俞勇：自带背景音乐的男人（我也想上荣誉班）

BGM列表：

- 梁祝
- 回家

数据结构

CH 0 课程试讲

名字结构：李子龙-计算机班-李世博

3 Credits. 48 shr.

听 · 看 · 想 · 做

- 算法谁好呢？算法很快、空间效率很高、时间效率很高。
- 能力怎么样？学到的数据结构、基本算法怎么去活学活用？
- 程序写的真漂亮？！计算机艺术家、语句写得很精炼、读起来一气呵成。
- 学习基本功。专业基础课。

动机 · 方法 · 能力 · 情怀

- 好奇心，积极、主动性：代码一天不写就浑身难受
- 理论、实践和工程结合：形成学习闭环-应用-学习
- 求解问题的综合能力：知识的延伸和扩展靠自己
- 关注团队合作：讨论交流-社会关注之问题

逻辑结构 · 实现思想 · 封装 · 经典应用

课堂环节 · 实践环节 · 研讨环节

- 系统性和实用性
- 渐进式教学模式
- 自主/研究型学习

教学资源

- 电子教材
- 微课教材
- 作业系统
- 课程视频 https://www.boyuai.com/elites/course/ https://www.boyuai.com/dartastructure

CH 1

挤、堵、乱

先下后上、井然有序、分门别类

数据管理

> 整理扑克

数字序列

插入排序

> 寻找最短路径

- 结点：城市
- 有向边：城市间的交通
- 边上的数：城市间的距离

正整数边（无负环）

> 数据结构

数据间的关系 + 存储结构 = 数据结构

求解问题的方法 + 求解问题的效率 = 算法

如何成为代码高手：

1. 读：经典程序
2. 仿：加深理解
3. 默：得心应手
4. 练：针对训练
5. 用：灵活运用

> 快递

邮编 = 省市2 0 邮区0 县市2 投递区4 0

地址 = 国家 - 省市 - 地区 - 街道 - 门牌

分发、查找、派送、改投

> 图书

上架 下架 查找 分类

国际标准书号 ISBN

> 学生

插入 删除 修改 查找 （标准的**增删查改**）

学号

> 数据结构：一组具有特定关系的同类数据的存储及处理

研究内容

- 数据之间的逻辑关系
- 逻辑关系的存储（存储模式）

- 在某种存储模式下，操作的实现（运算）

逻辑结构

- 操场上的小朋友<**集合结构**>：节点间的次序是任意的
- 排队上车的旅客<**线性结构**>：节点间的关系一对一
- 上下级关系<**树形结构**>：节点间的关系一对多
- 社交关系<**图形结构**>：节点间的关系多对多

操作

线性结构：创建、清除、...

- 插入
- 删除
- 搜索
- 更新
- 访问
- 遍历

存储实现

逻辑关系 + 操作

- 数据存储
- 关系存储

数据存储 - 数据元素的类型

- 基本类型：`int`,`char`,`long`,`float`,...
- 复合类型：由基本、复合类型组成

```
struct date {
    int day;
    int month;
    int year;
};

struct student {
    char[20];
    char sex;
    struct date birthday; //Complex structure
}
```

关系存储 - 节点间的关系

顺序存储

- 节点存放在一块连续的存储区域中

- 节点间的关系由节点的存储位置体现
- 如线性表中的数组

= 宿舍安排

链接存储

- 节点可分散地存放在一块存储区域中
- 节点间的关系由指针显式地指出
- 如线性结构中的单链表

= 地下党单线联络：暗号

= 手机联系

散列（哈希）存储 *Hash*

- 专用于集合结构的存储方式
- 节点均匀地分布在一块连续的存储区域中
- 由一个散列函数将节点和存储位置关联起来

= 学院方阵

= 宿舍安排

索引存储 *Index*

- 所有存储节点按照生成的次序连续存放
- 需要设置一个索引区域表示节点之间的关系

= 教科书

章节 - 页码 页码 - 内容

> 算法分析

评价指标

- **正确性**：能正确地实现预定的功能
- **易读性**：易于阅读和理解，以便调试、修改和扩充
- **健壮性**：面对非法输入不会产生不正确的运算结果
- **高效率**：具有较高的时间和空间性能

算法主要考虑的是**时空**性能 *Spcial and Temporal Preformance* = 效率分析

时间

影响程序运行时间的因素

- **问题规模**
- 输入数据的**分布**

- 编译器生成的**目标代码**
- 计算机系统的**性能**
- 算法的**质量**

时间函数	提速前的求解规模	提速是10倍后
$O(n)$		
$O(n\log n)$		
$O(n^2)$		
$O(n^3)$		
$O(2^n)$		

时间复杂度：运算量与问题规模之间的关系

数据分布：

- 最好情况：极端
- 最坏情况：极端
- 平均情况：一般

标准操作

- 将标准操作作作为一个抽象的运算单位
- 一般采用简单语句作为标准操作 算法运算量的计算
- 在给定的输入下有了多少标准操作

> 现在一组正整数，存放在数组A中，设计一个算法求数组中最大值与d的乘积。

```
int max1(int[] array, int d){
    int max=0,i;
    for (i=0;i<size;++i)    //1+n+1+n
        array[i]*=d;        //n*2
    for (i=0;i<size;++i)
        if (array[i]>max)
            max=array[i];
    return max;             //8n+4
}

int max2(int[] array, int d){
    int max=0,i;
    for (i=0;i<size;++i)
        if (array[i]>max)
            max=array[i];
    return max*d;           //4n+3
}
```

时间性能主要考虑：**问题规模很大时**运行时间随问题规模的变化规律。

采用渐进表示法：

- 不考虑具体的运行时间函数
- 只考虑运行时间函数的数量级

大O表示法[数量级]

- 定义
  - 如果存在一个正的常数c和自然数N_0，当$N \geq N_0$时，有$T(N) \leq cF(N)$成立（上界），则称$T(N)=O(F(N))$。是XX级或阶的。

大Ω表示法 $$ T(N)\geq cF(N) \Rightarrow T(N)=\Omega(F(N)) $$

大Θ表示法 $$ c_1F(N)\leq T(N)\leq c_2F(N) \Rightarrow T(N)=\Theta(F(N)) $$

小o表示法 $$ T(N)=o(F(N)) \Leftrightarrow T(N)=O(F(N))~&~T(N)\neq o(F(N)) $$

简化方法：

求和定理 顺序： $T_1(n)+T_2(n)=O(\max(f(n),g(n)))$

求积定理 嵌套 $T_1(n) \cdot T_2(n)=O(f(n)\cdot g(n))$

- 顺序语句-求和定理
- 条件语句-求和定理
- 循环语句-运行时间*循环次数
- 嵌套语句-求积定理
- 连续语句-求和定理

> 算法优化

> *最大连续子序列问题*：给定整数序列A_0,...,A_n 寻找（并标识）$\sum_{k=i}^j A_k$的值为最大的序列。

解法一 枚举法

```
for(int i=0;i<n-1;++i)
    for(int j=i;j<n;++j)
        //init. thisSum
        for(int k=i;k<=j;++k)
            //Calculating thisSum by adding.
        //Compare to maxSum.
```

$\frac{(n+2)(n+1)n}{6}=O(n^3)$

解法二

```
 for(int i=0;i<n-1;++i)
    //init. thisSum
    for(int j=i;j<n;++j)
```

```
        //Calculating thisSum by adding a[j].
        //Compare to maxSum.
```

$O(n^2)$

解法三 考虑性质：如果一个子序列的和为负，不可能是最大连续子序列的一部分。 设`f[x]`是以`a[x]`终止且包含`a[x]`的最大序列的和，有

```
f[0]=a[0];
f[x+1]=(f[x]>0?f[x]+a[x+1]:a[x+1]);
max f[i].
```

$O(n)$

不需要开数组，采用**动态规划。**

> 面向对象的数据结构

使用说明书 = 做什么：功能[功能] + 怎么用：操作[接口] + ~~怎么做：工艺[封装]~~

创建工具；存储和处理过程封装成一个个工具；逻辑结构的逻辑特性；如何封装更能适合用户的需求。

数据结构的逻辑特性

- 每种数据结构用一个抽象类描述
- 指出该数据结构提供的操作（接口）

数据结构的实现

- 每种数据结构可以有若干种实现的方法
- 每种实现就是一个类

泛型程序设计的实现

- 数据结构只关注数据元素间的关系是如何保存的
- 数据元素可以任意类型，C++中的工具成为模板
- 数据结构中的所有类都是类模板

CH 2 线性结构

> 建模：适当的、求解：有效的

- 具有相同特征的节点构成的有限序列

首节点--------尾结点
前驱节点与后继节点
或空或只含一个节点

线性表：仅通过结点之间的相对位置来确定他们之间的相互关系的线性结构。

时间有序表：按照节点到达的时间先后，作为确定节点之间关系的线性结构。

排序表：根据节点的关键字值来确定他们之间的相互关系的线性结构。

频率有序表：按照节点的使用频率确定他们之间的相互关系的线性结构。

线性表

节点序列：$a_0,a_1,\cdots ,a_{n-1}(n\geq 0)$

- 创建：`create()`
- 清除：`clear()`
- 线性表长度：`length()`
- 插入：`insert(i,x)`
- 删除：`remove(i)`
- 搜索：`search(x)`
- 访问：`visit(i)`
- 遍历：`traverse()`

抽象类

线性表的抽象类是一个类模板，抽象类包括了除`create`运算外的所有运算，增加了一个虚析构函数（非纯虚函数），否则派生类析构函数将不能执行，会导致内存泄漏。

```
template <class elemType>
class list{
    public:
        virtual void clear() = 0;   //In Derived only
        virtual int length() const = 0;
        virtual void insert(int i,const elemType &x) = 0;
        virtual void remove(int i) = 0;
        virtual int search(const elemType &x) const = 0;
        virtual elemtype visit(int i)const = 0;
        virtual void traverse() const = 0;
        virtual ~list() {}
}
```

```
ClxBase *pTest = new ClxDerived;
pTest->DoSomething();
delete pTest;
```

