# 约瑟夫环问题实验报告

杜金鸿,15338039

2017 年 3 月 9 日

## 目录

# 1 实验目的

## 1.1 问题描述

设有编号为 $1,2,\cdots,n$ 的 $n(n>0)$ 个人围成一个圈，每个人持有一个密码 $m$，从第 1 个人开始报数，报到 $m$ 时停止报数，报 $m$ 的人出圈，再从他的下一个人起重新报数，报到 $m$ 时停止报数，报 $m$ 的出圈,……，直到所有人全部出圈为止。当任意给定 $n$ 和 $m$ 后，求 $n$ 个人出圈的次序。

## 1.2 实验要求

1. 建立数据模型，确定存储结构;
2. 对任意 $n$ 个人，密码为 $m$, 实现约瑟夫环问题
3. 出圈的顺序依次输出

# 2 实验内容

## 2.1 约瑟夫环问题的存储结构

若将约瑟夫环看做链表，则每个人可用一个结点表示；若将约瑟夫环看做线性表（数组），则每个人可用一个整形数据表示。

## 2.2 建立约瑟夫环

分别采用不同存储方式的线性表作为存储结构，建立约瑟夫环。

1. 不带头结点的、由头指针指示的循环链表
2. 静态循环链表
3. 顺序表

## 2.3 设计约瑟夫环出圈算法

# 3 设计与编码

## 3.1 循环链表

1. 数据结构
   循环链表的数据元素用结点表示:

```
struct Node{
    int data;
    Node *next;
};
```

符号假设如下：

- *first*：头指针
- *pre*, *p*: 工作指针，分别指向当前结点的前驱和当前结点
- *count*：计数器

2. 算法流程

Algorithm 1 循环链表出圈算法

if $m == 1$ then
  依次输出链表中结点的编号
else
  $pre = first$;
  $p = first->next$;
  $count = 2$;
  while $p \neq pre$ do
    if $count == m$ then
      输出结点 $p$ 的编号;
      删除结点 $p$;$p = pre->next$;
      $count = 1$;
    end if
    $pre, p$ 后移;
    $count++$;
  end while
  输出最后一个结点的编号
end if

3. 代码示例

类的定义：

```
class Ring{
public:
    Ring(){create(10, 4);} // 构造函数
    Ring(int n,int m){create(n,m);}; // 复制构造函数
    ~Ring(){} // 析构函数
    void run(); // 出圈算法

private:
    Node *first; // 头指针
    void create(int n,int m); // 初始化
```

```
    void Delete(Node *p); // 某结点出圈
    int code; // 密码m
};
```

类的实现：

```
void Ring::create(int n, int m){
    Node *r,*s;
    first=new Node;
    r=first;
    r->data=1;
    for (int i=1; i<n; i++) {
        s=new Node;
        s->data=i+1;
        r->next=s;
        r=s;
    }
    r->next=first;
    code=m;
}

void Ring::Delete(Node *p){
    cout<<(p->data)<<'\t';
    Node *t;
    if(p->next!=p){
        if(first==p){
            first=p->next;
        }
        t=p;
        p=p->next;
        delete t;
    }
    else{
        delete p;
    }
}

void Ring::run(){
    Node *pre,*p;
    pre=first;
    p=first->next;
    if(code==1){
        while(pre->next!=first){
            cout<<pre->data<<endl;
            pre=pre->next;
        }
        cout<<pre->data<<endl;
    }
    else{
        int count=2;
        while(p!=pre){
            if(count==code){
                pre->next=p->next;
                Delete(p);
                p=pre->next;
                count=1;
```

```
            }
            pre=p;
            p=p->next;
            count++;
        }
        Delete(p);
    }
}
```

