数据

1 综合

1.1 ++

--

2 数据结构

2.1 表

--
存储:顺序 链式

2.2 栈

--
先入后出，后入先出
存储:顺序 链式

2.3 队列

--
先入先出，后入后出
存储:顺序 链式

2.4 二叉树

--
二叉树第i(i≥1)层上的节点最多为2^（i−1）个。
深度为k(k≥1)的二叉树最多有2^k−1个节点
在任意一棵二叉树中，树叶的数目比度数为2的节点的数目多一
满二叉树 :深度为k(k≥1)时有2^k−1个节点的二叉树
完全二叉树 :只有最下面两层有度数小于2的节点，且最下面一层的叶节点集中在最左边的若干位置上
当i＞1(不是根节点)时，有父节点，其编号为i/2
当2*i≤n时，有左孩子，其编号为2*i ,否则没有左孩子，本身是叶节点
当2*i＋1≤n时，有右孩子，其编号为2*i+1 ,否则没有右孩子
当i为奇数且不为1时，有左兄弟，其编号为i−1,否则没有左兄弟
当i为偶数且小于n时，有右兄弟，其编号为i＋1,否则没有右兄弟
存储:顺序 链式
遍历:先序 中序 后序 分层
节点是红色或者黑色

根节点是黑色
每个叶子的节点都是黑色的空节点（NULL）
每个红色节点的两个子节点都是黑色的。
从任意节点到其每个叶子的所有路径都包含相同的黑色节点。

2.5 图

——
有向图和无向图 稀疏图和稠密图 完全图和连通图
弧头与弧尾 路径与回路 边 权 网 子图
度:顶点相关联的边的条数 无向图:度 有向图:入度、出度
存储:顺序 一维数组 二维数组 链式 邻接表 逆邻接表 十字链表
遍历:深度优先DFS 广度优先BFS

3 算法

3.1 查找

顺序查找 折半查找 分块查找 hash查找
——
顺序

```
int SqSearch(int *p, int size, int key)
{
        int i;
        for (i = 0; i < size; i++)
        {
 if (key == p[i])
 {
 return i;
 }
        }
        return -1;
}
```

——
二分

```
int BinSearch(int *p, int size, int key)
{
        int low = 0;
        int high = size - 1;
        int min;
        while (low < high)
        {
 min = (low+high) / 2;
 if (key == p[min])
 {
 return min;
 }
 else if (key < p[min])
```

```
 {
 high = min;
 }
 else if (key > p[min])
 {
 low = min;
 }
        }
}
```

3.2 排序

——

稳定与不稳定排序 内与外排序

插入:直接插入 折半插入 链表插入

交换:冒泡排序 选择排序 快速排序 归并排序 堆排序

——

冒泡

```
void BubbleSort(int *p, int size)
{
        int i, j, t;
        for (i = 0; i < size – 1; i++)
        {
 for (j = 0; j < size –1 – i; j++)
 {
 if (p[j] < p[j+1])
 {
 t = p[j];
 p[j] = p[j+1];
 p[j+1] = t;
 }
 }
        }
}
```

——

选择

```
void BubbleSort(int *p, int size)
{
        int i, j, t;
        for (i = 0; i < size; i++)
        {
 for (j = i + 1; j < size; j++)
 {
 if (p[i] < p[j])
 {
 t = p[i];
 p[i] = p[j];
```

```
 p[j] = t;
 }        }
        }
}
```

——

快速

```
void q_sort(int *p, int start, int end)
{
 int i = start;
 int j = end;
 int x = p[start];
 if(i > j)
 return;
 while(i < j)
 {
 while(i < j && p[j] >= x)
 j--;
 if(p[j] < x)
 {
 p[i] = p[j];
 i++;
 }
 while(i < j && p[i] <= x)
 i++;
 if(p[i] > x)
 {
 p[j] = p[i];
 j--;
 }
 }
 p[i] = x;
 q_sort(p, start, i-1);
 q_sort(p, i+1, end);
}
```

4 STL

4.1 基本

——

容器：各种数据结构，如vector、list、deque、set、map等,用来存放数据，从实现角度来看，STL容器是一种class template。

算法：各种常用的算法，如sort、fifind、copy、for_each。从实现的角度来看，STL算法是一种function tempalte.