如果不使用虚函数，就会导致析构派生类时只调用基类的析构函数！内存泄漏！

顺序存储

用一组连续的存储单元依次存储数据元素，数据元素之间的逻辑关系由元素的存储位置来表示。

保存一个动态数组，需要三个变量

- 指向线性表元素类型的指针：data
- 数组规模（容量）：maxSize
- 数组中的元素个数（表长）：length
- 存储空间

顺序表类 私有数据成员：data, maxSize, length
私有成员函数：数组扩容doubleSpace()
共有成员函数：list

```
template <class elemtype>
class seqList:public list<elemType>{
    private:
    elemType *data;
    int currentLength;
    int maxSize;
    void doubleSpace();
    public:
    seqList(int initSize=10);
    ~seqList() {delete[] data;}
    void clear() {currentLength=0;}
    int length() const {return currentLength;}
    void insert(int i,const elemType &x);
    void remove(int i);
    int search(const elemType &x) const;
    elemType visit(int i) const {return data[i];}
    void traverse() const;
}

template <class elemType>
seqList<elemType>::traverse() const{
    cout<<endl;
    for (int i=0;i<currentLength;++i)
        cout<<data[i]<<' ';
}

template <class elemType>
seqList<elemType>::seqList(init initSize){
    data = new elemType[initSize];
    maxSize=initSize;
    currentLength = 0;
}

template <class elemType>
int seqList<elemType>::search(const elemType &x) const {
    int i;
    for (i=0;i<currentLength &&data[i]!=x;++i);
    if (i==currentLength) return -1;
    else return i;
}

template <class elemType>
void seqList<elemType>::search(int i,const elemType &x){
```

```
    if (currentLength==maxSize) doubleSpace();  //Full
    for (int j=currentLength;j>1;j--)   //From the last element
        data[j]=data[j-1];
    data[i]=x;
    ++currentLength;
}

template <class elemType>
void seqList<elemType>::doubleSpace(){
    elemType *tmp=data;
    maxSize*=2;
    data=new elemType[maxSize];
    for (int i=0;i<currentLength;++i)
        data[i]=tmp[i];
    delete[] tmp;   //Release tmp
}

template <class elemType>
void seqList<elemType>::remove(int i){
    for (int j=i;j<currentLength-1;j++)
        data[j]=data[j+1];
    --currentLength;
}
```

- `length`,`visit`,`clear` $O(1)$ 性能很好！不会改变逻辑次序和物理次序
- `traverse` $O(n)$
- `create` $O(1)$
- `insert`,`remove`,`search`,`resize`
    - best case: $O(1)$
    - worst case: $O(n)$
    - average case: $\frac{n}{2}=O(n)$ 不理想！要保持一致性！

比较适合静态的，经常做定位访问的。

单链表

用一组存储单元分散存储数据元素，数据元素之间的逻辑关系由存储单元中附加的指针给出

将每个节点放在一个独立的存储单元中，节点间的逻辑关系依靠存储单元中附加的指针给出；节点的存储单元在物理位置上可以相邻，也可以不相邻；姐弟啊的直接后继节点可由其附加指针来表示。

指针字段：`next`（最后一个元素`next=NULL`）

1. 创建 `x`
2. 链接后继
3. 链接前驱
4. 改变当前指针

顺序不能错！

头结点：通常在表头额外增加一个相同类型的特殊节点 它们不是线性表中的组成部分

使得在表头位置上进行插入和删除和在其他节点位置上是完全一致的。

插入结点：

- 插入第一个结点时：`new(p);p->next=head;head=p;`
- 插入第i个结点：`new(p);p->next=q->next;q->next=p;`

单链表类：

- 私有数据成员：存储表元素的单链表、头指针、表长
- 私有成员函数：移动到指定的节点地址
- 共有成员函数：`list`

```
template<class elemType>
class sLinkList: public list<elemType>{
    private:
        struct node{
            elemType data;
            node *next;
            node(const elemType &x, node *n=NULL){data=x;next=n;}
            node():next(NULL){}
            ~node(){}
        };
        node *head;
        int currentLength;
        node *move(int i) const;

    public:
    sLinkList();
    ~sLinkList(){clear();delete head;}
    void clear();
    int length() const {return currentLength;}
    void insert(int i,const elemType &x);
    void remove(int i);
    int search()
    /**/
}

template<class elemType>
sLinkList<elemType>::sLinkList(){
    head=new node();
    /**/
}

template<class elemType>
void sLinkList<elemType>::clear(){
    node *p=head->next,*q;
    head->next=NULL;
    while (p!=NULL){
        q=p->next;delete p;p=q;
    }
```

```
    currentLength=0;
}

template<class elemType>
void sLinkList<elemType>::insert(int i,const elemType &x){
    node *pos;
    pos=move(i-1);
    pos->next=new node(x,pos->next)
    ++currentLength;
}

remove()
```

找位置 & 移动的区别

insert $O(n)$

双链表

两个指针：prev和next head和tail

先安顿自己 先保护自己

移动到指定的节点地址

单循环链表

双循环链表

STL 线性表

C++ 标准模板库 Standard Template Library 容器

Vector：线性表的顺序实现 List：线性表的双链表的实现 迭代器：定位/操作容器中的对象

- 迭代器对象相当于指向容器中对象的指针
- 它封装了容器中对象的位置信息

++ 迭代器可行

vector cyclinder<->back

list front<->cyclinder<->back

共同操作： 求规模、清空、判空 表尾插/删 取表头/尾 list 特有： 表头插/删

迭代器操作 Begin/End ++ * 删。。。

```
template <class Node>
class List{
    Node *head;
    class iterator{
```

```
        Node *ptr;
        public:
            iterator(Node *p=0):ptr(p){}
            iterator &operator++(){ptr=ptr->next;return *this;} //前置
            iterator operator++(int) {iterator old=*this;ptr=ptr->next;return
old;}
            Node opeartor*() const {return *ptr;}
            bool operator==(const iterator &i) const {return ptr==i.ptr;}
            bool opeartor!=(const iterator &i) const {return ptr!=i.ptr;}
    }
    /*头指针和尾指针*/
}
```

- 维护一个C++的原始数组、容量和规模。

```
template <class object>
class vector {
    private:
        int theSize;
        int theCapacity;
        object *objects;
    public:
        explicit vector(int initSize=0);//不允许隐式转换
        vector(const vector &R);
        ~vector();
        void resize(int newsize);
        void reserve(int newCapacity);
        const vector &operator=(const vector &r);
        object &operator...
        /**/
}

vector<int> v;
v.size() v.capacity()
v.push_back(2);//放入
capacity 1,2,4
size 1,2,3
```

```
#include <list>

template <class object>
void printall(const list<object>&v){
    if (v.empty())
        cout<<"\n list is empty!";
    else
        /**/
}
```

虽然本次作业的 1202 超时了，但是我只能这样喽。

CH 3 栈的定义

> 进去的顺序与出来的顺序正好相反。

后进先出(LIFO, Last In First Out)
先进后出(FILO, First In Last Out)

栈底(bottom)
栈顶(top)
进栈(push)
出栈(pop)
空栈(isEmpty)

```
template<class elemType>
class stack{
public:
    //没有构造函数：不知道存储方式
    virtual bool isEmpty() const = 0;   //虚函数为了实现多态性
    virtual void push(const elemType &x) = 0;  //const保护对象不被修改
    virtual elemType pop() = 0; //纯虚函数让派生类实现
    virtual elemType top() const = 0;   //所有的数据成员都不能动
    virtual ~stack() {}     //可以调用子类的函数，先析构子类
}
```

栈的顺序实现

- 用连续的空间存储栈中的节点（数组）
- 进栈和出栈总是在栈顶一端进行
    - 栈顶为前端：第一个元素（X）
    - 栈顶为后端：最后一个元素（运用自如）
- 栈的顺序存储结构比较常见

```
create()    //top_p=-1(init.)
push(x)     //*(top_p++)=x, doublespace(减少判断开销)
pop()       //read; top_p--
top()       //read;
isEmpty()   //top_p==-1 ? true : false
```

```
template<class elemType>
class seqStack: public stack<elemType>{
private:
    elemType *elem;
```

```
    int top_p;
    int maxSize;
    void doubleSpace();
public:
    seqStack(int initSize=10){
        elem=new elemType[initSize];
        maxSize=initSize;
        top_p=-1;
    }
    ~seqStack(){
        delete[] elem;
    }
    bool isEmpty() const {
        return top_p==-1;
    }
    void push(const elemType &x){
        if (top_p==maxSize-1)
            doubleSpace();              //除了这步都是O(1)
        elem[++top_p]=x;    //++后再赋值
    }
    elemType pop(){
        return elem[top_p--];   //可以覆盖
    }
    elemType top() const {
        return elem[top_p];
    }
    void doubleSpace(){
        elemType *tmp=elem;
        elem = new elemType[2*maxSize];
        for (int i=0;i<maxSize;++i)
            elem[i]=tmp[i];
        maxSize*=2;
        delete[] tmp;
    }

}
```

栈的链接实现

- 单链表就可以了
- 不考虑头结点：**头指针指向栈顶**

```
template<class elemType>
class linkStack: public stack<elemType>{
private:
    struct node{
      elemType data;
      node *next;
      node(const elemType &x,node *N=NULL)  {data =x;next=N;}
      node():next(NULL){}
      ~node(){}
```

```
    };
    node *top_p;
public:
    linkStack(){
        top_p=NULL;
    }
    ~linkStack(){
        node *tmp;
        while (top_p!=NULL){
            tmp=top_p;
            top_p=top_p->next;
            delete tmp;
        }
    }
    bool isEmpty(){
        return top_p==NULL;
    }
    void push(const elemType &x){
        top_p=new node(x,top_p);    //四件事情
    }
    elemType pop(){
        node *tmp=top_p;
        elemType x= tmp->data;
        top_p=top_p->next;
        delete tmp;
        return x;
    }
    elemType top() const{
        return top->data;
    }
}
```

STL 的栈

栈是容器适配器。

- `elemType`
- `vector`, `list`, `deque`

push_back()
pop_back()
back()
...

```
#include <iostream>
#include <stack>
using namespace std;
int main(){
    stack <int, vector<int>> s1;
    stack <int, list<int>> s2;
```

```
    int i;
    for (i=0;i<10;++i)
        s1.push(i);
    while (!s1.empty()){
        cout<<s1.top()<<' ';
        s1.pop();   //删除栈顶
    }
    cout<<endl;
    for (i=0;i<10;++i)
        /*blah blah*/
}
```

递归函数的非递归实现

调用

- 保存实参、返回地址等信息
- 为被调用函数的局部变量分配存储区
- 将控制转移到被调用函数的入口

返回

- 保存被调函数的计算结果，并回复现场
- 释放被调函数的数据区
- 将控制转回到调用函数

自己调用自己

递归

```
void quicksort(int a[],int low,int high){
    int mid;
    if (low>=high) return ;
    mid = divide(a,low,high);       //确定中间节点的位置
    quicksort(a,low,mid-1);
    quicksort(a,mid+1,high);
}
```

