==NULL){
        cout<<"队列为空！";
        return ;
    }

     tmp = LQ->front;
     while(tmp)
     {
         cout<<setw(4)<<tmp->data;
         tmp = tmp->next;
     }
     cout<<endl;
}

//获取队首元素，不出队
int GetHead(LinkQueue *LQ,DataType *data)
{
    if (!LQ || IsEmpty(LQ))
```

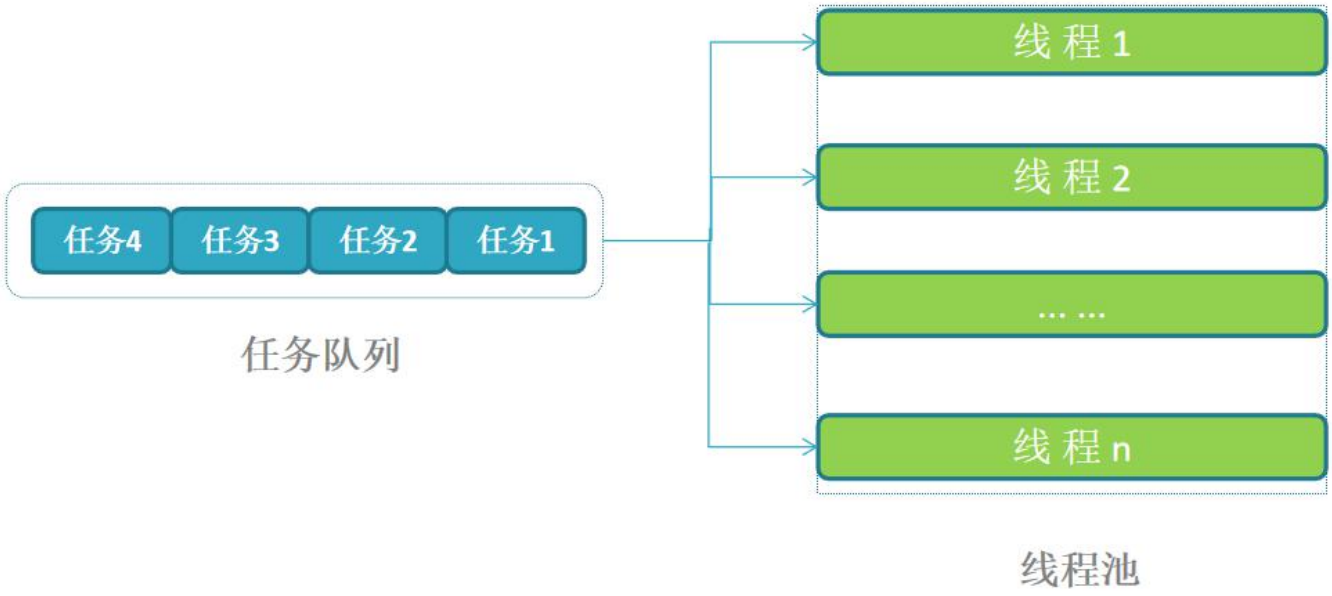
```
    {
        cout<<"队列为空!"<<endl;
       return 0;
    }

    if(!data) return 0;

    *data = LQ->front->data;
    return 1;
}

//清空队列
void ClearQueue(LinkQueue *LQ)
{
    if(!LQ) return ;

    while(LQ->front){
        QueuePtr tmp = LQ->front->next;
        delete LQ->front;
        LQ->front = tmp;
    }

    LQ->front = LQ->rear = NULL;
    LQ->length = 0;
}

//获取队列中元素的个数
int getLength(LinkQueue* LQ){
    if(!LQ) return 0;

    return LQ->length;
}

int main()
{
    LinkQueue *LQ = new LinkQueue;
    DataType data = -1;

    //初始化队列
    InitQueue(LQ);

    //入队
    for(int i=0; i<7; i++){
        EnterQueue(LQ, i);
    }

    //打印队列中的元素
```

```
    printf("队列中的元素(总共%d 个)：", getLength(LQ));
    PrintQueue(LQ);
    cout<<endl;

    //出队
    //for(int i=0; i<10; i++){
        if(DeleteQueue(LQ, &data)){
            cout<<"出队的元素是："<<data<<endl;
        }else {
            cout<<"出队失败！"<<endl;
        }
    //}

    //打印队列中的元素
    printf("出队一个元素后，队列中剩下的元素[%d]：", getLength(LQ));
    PrintQueue(LQ);
    cout<<endl;

    ClearQueue(LQ);
    cout<<"清空队列!\n";
    PrintQueue(LQ);


    //清理资源
    delete LQ;

    system("pause");
    return 0;
}
```

第 4 节 队列的企业级应用案例

4.1 线程池中的任务队列

线程池 - 由一个**任务队列**和一组处理队列的**线程**组成。一旦工作进程需要处理某个可能“阻塞”的操作，不用自己操作，将其作为一个任务放到线程池的队列，接着会被某个空闲线程提取处理。