迭代器：扮演了容器与算法之间的胶合剂，共有五种类型，从实现角度来看，迭代器是一种将operator* , operator-> , operator++,operator--等指针相关操作予以重载的class template. 所有STL容器都附带有自己专属的迭代器，只有容器的设计者才知道如何遍历自己的元素。原生指针(native pointer)也是一种迭代器。

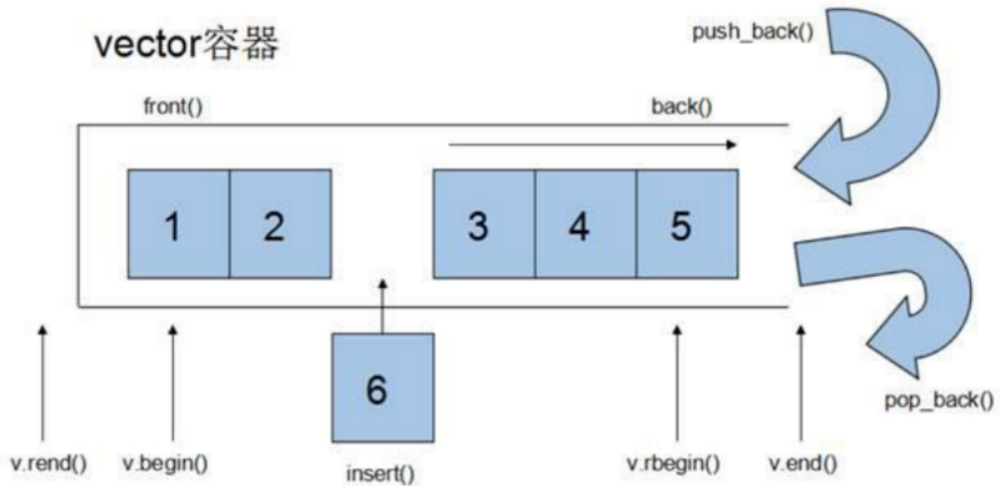
仿函数：行为类似函数，可作为算法的某种策略。从实现角度来看，仿函数是一种重载了operator()的class 或者class template
适配器：一种用来修饰容器或者仿函数或迭代器接口的东西。
空间配置：负责空间的配置与管理。从实现角度看，配置器是一个实现了动态空间配置、空间管理、空间释放的class tempalte.
STL六大组件的交互关系，容器通过空间配置器取得数据存储空间，算法通过迭代器存储容器中的内容，适配器可以修饰仿函数,仿函数可以协助算法完成不同的策略的变化.
谓词是指普通函数或重载的operator()返回值是bool类型的函数对象(仿函数)。如果operator接受一个参数，那么叫做一元谓词,如果接受两个参数，那么叫做二元谓词，谓词可作为一个判断

4.2 string 容器

--
string 封装了 char，管理这个字符串，是一个 char 型的容器

4.3 vector 容器

--
单端数组
array 是静态空间，一旦配置了就不能改变
vector 是动态空间，随着元素的加入，它的内部机制会自动扩充空间以容纳新元素
vector 实际配置的大小可能比客户端需求大一些，
一旦容量等于大小，便是满载，下次再有新增元素，整个 vector 容器就得另觅
所谓动态增加大小，并不是在原空间之后续接新空间(因为无法保证原空
间之后尚有可配置的空间)，而是一块更大的内存空间，然后将原数据拷贝新空间，并释放原空间。因此，对 vector 的任何操作，一旦引起空间的重新配置，指向原 vector 的所有迭代器就都失效了



