电子信息学院

算法与数据结构
(基于现代C++的方法及实践)
ALGORITHM & DATA STRUCTURE IN MODERN C++

第7章 栈与队列

王文伟 Wang Wenwei, Dr.-Ing.
Tel: 18971562600
Email: wwwang@aliyun.com
课程QQ群：培珈EIS数据结构与算法，668792335

电子信息学院 Table of Contents

本章位置

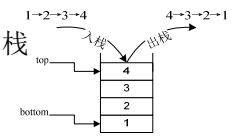
本章首先学习栈与队列的概念和抽象数据类型定义，然后分析它们的不同存储结构的实现和应用举例。

电子信息学院 Table of Contents

7.0 Introduction

- 栈和队列是两种特殊的线性结构。
  - 它们的数据元素之间也具有顺序的逻辑关系，都可以采用顺序存储结构和链式存储结构实现。
  - 线性表的插入和删除操作不受限制，可以在任意位置进行。
  - 栈的插入和删除操作只允许在结构的一端进行。
  - 队列的插入和删除操作则分别在结构的两端进行。
- 栈的特点是后进先出(LIFO)，队列的特点是先进先出(FIFO)。

7.1 栈的概念及类型定义



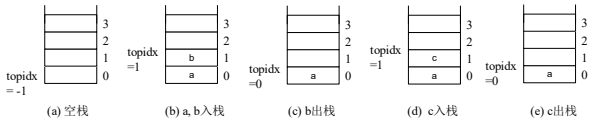
- 栈是一种“后进先出”（Last In First Out，LIFO）的线性结构。栈就像某种只有单个出入口的仓库，每次只允许一件件地往里面堆货物（入栈），然后一件件地往外取货物（出栈），不允许从中间放入或抽出货物。

7.1.1 栈的基本概念

- 栈（stack）是一种特殊的线性数据结构，元素之间具有顺序的逻辑关系，但插入和删除操作只允许在结构的一端进行。“后进先出”（Last In First Out，LIFO）。
- 向栈中插入数据元素的过程称为入栈(push)，删除数据元素的过程称为出栈(pop)。
- 允许插入和删除操作的一端称为栈顶（stack top），另一端称为栈底（stack bottom/back）。
- 栈顶指针：栈顶的当前位置随着插入和删除操作的进行而动态地变化，标识栈顶当前位置的变量称为栈顶指针。

栈状态随插入和删除操作而进行的变化

◆栈顶指针：标识栈顶的当前位置（动态）。



对于数据序列{a，b，c }依次进行
{ Push(a)，Push(b)，Pop( )，Push(c)，Pop( )}

7.1.2 抽象数据类型层面的栈

ADT 栈{
数据对象D：〈数据对象的定义〉
数据关系S：〈数据关系的定义〉
基本操作P：〈基本操作的定义〉
} ADT 栈

其中基本操作的定义格式为:

基本操作名（参数表）
初始条件：〈初始条件描述〉
操作结果：〈操作结果描述〉

抽象数据类型的描述方法

栈的数据对象和数据关系 D、S

◆栈是由若干数据元素组成的有限数据序列，元素间具有顺序的逻辑关系，入栈和出栈操作只能在栈顶。
◆对于由$n(n\geq 0)$个数据元素a_0，a_1，a_2，…，a_{n-1}组成的栈，记作：Stack = { a_0，a_1，a_2，…，a_{n-1}}
◆栈中的数据元素至少具有一种相同的属性，属于同一种抽象数据类型。
◆n表示栈中数据元素的个数，称为栈的长度。若n=0，则称为空栈。
◆可采用顺序存储结构和链式存储结构实现。顺序存储结构实现的栈称为顺序栈（Sequenced Stack），链式存储结构实现的栈称为链式栈（Linked Stack）。

栈的基本操作 P

◆Initialize: 栈的初始化。创建一个栈实例，并进行初始化操作，例如设置栈的状态为空。
◆Count: 数据元素计数。返回栈中元素个数。
◆Empty/Full: 判断栈的状态是否为空或满。
◆Push: 入栈。该操作将数据元素插入栈中作为新的栈顶元素。核心操作
◆Pop: 出栈。该操作取出当前栈顶数据元素，下一个数据元素成为新的栈顶元素。核心操作
◆Peek:探测栈顶。获得但不移除栈顶数据元素。栈顶指针不变。

7.1.3 C++中的栈类

◆C++标准库的栈类模板stack，刻画一种具有后进先出性质的数据集合。
强类型集合：stack<int>　stack<string>
◆stack类的成员函数：

公共构造函数
◆stack();　初始化新实例
◆stack (const container_type &);复制构造函数。

```
stack<int>  s1;
stack <int>  s2 (s1);
```


stack的公共方法

◆int size() const; 获取包含在栈中的元素数
◆bool empty() const; 测试栈是否为空
◆void push (const T& x); 将元素x添加到栈的顶部
◆void pop (); 移除位于栈顶部的元素
◆T& top(); 返回栈顶元素（的引用），可修改
◆const T& top() const; 返回栈顶元素（的引用），不修改

```
s1.push(5);
……
o = s1.top(); s1.pop()
```