## 3.2 静态链表

1. 数据结构

   静态链表的数据元素用整形数组的元素表示，在定义的时候根据其长度申请空间创建数组。与循环链表不同的是，

   （1）静态链表还需要另外一个数组保存数据元素之间的关系，初始化时最后一个元素对应的指示数组的值为第一个元素的的逻辑位置，其余元素则指向其下一元素。

   （2）在出圈过程，静态链表不必删除、移动元素，只需更改指示数组的值，

   符号假设如下：

   - *flag*：指示数组首地址指针
   - *p*: 逻辑位置数组首地址指针
   - *count*：计数器

2. 算法流程

   算法流程与循环链表相似，此处省略。

3. 代码示例

   类的定义：

```
class Ring2{
public:
    Ring2(){create(10, 4);} // 构造函数
    Ring2(int n,int m){create(n,m);}; // 默认构造函数
    ~Ring2(){ // 析构函数
        delete[] flag;
        delete[] p;
    }
    void run(); // 出圈算法

private:
    int* p; // 逻辑位置数组
    int* flag; // 指示数组
    void create(int n,int m); // 初始化
    void Delete(int n); // 某结点出圈
    int num; // 总人数n
    int code; // 密码m
};
```

   类的实现：

```
void Ring2::create(int n, int m){
    p=new int[n];
    flag=new int[n];
    for (int i=0; i<n; i++) {
        *(p+i)=i+1;
        *(flag+i)=i+1;
    }
    *(flag+n-1)=0;
    code=m;
    num=n;
}

void Ring2::run(){
    if(code==1){
        for (int i=0; i<num ; i++) {
            cout<<p[i]<<endl;
        }
    }
    else{
        int i,j;
        i=0;
        j=1;
        int count=2;
        while(i!=j){
            if(count==code){
                cout<<p[j]<<'\t';
                flag[i]=flag[j];
                j=flag[i];
                count=1;
            }
            i=j;
            j=flag[j];
            count++;
        }
        cout<<p[i]<<'\t';
    }
}
```

## 3.3 顺序表

1. 数据结构

   顺序表相比于链表，具有索引方便的优势，因此在出圈算法中因尽量利用这点，直接算出下一个出圈的位置，而不必利用计数器逐一判断。与此同时，顺序表也有其不利之处，每一个元素出圈（即删除元素），可能会有大量元素的移动，其时间复杂度为 $O(n)$。

   符号假设如下：

   - $p$: 逻辑位置数组首地址指针

2. 算法流程

Algorithm 2 顺序表出圈算法

if $m == 1$ then
  依次输出链表中结点的编号
else
  $i = m - 1$;
  while n>1 do
    输出第 $i$ 个元素的编号
    删除第 $i$ 个元素;
    $n--$;
    $i = (i + m - 1)\%n$;
  end while
  输出最后一个元素的编号
end if

3. 代码示例

类的定义：

```
class Ring3{
public:
    Ring3(){create(10, 4);} // 构造函数
    Ring3(int n,int m){create(n,m);}; // 默认构造函数
    ~Ring3(){delete[] p;} // 析构函数
    void run(); // 出圈算法

private:
    int* p; // 数据元素
    void create(int n,int m); // 初始化
    void Delete(int n); // 某元素出圈
    int num; // 总人数n
    int code; // 密码m
};
```

类的实现：

```
void Ring3::create(int n, int m){
    p=new int[n];
    for (int i=0; i<n; i++) {
        *(p+i)=i+1;
    }
    code=m;
    num=n;
}
void Ring3::Delete(int n){
    cout<<p[n]<<'\t';
    for (int i=n ; i<num+1 ; i++) {
        p[i]=p[i+1];
    }
    num--;
}
```

```
void Ring3::run(){
    if(code==1){
        for (int i=0; i<num ; i++) {
            cout<<p[i]<<endl;
        }
    }
    else{
        int i;
        i=code-1;
        while(num>1){
            Delete(i);
            i=(i+code-1)%num;
        }
        cout<<p[0]<<'\t';
    }
}
```