```
typedef struct _QNode {  //结点结构
    int          id;
    void         (*handler)();
    struct _QNode *next;
}QNode;

typedef QNode * QueuePtr;

typedef struct Queue
{
    int        length;  //队列的长度
    QueuePtr   front;   //队头指针
    QueuePtr   rear;    //队尾指针
}LinkQueue;
```

```
#include <stdio.h>
#include <assert.h>
```

```
#include <Windows.h>
#include <iostream>
#include <iomanip>

using namespace std;

#define MaxSize 1000      //队列的最大容量

typedef struct _QNode {   //任务结点结构
    int         id;
    void        (*handler)(void);
    struct _QNode *next;
}QNode;

typedef QNode * QueuePtr;

typedef struct Queue
{
    int      length;  //队列的长度
    QueuePtr front;   //队头指针
    QueuePtr rear;    //队尾指针
}LinkQueue;


//分配线程执行的任务节点
QueuePtr thread_task_alloc()
{
    QNode  *task;

    task = (QNode *)calloc(1,sizeof(QNode));
    if (task == NULL) {
        return NULL;
    }

    return task;
}


//队列初始化，将队列初始化为空队列
void InitQueue(LinkQueue *LQ)
{
    if(!LQ) return ;

    LQ->length = 0;
    LQ->front = LQ->rear = NULL;  //把对头和队尾指针同时置 0
 }
```

```
//判断队列为空
int IsEmpty(LinkQueue *LQ)
{
    if(!LQ) return 0;

    if (LQ->front == NULL)
    {
        return 1;
    }
    return 0;
}


//判断队列是否为满
int IsFull(LinkQueue *LQ)
{
    if(!LQ) return 0;

    if (LQ->length == MaxSize)
    {
        return 1;
    }
    return 0;
}

//入队,将元素 data 插入到队列 LQ 中
int EnterQueue( LinkQueue *LQ,QNode *node){
    if(!LQ || !node) return 0;

    if(IsFull(LQ)){
        cout<<"无法插入任务 "<<node->id<<", 队列已满!"<<endl;
        return 0;
    }

    node->next = NULL;

    if(IsEmpty(LQ)){//空队列
        LQ->front = LQ->rear = node;
    }else {
        LQ->rear->next =node;//在队尾插入节点 qNode
        LQ->rear = node;     //队尾指向新插入的节点
    }
    LQ->length++;

    return 1;
}
```

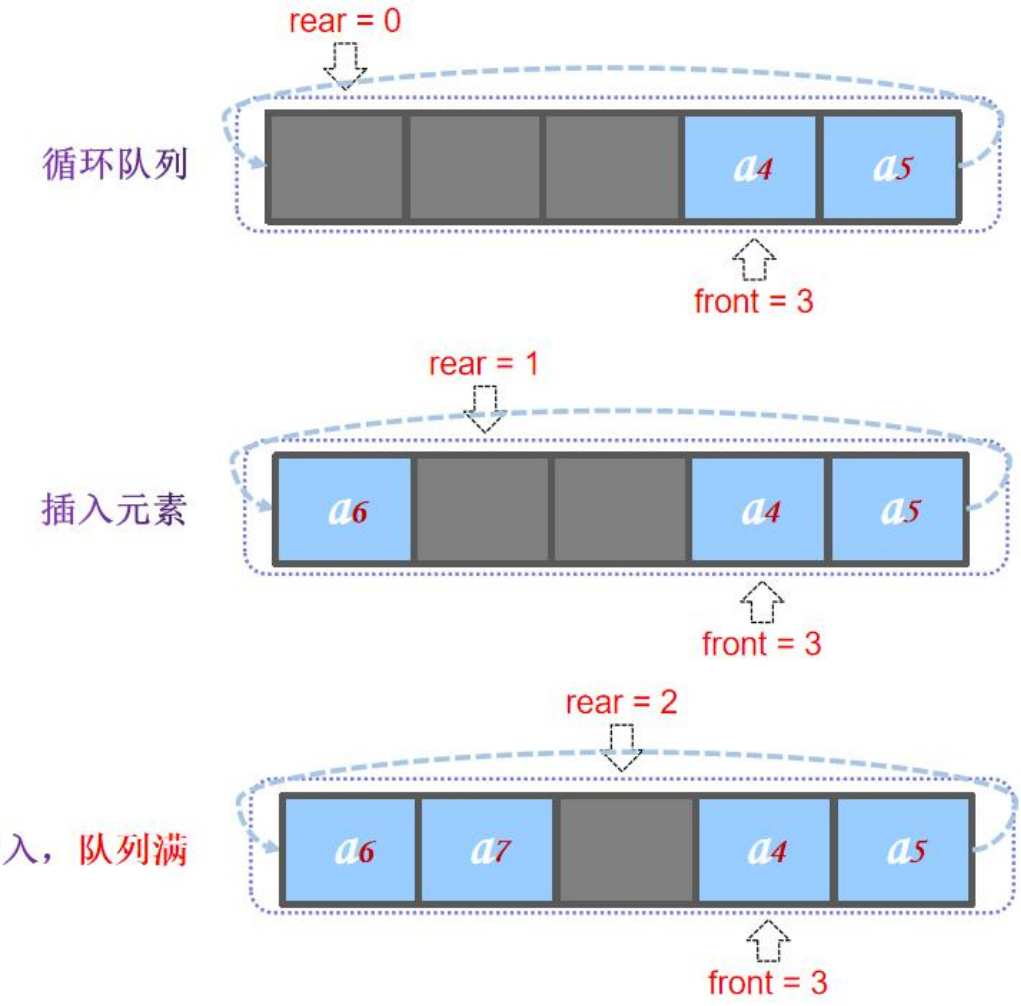
```
//出队,将队列中队头的节点出队，返回头节点
QNode * PopQueue(LinkQueue *LQ){
    QNode * tmp = NULL;