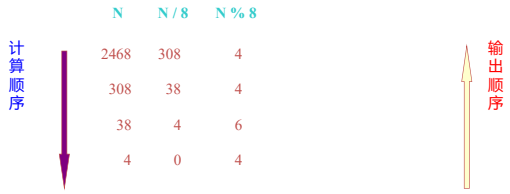
【例7.1】利用栈进行数制转换

$$N = a_n \times d^n + a_{n-1} \times d^{n-1} + \ldots + a_1 \times d^1 + a_0$$

数制转换就是要确定序列 $\{a_0, a_1, a_2, \ldots, a_n\}$

$$N = (N / d) \times d + N \% d$$

例如： $(2468)_{10}$ 转换成 $(4644)_8$ 的运算过程如下：




```
#include <iostream>   #include <stack>
void d2o(int n) {
  stack<int> s;
  cout << "十进制数:" << n << " -> 八进制:";
  while (n != 0) {
    s.push(n % 8);// "余数"入栈
    n = n / 8;     // 非零"商"继续运算
  }
  while(!s.empty()){// 和"求余"所得相逆的顺序输出八进制的各位数
    cout << s.top(); s.pop();
  }  cout << endl; }
int main(int argc, char* argv[]) {
     int n = 2468;
     if (argc >= 2)n = atoi(argv[1]);
     d2o(n);  }
```

7.2 栈的存储结构及实现

7.2.1 栈的顺序存储结构及操作实现
7.2.2 栈的链式存储结构及操作实现
7.2.3 栈的应用举例

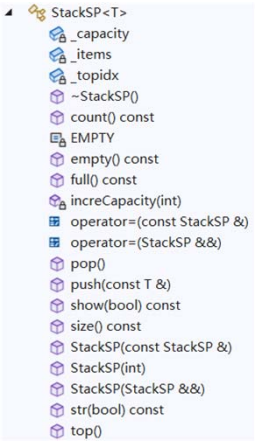
- 栈作为一种特殊的线性结构，可以如同一般线性表一样采用顺序存储结构和链式存储结构实现。顺序存储结构的栈称为顺序栈（Sequenced Stack），链式存储结构的栈称为链式栈（Linked Stack）。

7.2.1 栈的顺序存储结构及操作实现

- 栈的顺序存储结构：用一组连续的存储空间存放栈的数据元素。定义StackSP类刻画之。
- 成员变量_items定义为unique_ptr型智能指针，即准备用数组存储栈的元素。成员变量_topidx指示当前栈顶元素的下标，起着栈顶指针的作用。
- 用StackSP类构造的对象就是栈实例。通过对这个对象调用（公有的）成员函数进行相应的操作。

```
class StackSP {
private:    const int EMPTY = -1;
      unique_ptr<T[]> _items;
      int _topidx;   //_topidx为栈顶元素下标
      int _capacity;
public:     ……  };
```


顺序栈的类图



顺序栈的操作

- 栈的初始化: 构造方法
- 返回栈的元素个数：属性Count
- 判断栈的空与满状态：属性Empty/属性Full
- 入栈Push(k)：当栈不满时，栈顶元素下标topIdx加1，将k放入top位置，作为新的栈顶数据元素。
- 出栈Pop：当栈不空时，取走top处的元素， topIdx减1，下一位置的数据元素作为新的栈顶元素。
- 获得栈顶数据元素的值Top。当栈非空时，获得topIdx位置处的数据元素，此时该数据元素未出栈，topIdx值不变。

1）栈的初始化

- 构造函数初始化一个栈对象，它为_items数组申请指定大小的存储空间来存放栈的数据元素，设置栈初始状态为空。

```
StackSP(int initCapa = DefaultCapacity){
 // cout << "constructoring SequencedStack object" << endl;
  _capacity = initCapa;
  _items = make_unique<T[]>(_capacity);
  _topidx = EMPTY;     }
```

使用

```
SequencedStack<int> ss1(128);
SequencedStack<int> ss2;
```


copy constructor

```
StackSP(const StackSP& s) {
  _capacity = s._capacity;
  _items = make_unique<T[]>(_capacity);
  _topidx = s._topidx;
  for (int i = 0; i <= _topidx; i++)
    _items[i] = s._items[i];
}
```

使用

```
SequencedStack<int> ss3(128);……
SequencedStack<int> ss4(ss3);
```


2）返回栈的元素个数

```
int size() const{return _topidx + 1;};
int count() const{return _topidx+1;};
```

- const修饰表明这个函数不修改实例的状态，是个只读过程。

3）判断栈是否为空和判断栈是否为满

布尔类函数

- empty()：当_topidx为EMPTY时，表明栈为空状态，true。
- full()：当_topidx已指向当前的最后一个单元时，表明栈为满状态，true

```
bool empty() const{return _topidx==EMPTY;};
```

```
bool full() const{return _topidx>=_capacity-1;};
```


4）入栈Push(k)

- 过程：当栈不满时，栈顶元素下标_topidx自加1，将新数据k放入_topidx位置，作为新的栈顶元素。
- 当栈实例当前分配的存储空间已装满数据元素，在进行后续的操作前，需要调用本类中定义的increCapacity函数重新分配存储空间，并将原数组中的数据元素逐个拷贝到新数组。
- 入栈数据的类型：调用push函数实参的类型要与栈实例定义时声明的元素类型保持一致。例如：定义s为StackSP<string>类型，则以后入栈语句s.push(k)中的实参k必须为string类型。

```
StackSP<int> s;
s.push(14); s.push(8); s.push(27);
```


push(k)的代码

```
void push(const T& k) {
  if (full())increCapacity(DefaultCapacity);
    _topidx++;
    _items[_topidx] = k;  }
```

重新分配存储空间

```
void increCapacity(int amount=DefaultCapacity) {
   _capacity += amount;
   int cnt = _topidx + 1;
   auto newspace= make_unique<T[]>(_capacity);
   for(int i=0;i<cnt;i++)newspace[i]=_items[i];
   _items.reset();
    _items = move(newspace);}
```

push操作的时间复杂度

- 如果为栈预分配的空间合理，栈处于非满状态，push操作的时间复杂度为 O(1)。
- 如果经常需要增加容量以容纳新元素，则push操作的时间复杂度 成为 O(n)。