# 4 运行与测试

## 4.1 运行结果

以 $n=10, m=4$ 为例，主程序如下：

```
#include <iostream>
using namespace std;
#include "Ring.h"
int main(int argc, const char * argv[]) {
    cout<<"约瑟夫环问题求解"<<endl;
    cout<<"链式存储结构——循环链表："<<endl;
    Ring a;
    a.run();
    cout<<endl<<"----------------------------------------"<<endl;
    cout<<"链式存储结构——静态链表："<<endl;
    Ring2 b;
    b.run();
    cout<<endl<<"----------------------------------------"<<endl;
    cout<<"顺序存储结构——顺序表："<<endl;
    Ring3 c;
    c.run();
    cout<<endl;
    return 0;
}
```

输出结果：

```
约瑟夫环问题求解
链式存储结构——循环链表：
4 8 2 7 3 10 9 1 6 5
----------------------------------------
链式存储结构——静态链表：
```

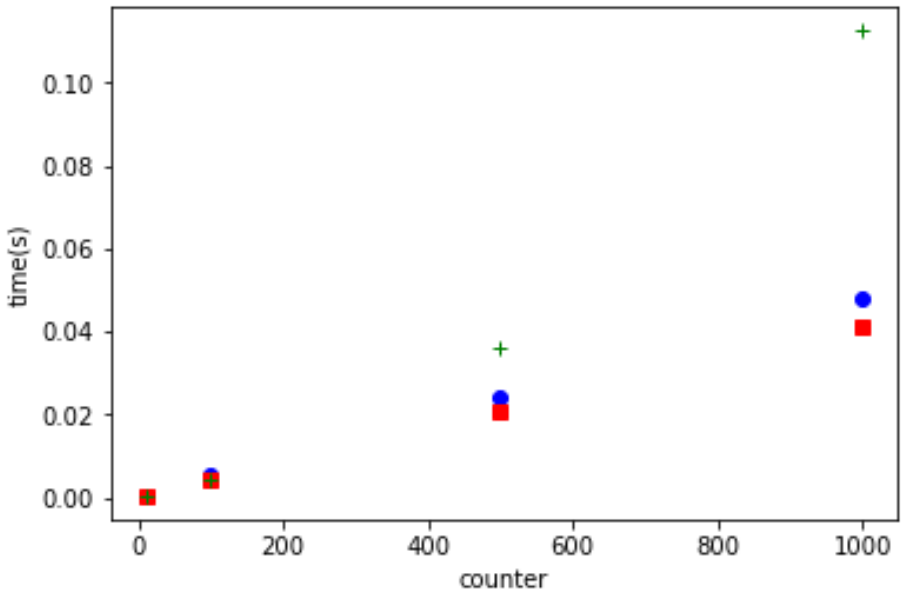
```
4 8 2 7 3 10 9 1 6 5
----------------------------------------
顺序存储结构——顺序表:
4 8 2 7 3 10 9 1 6 5
```

## 4.2 算法分析

通过重复多次实验，记录出圈算法运行时间，结果如下：

表 1: 算法时间比较（$s$）

| 次数 | $n$ | $m$ | 循环链表 | 静态链表 | 顺序表 |
|---|---|---|---|---|---|
| 100 | 10 | 4 | 0.000489 | 0.00043 | 0.000382 |
| 100 | 100 | 4 | 0.00563 | 0.004219 | 0.004537 |
| 100 | 500 | 4 | 0.024204 | 0.020498 | 0.036159 |
| 100 | 1000 | 4 | 0.048176 | 0.04102 | 0.112582 |




上图中正方形，圆形与加号分别代表循环链表，静态链表和顺序表可以看出，从总体上看，静态链表的算法最优；顺序表在 $\frac{m}{n}$ 较大时表现最好，而在该值相对较小的时候时间开销极大。究其原因，是在 $\frac{m}{n}$ 较小时，删除元素所花时间会线性增加，而静态链表实现约瑟夫环不需要删除元素，故表现最佳。

