STL中序列式容器剖析

对序列式容器的解释：容器中每个元素都有固定位置其取决于插入的时机和地点，与元素值无关

内容简要

- 向量(vector)
- 链表(list)
- 双端队列(deque)
- 队列(queue)
- 栈（stack）
- 每部分介绍包括：概述，数据结构的定义，关键基本操作具体实现的概述(1-2),应用(求解具体问题)

向量(vector)

- 概述
- vector的定义
- 关键基本操作具体实现的概述
- 应用(求解具体问题)

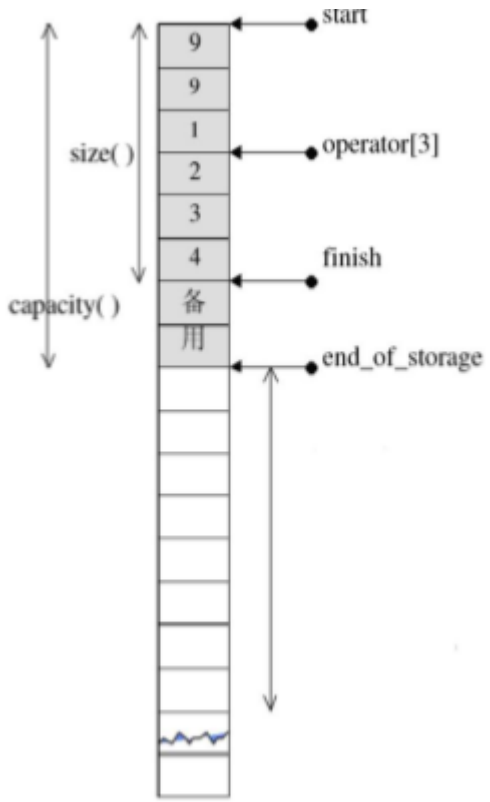
概述

Vector是一个能任意类型的动态数组，是基于类模板的动态数组，其内部定义了很多基本操作。

代码实现为一块连续的内存，可以理解为是动态扩张的array，即长度根据需要而自动改变的数组，其具有随机访问的特性，但不适合频繁的随机插入和删除

Vector的数据结构(定义)

- 采用线性连续空间
- 使用三个迭代器指示分配的空间及其使用情况（示意图如右）
  Start：指向目前使用空间的头
  Finish：指向目前使用空间的尾部
  End of storage: 指向可用空间的尾部(含备用空间)



关键基本操作具体实现的基本概述

Push_back操作

```
void
push_back(bool __x)
{
  if (this->_M_impl._M_finish._M_p != this->_M_impl._M_end_of_storage)
  //判断是否可以继续往容器中放入元素
    *this->_M_impl._M_finish++ = __x;
  else
  //不能则在尾部插入备用空间
    _M_insert_aux(end(), __x);
}
```

对是否能插入首先要进行一个判断

Pop_back操作

```
void
pop_back()
{ --this->_M_impl._M_finish; }
```

弹出操作主要为指针的移动，将指针进行移动

关键基本操作具体实现的基本概述

Size()操作

```
size_type
size() const _GLIBCXX_NOEXCEPT
{ return size_type(end() - begin()); }
```

说明：利用迭代器首尾位置之差来求vector容器的大小，即end()-begin()

Resize()操作

```
void resize(size_type _Newsize, _Ty _Val)
{   // determine new length, padding with _Val elements as needed
    if (size() < _Newsize)
        _Insert_n(end(), _Newsize - size(), _Val);
    else if (_Newsize < size())
        erase(begin() + _Newsize, end());
}
```

说明：比较现在大小和新的大小的关系，如果不够就扩充相应的大小，多余就删除多余的空间

应用(相关操作举例和求解具体问题)