- 设置一个栈，记录要排序的数据段
- 先将整个数据进栈，然后重复下列工作，至到栈空
    - 从栈中弹出一个元素，即一个排序区间
    - 将排序区间分成两半
    - 检查每一半，如果多余两个元素，则进栈

```
struct node{
    int left;
    int right;
};
```

```
void quicksort(int a[],int size){
    seqStack<node> st;
    int mid, start, finishl
    node s;

    if (sieze<=1)
    return;

    s.left=0;
    s.right=size-1;
    st.push(s);

    while (!st.isEmpty()){
        s=st.pop();
        start=s.left;
        finish=s.right;
        mid=divide(a,start,finish);//从后往前找比首元素小的元素位置，从前往后找比首元
素大的第一个元素，两元素交换，继续找
        if (mide-start>1){
            s.left=start;
            s.right=mid-1;
            st.push(s);
        }
        if (finish-mid>1){
            s.left=mid+1;
            s.right=finish;
            st.push(s);
        }
    }
}
```

符号配对问题

检查符号是否匹配

- 若用开闭括号是否相等?
- 用栈 `push` 与 `pop` （是否）成对

```
class Balance{
    //对象初始化时传给他一个源文件名
    //这个类提供一个成员函数CheckBalance检查源文件中的符号是否配对
    //输出所有不匹配的符号及所在的行号

private:
    ifstream fin;       //待检查的文件流
    int CurrentLine;    //正在处理的行号
    int Errors;         //已发现的错误数

    struct Symbol{
        char Token;
```

```
        int TheLine;
    };

    enum CommentType{SlashSlash, SlashStar};

    /*私有成员函数*/
    //symb1 = 栈顶符号，Symb2=当前符号，Line1栈顶行数，当前行数
    bool CheckMatch(char Symb1, char Symb2, int Line1, int Line2){
        if (Symb1 == '(' && Symb2!=')' || Symb1=='[' && Symb2!=']' || Symb1 == '{'
&& Symb2 != '}'){
            cout<</*  */<<endl;
            return false;
        }
        else
            return true;
    }
    char GetNextSymbol(){
    char ch1;
    while (ch=NextChar()){
        if (ch=='/'){
            ch=NextChar();
            if (ch=='*')
                SkipComment(SlashStar);
            else if (ch=='/')
                SkipComment(SlashSlash);
            else
                PutBackChar(ch);
        }
        else if (ch=='\''||ch=='"')
            SkipQuote(ch);
        else if (ch=='{'||ch=='['||ch=='('||ch=')'||ch==']'||ch=='}')
            return ch;
    }
    return NULL;
    }
    char NextChar(){
        char ch;
        if ((ch=fin.get())==EOF)
            return NULL;
        if (ch=='\n')
            CurrentLine++;
        return ch;
    }
    char PutBackChar(){
        fin.putback(ch);
        if (ch=='\n')
            CurrentLine--;
        //回车
    }
    char SkipQuote(char type){
        char ch;
        while (ch=NextChar()){
            if (ch==type)
                return ;
```

```
            else if (ch=='\n')
                break;
            else if (ch=='\\')
                ch==NextChar/..
        }
    }
    void SkipComment(enum CommentType type){
        /**/
    }

        //ch1=='/' 读ch2
        // ch2=='*'     Comment:    /**/
        // ch2=='/'     Comment:    //
        // else, back to ch2;
    // else
        // ch1=='\'' OR '"', skip 反义符号
        // ch1=='{' OR '[' OR '(' OR ')' OR ']' OR '}', back to ch1
    char GetNextSymbol(){
        char ch1

    }


public:
    //输入文件流（输入流）
    balance(const chat *s){
        fin.open(s);
        if (!fin) throw noFile();
        currentLine = 1;
        Errors = 0;
    }
    //检查文件流中的括号是否匹配
    int CheckBalance(){
    struct Symbol node;
    seqStack<Symbol> st;    //1: 创建空栈
    char LastChar,Match;    //读入的符号；栈顶的符号

    while (LastChar=GetNextSymbol()){ //2: 从文件中读取符号
        switch (LastChar) {
            case '(':case '[': case '{':
                node.Token=LastChar;
                node.TheLine=CurrentLine;
                st.push(node);
                break;
             case /* 闭括号情形 */
                if(st.isEmpty()){
                    Errors++;
                    cout<</**/<<endl;
                }
                else{
                    node = st.pop();
                    Match=node.Token;
                    if(!CheckMatch(Match, LastChar, node.TheLine, CurrentLine)
                        ++Errors;
```

```
                    break;
                }
            }
        }
        while(!st.isEmpty()){
            Errors++;
            node=st.pop();//不确定
            cout<</**/<<'\n';    //以后尽量少用endl，减少检查，提高效率
        }
        return Errors;
        }
        //3: 如果是开括号，进栈
        //4: 如果是闭括号，但是栈空，出错；否则，栈中符号出栈
           //5: 类型不匹配，出错
        //6: 继续，非空->3，空->7
        //7: 如果栈非空，报告出错，如果栈空括号匹配成功
    }

    class NoFile() {};
```

CH 4 表达式

main函数

```
int main(int argc,char *argv[],char *envp[])
    // int argc     参数个数
    // char *argv[] 参数
    // char *envp[] 环境变量
```

```
//#include "balance.h"

int main(int argc,const char **argv){
    char filename[80];
    balance *p;
    int result;

    try{
        if (argc==1){
            //cout<<
            cin>>filename;
            p=new balance(filename);
            result=p->CheckBalance();
            delete p;
            cout<</**/<<endl;
            return 0;
        }
        while (--argc){
            cout <<"检查文件"<<*++argv<<endl;
            p=new balance(*argv);
```

```
            result=p->CheckBalance();
            delete p;
            cout<</**/<<endl;
        }
        catch(noFile){
            cout<<"no such file"<<endl;
        }
        return 0;
    }
}
```

对于一个表达式：a+b
前缀式：+ab 波兰式
中缀式：a+b
后缀式：ab+ 逆波兰式 -- 优点：

- 可以不考虑运算符的优先级
- 距离符号最近的两操作数作运算

后缀式工作方式

1. 初始化一个栈
2. 依次读入后缀式的操作数和运算符
3. 若读到的是操作数，则将其进栈
4. 若独到的是运算符，则将栈顶的两个操作数出栈，后弹出的操作数为被操作数，先弹出的为操作数，将得到的操作数完成运算符所规定的的运算，并将结果进栈
5. 回到2. 继续读入操作
6. 当栈中只剩一个操作数时，弹出该操作数，它就是表达式的值

中缀式→后缀式

1. 若读入的是操作数，立即输出
2. 若读到的是开括号，则进栈
3. 若读入的是闭括号，则
    1. 将栈中运算符依次出栈，将其放在操作数序列后
    2. 出栈操作一直进行到遇到对应的开括号为止
    3. 将开括号出栈
4. 若读入的是运算符
    1. 如栈顶运算符优先级高，则栈顶运算符出栈
    2. 出栈操作一直进行到栈顶运算符优先级相同为止
    3. 将新读入的运算符进栈
5. 在读入操作结束时，将栈中所有的剩余运算符依次出栈，并放在操作数序列之后直到栈空为止

读剩的中缀式 栈中内容 输出的后缀式

> peek 读输入流但不提出，类似于栈的top

CLAC类实现

- 计算器中的表达式中的运算数都是常量

    - 当发现某个运算符可计算时，可直接执行运算，保存运算结果
    - 无需写出其后缀表达式：即可以将转换和计算两个步骤合并起来，边转换边计算
- 运算过程需要用到两个栈
    - 中缀表达式转换后缀表达式时的运算符栈
    - 执行后缀表达式运算时的运算数栈

同级先到先做

`calc<ElemType>`

按照面向对象的设计思想，先设计一个工具。

```
int main(){
    calc exp("3*(7+5)/6-2");
    cout<<exp.result()<<endl;

    calc exp("3*7+5");
    cout<<exp.result()<<endl;

    return 0;
}
```

```
//template<ElemType>
class calc{
private:
    char* expression;//存放表达式的字符串
    enum token{OPAREN,ADD,SUB,MULTI,DIV,EXP,CPAREN,VALUE,EOL};
    void BinaryOp(token op,seqStack<int> &dataStack){//执行一个算术运算
    }
    token getOp(int &value){
        //从表达式中取出一个合法符号
        while(*expression){
            while (*expression  //EOF \0
                && *expression==' ')
                ++expression;
            if (*expression<='9'&&*expression>='0'){
                value=0;
                while (*expresion >='0' && *expresssion<='9'){
                    value = value*10+*expression-'0';
                }
                return VAL;
            }

            switch(*expression){
                /*二元表达式的识别与返回*/
            }
            return EOL;

        }
    }
```

```
public:
    calc(char *s){
        expression=new char[strlen(s)+1];   // \0
        strcpy(expression,s);
    }
    ~calc(){delete expression;}//构造/析构函数
    int result(){//计算表达式结果
        //依次从表达式周静取出一个合法的符号，直到表达式结束
        {
            switch(/*当前符号*/){
                case /*数字*/:
                    //将数字存入运算数栈
                case '(':
                    //开括号进运算符栈
                case ')':
                    //开括号和闭括号间所有运算都可进行，即将运算符栈中运算符一次出栈，执
行相应运算，直到’(‘出栈
                case '^':
                    //乘方运算符进运算符栈
                case '*':case '/':
                    //运算符栈中的/,*,^退栈执行，当前运算符进栈
                case '+':case '-':
                    //运算符栈中运算符依次出栈执行，直到栈为空或遇到开括号，当前运算符进
栈
                }
            }
        }
        //运算符栈中所有的运算符出栈执行;
        if (/*运算数栈为空*/)
            //出错，无运算结果
        result_value = /*运算数栈出栈元素*/
        if (/*运算数非空*/)
            //出错缺运算符
        return result_value;
    }
}
```

队列的定义

按照结点到达的时间顺序，确定结点间关系的线性结构

`FIFO` 或 `LILO` 的结构

创建一个队列`create()`
进队`enQueue(x)`
出队`deQueue()`
读取队首元素`getHead()`
虚析构函数`isEmpty()`

```
template <class elemType>
class queue{
```

```
public:
    virtual bool isEmpty() const = 0;
    virtual void enQueue(const elemType &x) = 0;
    virtual elemType deQueue() = 0;
    virtual elemType getHead() const = 0;
    virtual ~queue() {}
};
```

队列的顺序存储

- 使用数组存储队列中的元素
  - 队列中的节点个数最多为 `MaxSize` 个
  - 元素下标范围从 `0` 到 `MaxSize-1`
- 顺序队列的三种组织方式
  - 队首位置固定
  - 队首位置不固定
  - 循环队列

队首位置固定在`0`，队列为空时，队尾位置为`-1`。

队首位置不固定，使用指针`front`和`rear`。
初始标志：`front = rear = -1`
队空标志：`front = rear`
队满标志：`rear = MaxSize - 1`

