数据结构复习纲要

> ---NonoTion

第2章 程序性能分析&第3章渐进记法

渐进记法

大O记法

f(n)=O(g(n)),f(n)渐进小于g(n)

渐进记法Ω和Θ

Ω渐进大于

Θ渐进等于

几种排序算法的性能分析

选择排序

```
template<class T>
void selectionSort(T a[],int n)
{
    for(int size=n;size>1;size--)
    {
        int j=indexOfMax(a,size);
        swap(a[size-1],a[j]);
    }
}
```

寻找最大值耗时 n+n-1+...+1=n(n+1)/2

时间复杂度O(n^2)

冒泡排序

一次冒泡过程

```
template<class T>
    void bubble(T a[],int n)
{
    for(int i=0;i<n-1;i++)
    {
        if(a[i]>a[i+1])
        swap(a[i],a[i+1]);
    }
}
```

冒泡排序

```
template<class T>
    void bubbleSort(T a[],int n)
{
    for(int i=n;i>1;i--)
    {
        bubble(a,i);
    }
}
```

O(n^2)

插入排序

```
template<class T>
    void insertionSort(T a[],int n)
{
    for(int i=0;i<n;i++)
    {
        T t=a[i];
        for(int j=i-1;j>=0&&t<a[j];j--)
        {
            a[j+1]=a[j];
        }
        a[j+1]=t;
    }
}
```

第5章 线性表的数组描述

数据对象和数据结构

数据对象（data object）是一组实例或值

数据结构（data structure）是一个数据对象，同时这个对象的实例和构成实例的元素都存在着联系，而且这些联系由相关的函数来规定

线性表数据结构

线性表，又叫有序表

除了元素之间由先后关系外，线性表不再有其他关系

抽象类 linearList

一个线性表的抽象类中应该有这些函数：

- 判空
- 返回线性表长度
- 根据索引查找元素
- 给定索引插入元素
- 给定索引删除元素
- 给定元素查找索引
- 从左到右输出线性表

数组描述

映射关系：

- location(i)=i 从索引为0开始存放
- location(i)=arrayLength-i-1 从表尾向表头存放
- location(i)=(i+x)%arrayLength 从索引为x地方开始，循环存放数据

变长一维数组

```
template<class T>
    void changeLength1D(T*& a,int oldLength,int newLength)
{
    if(newLength <0)
        throw illegalParameterValue("new length must be >= 0");
    T* temp=new T[newLength];
```

```
    int number=min(oldLength,newLenth);
    copy(a,a+number,temp);
    delete a;
    a=temp;
}
template<class T>
void copy(T* start, T* end, T* dest) {
    while (start != end) {
        *dest = *start;
        ++dest;
        ++start;
    }
}
```

类arrayList

类arrayList的类定义

```
template<class T>
class arrayList:public linearList<T>
{
public :
    arrayList(int initialCapacity=10);
    arrayList(const arrayList<T>&);
    ~arrayList(){delete []element};

    bool empty(){return listSize==0;}
    int size()const{return listSize;}
    T& get(int theIndex) const;
    int indexOf(const T& theElement)const;
    void erase(int theIndex);
    void insert(int theIndex,const T& theElement);
    void output(ostream &out)const;
    int capacity()const {return arrayLength;}
    protected:
    void checkIndex(int theIndex)const;
    int arrayLength;
    int listSize;
}
```

构造函数

```
template<class T>
    arrayList<T>::arrayList(int initialCapacity)
```

```
    {
        if(initialCapacity<1)
        {
            ostringstream s;
            s<<"InitialCapacity = "<<initialCapacity<<" Must be >= 0";
            throw ilegalParameterValue(s.str());
        }
        arrayLenth=initialCapacity;
        element=new T[arrayLenth];
        listSize=0;
    }
template<class T>
    arrayList<T>::arrayList(const arrayList<T>& theList)
    {
        arrayLength=theList.arrayLength;
        listSize=arrayList.listSzie;
        element=new T[arrayLength];
        copy(theList,theList+listSzie,element);
    }
```

其他功能实现

```
template<class T>
void arrayList<T>::checkIndex(int theIndex) const
{
    if(theIndex<0||theIndex>=listSize)
    {
        ostringstream s;
        s<<"index = "<<theIndex<<" size = "<< listSize;
        throw lilegalIndex(s.str());
    }
}
template<class T>
T& arrayList<T>::get(int theIndex)
{
    checkIndex(theIndex);
    return element[theIndex];
}
template<class T>
void arrayList<T>::erase(int theIndex)
{
    checkIndex(theIndex);
    copy(element+theIndex+1,element+listSize,element+theIndex);
```

```
    element[--listSize].~T();
}
template<class T>
    void arrayList<T>::insert(int theIndex,const T& theElement)
    {
        if(theIndex<0||theIndex>listSize)
        {
            ostringstream s;
            s<<"index =<< theIndex<<" size = "<<listSize;
            throw illegalIndex(s.str());
        }
    if(listSize==arrayLength)
    {
        changeLength1D(element,arrayLength,2*arrayLength)
       arrayLength*=2;
    }

 copy_backward(element+theIndex,element+listSize,element+listSize+1);
        element[theIndex]=theElement;
        listSize++;
    }
template<class T>
void copy_backward(T* start, T* end, T* dest_end) {
    while (start != end) {
        --end;
        --dest_end;
        *dest_end = *end;
    }
}
```

在一个数组中实现的多重表

空间的利用率更高，但在最坏的情况下，插入操作将耗费更多的时间。

第6章 链式描述

单向链表

在链式描述中，每个节点都包含一个相关节点的未知信息

链表节点结构

```
template<class T>
```

```
struct chainNode
{
  T element;
    chianNode<T>* next;
    chainNode()
    {
       next=nullptr;
    }
    chainNode(const T& theElement)
    {
        element=theElement;
        next=nullptr;
    }
    chainNode(const T& theElement,chianNode<T> *next)
    {
        element=theElement;
        this->next=next;
    }
};
```

类chain

```
template<class T>
class chian:public linearList
{
  public:
    chain(int initialCapacity=10);
    chain(const chain<T>&);
    ~chain();

    bool empty(){return listSize==0;}
    int size() const {return listSize;}
    T& get(int theIndex) const;
    int indexof(const T& theElement) const;
    void erase(int theIndex,const T& theElement);
    void output(ostream & out)const;
    protected:
    void checkIndex(int theIndex) const;
    chainNode<T>* firstNode;
    int listSize;
};
```