相关操作举例说明

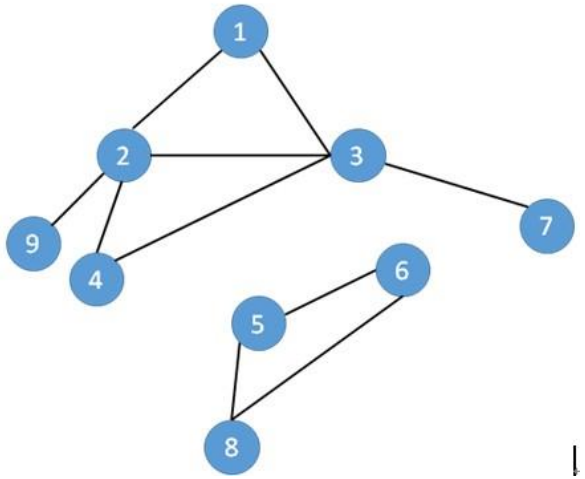
头文件 #include <vector>
定义变量 vector<int> v;
主要成员函数的调用说明
v.clear()//移除容器中所有数据
v.push_back（E）//在尾部插入一个数据E
v.pop_back()//删除最后一个元素
v.size()//返回容器中实际数据的个数
v.erase(pos)//删除pos处的元素，并返回下一个元素的值
v.insert(pos,cnt,E)//在pos 处插入cnt个E
v.begin() //返回的指针指向数组中第一个元素
v.end()//返回指向数组最后一个元素下一位的位置
v.empty()//判断容器是否为空

应用(相关操作举例和求解具体问题)

- 存储数据：在数组元素个数不确定的情况下，可以用数组使用，能更加有效的节省空间
- 用作邻接表存储图，解决图的相关问题

图的存储---邻接表

每个节点V对应一个一维数组(vector)，里面存放从V连出 去的边，边的信息包括另一顶点，还可能包含边权值等。具体如右所示，将右边的图存储在一个vector<int> G[9]的二位动态数组中



1	2	3		
2	1	4	9	3
3	1	4	7	2
4	2	3		
5	6	8		
6	5	8		
7	3			
8	5	6		
9	2			

遍历复杂度：O(n+e) n为节点数目，e为边数目

链表(list)

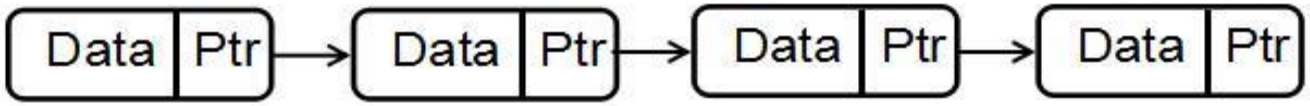
- 概述
- list的定义
- 关键基本操作具体实现的概述
- 应用(求解具体问题)

概述

List是一种双线性列表，只能顺序访问(从前向后或者从后向前)

实现上 链表在物理单元存储上非连续，采用动态分配内存，， 能够有效分配内存和利用内存资源。结构上，是一条相互链接的数据节点表，每个节点由数据和指向下一结点的指针构成，数据的组织形式如图所示

数据域

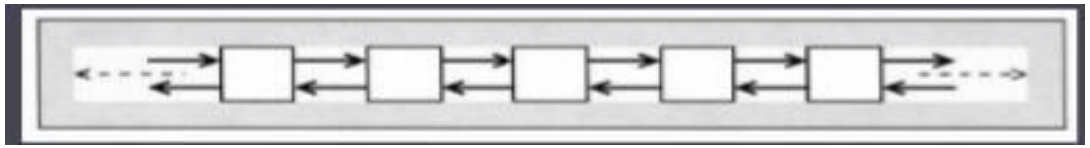


指针域

list的数据结构(定义)

1

链表是一种双线性表，列表中的元素构成一个线性逻辑次序，元素中物理地址可以任意。作为一种典型的动态存储结构，其中的数据分散为一系列的节点单位，节点之间通过指针相互索引和访问。



2

由于链表本身的结构特点，在任何位置上插入或者删除动作都非常迅速，内部只需调整一下指针的指向关系即可

关键步骤的实现和操作

```c
//思路：初始链表中的节点个数为0，故只需要一个头指针指向一个NULL，则表明该链表为空链表
void LinkListInit(LinkNode** phead)
{
 //判断非法输入
 if(phead==NULL)
 return;
 *phead=NULL;
}
```

```c
//思路：第一个参数接收链表的头指针，第二个参数接收字符串的地址，用于打印时显示关于打印的具体信息
void LinkListPrintChar(LinkNode* head,const char* msg)
{
 printf("[%s]\n",msg);
 LinkNode* cur=head;
 for(cur=head;cur!=NULL;cur=cur->next)
 {
    printf("[%p]|[%c] ",cur,cur->data);
 }
 printf("\n");
}
```