    if(!LQ || IsEmpty(LQ)){
        cout<<"队列为空！"<<endl;
         return 0;
    }

    tmp = LQ->front;

    LQ->front = tmp->next;
    if(!LQ->front) LQ->rear=NULL;//如果对头出列后不存在其他元素，则
rear 节点也要置空
    LQ->length--;

    return tmp;
}

//打印队列中的各元素
void PrintQueue(LinkQueue *LQ)
{   QueuePtr tmp;

     if(!LQ) return ;

    if(LQ->front==NULL){
        cout<<"队列为空！";
        return ;
    }

     tmp = LQ->front;
     while(tmp)
     {
         cout<<setw(4)<<tmp->id;
         tmp = tmp->next;
     }
     cout<<endl;
}

//获取队列中元素的个数
int getLength(LinkQueue* LQ){
    if(!LQ) return 0;

    return LQ->length;
}
```

```
void task1(){
    printf("我是任务 1 ...\n");
}

void task2(){
    printf("我是任务 2 ...\n");
}

int main()
{
     LinkQueue *LQ = new LinkQueue;
    QNode * task = NULL;

    //初始化队列
     InitQueue(LQ);

    //任务 1 入队
    task= thread_task_alloc();
    task->id = 1;
    task->handler = &task1;
    EnterQueue(LQ, task);

    //任务 2 入队
    task= thread_task_alloc();
    task->id = 2;
    task->handler = &task2;
    EnterQueue(LQ, task);


    //打印任务队列中的元素
     printf("队列中的元素(总共%d 个)：", getLength(LQ));
     PrintQueue(LQ);
     cout<<endl;

    //执行任务
    while( (task=PopQueue(LQ)) ){
        task->handler();
        delete task;
    }

    //清理资源
    delete LQ;
   system("pause");
   return 0;
}
```

4.2 循环队列

在队列的顺序存储中，采用出队方式 2，删除 front 所指的元素，然后加 1 并返回被删元素。这样可以避免元素移动，但是也带来了一个新的问题“**假溢出**”。

能否利用前面的空间继续存储入队呢？采用**循环队列**

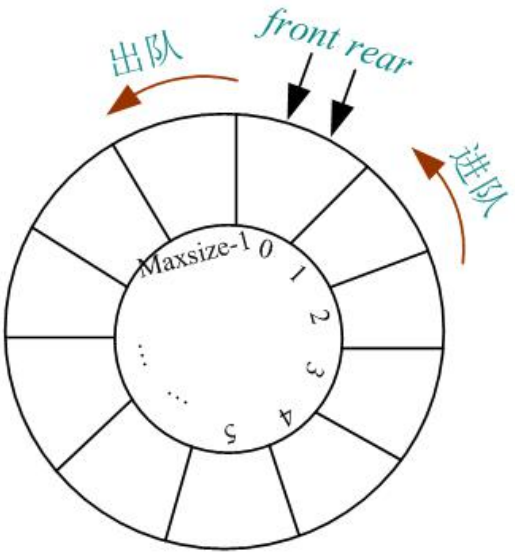


循环队列入队，队尾循环**后移**：　SQ->rear = （SQ->rear+1）%Maxsize;

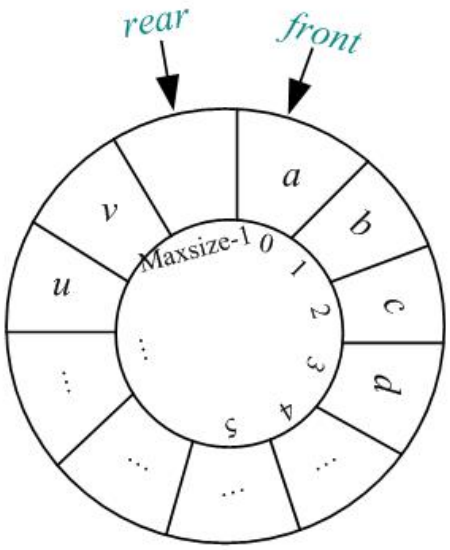
循环队列出队，队首循环**后移**：　SQ->front = （SQ->front+1）%Maxsize;

队空：SQ.front=SQ.rear; // SQ.rear 和 SQ.front 指向同一个位置

队满：　(SQ.rear+1) %Maxsize=SQ.front; // SQ.rear 向后移一位正好是 SQ.front



队空



队满

计算元素个数:

可以分两种情况判断：

- 如果 SQ.rear>= SQ.front：元素个数为 SQ.rear-SQ.front；
- 如果 SQ.rear<SQ.front：元素个数为 SQ.rear-SQ.front+ Maxsize；

采用取模的方法把两种情况统一为：(SQ.rear-SQ.front+Maxsize)% Maxsize

完整代码实现：