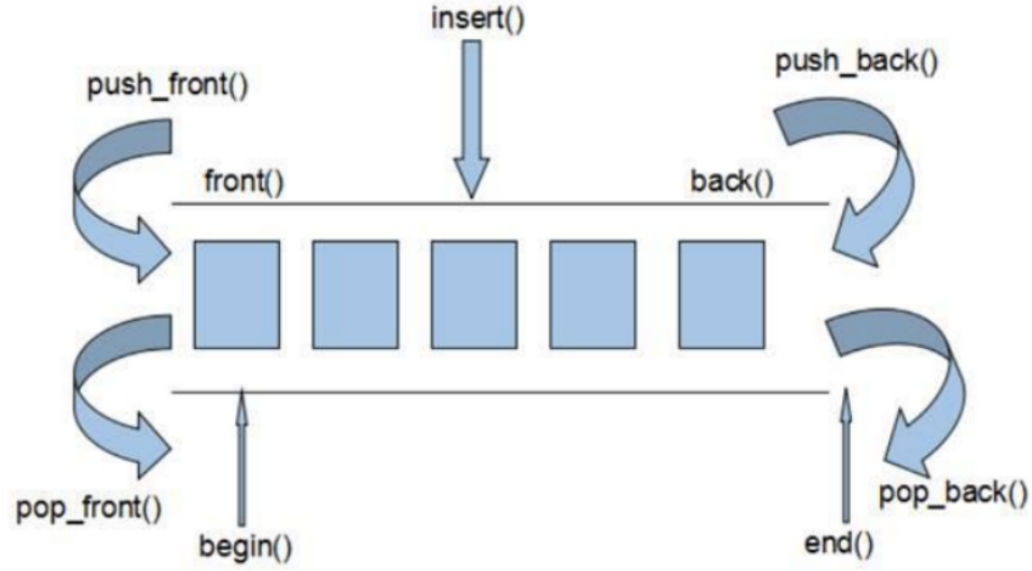
4.4 deque 容器

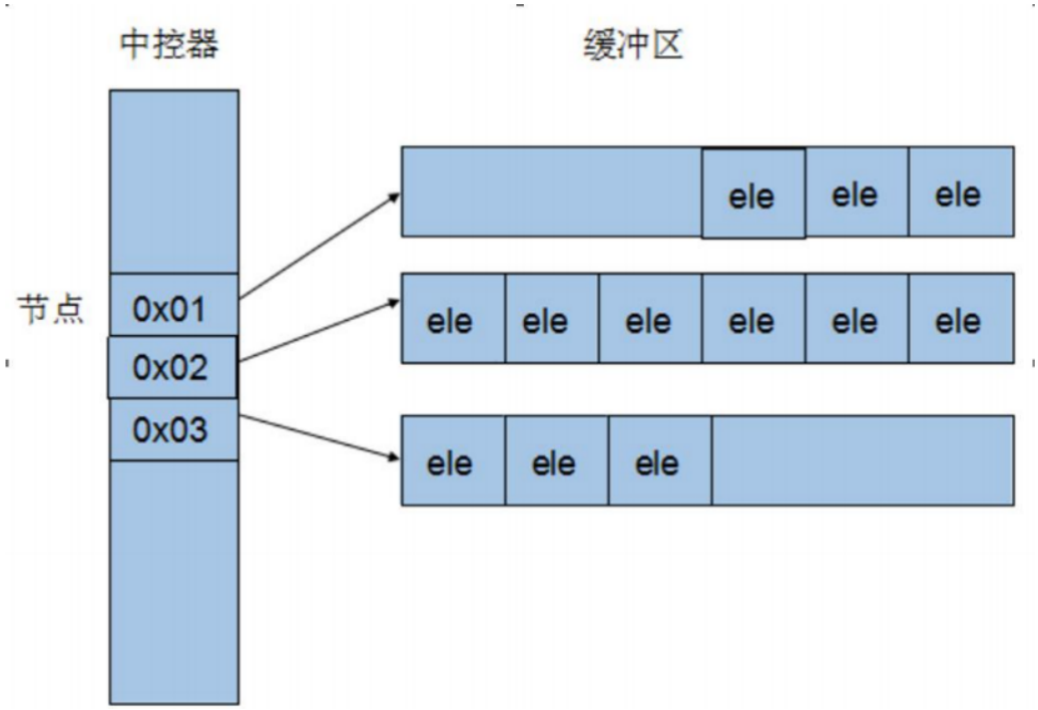
--
双端数组
vector 容器是单向开口的连续内存空间，deque 则是一种双向开口的连续线性空间差异，
一在于 deque 允许使用常数项时间对头端进行元素的插入和删除操作
二在于 deque 没有容量的概念，因为它是动态的以分段连续空间组合而成，随时可以增加一段新的空间并链接起来