构造函数和析构函数

```
template<class T>
```

```
chain<T>::chain(int initialCapacity)
{
    if(initialCapacity<1)
    {
        ostringstream s;
        s<<"Initial Capacity = "<<initialCapacity<<" Must be > 0";
    }
    firstNode=nullptr;
    listSize=0;
}
template<class T>
    chain<T>::chain(const chain<T> &theList)
    {
        listSize=theList.listSize;
        if(listSize==0)
        {
            firstNode=nullptr;
            return;
        }
        chainNode<T>* sourceNode=theList.fistNode;
        firstNode=new chainNode<T>(sourceNode->element);
        sourceNode=sourceNode->next;
        chainNode<T>* targetNode=firstNode;
        while(sourceNode!=nullptr)
        {
            targetNode->next=new chainNode<t>(sourceNode->element);
            targetNode=targetNode->next;
            sourceNode=sourceNode->next;
        }
        targetNode->next=nullptr;
        return;
    }
chain<T>::~chain()
{
    while(firstNode!=nullptr)
    {
        chainNode<T>* nextNode=firstNode->next;
        delete firstNode;
        firstNode=nextNode;
    }
}
```

方法get

```
template<class T>
T& chain<T>::get(int theIndex) const
{
    checkIndex(theIndex);
    chianNode<T>* currentNode=firstNode;
    for(int i=0;i<theIndex;i++)
    {
        currentNode=currentNode->next;
    }
    return currentNode->element;
}
```

方法indexOf

```
template<class T>
int chain<T>::indexOf(const T& theElement) const
{
    chainNode<T>* currentNode=firstNode;
    int index=0;
    while(currentNode!=nullptr&&currentNode-
>element!=theElement)
    {
        currentNode=currentNode->next;
        index++;
    }
    if(currentNode==nullptr)
        return -1;
    else
        return index;
}
```

方法erase

```
template<class T>
void chain<T>::erase(int theIndex)
{
    checkIndex(theIndex);
    chainNode<T>* deleteNode=firstNode,
                * p=nullptr;
    for(int i=0;i<theIndex;i++)
    {
        p=deleteNode;
        deleteNode=deleteNode->next;
```

```
    }
    if(deleteNode==firstNode)
    {
        firstNode=firstNode->next;
    }
    else
    {
        p->next=deleteNode->next;
    }
    delete deleteNode;
    listSize--;
}
```

方法insert

```
template<class T>
void chain<T>::insert(int theIndex,const T& theElement)
{
    if(theIndex<0||theIndex>listSize)
    {
        ostringstream s;
        s<<"index = "<<theindex<<" listSize = "<<listSize;
        throw ilegelIndex(s.str());
    }
    if(theIndex=0)
        firstNode=new chainNode<T>(theElement,firstNode);
    else
    {
        chainNode<T>* p=firstNode;
        for(int i=0;i<theIndex-1;i++)
        {
            p=p->next;
        }
        p=new chainNode<T>(theElement,p->next);
    }
    listSize++;
}
```

输出链表

```
template<class T>
    void chain<T>::output(ostream& out)
    {
        for(chainNode<T>*
currentNode=firstNode;currentNode!=nullptr;
```

```
            currentNode=currentNode->next)
        {
            out<<currentNode->element<<" ";
        }
    }
template<class T>
    ostream& operator<<(ostream& out,const chain<T> &x)
{
    x.output(out);
    return out;
}
```

链表的成员类iterator

```
class iterator
{
    public :
    typedef T value_type;
    typedef T& reference;
    typedef T* pointer;
    typedef std::forward_iterator_tag iterator_category;
    typedef ptrdiff_t difference_type;
    iterator(chainNode<T>* theNode=nullptr)
    {
        node=theNode;
    }
    T& operator*()const{return node->element;}
    T& operator&()const{return node->element;}
    iterator operator++(int)
    {
        iterator old=*this;
        node=node->next;
        return old;
    }
    iterator& operator++()
    {
        node=node->next;
        return *this;
    }
    bool operator!=(const iterator right)const
    {
        return node!=right.node;
    }
    bool operator==(const iterator right)const
```

```
    {
        return node==right.node;
    }
    protected:
    chainNode<T>*node;
}
```

类extendedChain

类chain中增加一个尾指针域lastNode，指向最后一个节点

循环链表和头节点

将最后一个元素的指针指向头节点

头节点：初始节点，不存放数据，next指针指向第一个节点

```
template<class T>
circularListWithHeader<T>::circluarListWithHeader()
{
    //构造函数
    headerNode=new chainNode<T>();
    headerNode->next=headerNode;
    listSize=0;
}
template<class T>
int circularListWithHeader<T>:: indexOf(const T& theElement)
const
{
    headerNode->element=theElement;
    chainNode<T>* currentNode=headerNode->next;
    int index=0;
    while(currentNode->element!=theElement)
    {
        currentNode=currentNode->next;
        index++
    }
    if(currentNode==headerNode)
        return -1;
    else
        return index;
}
```

双向链表

有next指针和previous指针，previous指针指向前一个节点

应用

箱子排序

用箱子排序堆学生的成绩进行排序

结构studentRecord

```
struct studentRecord
{
    int score;
    string *name;
    operator int() const {return score;}
};
ostream &operator<<(ostream &out,const studentRecord &x)
{
    out<<x.score<<" "<<*x.name<<endl;
    return out;
}
```

```
void binSort(chain<studentRecord>& theChain,int range)
{
    chain<studentRecord> *bin;
    bin=new chain<studentRecord>[range+1];
    int numberOfElements=theChain.size();
    for(int i=0;i<numberOfElements;i++)
    {
        studentRecord x=theChain.get(0);
        theChain.erase(0);
        bin[x.score].insert(0,x);
    }
    for(int j=range;j>=0;j--)
    {
        while(!bin[j].empty())
        {
            studentRecord x=bin[j].get(0);
            bin[j].erase(0);
            theChian.insert(0,x);
        }
    }
    delete[] bin;
```

```
}
```

基数排序

并查集

```
int *equivClass,n;
void initialize(int numberOfElements)
{
    n=numberOfElements;
    equivClass=new int[n+1];
    for(int e=1;e<=n;e++)
    {
        equivClass[e]=e;
    }
}
void unite(int classA,int classB)
{
    for(int k=1;k<=n;k++)
    {
        if(equivClass[k]==classB)
        {
            equivClass[k]=classA;
        }
    }
}
int find(int theElement)
{
    return equivClass[theElement];
}
```