```
#include <stdio.h>
#include <assert.h>
#include <Windows.h>
#include <iostream>
#include <iomanip>

using namespace std;

#define MaxSize 5  //循环队列的最大容量

typedef int DataType;  //循环队列中元素类型

typedef struct Queue
{
    DataType queue[MaxSize];
    int front;   //循环队头指针
    int rear;    //循环队尾指针
}SeqQueue;


//队列初始化，将循环队列初始化为空队列
void InitQueue(SeqQueue *SQ)
{
    if(!SQ) return ;

    SQ->front = SQ->rear = 0;  //把对头和队尾指针同时置 0
}

//判断队列为空
int IsEmpty(SeqQueue *SQ)
{
    if(!SQ) return 0;

    if (SQ->front == SQ->rear)
    {
        return 1;
    }
    return 0;
}


//判断循环队列是否为满
int IsFull(SeqQueue *SQ)
{
```

```
    if(!SQ) return 0;

    if ((SQ->rear+1)%MaxSize == SQ->front)
    {
        return 1;
    }
    return 0;
}

//入队,将元素 data 插入到循环队列 SQ 中
int EnterQueue( SeqQueue *SQ,DataType data){
    if(!SQ) return 0;

    if(IsFull(SQ)){
        cout<<"无法插入元素 "<<data<<", 队列已满!"<<endl;
        return 0;
    }

    SQ->queue[SQ->rear] = data; //在队尾插入元素 data
    SQ->rear=(SQ->rear+1)%MaxSize; //队尾指针循环后移一位
    return 1;
}


//出队，将队列中队头的元素 data 出队，出队后队头指针 front 后移一位
int DeleteQueue(SeqQueue* SQ,DataType* data)
{
    if (!SQ || IsEmpty(SQ))
    {
        cout<<"循环队列为空！"<<endl;
        return 0;
    }


    *data = SQ->queue[SQ->front];   //出队元素值
    SQ->front = (SQ->front+1)% MaxSize;      //队首指针后移一位
    return 1;
}

//打印队列中的各元素
void PrintQueue(SeqQueue* SQ)
{
    if(!SQ) return ;

    int i = SQ->front;
    while(i!=SQ->rear)
    {
```

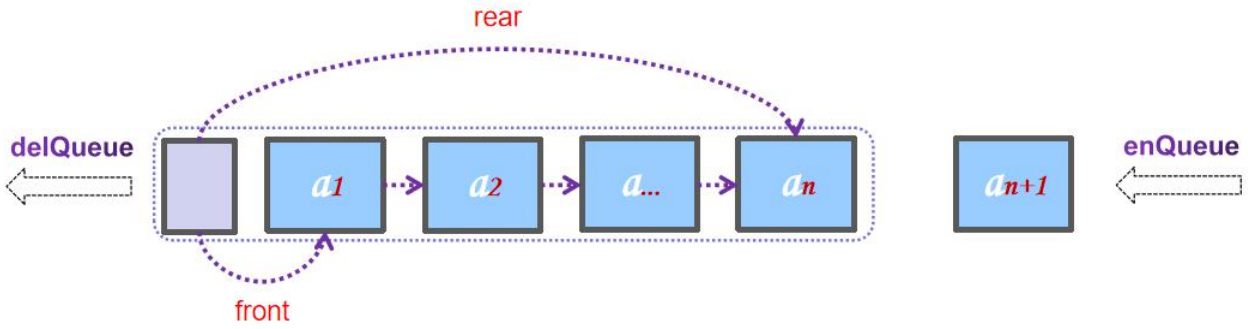
```
        cout<<setw(4)<<SQ->queue[i];
        i=(i+1)%MaxSize;
    }
    cout<<endl;
}

//获取队首元素，不出队
int GetHead(SeqQueue* SQ,DataType* data)
{
    if (!SQ || IsEmpty(SQ))
    {
        cout<<"队列为空!"<<endl;
    }
    return *data = SQ->queue[SQ->front];
}

//清空队列
void ClearQueue(SeqQueue* SQ)
{
    if(!SQ) return ;
    SQ->front = SQ->rear = 0;
}

//获取队列中元素的个数
int getLength(SeqQueue* SQ){
    if(!SQ) return 0;

    return (SQ->rear-SQ->front+MaxSize) % MaxSize;
}

int main()
{
    SeqQueue *SQ = new SeqQueue;
    DataType data = -1;

    //初始化队列
    InitQueue(SQ);

    //入队
    for(int i=0; i<7; i++){
        EnterQueue(SQ, i);
    }

    //打印队列中的元素
    printf("队列中的元素(总共%d 个)：", getLength(SQ));
    PrintQueue(SQ);
    cout<<endl;
```