## 4.3 算法的改进

从上述讨论过程中可以发现，链式存储与顺序存储结构实现约瑟夫环各有优劣，循环链表时间花销主要在于元素的遍历查找，顺序表的时间花销主要在于元素的删除。显然前者很难加以改进，但是后者我们可以作

出如下改进：

结合静态链表的优势，将指示数组作为数据存储在顺序表的数据元素中。

这是因为在约瑟夫环问题中，个体的编号信息是顺序排列的，正好与顺序表索引一一对应，因此可以不必花费空间存储这一部分信息。

类的定义：

```
class Ring4{
public:
    Ring4(){create(10, 4);} // 构造函数
    Ring4(int n,int m){create(n,m);}; // 默认构造函数
    ~Ring4(){delete[] p;} // 析构函数
    void run(); // 出圈算法

private:
    int* p; // 数据元素
    void create(int n,int m); // 初始化
    int num; // 总人数n
    int code; // 密码m
};
```

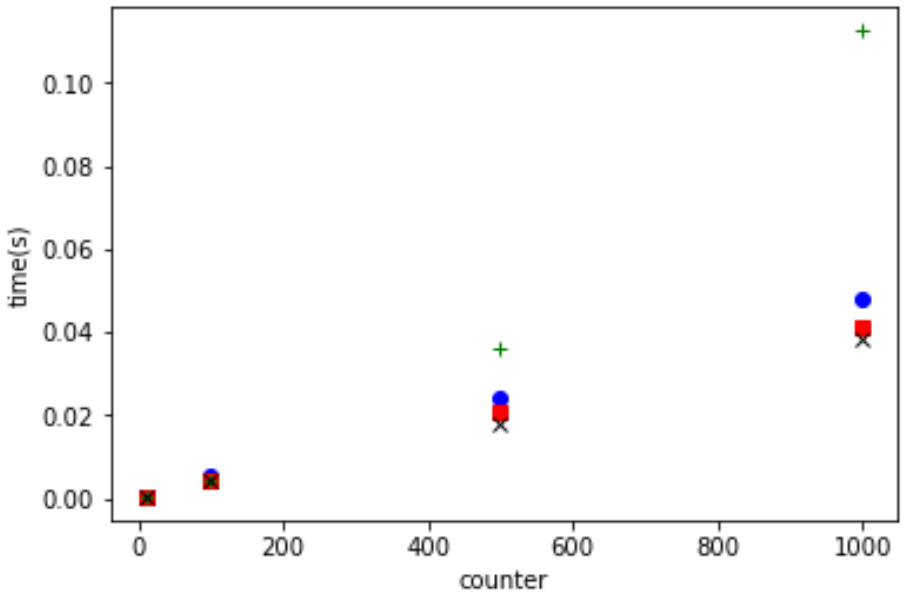
类的实现：

```
void Ring4::create(int n, int m){
    p=new int[n];
    for (int i=0; i<n; i++) {
        *(p+i)=i+1;
    }
    p[n-1]=0;
    code=m;
    num=n;
}


void Ring4::run(){
    if(code==1){
        for (int i=0; i<num ; i++) {
            cout<<i+1<<endl;
        }
    }
    else{
        int i,j;
        i=0;
        j=1;
        int count=2;
        while(i!=j){
            if(count==code){
                cout<<j+1<<'\t';
                p[i]=p[j];
                j=p[i];
                count=1;
```

```
            }
            i=j;
            j=p[j];
            count++;
        }
        cout<<i+1<<'\t';
    }
}
```

改进后测试结果如下：




可以看到，在时间复杂度方面，改进的顺序表出圈算法要略优于静态链表，猜测可能是减少了编号数组的索引而造成的结果。另一方面，改进的顺序表出圈的算法空间复杂度明显优于上述其它算法。

# 5 总结

线性表的链式存储与顺序存储结构各有优劣————链表的删除、插入等操作方便，但定位麻烦；顺序表的定位操作方便，但删除、插入等操作麻烦。因此应按照实际问题需要选择适当的存储结构。