因此，deque 没有必须要提供所谓的空间保留(reserve)功能

对 deque 进行的排序操作，为了最高效率，可将 deque 先完整的复制到一个 vector 中，对 vector 容器进行排序，再复制回 deque

deque 采取一块所谓的map(注意，不是 STL 的 map 容器)作为主控，这里所谓的 map 是一小块连续的内存空间，其中每一个元素(此处成为一个结点)都是一个指针，指向另一段连续性内存空间，称作缓冲区。缓冲区才是 deque 的存储空间的主体

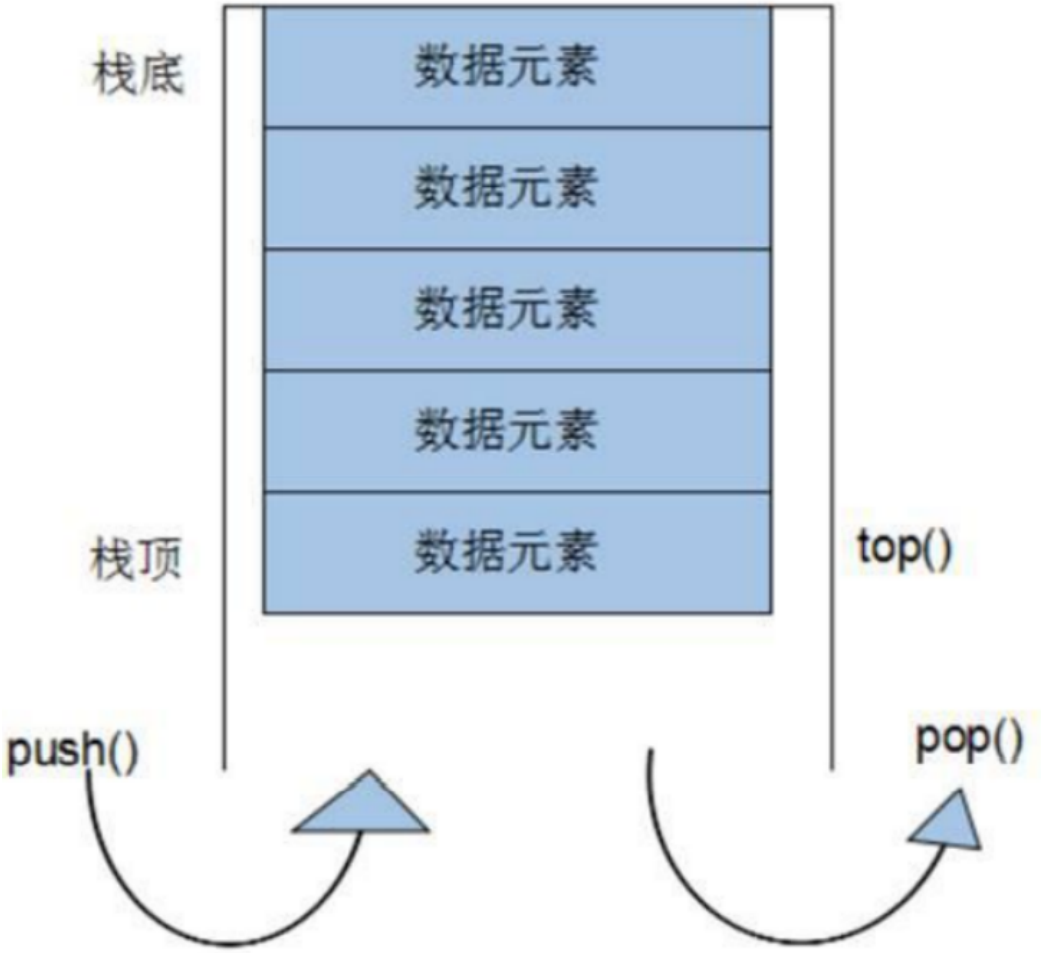




4.4 stack容器

--

堆栈



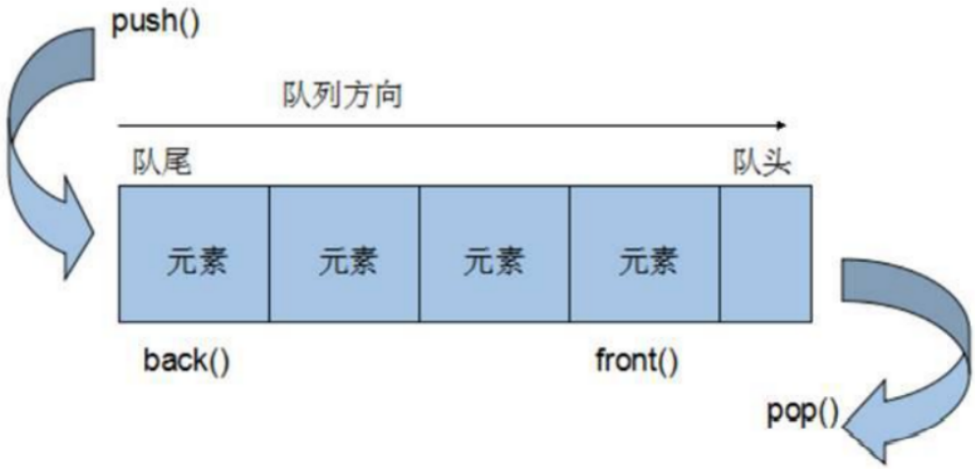
stack 是一种先进后出(First In Last Out,FILO)的数据结构，它只有一个出口，形式如图所示。stack 容器允许新增元素，移除元素，取得栈顶元素，但是除了最顶端外，没有任何其他方法可以存取 stack 的其他元素。换言之，stack 不允许有遍历行为，也不提供迭代器