5）出栈pop

- 过程：当栈不空时，_topidx自减1，（原）下一位置上的元素成为新的栈顶元素。
- 此操作的运算复杂度是 O(1)。

```
void pop() {
  if (!empty()){     //栈不空
    _topidx--;
    return; }
  else  //栈空时产生异常
    throw underflow_error("empty");
}
```


6）获得栈顶数据元素的值top()

- 过程：当栈非空时，获得_topidx位置处的元素，此时该数据并不出栈，变量_topidx保持不变。运算复杂度是 O(1)。

```
const T& top() const {
  if (!empty()) return _items[_topidx];
  else  throw underflow_error("empty");
}
T& top();
```

第一个形式提供读的功能，第二个形式提供设置的功能。

7）显示栈中每个数据元素的值

- 当栈非空时，从栈顶结点开始，直至栈底结点，依次在控制台输出各结点值。

```
void show(bool showTypeName = false) const{
  if (showTypeName)cout<<"SequencedStack: ";
  if (!empty())
    for(int i=_topidx; i >= 0; i--)
      cout << _items[i] << " -> ";
  cout << " |." << endl;}
```

A -> B-> C-> |.

```
string str(bool showTypeName = false) const;
```


【例7.2】使用顺序栈的基本操作

- 在控制台编译SequencedStackTest.cpp：

```
X> cl SequencedStackTest.cpp  ..\Debug\dsa.lib
X> SequencedStackTest
```

- 运行结果如下：

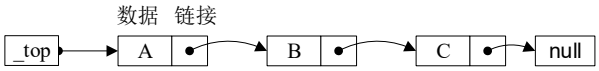
```
字符串型栈 Push: A B C
Push:  1 2   SequencedStack: 2 -> 1 -> C -> B -> A ->  |.
Pop:   2 1   SequencedStack: C -> B -> A ->  |.
Push:  3 4   SequencedStack: 4 -> 3 -> C -> B -> A ->  |.
Pop:   4 3   SequencedStack: C -> B -> A ->  |.
十进制数: 1357 -> 八进制: 2515
```



```
int i = 0, n = 4; StackSP<string> s1(20);
cout<<"字符串型栈 Push: A B C"<<endl;
s1.push("A"); s1.push("B"); s1.push("C");
for(i=1;i<=n;i+=2) { s1.push(to_string(i));
  s1.push(to_string(i+1));
  cout<<"Push: "<<i<<" "<<i+1<<"\t"; s1.show(true);
  cout<<"Pop:  "<<s1.top();s1.pop();
  cout<<"\t"<<s1.top()<<"\t"; s1.pop();
  s1.show(true);   }
int m = 1357; StackSP<int>  s(20);
cout << "十进制数:" << m << " ->八进制:";
while(m!=0){s.push(m%8); m=m/8;}
  int j = s.count();
  while(j>0){cout<<s.top(); s.pop();j--; }
  cout << endl;
```

7.2.2 栈的链式存储结构及操作实现

```
class LinkedStack {
private:
     SLNode<T>* _top;
     int _count;
public:
..... }
```



单向链结点类（以前的设计成果）

```
struct SLNode {
  T item;     //存放结点值
  SLNode<T>* next;//指向后继结点的指针
  // 构造函数，构造值为k的结点
  SLNode(const T& k):
     item(k),next(nullptr) { }
  // 缺省构造函数，构造缺省值的结点
  SLNode():next(nullptr), item{} {  }
  //析构函数
  ~SLNode() {     }   };
```


链式栈的操作

- 栈的初始化：构造函数
- 判断栈的空与满状态：empty()/full()
- 返回栈的元素个数：size()或count()
- 入栈：push()
- 出栈：pop()
- 获得栈顶数据元素的值，该数据元素未出栈Peek。

- 成员变量_top指向栈顶数据结点，结点类型为单向链结点类SLNode，结点数据域的类型为泛型T。入栈和出栈操作都是针对栈顶指针_top所指向的结点进行的
- 用LinkedStack类构造的对象就是一个个栈实例。

1）栈的初始化

- 构造函数创建和初始化一个栈实例（空栈）。

```
LinkedStack(){_top = nullptr; _count = 0;}
// copy constructor
LinkedStack(const LinkedStack& s) {
  _count=s._count; _top=nullptr;
  int n = 0; SLNode<T>* q = _top;
  SLNode<T>* p=s._top; SLNode<T>* t;
  while (n < _count) {
    t = new SLNode<T>(p->item);
    if (n == 0){_top = t;q = _top;}
    else{q->next = t; q = t ; }
    p = p->next; n++;     }     }
```


对象的销毁

```
//deconstructor. 析构函数
~LinkedStack() {
  if (_top != nullptr)
    ::dispose(_top);
}

void clear() {
  if (_top != nullptr)
    ::dispose(_top);
  _top = nullptr;    _count = 0;
}
```


2）返回栈的元素个数

```
int count() const { return _count; };
int size() const { return _count; };
```

3）判断栈状态是否为空或是否已满

- 当_top == nullptr时，栈为空；true
- 动态向系统申请存储空间，不必判断栈是否已满。

```
bool empty() const {
  return (_count == 0)&&(_top == nullptr);
};
```


4）入栈push

- 在栈顶结点_top之前插入一个结点来存放数据k，并使_top指向新的栈顶结点。

```
void push(const T& k) {
  SLNode<T>* t = new SLNode<T>(k);
  t->next = _top;      _top = t;
  _count++;
}
```