说明

- 链表的初始化
- 链表的打印操作实现，依次输出链表中存储的数据的值

关键步骤的实现和操作

在链表尾部插入一个元素值的操作实现

```
void LinkListPushBack(LinkNode** phead,LinkListType value)
{
 //判断非法输入
 if(phead==NULL)
 return;
 //链表为空时，是通过创建新节点并修改头指针的指向
 if(*phead==NULL)
 {
 LinkNode* new_node=CreateNode(value);
 *phead=new_node;
 return;
 }
 //链表非空时，通过创建新节点并遍历找到最后一个节点，将其的next进行修改
 LinkNode* cur=*phead;
 while(cur->next!=NULL)
{
 cur=cur->next;
 }
 cur->next=CreateNode(value);
}
```

新建一个结点的操作实现

```
LinkNode* CreateNode(LinkListType value)
{
 LinkNode* new_node=(LinkNode*)malloc(sizeof(LinkNode));
 new_node->data=value;
 new_node->next=NULL;
 return new_node;

}
```

应用(相关操作举例和求解具体问题)

相关操作举例说明

头文件 #include <list>
List<T,ALLoc>
支持操作
begin(),end(),size(),clear(),empty()
push_back(),pop_back()
push_front(),pop_front()
Insert() 时间复杂度 O(1)
Erase() 时间复杂度O(1)
Sort（）列表的排序，时间复杂度 O(nlogn),不同于algorithm中的sort()
Merge() :合并两个排序列表

应用(相关操作举例和求解具体问题)

求解具体问题

链表作为一种线性存储结构，在结构上便于插入和查找元素。
基于链表，我们可以实现一些特殊的数据结构，比如队列，双端队列，栈，实现一些基本算法，图的遍历，树的遍历，大数加法运算器等等。

大数加法运算器

问题描述:
编写一个基于动态链表的“大整数加法运算器”，来实现任意长度的的两个整数的加法运算。

应用(相关操作举例和求解具体问题)

问题分析:

- 由于加数很大超出了数据的存储范围，我们可以用单链表的形式来存储一个大整数。例如：对于整数135，可以创建一个单链表，该链表包含三个节点，分别用来存储1、3、5这三个数字。考虑到输入整数的长度是任意的（不超过80位），因此为了减少内存空间的浪费，在程序中必须采用动态链表的方法，即每一个链表节点都是根据需要动态创建的
- 可以编写函数：创建链表（CreateLList）、加法函数（AddLList）、打印链表（DisplayLList）

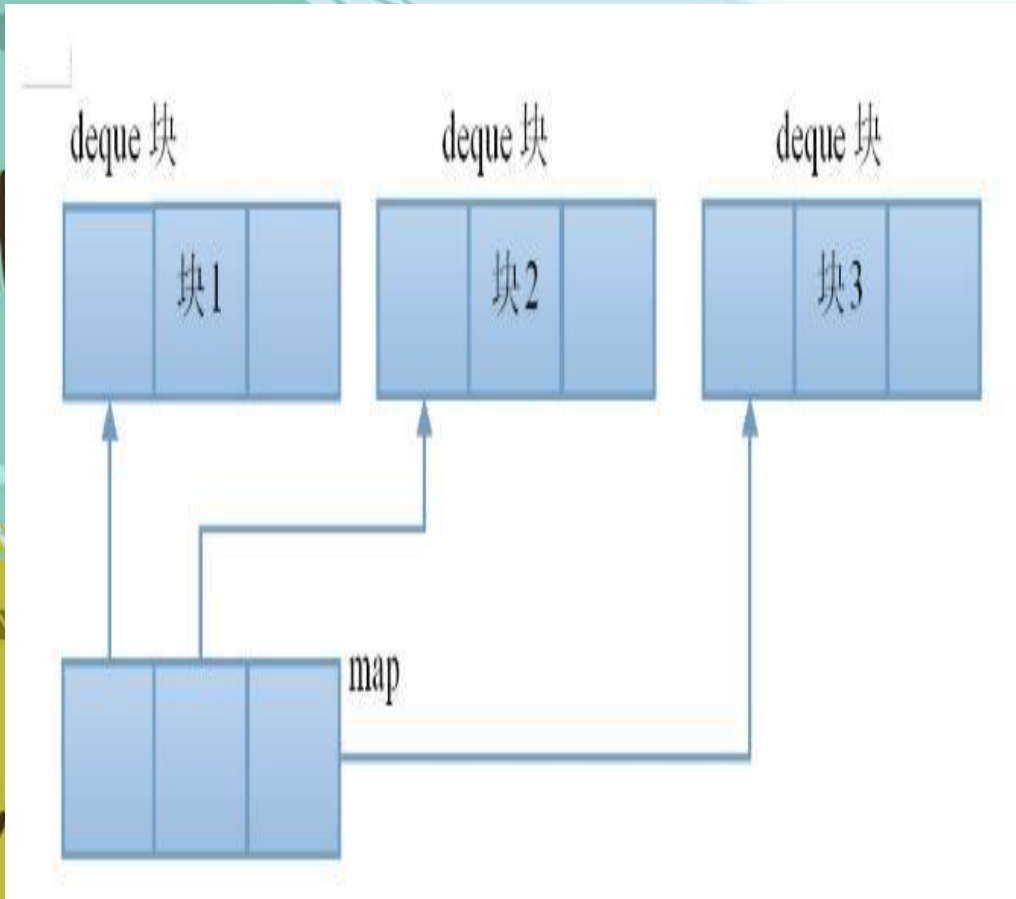
关键运算分析:

- 每次将运算结果存储到一个新建链表的结点中
- 要处理好进位，两个加数的长度大小关系。

简单实现如下:

```
while(1)                                //当p1,p2都不指向NULL时，进入循环
{
    p03=new Node;                       //为链表3开辟空间
    sum=p01->date+p02->date+x;
    if(sum>=10)                         //要进位
    {
        p03->date=sum-10;
        x=1;                            //进位标志为1
    }
    else if(sum>=0)
    {
        p03->date=sum;
        x=0;                            //进位标志为0
    }
    p01=p01->next;p02=p02->next;        //指向下一节点
    p3->next=p03;                       //这两步是连接链表3的节点
    p3=p03;
    if(p01==NULL||p02==NULL) break;     //当一数加完后退出循环
}
```

双端队列(deque)



- 概述
- deque的定义
- 关键基本操作具体实现的概述
- 应用(求解具体问题)

概述

deque（双端队列）是一种具有队列和栈的性质的数据结构。双端队列的元素可以从两端弹出，其限定插入和删除操作在表的两端进行

一般来说，当考虑到容器元素的内存分配策略和操作的性能时deque相当于vector更有优势。

deque的数据结构(定义)

- deque双向队列是一种双向开口的连续线性空间，可以高效的在头尾两端插入和删除元素，deque在接口上和vector非常相似）
- deque块在头部和尾部都可以插入和删除。而不需要移动任何元素deque需要一个指向map的指针，还有两个指针，start和finish指针，分别指向deque的第一个节点和最后一个节点的下一个位置。
- Deque可以随机存取数据(用索引直接存取），在数据头部和尾部添加和移除元素都非常迅速，但是中间安插元素比较费时

关键基本操作具体实现的基本概述

Push_back操作

```
void push_back(const value_type& t)
{
    if (finish.cur != finish.last - 1)
    {//尾部还有多余空间，一个以上的空间
        construct(finish.cur, t);//直接构造
        ++finish.cur;//调整缓冲区状态finish的cur+1
    }
    else
        push_back_aux(t);//没有或者只剩下一个，添加node，然后构造
}
```

Push_front操作

```
void push_front(const value_type& t)
{
    if (start.cur != start.first) {
        construct(start.cur - 1, t);
        --start.cur;
    }
    else
        push_front_aux(t);//没有或者只剩下一个，添加node，然后构造
}
```

应用(相关操作举例和求解具体问题)

相关操作举例说明

头文件 #include <deque>
定义变量 deque<int> dq;
主要成员函数
dq.clear()//移除容器中所有数据
dq.push_front(elem)//在队列首部加入一个数据
dq.pop_back()//删除队列尾部元素
dq.push_back(elem) //在队列尾部加入一个数据
dq.pop_back() //删除队列尾部元素
dq.empty()//判断队列是否为空，为空返回true
dq.size()//返回容器中实际数据的个数
dq.erase(pos)//删除pos位置处的数据，返回下一个数据的位置
dq.insert(pos,cnt,elem)//在pos处插入cnt个数据elem
dq.begin()//返回的指针指向数组中的第一个数据
dq.end()//实际上是取末尾加1，以便让循环正常运行