4.5 queue 容器

--

队列

queue 是一种先进先出(First In First Out,FIFO)的数据结构，它有两个出口，queue容器允许从一端新增元素，从另一端移除元素



只有 queue 的顶端元素，才有机会被外界取用。queue 不提供遍历功能，也不提供迭代器

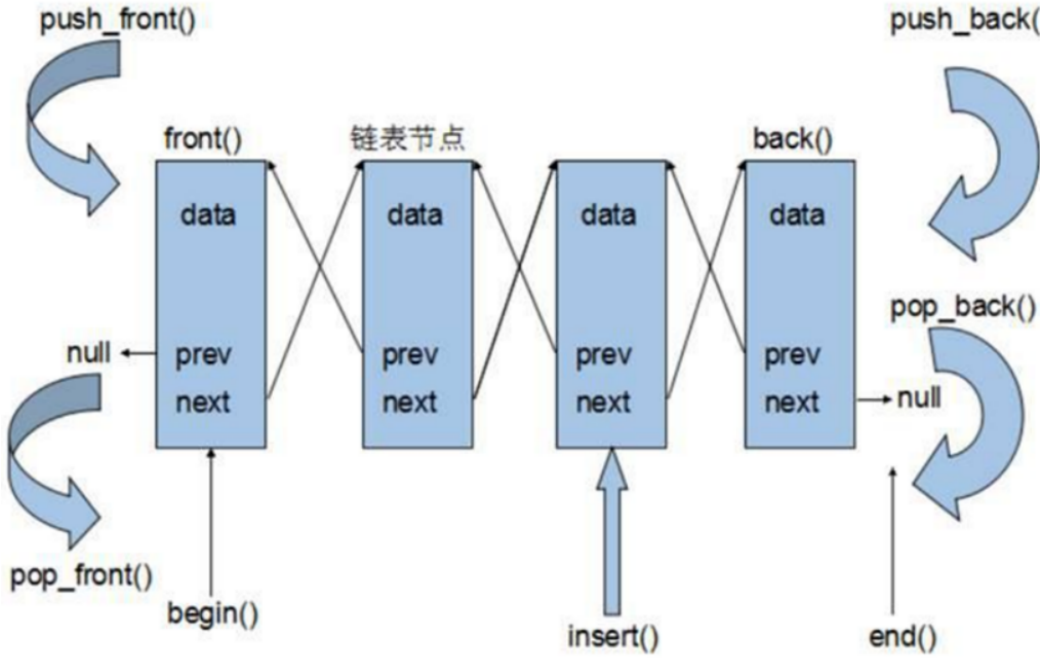
4.6 list容器

--

双向链表

链表是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的

元素的存储空间是随机的(不连续的存储空间)，所以在访问元素的时候，通过迭代器访问元素，不能用元素的序号访问



4.7 set/multiset容器

--

所有元素都会根据元素的键值自动被排序，set 的元素即是键值又是实值

multiset 特性及用法和 set 完全相同，唯一的差别在于它允许键值重复

不可通过迭代器修改键值

4.8 map/multimap容器

--

所有元素都会根据元素的键值自动排序。map 所有的元素都是pair,同时拥有实值和键值，pair 的第一元素被视为键值，第二元素被视为实值

Multimap 和 map 的操作类似，唯一区别 multimap 键值可重复

不可通过迭代器修改键值，可以修改实值

4.9 STL 容器使用时机

--

	vector	deque	list	set	multiset	map	multimap
典型内存结构	单端数组	双端数组	双向链表	二叉树	二叉树	二叉树	二叉树
可随机存取	是	是	否	否	否	对 key 而言：不是	否

	vector	deque	list	set	multiset	map	multimap
元素搜寻速度	慢	慢	非常慢	快	快	对 key 而言：快	对 key 而言：快
元素安插移除	尾端	头尾两端	任何位置	-	-	-	-

vector.at()比 deque.at()效率高，比如 vector.at(0)是固定的，deque 的开始位置 却是不固定的。

如果有大量释放操作的话，vector 花的时间更少，这跟二者的内部实现有关

deque 支持头部的快速插入与快速移除，这是 deque 的优点。

list 的使用场景：比如公交车乘客的存储，随时可能有乘客下车，支持频繁的不确实位置元素的移除插入。

set 的使用场景：比如对手机游戏的个人得分记录的存储，存储要求从高分到低分的顺序排列。

map 的使用场景：比如按 ID 号存储十万个用户，想要快速要通过 ID 查找对应的用户。二叉树的查找效率，这时就体现出来了。如果是vector 容器，最坏的情况下可能要遍历完整个容器才能找到该用户

4.10 常用算法