运算复杂度是 O(1)

5）出栈pop

- 当栈不为空时，成员_count自减1，栈顶的后继元素成为新的栈顶，销毁原栈顶元素。

```
void pop() {
  if (!empty()) {    //栈不空
    SLNode<T>* t = _top; _top = _top->next;
    _count--;   delete t;
    return;
  }
  else {//栈空时产生异常
    throw underflow_error("Stack is empty: ");
  }  }
```

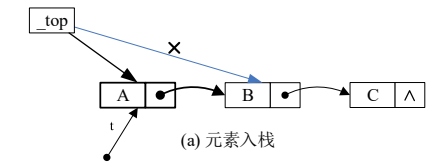
运算复杂度是 O(1)

6）获得栈顶数据元素的值top

- 当栈非空时，获得栈顶_top的数据，此时该数据元素不出栈，_top值不变。

```
const T& top() const {
  if (!empty()) {//栈不空
    return _top->item; //取得栈顶元素
  }else {//栈空时产生异常
    throw underflow_error("empty"); }
}
T& top();
```


链式栈的基本操作示意



(a) 元素入栈

(b) 元素出栈

应用中首先关注数据结构的抽象功能

- 由以上多个操作的算法实现分析可知，顺序栈StackSP和链式栈LinkedStack，都实现了“栈”这个抽象数据结构的基本操作。无论是StackSP类还是LinkedStack类，都可以用来建立具体的栈实例，通过栈实例调用入栈或出栈方法进行相应的操作。
- 一般情况下，解决某个问题关注的是栈的抽象功能，而不必关注栈的存储结构及其实现细节。

7.2.3 栈的应用举例(这一部分稍后更新)

栈是一种具有“后进先出”特性的特殊线性结构，适合作为求解具有LIFO特性问题的数学模型，因此栈成为解决相应问题算法设计的有力工具。

- 基于栈结构的函数嵌套调用
- 判断表达式中括号是否匹配
- 使用栈计算表达式的值

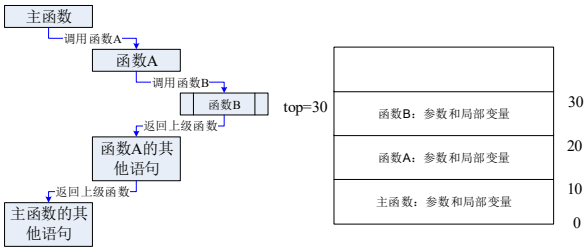
1. 基于栈结构的函数嵌套调用

- 程序中函数的嵌套调用是指：一个函数的执行语句序列中存在对另一个函数的调用，每个函数在执行完后应返回到调用它的函数中，对于多层嵌套调用来说，函数返回的次序与函数调用的次序正好相反，整个过程具有后进先出的特性。系统建立一个栈结构可以实现这种函数嵌套调用机制。系统栈
- 执行函数A时，A中的某语句调用函数B，系统要做一系列的入栈操作：
  - 将函数调用语句后的下一条语句作为返回地址信息保存在栈中，该过程称为保护现场；
  - 将A调用函数B的实参保存在栈中，该过程称为实参压栈；
  - 在栈中分配函数B的局部变量，开始执行函数B的其他语句。

子函数返回的过程

- B函数执行完成时，系统则要做一系列的出栈操作才能保证将系统控制返回调用B的函数A中
  - 退回栈中为函数B的局部变量分配的空间；
  - 退回栈中为函数B的参数变量分配的空间；
  - 取出保存在栈中的返回地址信息，该过程称为恢复现场，程序继续运行A函数。
- 函数返回的次序与函数调用的次序正好相反。可见，系统栈结构是实现函数嵌套调用或递归调用的基础。

函数嵌套调用时的系统栈



(a) 函数的嵌套调用与返回　　(b) 函数调用时的系统栈

2）判断表达式中括号是否匹配

如表达式中

（ [] （） ）或 [（ [ ] [ ] ）]

等为正确的括号匹配，

[（] ）或 （ [（）] 或 [（）] ）

均为括号不匹配。

检验括弧匹配：按“期待的急迫程度”进行处理 。

括弧匹配算法的设计思想

1）凡出现**左括弧(**，则**进栈**；

2）凡出现**右括弧)**，首先检查栈是否空？

若**栈空**，则表明该**“右括弧)”多余**

否则**和栈顶元素**比较，

若**相匹配**，则**“左括弧(”出栈**

否则表明**不匹配**

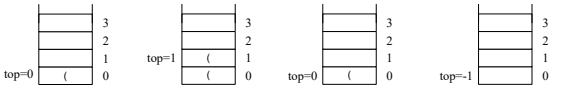
3）**表达式**检验**结束时**，

若**栈空**，则表明表达式中**匹配正确**

否则表明**“左括弧(”有余**

表达式括号匹配过程中栈状态的变化

((9-1)*(3+4))



(a) 第一个“(”入栈　(b) 第二个“(”入栈　(c) 遇到“)”时，出栈一个“(”，一对括号匹配　(d) 表达式检测完，栈为空：括号全部匹配


```
string MatchingBracket(const string& expstr) {
  StackSP<char> s1(30); //创建空栈
  char NextToken, OutToken; size_t i=0;bool LlrR= true;
  while (LlrR && i < expstr.size()) {
    NextToken = expstr[i++];
    switch (NextToken) {
    case '(':                          //遇见左括号时,入栈
      s1.push(NextToken); break;
    case ')':                          //遇见右括号时,出栈
      if (s1.empty()) LlrR = false;
      else {
        OutToken = s1.top(); s1.pop();
        if (OutToken!='(')LlrR = false;//判断出栈的是否为左括号
      }  break;   }   }
  if (LlrR)
    if (s1.empty()) return "OK!";
    else return "期望)!";
  else return "期望(!"; }
```