```
    //出队
    for(int i=0; i<5; i++){
        if(DeleteQueue(SQ, &data)){
            cout<<"出队的元素是："<<data<<endl;
        }else {
            cout<<"出队失败！"<<endl;
        }
    }

    //打印队列中的元素
    printf("出队五个元素后，队列中剩下的元素个数为 %d 个：",
getLength(SQ));
    PrintQueue(SQ);
    cout<<endl;

    //入队 4 个
    for(int i=0; i<4; i++){
        EnterQueue(SQ, i+10);
    }

    printf("\n 入队四个元素后，队列中剩下的元素个数为 %d 个：",
getLength(SQ));
    PrintQueue(SQ);

    system("pause");
    return 0;
}
```

4.3 优先队列

英雄联盟游戏里面防御塔都有一个自动攻击功能，小兵排着队进入防御塔的攻击范围，防御塔先攻击靠得最近的小兵，这时候大炮车的优先级更高（因为系统判定大炮车对于防御塔的威胁更大），所以防御塔会优先攻击大炮车。而当大炮车阵亡，剩下的全部都是普通小兵，这时候离得近的优先级越高，防御塔优先攻击距离更近的小兵。

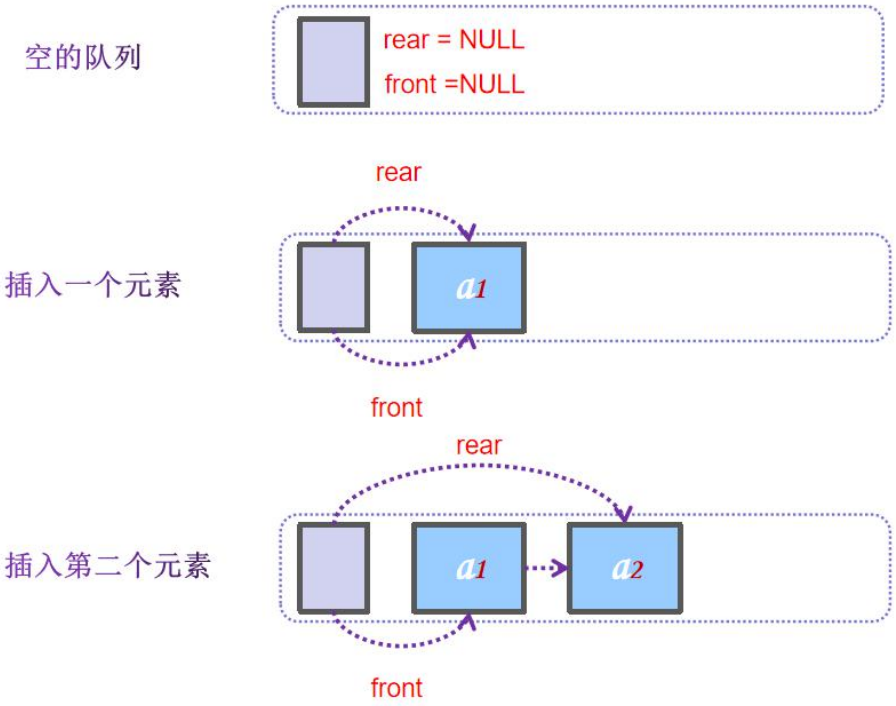
优先队列： 它的入队顺序没有变化，但是出队的顺序是根据优先级的高低来决定的。优先级高的优先出队。



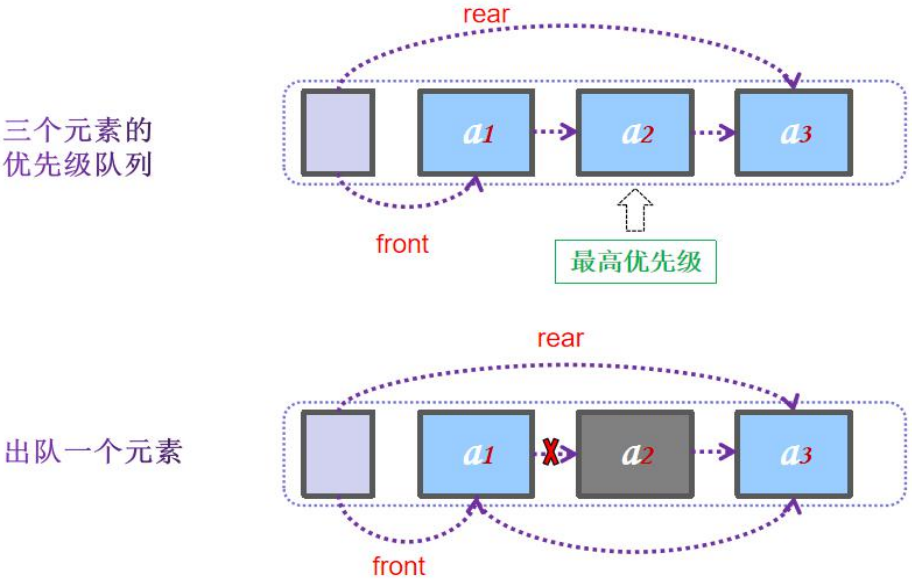