链表实现

结构equivNode

```
struct equivNode
{
    int equivClass,size,next;
};
```

```
equivNode *node;
int n;
void initialize(int numberOfElements)
{
    n=numberOfElements;
```

```
    node=new equivNode[n+1];
    for(int e=1;e<=n;e++)
    {
        node[e].equivClass=e;
        node[e].next=0;
        node[e].size=1;
    }
}
void unite(int classA,int classB)
{
    if(equivNode[classA].size>equivNode[classB].size)
        swap(classA,classB);
    int k;
    for(k=classA;node[k].next!=0;k=node[k].next)
        node[k].equivClass=classB;
    node[k].equivClass=classB;
    node[classB].size+=node[classA].size;
    node[k].next=node[classB].next;
    node[classB].next=classA;
}
int find(int theElement)
{
    return node[theElement].equivClass;
}
```

第7章 数组和矩阵

行主映射和列主映射

把数组的一个索引$[i_1][i_2]\dots[i_k]$映射为$[0,n-1]$范围中的一个数map($i_1,i_2,\dots,i_k$)

当数组维数为1时，

$$map(i)=i$$

当数组维数为2时，第一个索引相同的为同一行，第二个索引相同的为同一列

行主映射：每一行从左到右连续编号 u2为列数

$$map(i_1,i_2)=i_1u_2+i_2$$

列主映射：每一列从上到下连续编号

$$map(i_1,i_2)=i_1+i_2u_1$$

当数组维数为3时，

行主映射：

$$map(i_1,i_2,i_3)=i_1u_2u_3+i_2u_3+i_3$$

列主映射：

$$map(i_1,i_2,i_3)=i_1+i_2u_1+i_3u_2u_1$$

矩阵

一个mxn矩阵是一个m行，n列的表

类martix

```
template<class T>
class martix
{
    friend ostream& operator<<(ostream&,const martix<T>&);
    public:
    //构造方法，析构方法，加法减法乘法，返回各种属性的方法
    private:
    int theRows;
    int theColumns;
    T *element;
};
```

构造函数和复制构造函数

```
template<class T>
martix<T>::martix(int theRows,int theColumns)
{
    //验证行数和列数不能小于0，必须同时为0
    //创建矩阵
    this->theRows=theRows;
    this->theColumns=theColumns;
    element=new T[theRows*theColumns];
}
template<class T>
    martix<T>::martix(const martix<T>& m)
    {
        theRows=m.theRows;
        theColumns=m.theColumns;
        element=new T[theRows*theColumns];
        copy(m.element,m.element+theRows*theColumns,element);
```

```
    }
```

重载赋值操作符=

```
template<class T>
martix<T>& martix<T>::operator=(const martix<T>&m)
{
    if(this!=&m)
    {
        delete[] element;
        theRows=m.theRows;
        theColumns=m.theColumns;
        element=new T[theRows*theColumns];
        copy(m.element,m.element+theRows*theColumns,element);
    }
    return *this;
}
```

矩阵类martix对()操作符的重载

```
template<class T>
T& martix<T>::operator()(int i,int j) const
{
    //检查索引是否合法
    return element[(i-1)*theColumns+j-1]
}
```

矩阵加法

```
template<class T>
martix<T> martix<T>::operator+(const martix<T>& m)const
{
    //检查矩阵行列是否相等
    martix<T> w(theRows,theColumns);
    for(int i=0;i<theRows,theColumns;i++)
        w.element[i]=element[i]+m.element[i];
    return w;
}
```

矩阵乘法

```
template<class T>
martix<T> martix<T>::operator*(const martix<T>& m)const
{
    //检查矩阵是否能进行乘法
```

```
    martix<T> w(theRows,m.theColumns);

    int ct=0,cm=0,cw=0;
    for(int i=1;i<=theRows;i++)
    {
        for(int j=1;j<=m.theColumns;j++)
    {
        T sum=element[ct]*m.element[cm];
        for(int k=2;k<=theColumns;k++)
        {
            ct++;
            cm+=m.theColumns;
            sum+=element[ct]*m.element[cm];
        }
        w.element[cw++]=sum;
        ct-=theColumns-1;
        cm=j;
    }
        ct+=theColumns;
        cm=0;
    }
    return w;
}
```

特殊矩阵

特殊矩阵一般都是方阵

对角矩阵

用一个theRows个元素的数组表示

映射公式

$$map(i,j)=\begin{cases}0 & \text{i}\neq j\\ 1 & \text{i}=j\end{cases}$$

三对角矩阵

主对角线的元素 i=j

主对角线上的对角线元素i=j+1

主对角线下的对角线元素i=j-1

用一个3*theRows-2个元素的数组存储

```
swtich(i-j)
{
    case 1: //下对角线
        element[i-2];
    case 0://对角线
        element[n+i-2];
    case -1://上对角线
        element[2*n+i-2]
}
```

三角矩阵

用n*n/2个元素的数组存放

```
if(i>=j)
    element[i*(i-1)/2+j-1]
else
    0
```

对称矩阵 视作上三角或下三角矩阵

稀疏矩阵

用单个线性表描述

线性表的元素包含矩阵非零行号，列号和存储的数值

用多个线性表描述

firstNode表示头节点，有元素的列生成一个节点

第8章 栈

定义

栈(stack):是一种特殊的线性表，其插入(也叫入栈和压栈)和删除(也叫出栈和弹栈)操作都在表的同一端进行。这一端称为**栈顶(top)**，另一端称为**栈底(bottom)**。

后进先出 (LIFO)

抽象类栈

```
template<class T>
class stack
{
public:
virtual ~stack(){};
virtual int size() const =0;
virtual bool empty()const=0;
virtual T& pop()=0;
virtual void push(const T& theElement)=0;
};
```

数组描述

```
template<class T>
class arrayStack:public stack<T>
{
    public:
    //很多方法与arrayList类似，这里不再赘述
    T& pop()
    {
        if(stackTop==-1)
            throw stackEmpty();
        return stack[stackTop];
    }
    void pop()
    {
       if(stackTop==-1)
           throw stackEmpty();
        stack[stackTop--].~T();
    }
    void push(const T& theElement)
    {
        if(stackTop==arrayLength-1)
        {
            changLength1D(stack,arrayLength,arrayLength*2);
            arrayLength*2;
        }
        stack[++stackTop]=theElement;
    }
    private:
    int stackTop;
```

```
    int arrayLength;
    T *stack;
}
```