3）表达式求值

设 Exp = S1 + OP + S2

则称　OP + S1 + S2　为前缀表示法

S1 + OP + S2　为中缀表示法

S1 + S2 + OP　为后缀表示法

例如:　**Exp** = a × b + (c – d / e) × f

前缀式:　+ × a b × – c / d e f

中缀式:　a × b + c – d / e × f

后缀式:　a b × c d e / – f × +

结论:

5) 后缀式的运算规则为:

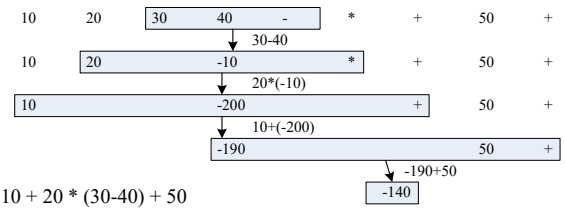
1) 操作数之间的相对次序不变;

运算符出现的顺序恰为表达式的运算顺序;每个运算符和在它之前出现且紧靠它的两个操作数构成一个最小表达式;

2) 运算符的相对次序不同;

后缀表达式的计算过程

- 后缀表达式中的运算符没有优先级，而且后缀表达式不需括号。
- 后缀表达式的求值过程能够严格地从左到右顺序进行，符合运算器的求值规律。从左到右按顺序进行运算，遇到某个运算符时，则对它前面的两个操作数求值

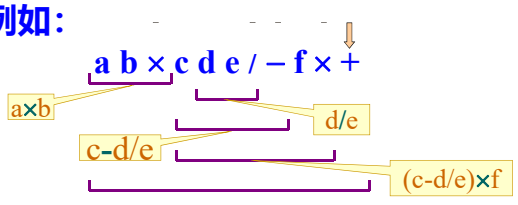


10 + 20 * (30-40) + 50

如何从后缀式求值？

- **先找运算符，**
- **再找操作数**

例如:



如何从原表达式求得后缀式？

分析 “原表达式” 和 “后缀式”中的运算符:

原表达式: a + b × c – d / e × f

后缀式: a b c × + d e / f × –

在后缀式中，优先权高的运算符领先于优先权低的运算符出现。

每个运算符的运算次序要由**它之后的一个运算符**来定。

从原表达式求得后缀式的规律

1) 设立暂存运算符的**栈**OPTR和暂存操作数的**栈**OPND

2) 设表达式的结束符为“#”，
予设运算符栈的栈底为“#”

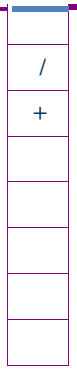
3) 若当前字符是操作数，则进入栈OPND;

4) 若当前运算符的优先数高于栈顶运算符，则进栈;

5) 否则，退出栈顶运算符发送给后缀式;

6) “(” 对它之前后的运算符起隔离作用，“)”可视为自相应左括弧开始的表达式的结束符。

a b c d e / + × f - × #



7.3 队列的概念及类型定义

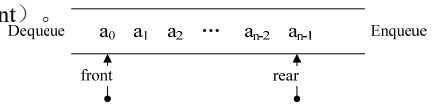
7.3.1 队列的基本概念

7.3.2 抽象数据类型层面的队列

7.3.3 C++中的队列类

7.3.1 队列的基本概念

- 队列(queue)是一种特殊的线性数据结构，其插入和删除操作分别限定在队列的两端进行，是一种“先进先出”（First In First Out，FIFO）的线性结构。
- “先到先服务，后到排队尾”。在计算机系统中，如果多个进程需要使用某个资源，它们就要排队等待该资源的就绪。此类问题的求解，需要队列数据结构。
- 向队列中插入元素的操作称为**入队**(enqueue/push)，删除元素的操作称为**出队**（dequeue/pop）。允许入队的一端为**队尾**（rear/back），允许出队的一端为**队头**（front）。



7.3.2 抽象数据类型层面的队列 D、S

- 队列是由若干数据元素组成的有限数据序列
- 对于由$n(n\geq 0)$个数据元素a_0，a_1，a_2，…，a_{n-1}组成的队列，记作：

  Queue = $\{a_0, a_1, a_2, \ldots, a_{n-1}\}$
- 队列中的数据元素至少具有一种相同的属性，属于相同的抽象数据类型。
- n表示队列的元素个数，称为队列的长度，若n=0，则称为空队列。
- 可以采用顺序存储结构和链式存储结构实现。顺序存储结构实现的队列称为顺序队列（sequenced queue），链式存储结构实现的队列称为链式队列（linked queue）。

队列的基本操作 P

- Initialize: 队列的初始化。创建一个队列，并进行初始化操作。
- Count: 返回队列中元素个数。
- Empty: 判断队列的状态是否为空。
- Full: 判断队列的状态是否已满。
- Enqueue: 入队。该操作将数据加入队列队尾处。在入队前须判断队列的状态是否已满。核心操作
- Dequeue: 出队。该操作取出队头元素。在出队前，须判断队列的状态是否为空。核心操作
- Peek:探测队首。获得但不移除队首数据元素。