循环队列

进队操作：

```
rear=(rear + 1) % MaxSize;
elem[rear]=x;
```

出队操作：

```
front = (front + 1) % MaxSize;
```

创建队列：

```
front = rear = 0;
```

"牺牲"一个单元，规定front指向单元不能存储队列元素（标志），后一个是队首元素。
队满条件：

```
(rear + 1) % MaxSize == front;
```

队空条件

```
front == rear;
```

甜甜圈

```
template<class elemType>
class seqQueue:public queue<elemType>{
private:
    elemType *elem;//存储队列元素的数组
    int maxSize;//数组的规模
    int front,rear;//队首队尾的标志
    void doubleSpace(){
        elemType *tmp=elem;
        elem=new elemType[2*maxSize];
        for (int i=1;i<maxSize;++i)
            elem[i]=tmp[(front+i)%maxSize];
        front = 0;
        rear=maxSize-1;
        maxSize*=2;
        delete[] tmp;
    }//数组扩容
public:
    seqQueue(int size=10){
        elem=new elemType[size];
        maxSize=size;
        front = rear =0;
    }//构造析构函数
    bool isEmpty() {return front==rear;}//判队列空
    void enQueue(const elemType &x){

    }
    void deQueue(){

    }
    //进队出队
    elemType getHead(){
        return elem[front];
    }//读队首
};
```

链接实现

单链表；记住首尾节点的位置 $O(1)$

初始化：

```
front = rear = NULL;
```

```
template<class elemType>
class linkQueue:public queue{
private:
    struct node{
        elemType data;
        node *next;
        node ( )...
    }//存储队列元素的链表
    //队首队尾标志
public:
    linkQueue();
    ~linkQueue();//构造析构函数
    bool isEmpty() const {return front==NULL;}//判队列空
    void enQueue(const elemType &x);//进队
    elemType deQueue();//出丢
    elemType getHead() {return head->data;}//读队首
}
```

CH 5

STL 中的队列

Deque: front - Deque - back

1049 变种 queue 方式：蜘蛛纸牌

有三根缓冲轨道的情况：每一条轨道是一个队列

依次取入轨的队首元素，直到队列为空：

- 进入一条最适合的缓冲轨道：选择大于队尾车厢编号且其中最大编号的车厢
- 检查每条缓冲轨道的队首元素，将可以出轨的元素出队，进入出轨

```
//重新排列车厢，使得车厢编号与其车站编号一致
void arrange(int in[],int n,int k){
    linkQueue<int> *buffer = new linkQueue<int>[k];
    int last = 0;
    for (int i=0;i<n;++i){
        if (!putBuffer(buffer,k,in[i])) return ;
        checkBuffer(buffer,k,last);
    }
}

//将车厢in进入合适的缓冲队列
```

```
bool putBuffer(linkQueue<int> *buffer,int size,int in){
    int avail = -1, max = 0;
    for (int j = 0;j<size;++j){
        if (buffer[j].isEmpty()){
            if (avail == -1)
                avail = j;
        }
        else if (buffer[j].getTail()<in &&
        buffer[j].getTail() > max   /*且接近最大编号的车厢*/ ){
            avail = j;
            max = buffer[j].getTail();
        }
    }
    if (avail!=-1){
        buffer[avail].enQueue(in);
        cout<< in <<endl;
        return true;
    }
    return false;
    }
}

//将缓冲轨道中值为last+1的队首元素出队，进入出轨
void checkBuffer(buffer,k,last){
    //检查出轨
}

//增加读队尾
```

排队系统的模拟

用计算机来模拟有很多优点：

1. 不需要真是的顾客就能得到统计信息
2. 由于计算机速度很快，使用计算机模拟比真实的实现快很多
3. 模拟结果可以简单地重现

模拟排队系统：

- 事件系列：顾客到达（请求服务），顾客离开（服务完毕）
- 统计信息：顾客：排队时间、等待时间，服务台：工作时间、空闲时间

特点：离散（不断发生、但不连续）

（时间不定：从统计上服从一定的概率分布、随机数生成器） 到达事件：

- 服务台忙 - 顾客排队
- 服务台空 - 生成顾客服务时间，结束后产生离开事件

离开事件：

- 有顾客：顾客离队、接受服务
- 没顾客：服务员休息

均匀分布的概率事件：

- 如顾客到达的间隔时间服从$[a,b]$之间的均匀分布
- 则可生成一个$[a,b]$之间的随机数x
- 表示前一顾客到达后，经过x时间后又有一个顾客到达

虚拟系统的时间模拟：

- 运行时间：取决于滴答数、不取决于实际时间
- 离散时间：取决于时间到达的顺序：当一个事件处理结束后直接把时间拨到下一个时间、不取决于时间的到达的时间

设计一个最简单的排队系统模拟器，单服务台模型。

银行中只有一个服务台，统计顾客的平均排队时间。

到达事件

排队 服务 离开

```
totalWaitTime = 0;
currentTime = 0;    //设置顾客开始到达的时间
//模拟生成所有的到达事件
for (i=0;i<customNum;++i){
    //生成下一顾客到达的间隔时间
    currentTime += 下一顾客到达间隔时间
    //将下一个顾客的到达时间进队
}
//从0时刻开始模拟
while(/* 顾客队列非空 */){  //为到达时间提供服务
    //取队首顾客
    if (/* 到达时间>当前时间 */)
        //直接将时钟拨到事件发生的时间
    else
        //收集该顾客的等待时间
    //生成服务时间
    //将时钟拨到服务完成的时间
}
return /* 等待时间 / 顾客数 */;
```

设计一个实现单服务台排队系统的工具：

```
class simulator {
private:
    int arrivalLow;//到达间隔时间下限
    //到达间隔时间上限
    //服务时间下限
    //服务时间上限
    //
public:
```

```
    simulator(){
    srand(time(NULL));
    }//

    int avgWaitTime() const {
        int currentTime = 0;
        int totalWaitTime = 0;
        int eventTime;
        linkQueue<int> customerQueue;
        int i;
        for (i=0;iMcustomNum;++i){
            //生成到达时间
        }
        currentTime=0;
        while(!customerQueue.isEmpty()){
            eventTime=customerQueue.deQueue();
            if ...
        }
    }
}

int main(){
    simulator sim;
    sim.avgWaitTime();
    return 0;
}
```

队列是一种先进先出的线性表 FIFO

字符串

字符串是由0个或多个字符组成的有限序列，字符串就是一个线性表

```
length(s)
disp(s)
equal/greater/greaterEqual/less/lessEqual(s1,s2)
copy(s1,s2)
cat(s1,s2)
substr(s,start,len)
insert(s1,start,s2)
remove(s,start,len)
```

```
c: cstring
c++: string
```

顺序串类 Time>Space

```
class seqString{
private:
    char *data;//字符串
    int len;//字符串长度
    //运算符重载
    friend seqString opeartor+(const seqString &s1, const seqString &s2);
    /* a lot of opeartor reloads */
public:
    //构造、析构
    seqString(const char *s=""){
        for (len=0;s[len]!='\0';++len);
        data = new char[len+1];
        for (len =0;s[len]!='\0';++len)
            data[len]=s[len];
        data[len]='\0';
    }
    seqString(const seqString &other){
        data = new char[other.len + 1];
        for (len=0;len<=other.len;++len){
            data[len]=other.data[len];
        }
    }
    ~seqString(){

    }
    //取字符串长度
    int length() const{

    }
    //赋值 运算符重载=
    seqString &operator=(const seqString &other){
        if (this == &other) return *this;
        data=new char[other.len+1];
        for (len=0;len<=other.len;++len)
            data[len]=other.data[len];
    }
    //取子串
    seqString subStr(int start,int num) const{
        if (start >= len - 1|| start <0)
            return "";
        seqString tmp;
        tmp.len = (start + num > len)?len-start:num;    //防止越界
        delete tmp.data;
        tmp.data = new char[tmp.len + 1];
        for (int i=0;i<tmp.num;++i)
            tmp.data[i]=data[start+i];
        tmp.data[i]='\0';
        return tmp;
    }
    //插入子字符串
    void insert(int start,const seqString &s){
        char *tmp=data;
        int i;
```

```
        if (start>len||start<0)
            return ;
        len+=s.len;
        data=new char[len+1];
        for (i=0;i<start;++i)
            data[i]=tmp[i];
        for (i=0;i<s.len;++i)
            data[start+i]=s.data[i];
        for (i=start;tmp[i]!='\0';++i)
            data[i+s.len]=tmp[i];
        data[i+s.len]='\0';
        delete tmp;
    }
    //删除子字符串
    void remove(int start,int num){
        if (start>=len-1||start <0)
            return ;
        if (start + num>=len){
            data[start]='\0';
            len=start;
        }
        else{
            for (len = start; data[len+num]!='\0';++len)
                data[len]=data[len+num];
            data[len]='\0';
        }
    }
};
```

链接实现

块状链表 提高了空间利用率 解决方案：允许节点有一定的空闲空间

```
class linkString {
private:
    stuct node{
        int size;
        char *data;
        node *next;
        node (int s=1;node *n=NULL){
            data = new char[s];
            size = 0;
            next = n;
        }
    }
    node *head;

    //字符串标识
    int len;//字符串长度
    int nodeSize;//每个节点容量

    //运算符重载
```

```
    void clear();//清除字符串
    //定位字符串的位置
    void findPos(int start,int &pos, node *&p) const{
        int count = 0;
        p = head -> next;
        while (count < start){
            if (count + p->size<start){
                count +=p->size;
                p = p->next;
            }
            else{
                pos = start - count; return ;
            }
        }
    }
    //分裂字符串
    void split(node *p,int pos){
        p->next=new node(nodeSize,p->next);
        for (int i=pos;i<p->size;++i)
            p->next->data[i-pos]=p->data[pos];
        p->next->size = i-pos;
        p->size = pos;
    }
    //合并字符串
    void merge(node *p){
        node *nextp=p->next;
        if (p->size+nextp->size<=nodeSize){
            for (int pos = 0;pos<nextp->size;++pos,++pos->size)
                p->data[p->size]=nextp->data[pos];
            p->next = nextp->next;
            delete nextp;
        }
    }

public:
    linkString(const char *s="")//构造、析构
    linkString(const linkString &other);
    ~linkString;
    int length() const;//取字符串长度
    //赋值、运算符重载
    //取子字符串

    //插入子字符串
    void insert(int start,const linkString &s){
        node *p,*nextp,*tmp;
        int pos;
        if (start<0||start>len)
            return ;
        findPos(start,pos,p);

        split(p,pos);//分裂
        nextp=p->next;
        tmp=s.head->next;
        while (tmp){
```

```
            for (pos=0;pos<tmp->size;++pos){
                if (p->size==nodeSize)
                    p=p->next=new node(nodeSize);
                p->data[p->size]=tmp->data[pos];
                ++pos->size;
            }
            tmp=tmp->next;
        } //插入
        len+=s.len;
        p->next=nextp;
        merge(p);//归并
    }

    //删除子字符串
    remove(int start,int num){
        if (start<0||start>=len-1)
            return ;
        node *startp;
        int pos;
        findPos(start,pos,startp);
        split(startp,pos);//分裂起始节点
        if (start+num>=len){    //计算实际被删长度
            num=len-start;
            len=start;
        }
        else{
            len-=num;
        }
        while(true){
            node *nextp=start->next;
            if (num>nextp->size){
                num-=nextp->size;
                startp->next=nextp->next;
                delete nextp;//删除中间节点(整个)
            }
            else{//被删子串重点所在的节点
                split(nextp,num);//分裂终止节点
                startp->next=nextp->next;
                delete nextp;
                break;
            }
        }
        merge(startp); //归并起始和终止节点
    }
};
```