链表描述

```
template<class T>
class linkedStack:public stack<T>
{
    public:
    T& top()
    {
        if(stackSize==0)
            throw stackEmpty();
        return stackTop->element;
    }
    T& pop()
    {
        if(stackSize==0)
            throw stackEmpty();
        else
        {
            chainNode<T>* deleteNode=stackTop;
            stackTop=stackTop->next;
            delete deleteNode;
            stackSize--;
        }
    }
    T& push(const T& theElement)
    {
        stackTop=new chainNode<T>(theElement,stackTop);
    }
    private:
    chainNode<T>* stackTop;
    int stackSize;
}
```

应用

括号匹配

汉诺塔

列车轨道重排

开关布线盒

离线等价类

第9章 队列

数组描述

映射公式

$$location(i) = i$$

队列为空时，queueBack=-1,queueFront=-1,队列长度=queueBack+1

插入O(1),删除O(n)

$$location(i) = location(\text{队首元素}) + i$$

删除 O(1)，插入最坏情况变为O(arrayLength)

$$location = (loacation(\text{队首元素}) + i)$$

插入和删除都是O(1)

但判空和判满条件冲突，插入一个元素之前，查看是否会使队列变满，如果是，要先延长队列

```
T *newQueue =new T[2*arrayLength];
int start=(theFront+1)%arrayLength;
if(start<2)
    //没有形成环
copy(queue+start,queue+start+arrayLength-1,newQueue);
else
{//队列成环
    copy(queue+start,queue+arrayLength,newQueue);
    copy(queue,queue+theBack+1,newQueue+arrayLength-start);
}
theFront=2*arrayLength-1;
theBack=arrayLength-2;
arrayLength*=2;
delete[] queue;
queue=newQueue;
```

链表描述

插入和删除操作

```
template<class T>
void linkedQueue<T>::push(const T& theElement)
{
    chainNode<T>* newNode=new chainNode<T>(theElement,nullptr);
    if(queueSize==0)
    {
        queueFront=newNode;
    }
    else queueBack->next=newNode;
    queueBack=newNode;
    queueSize++;
}
template<class T>
    void linkedQueue<T>::pop()
    {
        chainNode<T>* deleteNode=queueFront;
        if(queueFront=queueBack)
        {
            queueFront=queueBack=nullptr;
        }
        else
        {
            queueFront=queueFront->next;
        }
        delete deleteNode;
        queueSize--;
    }
```

第10章 跳表和散列

跳表

通过维护多个链表，降低链表插入，删除，查询的时间复杂度，O(logn)

i级链表每2^i个数对选一个

散列表

用于一个**散列函数（hash函数）**把字典的数对映射到一个**散列表（哈希表)**

散列表的每一个位置叫做一个**桶(bucket)**;关键字为k的数对，**f(k)是起始桶(home bucket)**

除法散列函数是最常见的散列函数

冲突和溢出

冲突：关键字为k的元素插入时，起始桶不为空

溢出：起始桶满了，不能再插入新元素

解决溢出的方法：

1. **线性探查**

如果起始桶被占用，继续向下寻找可以使用的桶，已经使用的桶只能存储相同关键字的元素

2. **链式散列**

用链表数组来描述散列表

线性探查和链式散列的优缺点分析

线性探查的优缺点：

优点：
不需要额外的空间（如指针、链表、溢出区）
探测序列具有局部性，可以利用系统缓存，减少I/O
缺点：
插入和查找可能需要多次探测，耗费时间
容易产生“聚集”现象，即多个哈希地址不同的关键字争夺同一个后继哈希地址
链式散列的优缺点

优点：
不采用连续的存储空间，内存空间利用率比较高
插入或删除元素时，不需要移动大量的元素
缺点：
需要额外的空间来表达数据之间的逻辑关系
不支持下标访问和随机访问
总的来说，线性探查在空间使用上有一定的优势，但在处理冲突时可能需要多次探

测。而链式散列在处理冲突时效率较高，但需要额外的空间来存储数据之间的逻辑关系。

跳表和散列的优缺点分析

跳表和哈希表都是非常重要的数据结构，它们各自有其优点和缺点。

跳表的优缺点：

优点：
原理简单，实现简单，方便扩展，效率更高。
插入和删除操作只需要修改相邻节点的指针，操作简单又快速
适合做范围查找，比平衡树操作要简单
平均每个节点包含的指针数目可以通过参数调整，具有一定的空间优势
缺点：
需要额外维护多个链表，占用额外的空间
插入、删除时间复杂度为O(logn)，插入和删除会导致索引不工整
哈希表的优缺点

优点：
插入和查找速度快，时间复杂度为O(1)
缺点：
扩展性差，需要提前预测数据量的大小
不能有序遍历数据
当数组被填满后，性能会受到很大的影响
总的来说，哈希表更适用于数据的插入和删除，对遍历则相对来说较为劣势。而跳表则在空间使用和范围查找上有一定的优势。

第11章 二叉树和其他树

树

一棵树t是一个非空的有限元素的集合,其中一个元素为**根(root)**,其余的元素(如果有的话)组成t的子树

节点之间的关系:

- 父母 parent
- 孩子 child
- 祖先 ancestor
- 后代 descendent
- 兄弟 sibling

级(level):树结构的一层就是一级

一棵树的**高度(height)**或**深度(depth)**就是树的级数

一个元素的**度(degree)**是指其孩子节点的个数

一棵树的**度**是其元素的度的最大值

二叉树

一棵二叉树(binary tree) t 是有限个元素的集合(可以为空),二叉树为非空时,其中有一个元素为根,其他元素被划分为两棵二叉树,分别称为t的左子树和右子树

二叉树和树的**根本区别**是:

- 二叉树的每个元素都恰好有两棵子树(可能为空)
- 二叉树中,每个元素的子树都是有序的,有左子树和右子树之分
- 二叉树可以为空,树不能为空

二叉树的**性质**:

- 一棵二叉树有n个元素,n-1条边
- 一棵二叉树的高度为h,最少有h个元素,最多有$2^h - 1$个元素
- 一棵二叉树有n个元素,高度最高为n,最低为log_2^{n+1}

满二叉树(full binary tree): 恰好有$2^h - 1$元素的树

完全二叉树(complete binary tree): 除最低一级,每一级的元素都是满的二叉树

完全二叉树的**特性:**

设完全二叉树的一元素的编号为i,$1 \leq i \leq n$有一下关系成立

- 如果i=1,则该元素为二叉树的根.若i>1,该元素父节点的编号为$\lceil \frac{i}{2} \rceil$
- 如果2i>n,则该节点无左孩子.否则,其左孩子编号为2i
- 如果2i>n+1,则该节点无右孩子.否则,其右孩子编号为2i+1