7.3.3 C++中的队列类

- C++标准库包含一个队列类模板queue，队列类刻画了一种数据先进先出的集合，是编程中常用的数据集合类型。
- queue类的public函数：

  公共构造函数
- queue(); 初始化新实例
- queue (const container_type &); 复制构造函数

公共成员函数

- int size() const;
  获取包含在队列中的元素数

```
queue<int> x;
int i = x.size();
```


queue的公共函数

- void push (const T& x); 将对象x添加到队尾
- void pop (); 移除队尾元素
- T& front();返回队首元素（的引用），可修改
- const T& front () const;
  返回队首元素（的引用），不修改
- T& back();返回队尾元素（的引用），可修改
- const T& back () const;
  返回队尾元素（的引用），不修改

```
queue<string> x;
x.push("WHU");
```


【例7.5】创建string型队列，向其添加值并打印出其值

```
queue<string> q; q.push("First");
q.push("Second"); q.push("Third");
queue<string> q1 = q;
cout<<"Queue q, "<<"  \tCount:   "<<q.size();
cout << "\n q1=q, \tValues:  ";
int cnt = q1.size();
for (int i = 0; i < cnt; i++) {
  cout << q1.front() << "\t"; q1.pop(); }
cout << endl;
```

程序运行结果

```
Queue q, Count:  3
q1 = q,  Values: First  Second  Third
```

输出序列的顺序与入队的顺序相同，这是队列的先进先出（FIFO）特性的体现。

7.4 队列的存储结构及实现

- 队列作为一种特殊的线性结构，可以如同栈和线性表一样，采用顺序存储结构和链式存储结构实现。顺序存储结构的队列称为顺序队列（Sequenced Queue），链式存储结构的队列称为链式队列（Linked Queue）。

7.4.1 队列的顺序存储结构及操作实现

7.4.2 队列的链式存储结构及操作实现

7.4.3 队列的应用举例

7.4.1 队列的顺序存储结构及操作实现

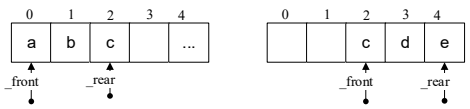
```
class QueueSP {
private:
  int _capacity;
  int _front, _rear;
  unique_ptr<T[]> _items;
public:  ……  }
```

- 顺序队列用一组连续的存储空间存放队列的元素。成员变量_items用数组存储加入队列的元素，成员变量_front和_rear分别作为队头元素下标和队尾元素下标，构成队头指针和队尾指针。
- 元素入队或出队时，需要相应修改_front或_rear变量的值：一个元素入队时_rear自加1，而一个元素出队时_front自加1。

顺序“循环”队列的操作

- 队列的初始化：QueueSP构造方法
- 返回队列的元素个数：Count
- 判断队列的空与满状态：Empty/Full
- 入队：Enqueue
- 出队：Dequeue
- 获得队首对象，但不将其移除：Peek

顺序队列的“假溢出”问题（图需修改）



(a) 队列的顺序存储结构　　(b) 顺序队列的“假溢出”

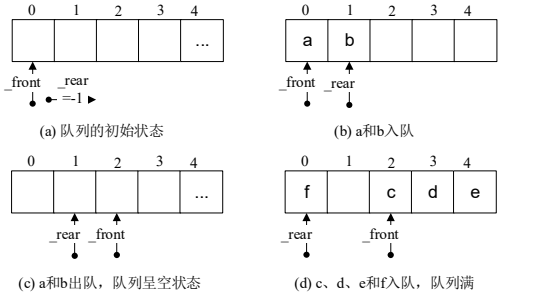
- 设先有(a，b，c)依次入队，那么_front=0，_rear=2。接着a和b出队，d和e入队，_front=2，+rear=4。如果再有新数据入队，应存放于_rear+1=5位置处，数组下标溢出。
- 顺序队列因多次入队和出队操作后出现的有存储空间但不能进行入队操作的溢出现象称为**假溢出**。
- 假溢出缺陷是因为顺序队列没有重复使用存储单元的机制。解决办法是将顺序队列设计成“环形”结构。

顺序“循环”队列

取模运算

```
_front = (_front + 1) % _capacity;
_rear =  (_rear + 1) % _capacity;
```

出队时　入队时



1）队列的初始化

- 构造方法初始化一个队列对象，它为items数组变量申请指定大小的存储空间，以存放加入队列的数据元素，设置队列初始状态为空。

```
QueueSP(int n=DefaultCapacity) {
  cout<<"constructing Queue"<<endl;
  _capacity = n+1;预留一个单元以防假溢出
  _items= make_unique<T[]>(_capacity);
  _front = 0; _rear = -1; }
```

使用

```
SequencedQueue<int> sq1;
SequencedQueue<int> sq2(128);
```


copy constructor

```
QueueSP(const QueueSP& otherq) {
  //cout<<"copy-constructing" << endl;
  _capacity = otherq._capacity;
  _items = make_unique<T[]>(_capacity);
  _front=otherq._front;_rear= otherq._rear;
  int cnt= (_rear-_front+1+_capacity)% _capacity;
  for(int i=0, j; i < cnt; i++) {
    j = (_front + i) % _capacity;
    _items[i] = otherq._items[j];
  }
}
```

2）返回队列中元素的个数

```
// return number of elements in queue
int count() const {
  return (_rear-_front+1+_capacity)%_capacity;}
int size()  const {
  return (_rear-_front+1+_capacity)%_capacity; }
```

(rear-front-1 +_capacity) % _capacity

3）判断队列的空与满状态

- 队列为空， empty应指示true，否则指示false：当 _front== (_rear+1)%_capacity时，队列中没有数据元素。

```
bool empty() const {
  return _front==(_rear + 1)%_capacity;}
```