应用(相关操作举例和求解具体问题)

描述

定义一个双端队列，进队操作与普通队列一样，从队尾进入。出队操作既可以从队头，也可以从队尾。编程实现这个数据结构。

输入

第一行输入一个整数t，代表测试数据的组数。

每组数据的第一行输入一个整数n，表示操作的次数。

接着输入n行，每行对应一个操作，首先输入一个整数type。

当type=1，进队操作，接着输入一个整数x，表示进入队列的元素。

当type=2，出队操作，接着输入一个整数c，c=0代表从队头出队，c=1代表从队尾出队。

n <= 1000

输出

对于每组测试数据，输出执行完所有的操作后队列中剩余的元素,元素之间用空格隔开，按队头到队尾的顺序输出，占一行。如果队列中已经没有任何的元素，输出NULL。

问题求解，可以利用数组来模拟双端队列的实现，再根据输入与操作运算符的关系进行入队，出队

```
if(type==1)
{
        a[co++]=val;
        top2++;//与数组下标保持一致
}
else
{
        if(val)//从队尾出
        {
                a[top2]=-1;
                top2--;
                co--;//注意元素个数要跟着递减
        }
        else //从队首出
        {
                a[top1]=-1;//标记出队
                top1++;
        }
}
```

队列(queue)

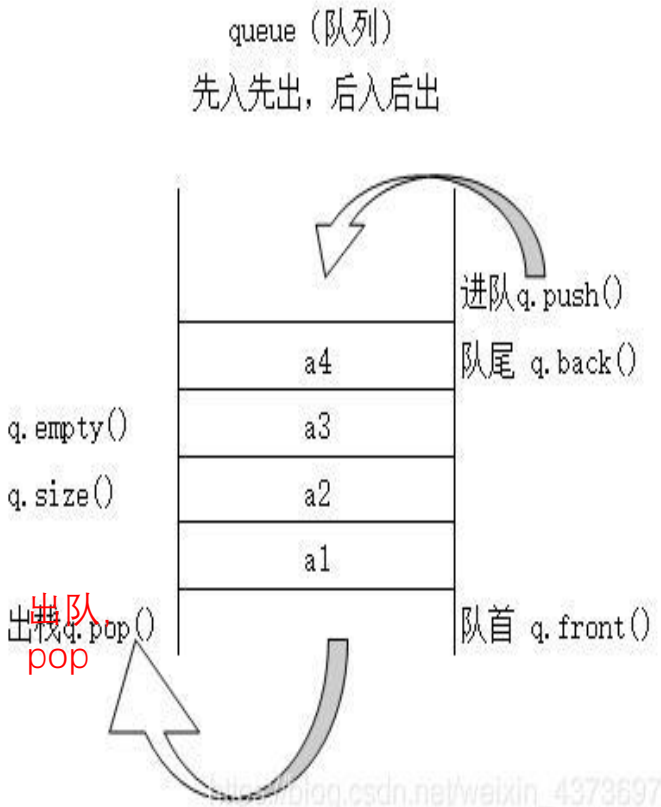
- 概述
- queue的定义
- 关键基本操作具体实现的概述
- 应用(求解具体问题)

概述

Queue是一种先进先出的数据结构，(FIFO),举个简单的例子，我们可以类比类比食堂中排队打饭，每个人都要排到队伍的最后面，而排在队伍最前的人(即先入队的人），则打饭出队。

- 实现上 队列实际上是一种有限制的线性表，只能在队的前端进行删除。
- 在队的后端进行插入.队列可以看做是容器的容器，内部使用其他容器来存放具体数据，加上一个外壳，使得我们对数据的操作只能是在头或者是在尾。其内部默认数据存放容器是deque

queue的数据结构(定义)



1

一般来说队列的实现需要一个队首指针front来指向队首元素的前一个位置，使用一个队尾指针rear来指向队尾元素。队列的实现基于线性表的实现。

2

STL对queue队列的泛化，是通过模板类型，将默认的deque双端队列类型导入，在内部创建一个序列容器对象，来处理 queue队列的数据存储和操作，包括queue队列是否为空、取队首元素、取队尾元素、元素入队和元素出队等