```
typedef int DataType;     //队列中元素类型

typedef struct _QNode {  //结点结构
    int priority;  //每个节点的优先级,9 最高优先级，0 最低优先级，优先级相同，取第一个节点
    DataType data;
    struct _QNode *next;
}QNode;
```

```
typedef QNode * QueuePtr;

typedef struct Queue
{
    int      length;  //队列的长度
    QueuePtr front;   //队头指针
    QueuePtr rear;    //队尾指针
}LinkQueue;
```

空的任务队列 & 插入元素



删除一个节点



源码实现

```
#include <stdio.h>
#include <assert.h>
#include <Windows.h>
#include <iostream>
#include <iomanip>

using namespace std;

#define MaxSize 5          //队列的最大容量

typedef int DataType;     //任务队列中元素类型

typedef struct _QNode {  //结点结构
    int priority; //每个节点的优先级,0 最低优先级，9 最高优先级，优先级相同，取第一个节点
    DataType data;
    struct _QNode *next;
}QNode;

typedef QNode * QueuePtr;

typedef struct Queue
{
    int       length;  //队列的长度
    QueuePtr front;   //队头指针
    QueuePtr rear;    //队尾指针
}LinkQueue;


//队列初始化，将队列初始化为空队列
void InitQueue(LinkQueue *LQ)
{
    if(!LQ) return ;

    LQ->length = 0;
    LQ->front = LQ->rear = NULL;  //把对头和队尾指针同时置 0
 }

//判断队列为空
int IsEmpty(LinkQueue *LQ)
{
    if(!LQ) return 0;

    if (LQ->front == NULL)
    {
```

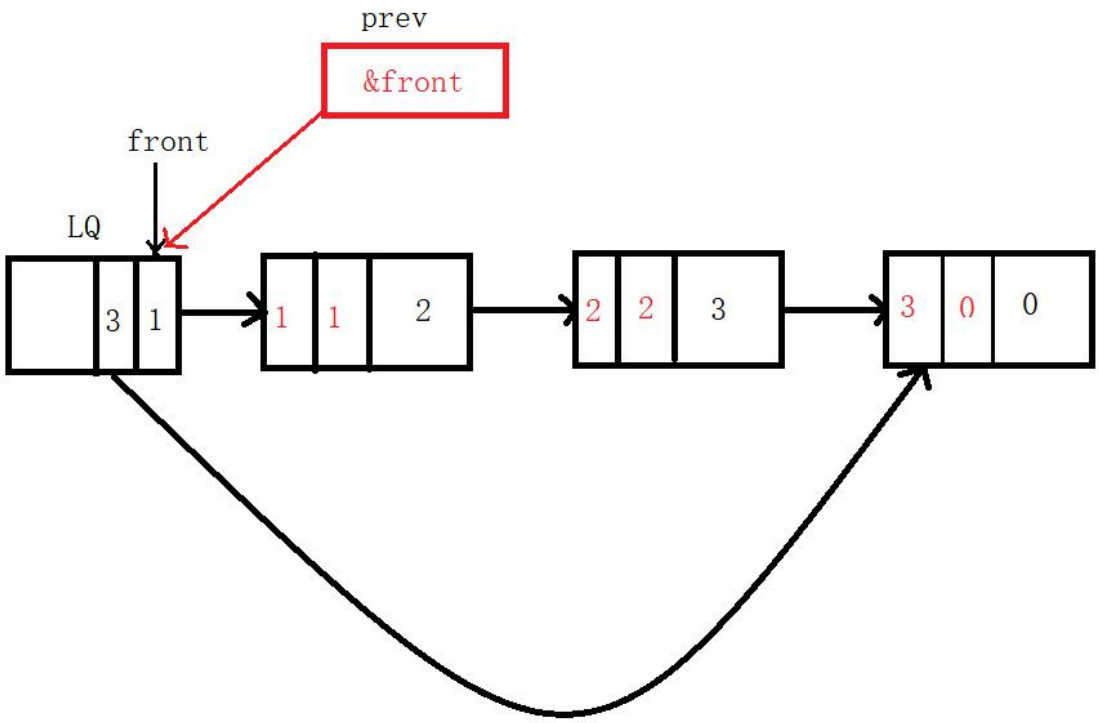
```
        return 1;
    }
    return 0;
}


//判断队列是否为满
int IsFull(LinkQueue *LQ)
{
    if(!LQ) return 0;

    if (LQ->length == MaxSize)
    {
        return 1;
    }
    return 0;
}

//入队,将元素 data 插入到队列 LQ 中
int EnterQueue( LinkQueue *LQ,DataType data, int priority){
    if(!LQ) return 0;

    if(IsFull(LQ)){
        cout<<"无法插入元素 "<<data<<", 队列已满!"<<endl;
        return 0;
    }