- 队列已满，full应指示true：当(_rear+2)% _capacity ==_front 时，表明此时_items数组中仍有一个空位置。

```
bool full()  const {
  return _front==(_rear+2)%_capacity;}
```


4）入队

- 过程：当队列不满时，_rear循环加1，将参数*k*表示的元素存放在_rear位置，作为新的队尾数据元素。
- 当队列当前分配的存储空间已装满数据元素，在进行后续的操作前，需要重新分配存储空间，将原数组中的数据元素逐个拷贝到新数组，并相应调整队首与队尾指针。increCapacity
- 参数*k*（即入队的数据元素）声明为T类型，在调用该操作时，实参的类型要与队列定义时声明的类型保持一致。

```
void enqueue(const T& k) {
  if (full())increCapacity();
  _rear = (_rear + 1) % _capacity;
  _items[_rear] = k;}
```

```
QueueSP<int> q;
q.enqueue(14); q.enqueue(3);
```


重新分配存储空间

```
void increCapacity(int amount=DefaultCapacity) {
  int cnt=(_rear-_front+1+_capacity)%_capacity;
  int newcapa= _capacity+amount;
  auto newspace= make_unique<T[]>(newcapa);
  for (int i=0, j; i<cnt; i++) {
     j = (_front + i) % _capacity;
     newspace[i] = _items[j];  }
  _items.reset();_front=0;_rear=cnt-1;
  _capacity = newcapa;
  _items = move(newspace);}
```


Enqueue操作的时间复杂度

- 如果为队列预分配的空间大小合理，队列处于非满状态，入队操作的时间复杂度为 O(1)。
- 如果经常需要重新分配内部数组以容纳新元素，则此操作成为时间复杂度 O(n) 级的操作。

5）出队

- 过程：当队列不空时，_front循环加1，指向新的队首元素。 void，不需返回值的过程。C++中，不易安全高效地实现既返回对象值又可能包含（隐式）销毁它的过程，而是将这种功能留给另一成员函数front()。
- 运算复杂度是 O(1)。

```
void dequeue() {
  if (empty())throw underflow_error("empty");
  _front = (_front + 1) % _capacity;
}
```

6）获得队首对象

- 获得但不移除队首对象。当队列不为空时，取走_front位置上的队首数据，_front不变。

```
const T& front() const{
  if (empty()) throw underflow_error("empty: ");
  return _items[_front];
}
T& front();
```


7）输出队列中所有数据元素的值

- 当队列非空时，从队首结点开始，直至队尾结点，依次输出结点值。

```
void show(bool showTypeName=false)const{
  if (showTypeName)cout<<"Queue: ";
  if (!empty()) {
    int cnt=(_rear-_front+1+_capacity)%_capacity;
    for (int i=0, j; i<cnt; i++) {
      j = (_front + i) % _capacity;
      cout << _items[j] << " -> ";  }
  } cout << "|." << endl;
}
```

string str(bool showTypeName=false);

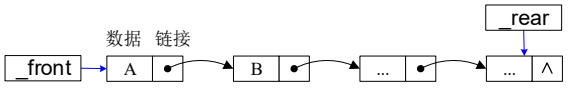
顺序"循环"队列在设计上的改进小结

- 入队时只改变下标_rear，出队时只改变下标_front，它们都做循环移动，取值范围是0 ~ _capacity-1，这使得存储单元可以重复使用，避免"假溢出"现象。
- 在队列中设立一个空位置。如果不设立一个空位置，则队列满的条件也是_front == _rear，那么就无法与队列空的条件相区别。而设立一个空位置，则队列满的条件是
  (_front - _rear+ _capacity) % _capacity == 2
  队列空的条件是
  (_front - _rear+ _capacity) % _capacity == 1

7.4.2 队列的链式存储结构及操作实现

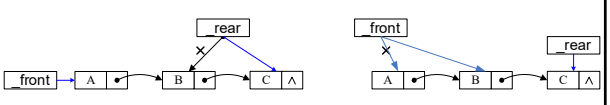
```
class LinkedQueue {
private:
     SLNode<T> *_front, *_rear;
     int _count;
public: …… }
```

- 成员变量_front和_rear分别指向队头和队尾结点，结点类型为单链表结点结构SLNode<T>。
- LinkedQueue类的实例即为具体的队列。



链式队列队列的操作

- 队列的初始化：构造方法
- 返回队列的元素个数：Count
- 判断队列的空与满状态：Empty/Full
- 入队：Enqueue
- 出队：Dequeue
- 获得队首对象：Peek



(a) 元素C入队　　(b) 元素A出队

1）队列的初始化

- 构造函数创建一条单向链表准备用以存储队列数据，设置初始状态为空 。

```
LinkedQueue() {
  _front = nullptr;
  _rear = nullptr;
  _count = 0;
}
```

copy constructor

```
LinkedQueue(const LinkedQueue& s) {
  _count = s._count;   _front = nullptr;
  int n = 0;  SLNode<T> *t,*q;
  SLNode<T>* p = s._front;
  while (n < _count) {
    t = new SLNode<T>(p->item);
    if(n==0){_front=t; q=_front;}
    q->next=t; q = t;
    p = p->next; n++; }
  _rear = q; }
```


析构函数

```
//deconstructor
~LinkedQueue() {
  if(_front!=nullptr)
    ::dispose(_front); }
```

```
void clear() {
  if (_front != nullptr)
    ::dispose(_front);
  _front = nullptr;
  _rear = nullptr; _count = 0; }
```


2）返回队列中元素的个数

```
int count() const {
  return _count;
};
int size() const {
  return _count;
};
```