趁热打铁。

STL 字符串

轮子。

CH 6 树

树是n个结点的有限集合T，且满足

- 有一个被称之为根节点
- 其余结点可分为m个互不相交的集合$T_1,T_2,\cdots,T_m$
    - 子树，每棵子树也有结点

树：连通无回路的无向图

结点的度：子树的数目
叶子结点：度数为0的节点
内部节点：度数不为0的节点
儿子节点：子树的根节点
父亲节点：相对于儿子节点
兄弟节点：具有同一父亲的儿子节点
祖先节点：从根到该节点父亲的路径上的所有节点
子孙节点：该节点所有子树中的全部节点
节点的层次：其父亲节点的层次+1，根节点的层次为1
树的高度：一棵树中的最大层次
有序树：为所有节点规定了其儿子节点的次序
无序树：相对于有序树
森林：n棵互不相交的树的集合

建树`create()`：创建一棵空树
清空`clear()`
判空`isEmpty()`
找根节点`root()`
找父节点`parent()`
找子节点`child(x,i)`
剪枝`remove(x,i)`
遍历`traverse()`

```
template <class Type>
class tree{
public:
    virtual void clear() = 0;
    virtual bool isEmpty() const = 0;
    virtual T root(T flag) const = 0;
    virtual T parent(T x,T flag) const = 0;
    virtual T child(T x,int i, T flag) const = 0;
    virtual void remove(T x,int i) = 0;
    virtual void traverse() const = 0;
    virtual ~tree() { }
}
```

二叉树

二叉树（Binary Tree）

- 必须严格区分左右子树。
- 一定是有序的。

二叉树的五种基本形式

- 空二叉树
- 根和空的左右子树
- 根和左右子树
- 根和左子树
- 根和右子树

满二叉树（丰满树）

- 任意一层的节点个数都达到了最大值。
- 或一棵高度为k并且节点个数为2^k-1的二叉树。

完全二叉树

- 在满二叉树的最底层自右至左依次去掉若干个结点得到的二叉树。
- 所有的叶节点都出现在最低的两层上
- 对任一节点，如果其右子树的高度为k，则其左子树的高度为k或$k+1$。

二叉树的性质

1. 一棵非空二叉树的第i层上最多有2^{i-1}个结点($i \geq 1$)
   数学归纳法。
2. 一棵高度为k的二叉树，最多具有2^k-1个结点
   等比数列求和。
3. 对于一棵非空二叉树，如果叶子结点数为n_0，度数为2的节点数为n_2，则$n_0=n_2+1$
   设n为总节点数，$n=n_0+n_1+n_2$
   树枝数$B=n-1$
   $B=n_1+2n_2$
   $n_0=n_2+1$
4. 具有n个结点的完全二叉树的高度$k=\lfloor \log_2 n\rfloor +1$
   $2^{k-1}\leq n<2^k$
   $k-1 \leq \log_2 n <k$
5. 对于一棵有n个结点的完全二叉树的节点按程序自上而下每一层自左至右依次编号 对任一编号为i的节点：

- $i=1$，根节点
- $i>1$，父亲节点编号$\lfloor \frac{i}{2} \rfloor$
- 如果$2i>n$，则编号为i的节点为叶节点，无左儿子；否则，其左儿子的编号$2i$
- 如果$2i+1>n$，则编号为i的节点无右儿子；否则，其右儿子的编号为$2i+1$

```
template <class T>
class bTree{
public:
    virtual void clear() = 0;
```

```
    virtual bool isEmpty() const = 0;
    virtual T root(T flag) const = 0;
    virtual T parent(T x,T flag) const = 0;
    virtual T lchild(T x,T flag) const = 0;
    virtual T rchild(T x,T flag) const = 0;
    virtual void delLeft(T x) = 0;
    virtual void delRight(T x) = 0;
    virtual void preOrder() const = 0;
    virtual void midOrder() const = 0;
    virtual void postOrder() const = 0;
    virtual void levelOrder() const = 0;
    virtual ~bTree() {}
}
```

遍历方法：前序、后序、中序

根节点N、左子树L、右子树R

- 前序：NLR 如果二叉树为空，则操作为空； 否则访问根节点；前序遍历左子树；前序遍历右子树
- 中序：LNR 空；否则中序遍历左子树；访问根节点；中序遍历右子树
- 后序：LRN

前序+中序、后序+中序：唯一确定一棵二叉树

- 前序确定根A
- 中序按根分
- 左部前序确定根

前序+后序 x

层次遍历：从上到下、从左到右

机考：四五次

顺序存储结构：适用于二叉树上的节点个数已知
可以省略左右儿子字段（当且仅当完全二叉树）
最坏情况：右单支树 补上

特殊场合：

- 静态的
- 节点个数已知的完全二叉树或接近完全二叉树

二叉链表、三叉链表

标准形式： `left,data,right.`

广义形式： `left,data,right,parent.`

```
class BinaryTree : public bTree<T>{
    friend void printTree(const BinaryTree t);//打印树
private:
```

```
    struct Node{
        Node *left,*right;
        T data;
        Node(): left(NULL),right(NULL) {}
        Node(T item,Node* L=NULL,Node *R=NULL):data(item),left(L),right(R){ }
    }
    //存储树元素的二叉链表
    Node *root;//树根指针

    clear(Node *&t){
        if(t==NULL) return ;
        clear(t->left);
        clear(t->right);
        delete t;
        t=NULL;
    }
    //寻找指定节点
    //删除二叉树所有节点
    void preOrder(Node*t) const{
        if(t==NULL) return ;
        cout<<
        preOrder(t->left);
        preOrder(t->right);
    }

    //前序遍历二叉树
public:
    binaryTree(): root(NULL) {}
    binaryTree(const T & value){root = new Node(value);}
    ~binaryTree(){clear();}//构造/析构函数
    void clear() {clear(root);}//删除二叉树所有节点
    bool isEmpty() const {return root==NULL;}//判二叉树空否
    //寻找根/左右子树
    //删除左/右子树
    void preOrder() const{
        preOrder(root);
    }

    void levelOrder() const {
        linkQueue<Node *> que;
        Node *tmp;
        que.enQueue(root);
        while(!que.isEmpty()){
            tmp=que.deQueue();
            cout<<
            if(tmp->left) que.enQueue(tmp->left);
            if(tmp->right) que.ennQueue(tmp->right);
        }
    }
    //前中后序/层次遍历
    //创建完全二叉树 按照层次创建，队列
    void createTree(T flag){
        linkQueue<Node*> que;
        Node *tmp;
```

```
        T x,ldata,rdata;
        ....
        while(!que.isEmpty()){
            tmp=que.deQueue();
            cout<<tmp->data;
            cin>>ldata>>rdata;
            if(ldata!=flag)
                que.enQueue(tmp->left=new Node(ldata));
                ...
        }

    }
    //寻找父亲节点

}
```

最长连续子序列问题

情况一：整个位于前半部，可递归计算
情况二：整个位于后半部，可递归计算
情况三：从前半部开始但在后半部结束

分治法 子问题方法

$$ T(1)=1, T(n)=aT(\frac{n}{b})+cn\ T(n)=O(n\log n) $$

舍弃一单元 存一下

树的数学定义

CH 7

非递归实现

递归=> 优点：结构清晰、明了、美观
弱点：程序时间、空间的效率比较低

二叉树遍历=> 递归函数非常简洁，有时需要速度更快的非递归函数

- 设置一个栈，保存将要访问的树根
- 开始时，把二叉树的根节点存入栈中
- 然后重复以下过程，直到栈为空
    - 从栈中取出一个节点，输出根节点的值
    - 然后把**右子树**，**左子树**放入栈中

算法（前序）

1. 根节点进栈
2. 节点出栈，被访问
3. 节点的右、左儿子（非空）进栈
4. 反复执行第 2、3 步，直至栈空

```
template<class Type>
void BinaryTree<Type>::preOrder() const {
    linkStack<Node *> s;
    Node *current;

    s.push(root);
    while(!s.isEmpty()){
        current = s.pop();
        cout<< current->data;
        if(current->right!=NULL)
            s.push(current->right);
        if(current->left!=NULL)
            s.push(current->left);
    }

}
```

算法（中序）-- 这种算法可以减小 Stack Overflow 的情况

1. 根节点进栈，并计数
2. 如栈顶首次出栈，则再进栈，左儿子进栈
3. 如栈顶再次出栈，则被访问，右儿子进栈
4. 反复执行第2-4步，直至栈空

```
struct StNode{
    Node *Node;
    int TimesPop;
    StNode (...): ..., TimesPop(0) {}
};

template<class Type>
void BinaryTree<Type>::midOrder() const {
    linkStack<StNode> s;
    s.push(current);
    while(!s.isEmpty()){
        current = s.pop();
        if(++current.TimesPop==2){
            cout<<current.node->data;
            if(current.node->right!=NULL)
                s.push(StNode(current.node->right));
        }
        else{
            s.push(current);
            if(current.node->left!=NULL)
                s.push(StNode(current.node->left));
        }
    }
}
```

算法（后序）

1. 根节点进栈，并计数
2. 如栈顶首次出栈，则再进栈，左儿子进栈
3. 如栈顶再次出栈，则三进栈，右儿子进栈
4. 如栈顶三次出栈，则被访问
5. 反复执行第 2 - 4 步，直至栈空

```
struct StNode{
    Node *Node;
    int TimesPop;
    StNode (...): ..., TimesPop(0) {}
};

template<class Type>
void BinaryTree<Type>::backOrder() const {
    linkStack<StNode> s;
    s.push(current);
    while(!s.isEmpty()){
        current = s.pop();
        if(++current.TimesPop==3){
            cout<<current.node->data;
            continue;
        }
        s.push(current);
        if(current.TimesPop==1)
            if(current.node->left!=NULL)
                s.push(StNode(current.node->left));
        }
        else{
            if(current.node->right!=NULL)
                s.push(StNode(current.node->right));
        }
    }
}
```