关键步骤的实现和操作

```
template <class Type> void Queue<Type>::push(const Type &val)
{
    // allocate a new QueueItem object
    QueueItem<Type> *pt = new QueueItem<Type>(val);
    // put item onto existing(目前) queue
    if(empty())
        head = tail = pt;    // the queue now has only one element
    else {
        tail->next = pt;     // add new element to end of the queue
        tail = pt;
    }
}
```

```
template <class Type> void Queue<Type>::pop()
{
    // pop is unchecked: Popping off an empty Queue is undefined
    QueueItem<Type>* p = head;
    head = head->next;
    delete p;
}
```

说明

- Push函数首先判断队列是否为空，然后确定插入元素方式，具体类模板实现如右
- Pop函数，通过中间变量，改变队列首部指针，使之指向下一个元素

关键步骤的实现和操作

取队首元素front

```
Type& front() {return head->item; }
```

判空操作empty

```
bool empty()const{
//true if no elements in the Queue
        return head == 0;
}
```

应用(相关操作举例和求解具体问题)

相关操作举例说明

头文件#include <queue>
常用函数
Queue<int> Q 声明一个int型的空队列
Push() 将一个新元素接到队列的末端
Pop()弹出队列中的第一个元素，返回的是一个void
Front()\back()存取队列中的第一个元素和最后一个元素，返回的是一个引用
Empty() 判断队列是否为空
Size () 获得队列中元素的个数
队列中元素的访问只能通过front或者back来访问

应用(相关操作举例和求解具体问题)

- **问题描述**

一个迷宫由R行C列格子组成，有的格子里有障碍物，不能走；有的格子是空地，可以走。
给定一个迷宫，求从左上角走到右下角最少需要走多少步(数据保证一定能走到)。只能在水平方向或垂直方向走，不能斜着走。

- **输入**
  - 第一行是两个整数， R和C，代表迷宫的长和宽。
  - （ 1<= R，C <= 40)
  - 接下来是R行，每行C个字符，代表整个迷宫。
  - 空地格子用'.'表示，有障碍物的格子用'#'表示。
  - 迷宫左上角和右下角都是'.'。
- **输出**
- 输出从左上角走到右下角至少要经过多少步，计算时包括起点和终点

□ 实现广度优先搜索，解决迷宫问题

算法伪代码(主要部分)

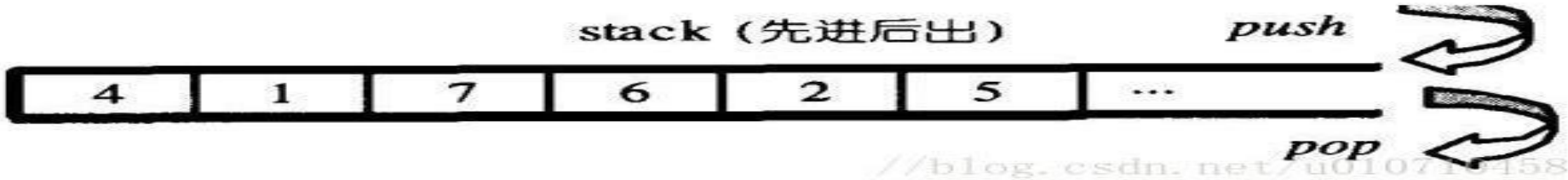
```
//d[]为改变方向的数组
While(队列不为空)
{
        Pos t=q.front()//Pos为结构体
    if(到达终点)
    {cout<<step;}
     else{
        int r=t.r,c=t.c,step=t.step+1;//步数加1
            for(int i=0;i<4;i++){
                if(没有标记过)
                {q.push(Pos(r+d[i].r,c+d[i].c,step)
                标记数组}
            }
            q.pop();
}
```

栈(stack)

- 概述
- stack的定义
- 关键基本操作具体实现的概述
- 应用(求解具体问题)

概述

栈是一种后进先出的数据结构，它限定在表的一段进行插入和删除操作的线性表，进行插入和删除操作的一端称为栈顶，另外一段称为战栈底。不含元素的栈称为空栈



stack的数据结构(定义)