二叉树的描述

数组描述

把二叉树看作缺少了部分元素的完全二叉树,对元素进行编号存储在数组中

链表描述

每个元素用一个节点结构表示,每个节点都有两个指针leftChild,rightChild指向它的左右子树,有一个element域存储数据

```
template<class T>
    struct binaryTreeNode
    {
        T element;
        binaryTreeNode<T> *leftChild,*rightChild;
        binaryTreeNode() {leftChild=rightChild=nullptr;}
        binaryTreeNode(const T& theElement)
        {
            element(theElement);
            leftChild=rightChild=nullptr;
        }
binaryTreeNode(const T&theElement,binaryTreeNode
*theLeftChild,binaryTreeNode *theRightChild)
{
    element(theElement);
    leftChild=theLeftChild;
    rightChild=theRightChild;
}
    }
```

二叉树的常用操作

- 确定高度
- 确定元素数目
- 复制
- 显示或打印二叉树
- 确定两棵二叉树是否一致
- 删除整棵树

这些操作可以通过遍历二叉树来完成

二叉树的遍历

前序遍历

```
template<class T>
    void preOrder(binaryTreeNode<T>*t)
{
   if(t!=nullptr)
   {
       visit(t);
       preOrder(t->leftChild);
       preOrder(t->rightChild);
   }
}
```

中序遍历

```
template<class T>
    void inOrder(binaryTreeNode<T>*t)
{
   if(t!=nullptr)
   {
       inOrder(t->leftChild);
       visit(t);
       inOrder(t->rightChild);
   }
}
```

后续遍历

```
template<class T>
void postOrder(binaryTreeNode<T>*t)
{
   if(t!=nullptr)
   {
       postOrder(t->leftChild);
       postOrder(t->rightChild);
       visit(t);
   }
}
```

层次遍历

```
template<class T>
void levelOrder(binaryTreeNode<T> *t)
{
    arrayQueue<binaryTreeNode<T>*> q;
    q.push(t);
```

```
    while(!q.empty())
    {
        binaryTreeNode<T>* currentNode;
        currentNode=q.front();
        q.pop();
        visit(currentNode);
        if(currentNode->leftChild!=nullptr) q.push(currentNode->leftChild);
        if(currentNode->rightChild!=nullptr) q.push(currentNode->rightChild);
    }
}
```

四种遍历所需的时间复杂度都是O(n) 最坏情况下 递归栈空间为O(n)

抽象数据类型BinaryTree

抽象类

```
template<class T>
    class binaryTree
    {
        public :
        virtual ~binaryTree();
        virtual bool empty();
        virtual int size() const=0;
        virtual void preOrder(vois(*)(T *))=0;
        virtual void inOrder(vois(*)(T *))=0;
        virtual void postOrder(vois(*)(T *))=0;
        virtual void levelOrder(vois(*)(T *))=0;
    }
```

类linkedBinaryTree

```
template<class T>
class linkedBinaryTree:public binaryTree<binaryTreeNode<T>>
{
    public :
    linkedBinaryTree(){root=nullptr;treeSize=0;}
    ~linkedBinaryTree(){erase();}
    bool empty() const{return treeSize==0;}
    int size()const{return treeSize;}
    void preOrder(void(*theVisit)(binaryTreeNode<T>*))
    {
```

```
        visit=theVisit;
        preOrder(root);
    }
    void inOrder(void(*theVisit)(binaryTreeNode<T>*))
    {
        visit=theVisit;
        inOrder(root);
    }
    void postOrder(void(*theVisit)(binaryTreeNode<T>*))
    {
        visit=theVisit;
        postOrder(root);
    }
    void levelOrder(void(*theVisit)(binaryTreeNode<T>*))
    {
        visit=theVisit;
        levelOrder(root);
    }
    void erase()
    {
        postOrder(dispose);
        root=nullptr;
        treeSize=0;
    }
    private:
    binaryTreeNode<T> *root;
    int treeSize;
    static void (*visit)(binaryTreeNode<T>*);
    static void preOrder(binaryTreeNode<T>*);
    static void inOrder(binaryTreeNode<T>*);
    static void postOrder(binaryTreeNode<T>*);
    static void levelOrder(binaryTreeNode<T>*);
}
```

确定二叉树的高度

```
int height()const{return height(root);}
template<class T>
int linkedBinaryTree<T>::height(binaryTreeNode<T> *t)
{
    if(t==null)
        return 0;
    int hl=height(t->leftChild);
    int hr=height(t->rightChild);
    if(hl>hr)
        return ++hl;
    else
        return ++hr;
}
```

应用

设置信号放大器

用树来表示传输过程

放大器可以使子节点的信号与其父节点相同

每条边上的数字是信号从父节点传到子节点的衰减量

求解策略

设degradeFromParent(i)表示节点i与其父节点的衰减量

从节点i到达叶子节点的衰减量的最大值设为degradeToLeaf(i)

若节点i为叶子节点degradeToLeaf(i)=0

如果节点i不是子节点,j是其子节点则

degradeToleaf(i)=$max_{j是i的孩子}${degradeFromParent(j)+degradeToleaf(j)}

如果degradeFromParent(j)+degradeToleaf(j)大于容忍值,那么应该在j放置信号放大器

```
struct booster
{
    int degradeToleaf,degradeFromParent;
    bool boosterHere;//是否安放了放大器
    void output(ostream&out) const
    {
        out<<"boosterHere"<<" "<<degradeToleaf<<" "
<<degradeFromParent<<" ";
```

```
    }
}
//重载操作符<<
ostream & operator<<(ostream& out,booster x)
{
    x.output(out);
    return out;
}
```

```
void placeBoosters(binaryTreeNode<booster> *x)
{
    x->element.degradeToLeaf=0;
    //计算x的左子树的衰减值.若大于容忍值,则在x的左孩子放大一个放大器
    binaryTreeNode<booster> *y=x->leftChild;
    if(y!=nullptr)
    {
        int degradation=y->element.degradeToleaf+y->element.degradeFromParent;
        if(degradation>tolerance)
        {
            y->element.boosterHere=true;
            x->element.degradationToLeaf=degradation;
        }
        if(x->element.degradationToLeaf<degradation)
            x->element.degradationToLeaf=degradation;
    }
    y=x->rightChild;
    if(y!=nullptr)
    {
        int degradation=y->element.degradeToleaf+y->element.degradeFromParent;
        if(degradation>tolerance)
        {
            y->element.boosterHere=true;
            x->element.degradationToLeaf=degradation;
        }
        if(x->element.degradationToLeaf<degradation)
            x->element.degradationToLeaf=degradation;
    }
}
```