企业眼中 ACM 班 > 博 士

表达式树

运算符

/
被操作数 操作数

使用栈可以 我们也可以用树 (不考虑括号)

顺序扫描中缀表达式

如果扫描到运算数，判当前表达式树存在否:

- 不存在，则表示第一个运算数，保存
- 存在，则将此运算数作为**前一个运算符**的右儿子

如果扫描到+或-

- 则将其作为根节点，原树作为其左子树

如果扫描到的是*或/，则与根节点比较

- 如根节点也是*或/，则将当前运算符作为根节点，原树作为其左子树。（原根节点应当先执行）
- 如根节点是+或-，则将其作为右子树的根，原右子树作为其左子树。（当前运算符应先执行）

在遇到运算数时，如何知道它前面的运算符是谁

- 这只需要判别根节点有没有右孩子
- 如没有右孩子，则运算数是根节点的右运算数
- 否则就是根节点右孩子的右运算数

(考虑括号) 遇到'('，则开始构建子树，遇到')'结束

```
class calc {
private:
    enum Type{DATA,ADD,SUB,MULTI,DIV,OPAREN,CPAREN,EOL};

    //指向树根的指针
    node *root;
    //节点的类型：运算数/运算符/括号等
    //节点的值：值、+-*/()
    //指针：左子树、右子树
    struct node {
        Type type;
        int data;
        node *lchild, *rchild;
        node(Type t,int d=0,node *lc=NULL/*左儿子*/,node *rc=NULL/*右儿子*/){
            type = t;int d=0;lchild=lc;rchild=rc;
        }
    };

    //从表达式构建一棵树
    //叶节点：运算数
    //非叶节点：运算符
    node *create(char *&s){
        node *p,*root=NULL;
        Type returnType;
        int value;
        while(*s){
            returnTYpe = getToken(s,value);
```

```
            switch(returnType){
                case DATA:case OPAREN:
                    if(returnType == DATA)
                        p=new node(DATA,value);
                    else
                        p=create(s);    //创建一个子表达式
                    if(root !=NULL)
                        if (root->rchild == NULL)
                            root->rchild=p;
                        else
                            root->rchild->rchild = p;
                    break;
                case CPAREN:case EOL:
                    return root;
                case ADD: case SUB:
                    if(root==NULL)
                        root=new node(returnType,0,p);
                    else
                        root=new node(returnType,0,root);
                    break;
                case MULTI:case DIV:
                    if(root==NULL)
                        root=new node(returnType,0,p);
                    else if (root->type==MULTI||root->type==DIV)
                        root=new node(returnType,0,root);
                    else
                        root->rchild = new node(returnType,0,root->rchild);
                    break;
            }
        }
        return root;
    }

    //从表达式中取出一个语法单位（运算符、运算数、表达式）
    Type getToken(char *&s,int &data){
        char type;
        while (*s=='') ++s;
        if(*s>='0'&&*s<=9){
            data = 0;
            while(*s>='0'&&*s<='9'){
                data=data*10+*s-'0';
                ++s;
            }
            return DATA;
        }
        if(*s=='\0')    return EOL;
        type=*s;++s;
        switch(type){
            case '+': return ADD;
            case '-': return SUB;
            case '*': return MULTI;
            case '/': return DIV;
            case '(': return OPAREN;
            case ')': return CPAREN;
```

```
            default: return EOL;
        }
    }

    int result (node *t){
        int num1,num2;
        if(t->type==DATA)
            return t->data; //叶子
        num1=result(t->lchild);
        num2=result(t->rchild);
        switch(t->type){
            case ADD: t->data = num1+num2;break;
            case SUB: t->data = num1-num2;break;
            case MULTI:
            case DIV:
        }
        return t->data;
    }

public:
    //带递归参数的构造函数
    calc(char *s){
    root = create(s);
    }

    //计算表达式结果
    int result(){
        if (root == NULL)
            return 0;
        return result(root);
    }

    //用后序遍历计算表达式结果
}
```

- 没有考虑表达式不正确的情况
- 没有考虑乘方运算

哈夫曼树与哈夫曼编码

ASCII 码

在计算机中每一个字符是用一个编码表示

大多数编码系统都采用等长编码

每个字符的编码都不可能是其他字符编码的前缀：

- 因为字符只放在叶节点
- 前缀编码可被唯一编码

哈夫曼树是一棵最小代价的二叉树，在这棵树上，所有字符都包含在叶节点上。

要使得整棵树的代价最小

- 权值大的叶子应当尽量靠近树根
- 权值小的叶子应当尽量远离树根

哈夫曼算法

给定一个具有n个权值${\omega_1,\omega_2,\cdots,\omega_n}$的节点集合$F={T_1,T_2,\cdots,T_n}$。初始时，设集合$A=F$

执行$i=1$至$n-1$次循环，在每次循环时，执行以下操作：

- 从A中选取权值最小、次最小的两个节点T_r,T_s
- 生成新的内部节点b_i，T_r,T_s分别为b_i的左右儿子，b_i的权值为其左右儿子权值之和$\omega_r+\omega_s$
- 在A中去除节点T_r,T_s，并将内部节点置入

CH 8

```
template<class Type>
class hfTree{
private:
    struct Node{
        Type data;
        int weight;
        int parent,left,right;
    };
    Node *elem;
    int length;
public:
    struct hfCode{
        Type data;
        string code;
    };
    hfTree(const Type *v,const int *w,int size){
        const int MAX_INT=32767;
        int min1,min2;
        int x,y;

        length = 2*size;//1+n-1+n
        elem = new Node[length];
        for(int i=size;i<length;++i){
            elem[i].weight=w[i-size];
            elem[i].data=v[i-size];
            elem[i].parent=elem[i].left=elem[i].right=0;
        }

        for(i=size-1;i>0;--i){
            min1=min2=MAX_INT;x=y=0;
            for(int j=i+1;j<length;++j)
                if(elem[j].parent==0)
                    if(elem[j].weight<min1){
```

```
                        min2=min1;min1=elem[j].weight;x=y;y=j;
                    }//确保元素j最小
                    else if(elem[j].weight<min2){
                        min2=elem[j].weight;x=j;
                    }//确保元素j次小,, 次小比最小大
            elem[i].weight=min1+min2;
            elem[i].left=x;elem[i].right=y;elem[i].parent=0;
            elem[x].parent=i;elem[y].parent=i;
        }
    }

    template<class Type>
    void getCode(hfCode result[]){
        int size = length/2;    //后一半
        int p,s;
        for(int i=size;i<length;++i){
            result[i-size].data=elem[i].data;
            result[i-size].code="";
            p=elem[i].parent;s=i;
            while(p){
                if(elem[p].left==s)
                    result[i-size].code='0'+result[i-size].code;
                else
                    result[i-size].code='1'+result[i-size].code;
                s=p;p=elem[p].parent;
            }
        }
    }
}
```

10个结点有9个儿子 $kn-(n-1)=(k-1)\times n +1$

儿子链表示法

- 存储数据元素值的数据部分
- 指向儿子链的指针

儿子兄弟链表示法

- 数据场
- 指向它的第一棵子树树根的指针场
- 指向它的兄弟节点的指针场

k叉树可以变成二叉树

双亲表示法

树的遍历

前序

树和森林

二叉树是一种结构最简单、结构最简便的树形结构

树可以使用儿子兄弟链表示法表示成二叉树的形式

存储森林中的每一棵树

将B_i作为B_{i-1}根节点的右子树

优先队列

- 队列
  - 结点间的关系是由进队的先后次序决定
- 优先级队列
  - 优先级决定

（线性结构）

- 方法一：
  - 进队：按照优先级在队列中寻找合适的位置，将新进队的元素插入在此位置。$O(N)$
  - 出队：直接从队首获取元素 $O(1)$
- 方法二：
  - 进队：保持不变 $O(1)$
  - 出队：查找优先级最高的元素出队 $O(N)$

（树）

二叉堆：对于给定的n个元素的序列$k_1,k_2,\cdots,k_n$称为堆

最小化堆 $k_i\leq k_{2i}~&~k_i\leq k_{2i+1} (i=1,2,\cdots,\lfloor \frac{n}{2}\rfloor)$

最大化堆 $k_i\geq k_{2i}~&~k_i\geq k_{2i+1} (i=1,2,\cdots,\lfloor \frac{n}{2}\rfloor)$

结构性 符合完全二叉树结构

有序性

可以采用顺序存储

$\log n$好

假设数值越大，优先级越高，则可以用一个最大化堆存储优先级队列

在最大化堆中，最大元素是根元素，而根节点永远存放在数组的下标为1的元素中：

- 读取操作：返回下标为1的数组元素值（队首）
- 出队操作：删除下标为1的数组元素中的值...
- 进队操作：在数组中天机一个元素...