    QNode *qNode = new QNode;
    qNode->data = data;
    qNode->priority = priority;
    qNode->next = NULL;

    if(IsEmpty(LQ)){//空队列
        LQ->front = LQ->rear = qNode;
    }else {
        LQ->rear->next =qNode;//在队尾插入节点 qNode
        LQ->rear = qNode;     //队尾指向新插入的节点
    }
    LQ->length++;

    return 1;
}


//出队，遍历队列，找到队列中优先级最高的元素 data 出队
int DeleteQueue(LinkQueue *LQ, DataType *data){
    QNode **prev = NULL, *prev_node=NULL;//保存当前已选举的最高优先级
```

```
节点上一个节点的指针地址。
    QNode *last = NULL, *tmp = NULL;

    if(!LQ || IsEmpty(LQ)){
        cout<<"队列为空！"<<endl;
         return 0;
    }

    if(!data) return 0;
    //prev  指向队头 front 指针的地址
    prev = &(LQ->front);
    printf("第一个节点的优先级: %d\n", (*prev)->priority);
    last = LQ->front;
    tmp = last->next;
    while(tmp){
        if(tmp->priority >(*prev)->priority){
            printf("抓到个更大优先级的节点[priority: %d]\n",
tmp->priority);
            prev = &(last->next);
            prev_node= last;

        }
        last=tmp;
        tmp=tmp->next;
    }

    *data = (*prev)->data;
    tmp = *prev;
    *prev = (*prev)->next;
    delete tmp;

    LQ->length--;

    //接下来存在 2 种情况需要分别对待
    //1.删除的是首节点,而且队列长度为零
    if(LQ->length==0){
        LQ->rear=NULL;
    }

    //2.删除的是尾部节点
    if(prev_node&&prev_node->next==NULL){
        LQ->rear=prev_node;
    }

    return 1;
}
```

```
//打印队列中的各元素
void PrintQueue(LinkQueue *LQ)
{   QueuePtr tmp;

    if(!LQ) return ;

    if(LQ->front==NULL){
        cout<<"队列为空！";
        return ;
    }

    tmp = LQ->front;
    while(tmp)
    {
        cout<<setw(4)<<tmp->data<<"["<<tmp->priority<<"]";
         tmp = tmp->next;
    }
    cout<<endl;
}

//获取队首元素，不出队
int GetHead(LinkQueue *LQ,DataType *data)
{
    if (!LQ || IsEmpty(LQ))
    {
         cout<<"队列为空!"<<endl;
        return 0;
    }

    if(!data) return 0;

    *data = LQ->front->data;
    return 1;
}

//清空队列
void ClearQueue(LinkQueue *LQ)
{
    if(!LQ) return ;

    while(LQ->front){
        QueuePtr tmp = LQ->front->next;
        delete LQ->front;
        LQ->front = tmp;
    }
```

```
    LQ->front = LQ->rear = NULL;
    LQ->length = 0;
}

//获取队列中元素的个数
int getLength(LinkQueue* LQ){
    if(!LQ) return 0;

    return LQ->length;
}

int main()
{
    LinkQueue *LQ = new LinkQueue;
    DataType data = -1;

    //初始化队列
    InitQueue(LQ);

    //入队
    for(int i=0; i<5; i++){
        EnterQueue(LQ, i+10, i);
    }

    //打印队列中的元素
    printf("队列中的元素(总共%d 个)：", getLength(LQ));
    PrintQueue(LQ);
    cout<<endl;

    //出队
    for(int i=0; i<5; i++){
        if(DeleteQueue(LQ, &data)){
            cout<<"出队的元素是："<<data<<endl;
        }else {
            cout<<"出队失败！"<<endl;
        }
    }

    //打印队列中的元素
    printf("出队五个元素后，队列中剩下的元素[%d]：\n", getLength(LQ));
    PrintQueue(LQ);
    cout<<endl;

    ClearQueue(LQ);
    cout<<"清空队列!\n";
    PrintQueue(LQ);
```

```
    //清理资源
    delete LQ;