通过后序遍历可以计算节点的degradeToleaf值

并查集

用树来描述集合,同一集合的树的根节点是相同的

```
void initialize(int numberOfElements)
{
    parent=new int[numberOfElements+1];
    for(int e=1;e<=numberOfElements;e++)
        parent[e]=0;
}
int find(int theElement)
{
    while(parent[theElement]!=0)
        theElement=parent[Element];
    return theElement;
}
int unite(int rootA,int rootB)
{
    //合并两棵根节点不同的树
    parent[rootB]=rootA;
}
```

合并函数的性能改进

对根为i和根为j的树进行合并操作时,利用**重量**规则和**高度**规则,可以提高并查集算法的性能

重量规则:若根为i的树的节点数少于根为j的树的节点树,则将j作为i的父节点,否则将i作为j的父节点

高度规则: 若根为i的树高度小于根为j的树的高度,则j作为i的父节点,否则将i作为j的父节点

重量规则时使用的结构

```
struct unionFindNode
{
    int parent;//若为真,表示树的重量,否则是父节点的指针
    bool root;//当且仅当是根时为真
    unionFindNode()
    {
        parent=1;
        root=true;
    }
}
```

```
void initialize(int numberOfElements)

{

node=new unionFindNode[numberOfElements+1];

}
```

```
int find(int theElement)
{
    while(!node[Element].root)
    {
        theElement=node[theElement].parent;
    }
    return theElement;
}
void unite(int rootA,int rootB)
{
    if(node[rootA].parent<node[rootB].parent)
    {
        node[rootB].parent+=node[rootA].parent;
        node[rootA].root=false;
        node[rootA].parent=rootB
    }
    else
    {
        node[rootA].parent+=node[rootB].parent;
        node[rootB].root=false;
        node[rootB].parent=rootA
    }
}
```

查找函数的性能优化

- 路径紧缩---从待查节点到根节点的路径上,每一个节点的parent指针都被指向根节点
- 路径分割---除根节点和其子节点外,每个节点的parent指针指向祖父节点
- 路径对着 ---除根节点和其子节点外,每隔一个节点,节点的parent指针指向祖父节点

```
int find(int theElement)
{
int theRoot=theElement;
```

```
while(!node[theRoot].root)
    theRoot=node[theRoot].parent;
int currentNode=theElement;
while(currentNode!=theRoot)
{
    int parentNode=node[currentNode].parent;
    node[currentNode].parent=theRoot;
    currentNode=parentNode;
}
return theRoot;
}
```

第12章 优先级队列

定义和应用

优先级队列(priority queue)是0个或多个元素的集合，每个元素都有一个优先权或值。

对优先级队列进行的操作有：

- 查找一个元素
- 插入一个元素
- 删除一个元素

在**最小优先级队列(min priority queue)**中，查找和删除的元素都是优先级最小的元素

在**最大优先级队列(max priority queue)**中，查找和删除的元素都是优先级最大的元素

抽象数据类型

```
//抽象类maxPriorityQueue
template<class T>
    class maxPriorityQueue
    {
        public:
        virtual ~maxPriorityQueue(){}
        virtual bool empty()const =0;
        virtual int size() const=0;
        virtual const T&top()=0;
        virtual void pop()=0;
        virtual void push(const T& theElement)=0;
    }
```

线性表

描述最大优先级队列的最简单方法是无序线性表

使用链表插入操作为O(1)，删除操作为O(n)

使用数组插入操作为O(1),删除操作为O(n)

采用有序线性表，插入操作为O(n),删除操作为O(1)

堆

定义:

一棵**大根树(小根树)**是这样一颗树，其中每个节点的值都小于或等于其子节点(如果有的话)的值

定义:

一个**大根堆(小根堆)**既是大根树(小根树),也是完全二叉树

大根堆的插入操作

把新元素插入新节点，然后沿着从新节点到根节点的路径，执行一趟气泡操作，将新元素和其父节点的元素比较交换，直到后者大于或等于前者为止

时间复杂度为O(logn)

大根堆的删除操作

在大根树中删除一个元素，就是删除根节点的元素

把最后一个位置的节点取出，删除

从根节点开始，将这个节点与根节点的左右子树根节点比较，如果该元素最大，直接将该元素放在根节点上，算法终止；否则，选取左右子树的较大根节点放到根节点上，该元素再与较大根节点的左右子树根节点比较，直到找到放置位置为止

大根堆的初始化

策略1：在空堆中执行n次插入操作O(nlogn)

策略2：n个元素的数组，从i=n/2向下取整的节点开始，检查以该节点为根节点的完全二叉树是否是大根堆，如果是，检查第i-1个节点；如果不是，将其调整为大根堆后，再检查第i-1个节点。直到检查到这个数组的第0号元素为止

类maxHeap

大根堆的插入

```
template<class T>
void maxHeap<T>::push(const T& theElement)
{
    if(heapSize==arrayLength-1)
    {
        changeLength1D(heap,arrayLength,2*arrayLength);
        arrayLength*=2;
    }
    int currentNode=++heapSize;
    while(currentNode!=1&&heap[currentNode/2]<theElement)
    {
        heap[currentNode]=heap[currentNode/2];
        currentNode/=2;
    }
    heap[currentNode]=theElement;
}
```

大根堆的删除

```
template<class T>
    void maxHeap<T>::pop()
    {
        if(heapSize==0)
            throw queueEmpty();
        heap[1].~T();
        T lastElement=heap[heapSize--];
        int currentNode=1,child=2;
```

```
        while(child<=heapSize)
        {
            if(child<heapSize&&heap[child]<heap[child+1])
            {
                child++;
            }
            if(lastElement>=heap[size])
                break;
            heap[currentNode]=heap[child];
            currentNode=child;
            child*=2;
        }
        heap[currentNode]=lastElement;
    }
```

堆排序

先用n个待排序的元素来初始化一个大根堆，再从堆种逐个提取，删除元素

初始化O(nlogn)

提取元素O(1)

删除元素O(logn)

2nlogn+n

总时间O(nlogn)