如何保持结构性和有序性

进队操作：$\Theta (\log N)$

在堆中插入新元素

- 在最大序号的元素之后插入新的元素或节点 新节点向上移动：向上过滤（交换数据）
- 如果新元素放入后，没有违反堆的有序性，那么操作结束
- 否则，让该节点向父节点移动，直到满足有序性或到达根节点

2.6次比较 1.6层上移

内部不必完全有序

出队操作：$O(\log N)$

- 删除
- 空节点
- 把最后一项放在此空节点中，如果不满足堆的有序性
    - 把某些项放入空节点，然后移动空节点

向下过滤

0不用，结束后建堆$O(N\log N)\rightarrow O(N)$

```
template<class Type>
class priotityQueue: public queue{
private:
    //存储队列元素的数组
    int currentSize;
    //队列长度
    Type *array;
    //数组容量
    int maxSize;

    //数组扩容
    void doubleSpace();
    //建堆
    void buildHeap(){
        for(int i=currentSize/2;i>0;i--)    //从倒数第二层开始
            percolateDown(i);
    }
    //向下过滤
    void percolateDown(int hole){
        int child;
        Type tmp=array[hole];

        for(;hole*2<=currentSize;hole=child){    //为tmp找到了位置
            child=hole*2;//向下一层
            if(child!=currentSize&&array[child+1]>array[child]) //比较左右节点
                child++;
            if(array[child]>tmp)
                array[hole]=array[child];    //儿子节点上移
            else
                break;  //找到了位置
        }
```

```
            array[hole]=tmp;
        }
    public:
        //构造/析构函数
        priorityQueue(int capacity=100){
            array = new Type[capacity];
            maxSize = capacity;
            currentSize = 0;
        }

        priorityQueue(const Type *items,int size){
            //...
            buildHeap();
        }

        ~priorityQueue(){
            delete[] array;
        }
        //判队列空
        bool isEmpty() const{return currentSize==0;}
        //进队
        void enQueue(const Type &x){
            if(currentSize==maxSize-1)
                doubleSpace();

            //向上过滤
            int hole==++currentSize;
            //当前节点大于父节点，则父节点下移
            for(;hole>1&&x>array[hole/2];hole/=2)
                array[hole]=array[hole/2];
            array[hole]=x;

        }
        //出队
        Type deQueue(){
            Type maxItem;
            maxItem=array[1];
            array[1]=array[currentSize--];
            percolateDown(1);
            return maxItem;
        }
        //读队首
        Type getHead() const {return array[1];}
    };
```

堆是递归定义的

1. 左子堆递归调用 buildHeap
2. 右子堆递归调用 buildHeap
3. 对根节点调用 percolateDown 保持堆的有序性

非递归

叶子结点符合堆的定义

小堆建堆 再大堆

- 自下而上调整每一个子堆
- 在调整每一个堆的时候，假设除根以外，所有节点满足堆的定义
- 根节点的调整与删除一样，可以通过调用 PercolateDown 实现。

交换次数与高度有关。 建堆时间是$O(n)$

定理：对于一棵高度为h，包含了$N=2^h-1$个结点的满二叉树，所有节点的交换次数和为$N-h$

高度为1的节点有1个，... 高度为i的节点有2^{i-1}个。

$s=\sum_{i=1}^h 2^{i-1}(h-i)=(2^h-1)-h=N-h$

D堆

有D个儿子，堆更矮

- 插入$O(\log_D N)$
- 删除$O(D\log_D N)$ 要找D个元素中最小的或最大的。

外存

排队系统的模拟

不是单服务台，而是多服务台 多线程

先到新服务 但可能后离开

事件队列、等待队列、柜台

离开事件

```
while(/*事件队列非空*/){
    //队首元素出队；设置当前时间为该事件的发生时间；
    switch(/*事件类型*/){
        case /*到达*/:
        if(/*柜台有空*/){
            柜台数--;
            生成所需的服务时间
            事件类型改为离开
            设置离开事件发生时间
            重新存入事件队列
        }
        else
            将该事件存入等待队列
        case /*离开*/:
        if(/*等待队列非空*/){
            队首元素出队
            统计该顾客等待时间
            生成所需的服务时间
```

```
                时间类型改为离开
                设置时间发生时间
                存入队列
            }
            else
                空闲柜台数++
        }
        计算平均等待时间
        返回
    }
```

CH 9

集合关系

集合中的元素除同属一个集合外，没有任何逻辑关系

逻辑关系：因果关系、依赖关系

唯一表示：键值、关键字值

```
template<class KEY,class OTHER>
struct SET{
    KEY key;
    OTHER other;
};
```

集合的主要运算

- 添加/删除：添加/删除某一元素
- 查找：某一元素是否存在
- 排序：按元素的唯一标识（关键字）排序 --为了查找的方便

集合的存储

- 线性表
- 树

最适合的数据结构（仅适合于**集合**的存储和处理）

- 散列表：通过将关键字值直接映射到表中的某个位置；将该关键字对应的数据元素存储在这个位置中

元素1：关键字1 元素2：关键字2 … 元素K：关键字K

$O(1),K<<M$

查找的基本概念

- 查找是集合最基本的操作
    - 用于查找的集合称之为查找表

    - 查找是确定关键字的元素是否存在
    - 一般关键字是唯一的，如不唯一称为是多重集
- 查找表的分类
    - 静态查找表 → 集合是不变的（顺序表）
    - 动态查找表 → 集合是变化的（查找树、散列表）
    - 内部查找 → 集合存放在内训中（比较次数）
    - 外部查找 → 集合存放在外存中（访问次数）

静态查找表的定义

- 静态查找表
    - 给定某个数据元素的关键字X和数组A
    - 返回关键字值为X的数据元素在A中的位置（或不存在的标志）
- 如：词典

无序表的查找

无序表：数组中的元素是无序的

无序表的查找：只能做线性的顺序查找

查找表[0]不用。

```
template<class KEY,class OTHER>
int seqSearch(SET<KEY,OTHER> data[],int size,const KEY &x){
    data[0].key=x;
    for(int i=size;x!=data[i].key;--i)  //从后往前查找
    return i;
}
```

- 查找不成功
    - 没有要找的元素，`data[size~0]` $O(n)$
- 最好情况
    - `data[size]`就是要找到元素 $O(1)$
- 最坏情况
    - `data[size~1]` $O(n)$
- 平均情况
    - 平均查找时间为
        - $\sum_{i=n}^1 \frac{1}{n}(n-i+1)=\frac{n+1}{2}$ $O(n)$

综合考虑查找成功与否

$p(0\leq p\leq 1)$，不成功$1-p$

$ASL=\sum_{i=n}^1((n-i+1)p+n(1-p))$

$O(n)$

有序表查找

```
template<class KEY,class OTHER>
int seqSearch(SET<KEY,OTHER> data[],int size,const KEY &x){
    data[0].key=x;
    int i;
    for(i=size;x<data[i].key;--i);
    if(i==0||x!=data[i]) return 0;
    else return i;
}
```

二分查找

- 每次检查排在最中间的元素
- 如果中间元素与待查找的元素相等，则查找完成
- 否则，确定待查找的元素实在前一半还是后一半
- 缩小范围，在前一半或后一半内继续查找

算法：

- 数组A：递增序`A[i].key<=A[i+1].key`$(i=1,2,\cdots,n-1)$
- 查找范围：`low=1`至`high=n-1`
- 比较对象：中点元素 $\text{mid}=\lfloor (\text{low}+\text{high})/2\rfloor$
- **mid+1 mid-1**

$n=1,T(1)=1$ n为奇数，则中点元素的左右段各有$\frac{n-1}{2}$个元素 n为偶数，则中点元素的左段有$\lfloor \frac{n}{2}\rfloor-1$个元素，右段有$\lfloor \frac{n}{2}\rfloor$个元素

$T(n)=1+T(\lfloor \frac{n}{2} \rfloor)=k+T(\lfloor \frac{n}{2^k} \rfloor)=\lfloor \log_2 n \rfloor+1$

$k=\lfloor \log_2 n \rfloor$

$O(\log n)$

平均情况

丰满树 $n=2^t-1(t=1,2,3,\cdots)$

比较覆盖的节点数 -- 高度 2^t-1

$ASL=\sum_{i=1}^t\frac{2^{i-1}i}{n}=\log_2(n+1)\frac{n+1}{n}-1$

$ASL=\lceil\log_2(n)\rceil-1$

考虑成功与否

成功 n 种情况 不成功 n+1 种情况

$ASL=\frac{\sum_{i=1}^t 2^{i-1}i+t(n+1)}{2n+1}=\log_2n-\frac{1}{2}$

```
int low=1,high=size,mid;
while(low<=high){
```

```
    mid=(low+high)/2;
    if(x==data[mid].key) return mid;
    if(x<data[mid].key) high=mid-1;
    else low=mid+1;
}
```

插值查找

- 适用于数据分布比较均匀 $\verb"next"=\verb"low"+\frac{x-A[\verb"low"]}{A[\verb"high"]-A[\verb"low"]}(\verb"high"-\verb"low")$

$\verb"mid"=\verb"low"+\frac{1}{2}(\verb"high"-\verb"low")$

根据x所处的位置，让next更接近x

计算量大（浮点运算）

外存访问、相当均匀

分块查找

- 索引顺序块的查找

块内最大关键字 块起始地址

块与块之间有序

$ASL=\sqrt{n}+1$

STL 中的静态查找表

find binary_search

```
#include<iostream>
#include<vector>
#include<algorithm>
using namespace std;
int main(){
    vector<int> v;
    vector<int>::iterator itr;
    if(!binary_search(v.begin(),v.end(),13))
        //不存在
    else{
        itr=find(v.begin(),v.end(),13)
        cout<<*itr;
    }
}
```

动态查找表

```
template<class KEY,class OTHER>
class dynamicSearchTable{
public:
    virtual SET<KEY,OTHER> *find(const KEY &x) const = 0;
    virtual void insert(const SET<KEY,OTHER> &x) = 0;
    virtual void remove(const KEY &x)=0;
    virtual ~dynamicSearchTable() {}
};
```

二叉查找树： 空；

- 如果p的左子树非空，左小
- 右大
- p的左右都是二叉查找树

中序遍历是递增的

有序！与堆不同！

二叉查找树的存储实现

```
template<class KEY,class OTHER>
class BinarySearchTree(): public dynamicSearchTable<KEY,OTHER>{
private:
    //树根
    struct BinaryNode{
        SET<KEY,OTHER> data;
        BinaryNode *left;
        BinaryNode *right;

        BinaryNode(const SET<KEY,OTHER> &thedata, BinaryNode *lt =NULL,BinaryNode
*rt =NULL):data(thedata),left(lt),right(rt) { }
    };
    BinaryNode *root;
    //插入
    void insert(const SET<KEY,OTHER> &x,BinaryNode *&t){
        if(t==NULL)
            t=new BinaryNode(x,NULL,NULL);
        else if(x.key<t->data.key) insert(x,t->left);
        else if(t->data.key<x.key)
            insert(x,t->right);
    }
    //删除
    BinaryTree<KEY,OTHER> remove(const KEY &x,BinaryNode *&t){
        //叶子结点，无影响
        //有一个儿子，接到父亲上，保持同侧
        //有两个儿子，替身：左子树最大（最多只有左儿子）或右子树最小（最多只有右儿子），
在中序遍历中仅靠被删结点的节点（投影）

        if(t==NULL) return;//空树，返回
        if(x<t->data.key) remove(x,t->left);//被删结点<根节点 递归于左子树删除
```

```
        else if(t->data.key<x) remove(x,t->right);//被删结点>根节点 递归在右子树删除
        else //被删结点==根节点，判断根
            if(t->left!=NULL&&t->right!=NULL){
            //有两个儿子，右子树最小者替代根（删除根），最小者的右子树替代原来的位置（删除最小者）
            BinaryNode *tmp=t->right;
            while(tmp->left!=NULL) tmp=tmp->left;
            t->data=tmp->data;
            remove(t->data,t->right);}
        else {
            BinayNode *oldNode=t;
            t=(t->left!=NULL)?t->left:t->right;
            delete oldNode;
            //只有一个儿子，相应子树取代根（删除原根）
            //没有儿子（根为叶子），直接删除
        }
    }
    //查找
    SET<KEY,OTHER> find(const KEY &x,BinaryNode *t) const {
        if(t==NULL||t->data.key==x)
            return (SET<KEY,OTHER>) *t; //找到或不存在，强制类型转换
        if(x<t->data)
            return find(x,t->left);
        else
            return find(x,y->right);
    }
    //清空
   void makeEmpty(BinaryNode *t){
       if(t==NULL) return;
       makeEmpty(t->left);
       makeEmpty(t->right);
       delete t;
   }
public:
    //构造/析构
    BinarySearchTree() {
        root=NULL;
    }
    ~BinarySearchTree( {
        makeEmpty(root);
    }
    //插入
    //叶子结点
    void insert(const SET<KEY,OTHER> &x){
        insert(x,root);
    }
    //删除
    void remove(const KEY &x){
        remove(x,root);
    }
    //查找
    SET<KEY,OTHER> find(const KEY &x) const {
        return find(x,root);
```

```
    }
}
```