    system("pause");
    return 0;
}
```



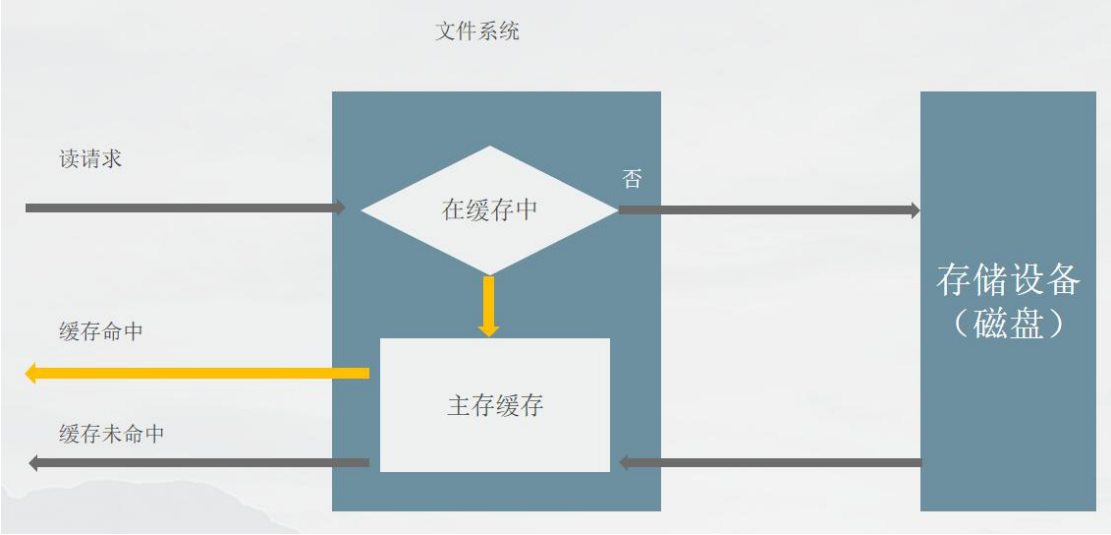
<图示>优先队列出队

4.4 动态顺序队列

使用链表动态存储的队列即为动态顺序队列，前面已经实现，故不再重复！

4.5 高并发 WEB 服务器队列的应用

在高并发 HTTP 反向代理服务器 Nginx 中，存在着一个跟性能息息相关的模块 - 文件缓存。



经常访问到的文件会被 nginx 从磁盘缓存到内存，这样可以极大的提高 Nginx 的并发能力，不过因为内存的限制，当缓存的文件数达到一定程度的时候就会采取淘汰机制，优先淘汰进入时间比较久或是最近访问很少(LRU)的队列文件。

具体实现方案：

1. 使用双向循环队列保存缓存的文件节点，这样可以实现多种淘汰策略：

   比如：如果采用淘汰进入时间比较久的策略，就可以使用队列的特性，先进先出

   如果要采用按照 LRU，就遍历链表，找到节点删除。

源码实现

nginx_queue.h

```
#ifndef  _NGX_QUEUE_H_INCLUDED_
#define  _NGX_QUEUE_H_INCLUDED_

typedef struct ngx_queue_s  ngx_queue_t;

struct ngx_queue_s {
    ngx_queue_t  *prev;
    ngx_queue_t  *next;
};


#define ngx_queue_init(q)            \
    (q)->prev = q;                   \
    (q)->next = q


#define ngx_queue_empty(h)           \
    (h == (h)->prev)


#define ngx_queue_insert_head(h, x)      \
    (x)->next = (h)->next;               \
    (x)->next->prev = x;                 \
    (x)->prev = h;                       \
    (h)->next = x


#define ngx_queue_insert_after   ngx_queue_insert_head


#define ngx_queue_insert_tail(h, x)         \
    (x)->prev = (h)->prev;                  \
    (x)->prev->next = x;                    \
    (x)->next = h;                          \
    (h)->prev = x


#define ngx_queue_head(h)                   \
    (h)->next


#define ngx_queue_last(h)                   \
    (h)->prev
```

```
#define ngx_queue_sentinel(h)                \
    (h)


#define ngx_queue_next(q)                    \
    (q)->next


#define ngx_queue_prev(q)                    \
    (q)->prev


#define ngx_queue_remove(x)                   \
    (x)->next->prev = (x)->prev;              \
    (x)->prev->next = (x)->next


#define ngx_queue_data(q, type, link)                \
    (type *) ((char *) q - offsetof(type, link))


#endif
```

Nginx_双向循环队列.cpp

```
#include <Windows.h>
#include <stdlib.h>
#include <iostream>
#include "nginx_queue.h"
#include <time.h>

using namespace std;


typedef struct ngx_cached_open_file_s {
    //其它属性省略...
    int fd;
    ngx_queue_t              queue;

}ngx_cached_file_t;
```

```
typedef struct {
    //其它属性省略...
    ngx_queue_t                expire_queue;
    //其它属性省略...
} ngx_open_file_cache_t;

int main(void){
    ngx_open_file_cache_t *cache = new ngx_open_file_cache_t;
    ngx_queue_t        *q;

    ngx_queue_init(&cache->expire_queue);

    //1. 模拟文件模块，增加打开的文件到缓存中
    for(int i=0; i<10; i++){
        ngx_cached_file_t *e = new ngx_cached_file_t;
        e->fd = i;
        ngx_queue_insert_head(&cache->expire_queue, &e->queue);
    }

    //遍历队列
    for(q=cache->expire_queue.next;
q!=ngx_queue_sentinel(&cache->expire_queue); q=q->next){
        printf("队列中的元素：%d\n", (ngx_queue_data(q,
ngx_cached_file_t, queue))->fd);
    }
    //模拟缓存的文件到期，执行出列操作
    while(!ngx_queue_empty(&cache->expire_queue)){
        q=ngx_queue_last(&cache->expire_queue);
        ngx_cached_file_t *cached_file = ngx_queue_data(q,
ngx_cached_file_t, queue);
        printf("出队列中的元素：%d\n", cached_file->fd);
        ngx_queue_remove(q);
        delete(cached_file);
    }


    system("pause");
    return 0;
}
```