霍夫曼编码

在一棵扩展二叉树中，从根到外部节点的路径可用来编码，方法使用0表示向左子树移动一步，用1表示向右子树移动一步。

令S是由一些字符组成的字符串，F(x)是x字符出现频率。编码位串长度是

$$WEP=\sum_{i=1}^{n}L(i)F(i)$$

L(i)：从根节点到外部节点i的路径长度

WEP:二叉树的加权外部路径长度

使得WEP最小的数叫做霍夫曼树

利用霍夫曼编码堆一个字符串进行编码的步骤:

- 确定字符串的符号以及其出现的频率
- 建立霍夫曼树，其中外部节点用字符串中的符号表示，外部节点的权值用相应的符号表示
- 沿着根到外部节点的路径遍历，获得每个符号的代码
- 用代码代替字符串中的符号

掌握编码过程即可

第14章 搜索树

二叉搜索树和索引二叉搜索树

二叉搜索树(binary search tree):

二叉搜索树是一棵二叉树，可能为空；若不为空，则它满足以下特征：

- 每个元素有一个关键字，且此关键字不能重复
- 在根节点的左子树中，所有元素的关键字都小于根节点的关键字
- 在根节点的右子树中，所有元素的关键字都大于根节点的关键字
- 根节点的左子树、右子树也是一棵二叉搜索树

其中第一个要求中的关键字不能重复可以去掉，并且将第二个和第三个要求中的小于、大于更改为小于等于、大于等于，所形成的这种二叉树称为**有重复值的二叉搜索树（binary search tree with duplicates)**

二叉索引搜索树:

源自于普通二叉搜索树，只是在每个节点中添加一个leftSize域，用于记录该节点左子树元素的个数

抽象数据类型

C++抽象类bsTree

```
template<class K,class E>
class bsTree :public dictionary<K,E>
{
    public :
    virtual void ascend()=0;
}
```

C++抽象类indexedBSTree

```
template<class K,class E>
class indexedBSTree : public bsTree<K,E>
{
    public:
    virtual pair<const K,E> *get(int) const=0;//根据给定的索引，返回其数对的指针
    virtual void delete<int> =0;//根据所给索引，删除其数对
}
```

二叉搜索树的操作和实现

类binarySearchTree

如果继承上一章中的linkedBinaryTree，则ascend方法的实现

```
void ascend(){inOrderOutput();}
```

搜索

从根开始查找关键字为theKey的元素，如果结果为空，则查找失败；如果找到元素，返回数对的指针

```
template<class K,class E>
pair<const K,E>* binarySearchTree<K,E>::find(int theKey)
{
    binarySearchTreeNode<pair<Const K,e>> *p=root;
    while(p!=NULL)
    {
        if(p->element.first==theKey) return p->element;
        if(p->element.first>theKey) p=p->leftChild;
        else
            p=p->rightChild;
    }
    return NULL;
}
```

时间复杂度 O(h)

插入

首先通过查找来确定，在树中是否存在某个元素的关键字与插入元素的关键字相同，如果有，则替换该元素的值；如果没有，就将插入元素当作中断节点的子节点插入

```
template<class K,class E>
void binarySearchTree<K,E>::insert(const& pair<const K,E>
thePair)
{
    binaryTreeNode<pair<const K,E>>* p=root,*pp=NULL;
    while(p!=NULL)
    {
        pp=p;
        if(thePair.first>p->element.first)
        {
            p=p->rightChild;
        }
        else if(thePair.first<p->element.first)
        {
            p=p->leftChild;
        }
        else
        {
            p->element.second=thePair.second;
            return;
        }
    }
    binaryTreeNode<pair<const K,E>> *newNode=new
binaryTreeNode<pair<const K,E>>(thePair);
    if(root!=NULL)
    {
    if(thePair.first>pp->element.first)
        pp->rightChild=newNode;
    else pp->leftChild=newNode;
    }
    else root=newNode;
    treeSize++;
}
```

时间复杂度 O(h)

删除

考虑三种情况：

- 删除节点是叶子节点——直接删除即可
- 删除节点有一棵非空子树——用这棵子树的根节点代替这个删除节点的位置
- 删除节点有两棵非空子树

将删除节点的元素替换为它的左子树的最大元素，然后删除左子树的最大元素

```
template<class K,E>
void binarySearchTree::erase(const K& theKey)
{//删除关键字为theKey的数对
    binaryTreeNode<pair<const K,e>> *p=root,
                                    *pp=nullptr;
    while(p!=NULL&&p->element.first!=theKey)
    {
        pp=p;
        if(theKey>p->element.first)
            p=p->rightChild;
        else p=p->leftChild;
    }
    if(p==NULL) return;
    if(p->leftChild!=nullptr&&p->leftChild!=nullptr)
    {
        binaryTreeNode<pair<const K,E>> *s=p->leftChild,*ps=p;
        while(s->rightChild!=nullptr)
        {
            ps=s;
            s=s->rightChild;
        }
binaryTreeNode<pair<const K,E>>*q=new binaryTreeNode(s-
>element,p->leftChild,p->rightChild);
        if(pp==nullptr) root=q;
        else if(p==pp->leftChild) pp->leftChild=q;
        else pp->rightChild=q;
        if(ps==p) pp=q;// 删除最大元素
        else pp=ps;
        delete p;
        p=s;
    }
    binaryTreeNode<pair<const K,E>> *c;
    if(p->leftChild!=NULL)
        c=p->leftChild;
    else c=p->rightChild;
    if(p==root)
        root=c;
    else
```

```
    {
        if(p=pp->rightChild)
        pp->rightChild=c;
        else
        pp->leftChild=c;
    }
    treeSize--;
    delete p;
}
```

时间复杂度 O(h)

二叉搜索树的高度

一棵n个元素的二叉搜索树，高度h的取值范围为

$$logn \leq h \leq n$$

这说明如果我们将二叉搜索树的高度限定在logn时，插入、删除、搜索的时间复杂度都是O(logn)，这就是下一章平衡搜索树的原理.