CH 10

二叉查找树的性能

访问代价 $O(\log N)~~~\Theta(N)$

- 平均情况分析（在成功查找的情况下）

一般情况，设$P(n,i)$为其左子树节点个数为i时的平均查找长度。 $P(n,i)=\frac{1+(P(i)+1)i+(P(n-i-1)+1)(n-i-1)}{n}$

$P(n)=\sum_{i=0}^{n-1}\frac{P(n,i)}{n}=-\frac{3n}{n+1}+2\sum_{i=1}^n \frac{1}{i}\leq 2(1+\frac{1}{n})\ln n \approx 1.38 \log_2 n=O(\log n)$

最优查找树 $\neq$ 最优二叉树（哈夫曼树）

AVL树的定义（平衡树）

避免最坏情况出现

某种平衡条件 To

- 容易维护
- 保证树的高度是 $O(\log N)$

二叉平衡查找树 AVL(Adelson-Velski Landis)

```
节点平衡因子（平衡度）= 其左子树高度 - 其右子树高度
```

每一个节点 +1 -1 0

高度 $\log_2(N+1)\leq h\leq1.44\log_2(N+1)-0.328$

$\max h=\sqrt{5}\log_\alpha(N+1)-2,\alpha=\frac{1+\sqrt{5}}{2}$

左 完全二叉树定理

右 构造一系列平衡树 $T_1,T_2,\cdots,T_H$，高度分别为$1,2,\cdots,H$，节点个数最少的二叉树，$h\leq H$

H	$# N$
1	1
2	2
3	4
4	7

$S(H)=S(H-1)+S(H-2)+1$

$N\geq S(H) = f(H+2) - 1 = \frac{\alpha^{H+2}-\beta^{H+2}}{\sqrt{5}}-1\approx \frac{\alpha^{H+2}}{\sqrt{5}}-1$

(当$H\geq 3,\beta项<0.09$)

```
class AvlTree{
private:
    //节点类
    struct AvlNode{
        SET<KEY,OTHER> data;
        AvlNode *left,*right;
        int height;
        AvlNode(const SET<KEY,OTHER>&element,AvlNode*lt,AvlNode*rt,int
h):data(element),left(lt),right(rt),height(h) {}
    };
    //AVL树根
    AvlNode *root;

    //旋转
    //LL  顺
    void LL(AvlNode *&t){
        AvlNode &t1 = t->left;
        t->left = t1->right;
        t1->right = t;
        t->height = max(height(t->left),height(t->right))+1;
        t1->height = max(height(t1->left),height(t))+1;
        t = t1;
    }
    //LR  逆RR 顺LL
    void LR(AvlNode *&t){
        RR(t->left);
        LL(t);
    }
    //RL  顺 逆
    void RL(AvlNode *&t){
        LL(t->right);
        RR(t);
    }
    //RR  逆
    void RR(AvlNode *&t){
        AvlNode &t1 = t->right;
        t->right = t1->left;
        t1->left = t;
        t->height = max(height(t->right),height(t->left))+1;
        t1->height = max(height(t1->right),height(t))+1;
        t = t1;
    }

    //插入

    //修改父节点平衡度，直到不必修改 原来为0，不可能失衡继续向上回溯，重新计算后合法，调
```

```
整结束
    //否则 不平衡了，调整树的结构
    //找到危机节点
    //以危机节点为根的子树高度应当保持不变
    void insert(const SET<KET,OTHER> &x,AvlNode *&t){
        if(t==NULL) t=new AvlNode(x,NULL,NULL);
        else if(x.key<t->data.key){
            insert(x,left);
            if(height(t->left)-height(t->right)==2)
                if(x.key<t->left->data.key) LL(t);
                else LR(t);
        }
        else if(t->data.key<x.key){
            insert(x,t->right);
            if(height(t->right)-height(t->left)==2)
                if(t->right->data.key<x.key) RR(t);
                else RL(t);
        }
        t->height = max(height(t->left),height(t->right))+1;
    }
    //删除
    // X 0 树高不变
    // 较高的子树上删去一个节点 树变矮了
    // 较矮的子树上删除一个节点 再平衡因子上相当于删除->插入
    bool remove(const KEY &x,AvlNode *&t){
        if(t==NULL) return true;
        if(x==t->data.key){
            if(t->left==NULL||t->right==NULL){
                AvlNode *oldNode = t;
                t = (t->left!=NULL)?t->left:t->right;
                delete oldNode;
                return false;
            }
            else{
                AvlNode *tmp = t->right;
                while(tmp->left!=NULL) tmp=tmp->left;
                t->data=tmp->data;
                if(remove(tmp->data.key,t->right)) return true;
                return adjust(t,1);
            }
        }
        if(x<t->data.key){
            if(remove(x,t->left))
                return true;
            return adjust(t,0);
        }
        else {
            if(remove(x,t->right))
                return true;
            return adjust(t,1);
        }
    }

    //清空
```

```
    void makeEmpty(AvlNode *t);
    //树高
    int height(AvlNode*t) const {return t==NULL?0:t->height;}

    //最大值
    int max(int a,int b){retunr (a>b)?a:b;}
    //调整
    bool adjust(AvlNode*&t,int subTree){
        if(subTree){    //右
            if(height(t->left)-height(t->right)==1) return true;
            if(height(t->right)==height(t->left)){
                --t->height;return false;
            }
            if(height(t->left->right)>height(t->left->left)){ LR(t); return
false;}
            LL(t);
            if(height(t->right)==height(t->left)) return false;
            else return true;
        }
        else{
            if(height(t->right)-height(t->left)==1) return true;
            if(height(t->right)==height(t->left)){--t->height; return false;}
            if(height(t->right->left)>height(t->right->right)){RL(t);return
false;}
            RR(t);
            if(height(t->right)==height(t->left)) return false;
            else return true;
        }
    }
public:
    //构造
    AvlTree(){ root=NULL; }
    //析构
    ~AvlTree(){makeEmpty(root);}
    //插入
    void insert(const SET<KEY,OTHER> &x){ insert(x,root); }

    //删除
    void remove(const KEY &x,AvlNode*&t){ remove(x,root); }

    //查找
    SET<KEY,OTHER> find(const KEY &x) const {
        AvlNode *t = root;

        while(t!=NULL && t->data.key!=x)
            if(t->data.key>x) t = t->left;
            else t=t->right;

        if(t==NULL) return NULL;
        else return (SET<KEY,OTHER>*) t;
    }

};
```

在同一层的多个子树可一次相差1层的高度，n棵子树最大高度差为$\log_2 n$

$m=\frac{h-1}{2}$

红黑树

根黑色 红色 其儿子都是黑的 任一节点到空节点的路径必须具有相同数量的黑色节点（空间点为黑色）

全黑 满二叉树 黑节点总在红节点后（除去根）

高度为$h(h\leq 2H)$，节点数为n

丰满树不可能有空节点（#1-#H） 2^H-1

$h\leq 2\log(n+1)$

```
class RedBlackTree:public dynamicSearchTree<KEY,OTHER> {
private:
    //节点类 节点颜色
    enum clourT{RED,BLACK};
    struct RedBlackNode{
        SET<KEY,OTHER> data;
        RedBlackTree
    };
    //树根

    //清空
    //旋转
    //插入后调整
    //删除后平衡

public:
    //构造
    //析构
    //插入
    //删除
    //查找
};
```

CH 11

父亲节点是红色的

父节点P的兄弟S是黑色的

- X是G的外侧节点：LL-B,RR-B
- X是G的内侧节点：LR-B,RL-B

父节点P的兄弟S是红色的

- 重新着色 消除连续红节点

调整：重新着色、旋转

删除

- 删除叶节点：红色直接删除 黑色
- 删除只有一个儿子的节点
- 删除有两个儿子的节点：找替身--删除1/0个儿子的情况

被删结点是红色：直接删除 被删结点时黑色： X有两个黑儿子

- T有两个黑儿子
- T有一个外侧的红儿子
- T有两个红儿子

如果X是黑根，虚拟父亲

X至少有一个红儿子

AA树

左孩子不能为红色的红黑树

优点

- 省去了一半的需要重构的情况
- 简化了重新平衡的过程

将指向红节点的链画成水平的（水平右链）

所有叶子的高度都是相同的

如节点是叶子，层次是1 如节点是红色的，它就与其父亲层次相同 如节点是黑色的，它就比其父亲的层次低一层

颜色没用了

- 水平链只能是右儿子链
- 不可能有两条连续的水平链
- 在层次2或2以上的节点必有2个儿子（黑节点数目）
- 如一个节点没有右水平链，则其两个儿子在同一层

```
template<class Type>
class AATree: public dynamicSearchTable<KEY,OTHER> {
private:
    struct AANode {
        Type data;
        AANode *left;
        AANode *right;
        ...
    };
};
```

插入红叶子

- 插入左孩子
- 插入右孩子

可以通过旋转调整

水平左链（LL旋转）

连续右链（RR旋转）

AA树的删除归结到删除一个第一层的节点

被删结点与父亲同层：直接删除

被删结点与父亲不同层

- 有一个儿子：将此儿子替代被删结点
- 无儿子（叶子）：会影响平衡，要调整

化归

伸展树

平衡树的缺陷：

平衡树 $\Theta(\log N)$

每个节点需要存储额外的平衡信息

实现复杂

- 插入和删除代价很大，容易出错

90-10规则

- 百分之九十的访问都是针对百分之十的数据
- 但平衡查找树没有利用这条规则的优势

均摊法分析（amortized analysis）限定了一系列操作的代价

- 不需要加倍，单操作代价$O(1)$
- 需要加倍，$O(N)$

$$