——

常用的遍历算法有哪些

遍历容器元素

for_each(iterator beg, iterator end, _callback);

将指定容器区间元素搬运到另一容器中

transform(iterator beg1, iterator end1, iterator beg2, _callbakc);

——

常用的查找算法有哪些

查找元素

find(iterator beg, iterator end, value);

条件查找

find_if(iterator beg, iterator end, _callback);

查找相邻重复元素

adjacent_find(iterator beg, iterator end, _callback);

二分查找法

bool binary_search(iterator beg, iterator end, value);

统计元素出现的次数

count(iterator beg, iterator end, value);

——

常用拷贝和替换算法

--

常用的排序算法有哪些

容器元素合并

merge(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);

容器元素排序

sort(iterator beg, iterator end, _callback);

对指定范围内的元素随机调整次序

random_shuffle(iterator beg, iterator end);

反转指定范围的元素

reverse(iterator beg, iterator end);

--

常用拷贝和替换算法

copy算法 将容器内指定范围的元素拷贝到另一容器中

copy(iterator beg, iterator end, iterator dest)

replace算法 将容器内指定范围的旧元素修改为新元素

replace(iterator beg, iterator end, oldvalue, newvalue)

replace_if算法 将容器内指定范围满足条件的元素替换为新元素

replace_if(iterator beg, iterator end, _callback, newvalue)

swap算法 互换两个容器的元素

swap(container c1, container c2)

--

常用算法生成算法

accumulate算法 计算容器元素累计总和

accumulate(iterator beg, iterator end, value)

fill算法 向容器中添加元素 fill(iterator beg, iterator end, valu