- 以某种既有容器作为底部结构，将其接口改变，使之符合“先进先出”的特性，形成一个栈stack。deque是双端开口的数据结构，以deque为底结构并封闭其头端开口，形成一个stack。
- 栈也可以使用vector，list中的其中任何一个作为stack的底层模型

关键基本操作具体实现的基本概述

```
void push(int value)
{
  arr[++top]=value;
}
```

Push函数，元素入栈操作

```
Stack<E> pop()
{
  return arr[top--];
}
```

Pop函数，元素出栈

```
Bool isEmpty()
{
 return top ==-1;
}
```

判断栈是否为空

```
Stack<E> peek()
{
  return arr[top];
}
```

查找栈中的元素

应用(相关操作举例和求解具体问题)

相关操作举例说明

头文件 #include <stack>
常用函数
Stack<int> Q 声明一个int的空栈Q
push() 将一个新元素接到栈的末端
Pop() 弹出占中的末端元素，返回的是一个void
Top()存取栈中的最后一个元素，返回的是一个引用
Empty() 判断栈是否为空
Peek()查看栈中元素
Size（）获取栈中的元素个数

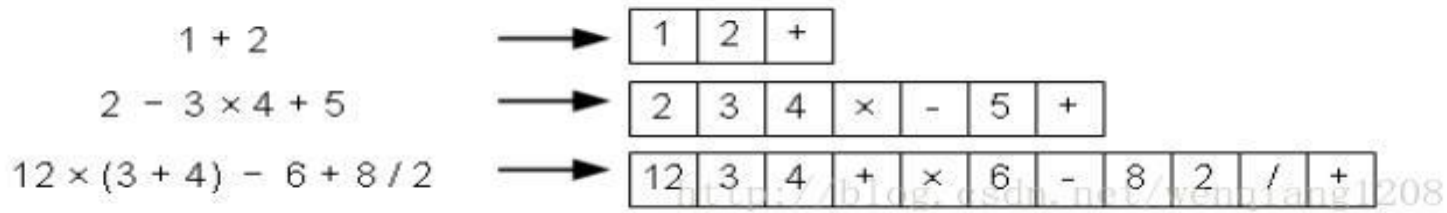
应用(相关操作举例和求解具体问题)

- 模拟实现一些递归，防止程序对栈内存的限制而导致程序运行出错
- 匹配括号串问题，计算逆波兰表达式

逆波兰表达式(计算运算式)

什么是逆波兰表达式?

通常我们计算一个算式：X+Y,(操作数X，操作符+，操作数Y)；逆波兰表达式是(操作数X,操作数Y，操作符+)；



在后缀表达式中看，不存在括号，也不存在运算符优先级的差别，计算过程完全按照运算符出现的先后次序进行，整个计算过程仅需扫描一遍便可完成。

应用(相关操作举例和求解具体问题)

如何用逆波兰表达式(计算运算式)

为了转换正确，必须设定一个运算符栈stack，并在栈底放入一个特殊算符，假定为？，让它具有最低的运算符优先级，此栈用来保存扫描中缀表达式得到的暂不能放入后缀表达式中的运算符，待它的两个运算对象都放入到后缀表达式之后，再令其出栈并写入到后缀表达式中。
转换过程如下:
从头到尾扫描中缀表达式，(1)若遇到数字则直接写入后缀表达式，(2)若遇到运算符，则比较栈顶元素和该运算符的优先级,(2.1)(* 或者/)当该运算符的优先级大于栈顶元素的时候，表明该运算符的后一个运算对象还没有进入后缀表达式，应该把该运算符暂存于运算符栈中，然后把它的后一个运算对象写入到后缀表达式中，再令其出栈并写入后缀表达式中;(比如说*和/) (2.2)(- 或者+)若遇到的运算符优先级小于等于栈顶元素的优先级，表明栈顶运算符的两个运算对象已经被写入后缀表达式，应将栈顶元素出栈并写入后缀表达式，对于新的栈顶元素仍进行比较和处理，直到栈顶元素为#，然后将新元素进栈。

THANKYOU！

参考文献：
【1】 胡凡，曾磊 。算法笔记【M】。北京：机械工业出版社，2016：359-367.
【2】 严蔚敏 吴伟民 编著 。数据结构(C语言版） 【M】。北京：清华大学出版社，1997.4(2004.11重印).
【3】百度百科
【4】CDSN博客