3）判断队列的空与满状态

- 当_front==nullptr且_count==0时，队列为空：empty()应指示true，其他情况指示false。
- 不需要判断队列是否已满。

```
bool empty() const {
  return (_count==0)&&(_front==nullptr);
}
```


4）入队

- 过程：在_rear指向的队尾结点之后插入一个结点存放k，并更新_rear指向新的队尾。入队操作的运算复杂度是O(1)。

```
void enqueue(const T& k) {
  SLNode<T>* t = new SLNode<T>(k);
  if (_count==0){ _front=t; _rear=t;}
  else if(_count==1)_front->next=t;
  else _rear->next = t;
  _rear = t;  _count++;
}
```


5）出队

- 过程：当队列不为空时，队首的后继元素成为新的队首，销毁原队首元素，_count自减1。此方法的运算复杂度是O(1)。

```
void dequeue() {
  if (!empty()){ //队列不空时
    SLNode<T>* t = _front;
    _front = _front->next;
    if(_front==nullptr) _rear = nullptr;
    _count--; delete t; return; }
  else throw underflow_error("empty");
}
```

6）获得队首对象

- 获得但不移除队首对象。当队列不为空时，取走_front位置上的队首数据元素，_front不变。

```
const T& front() const {
  if(!empty())return _front->item;
  else {//队列空时产生异常
    throw underflow_error("empty"); }
}
T& front();
```


应用中首先关注数据结构的抽象功能

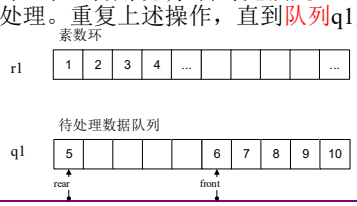
- 由以上多个操作的算法实现分析可知，顺序队列QueueSP和链式队列LinkedQueue，都实现了“队列”这个抽象数据结构的基本操作。无论是QueueSP类还是LinkedQueue类，都可以用来建立具体的队列实例，通过实例调用入队或出队等方法进行相应的操作。
- 一般情况下，解决某个问题关注的是队列的抽象功能，而不必关注队列的存储结构及其实现细节。

7.4.3 队列的应用举例

- 队列是一种具有“先进先出”FIFO特性的特殊线性结构，可以作为求解具有“先进先出”特性问题的数学模型，因此队列结构成为解决相应问题算法设计的有力工具。
- 在计算机系统中，某些过程需要按一定次序等待特定资源就绪，系统需设立一个具有“先进先出”特性的队列以解决这些过程的调度问题：处理“等待服务”问题时常使用队列。
- 实现广度遍历算法时使用队列。
- 解素数环问题。将n个数排列成环形，使得每相邻两数之和为素数，构成一个素数环。

[例7.7] 解素数环问题 试探法

- 创建一个线性表对象rl存放素数环的数据元素，创建一个队列对象q1，存放待检测的数据元素。方法IsPrime(k) 判断*k*是否为素数。素数环初始只有“1”。
- 首先将2~*n*的数全部入队q1；将出队数据*k*与素数环最后一个数据元素相加，若两数之和是素数，则将*k*加入到素数环rl中，否则说明*k*暂时无法处理，必须再次入队等待处理。重复上述操作，直到队列q1为空。




```
LinkedQueue<int> q1;// QueueSP<int> q1;
SequencedList<int> ring1(n);
ring1.push_back(1);//1添加到素数环中
for (i = 2; i <= n; i++) q1.enqueue(i); i = 0;
while (!q1.empty()) {
  k = q1.front(); q1.dequeue();
  cout << "Queue front: " << k << "\t";
  j = ring1[i] + k;
  if (IsPrime(j)){//判断j是否为素数
    ring1.push_back(k); //k添加到素数环中
    cout<< "add into ring\t";  i++;}
  else{q1.enqueue(k);cout<<"into queue again\t";}
  q1.show(true);}
cout<<endl; cout<<ring1.str(true)<<endl;
```

本章学习要点

1. 掌握栈和队列类型的特点，并能在相应的应用问题中正确选用它们。
2. 熟练掌握栈类型的两种实现方法，特别应注意栈满和栈空的条件以及它们的描述方法。
3. 熟练掌握顺序循环队列和链式队列的基本操作实现算法，特别注意队满和队空的描述方法。

作业

7.1 填空：线性表、栈和队列都是_____结构，可以在线性表的_____位置插入和删除元素；栈是一种特殊的线性表，允许插入和删除操作的一端称为_____。不允许插入和删除运算的一端称为_____，所以栈又称为__进__出型线性表。队列是只能在_____插入和_____删除元素的特殊线性表，所以队列又称为____进出型结构。

7.3 说明顺序队列的“假溢出”是怎样产生的，并说明如何用循环队列解决。循环队列的基本操作，如初始化，判断队列满、判断队列空、返回队列元素个数、入队、出队等是如何实现的？

作业（II）

7.4分别在StackSP、QueueSP、LinkedStack和LinkedQueue类中编程实现检测数据结构中是否包含某数据的操作：`bool contains(const T& k);`

7.7 说明以下算法的功能（Stack和Queue分别是C#类库中的栈类和队列类）。。

```
void funcA(Queue q){
    Stack s = new Stack();  object d;
    while(q.Count!=0){
        d = q.Dequeue( ); s.Push(d); };
    while(s.Count!=0){
        d = s.Pop( );   q.Enqueue(d);}
}
```


实习

- 实验目的
  理解栈的基本概念及其基本操作。
- 题意
  将中缀表达式转换为后缀表达式，再求后缀表达式的值。