带有相同关键字的二叉搜索树

当一棵二叉搜索树可以具有两个或多个关键字相同的元素时，对应的类称为dBinarySearchTree,要实现这个类，只需修改二叉树中的插入函数中的while语句即可

```
while(p!=nullptr)
{
    pp=p;
    if(thePair.first<p->element.first)
        p=p->leftChild;
    else
        p=p->rightChild;
    //删除了相同关键字覆盖原元素值的操作，可以有重复的元素了！
}
```

索引二叉搜索树

索引二叉搜索树的搜索是按索引而非关键字来搜索的

假设查找元素的索引为index

从根节点开始查找，如果根节点leftSize域=index-1，那么根节点的元素就是要查找的元素

如果leftSize<index-1,要查找的元素在根节点的右子树中，索引为index-(leftSize+1)

如果leftSize>index-1,要查找的元素在根节点的左子树中，索引为index

应用

直方图

定义类binarySearchTreeWithVisit是类binarySearchTree的扩展

增加了下面一个公有成员函数

```
void insert(const pair<const K,E>& thePair, void (*visit)(E&));
```

将元素thePair插入搜索树，如果存在关键字等于thePair.first的元素p,则调用函数visit(p.second)

使用搜索树的直方图

```
int main(void)
{
    int n;
    cout<<"Enter number of elements"<<"\n";
    cin>>n;
    binarySearchTreeWithVisit<int,int> theTree;
    for(int i=0;i<n;i++)
    {
        pair<int,int> thePair;
        cout<<"Enter element"<<i<<"\n";
        cin>>thePair.first;
        thePair.second=1;
        insert(thePair,add1);
    }
    return 0;
}
```

箱子装载问题的最优匹配法

在实现最优匹配法时,使用带有重复关键字的二叉搜索树,我们能够在O(nlogn)时间内实现最优匹配法.使用平衡搜索树,在最坏情况下时间复杂度是O(nlogn).

搜索树的每一个元素代表一个正在使用且剩余容量不为0的箱子

为了完成箱子装载问题的最优匹配法,我们要搜索树类的定义,增加公有成员函数findGE(theKey),返回值是剩余容量及大于等于theKey又是最小的箱子.

```
template<class K,E>
```

```
pair<const K,E>* dBinarySearchTreeWithGE::findGE(const K&
theKey) const
{
    binaryTreeNode<pair<const K,E>> *currentNode=root;
    pair<const K,E> *bestElement=NULL;
    while(currentNode!=NULL)
    {
        if(currentNode->element.first>=theKey)
        {
            bestElement=currentNode->element;
            currentNode=currentNode->leftChild;
        }
        else
        {
            currentNode=currentNode->rightChild;
        }
    }
    return bestElement;
}
```

```
void bestFitPack(int *objectSize,int numberOfObjects,int
binCapacity)
{
    int n=numberOfObjects;
    int binsUesd=0;
    dBinarySearchTreeWithGE<int,int> theTree;
    pair<int,int> theBin;
    for(int i=1;i<=n;i++)
    {
        pair<const int,int>
*bestBin=theTree.findGE(objectSize[i]);
        if(bestBin==nullptr)
        {
            //没有足够大的箱子,启用一个新箱子
            theBin.firtst=binCapacity;
            theBin.second=++binUsed;
        }
        else
        {//从树theTree中删除最匹配的箱子
            theBin=*bestBin;
            theTree.erase(bestBin->first);
        }
        cout<<"Pack Objeact "<<i<<" in bin "
<<theBin.second<<endl;
```

```
            theBin.first-=Objeact[i];
            if(theBin.first>0)
                theTree.insert(theBin);
        }
    }
```

第15章 平衡搜索树

AVL树

定义：

如果搜索树的高度总是O(logn),我们就能保证查找、插入和删除的时间为O(logn)。最坏情况下的高度为O(logn)的树为**平衡树(balanced tree)**。比较流行的一种平衡树是AVL树

一棵空的二叉树是 AVL树;

如果T是一棵非空的二叉树，T_L和T_R分别是其左子树和右子树，那么当T满足以下条件时，T是一棵AVL树:

1)T_L和T_R是AVL树:

2)$|h_L - h_R| \leq 1$，其中h_L和h_R分别是TL和TR的高。

一棵AVL搜索树既是一棵AVL树，也是一棵二叉树

AVL树的一些特征：

- n个元素的AVL树，高度为O(logn)
- 对于大于0的任意一个n，都有一棵AVL树
- AVL搜索树的查找，插入，删除操作的时间复杂度都为O(logn)

AVL树的高度

对一颗高度为h的AVL树，令N_h使其最少的节点数。在最坏情况下，根的一棵子树高度是h-1,另一棵子树的高度是h-2,而且两棵子树都是AVL树，因此有

$$N_h = N_{h-1} + N_{h-2} + 1$$

如果树中有n个节点，那么树的最大高度为 O(logn)

AVL树的描述

AVL一般采用链表描述。为了简化插入和删除操作，为每一个节点增加一个平衡因子bf。节点x的平衡因子$bf(x)$定义为：

x的左子树高度-x的右子树高度，由AVL树的定义可知，平衡因子的取值可能由1，0，-1

AVL搜索树的搜索

用二叉搜索树的搜索即可，搜索时间为O(logn)

AVL搜索树的插入

用二叉搜书树的插入在AVL树中插入一个节点生成一棵搜索树，如果它有一个或更多的节点其平衡因子不再是-1、0、1那么搜索树就是不平衡的，需要移动不平衡子树来恢复平衡

导致不平衡的情形：

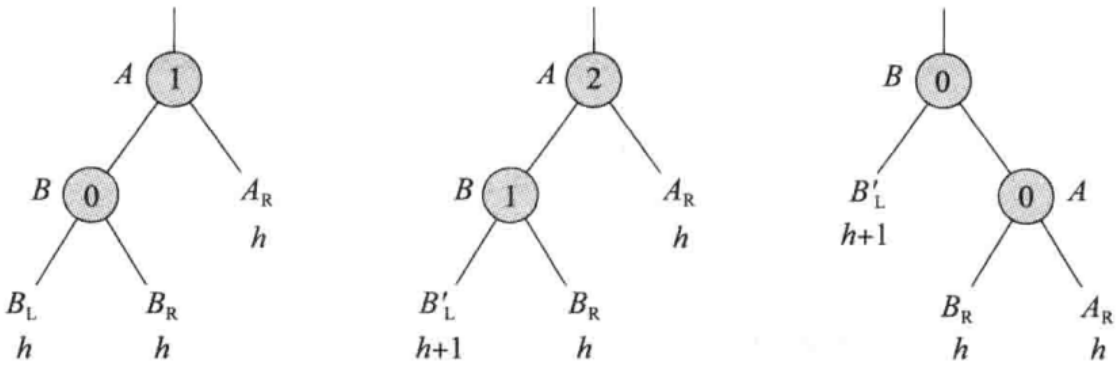
- 插入操作之后，平衡因子从-1变为-2(在X的右子树插入)
- 插入操作之后，平衡因子从1变为2(在X的左子树插入)

不平衡的情况有两种:

1. L型不平衡

又可细分为LL型和LR型不平衡

LL型不平衡

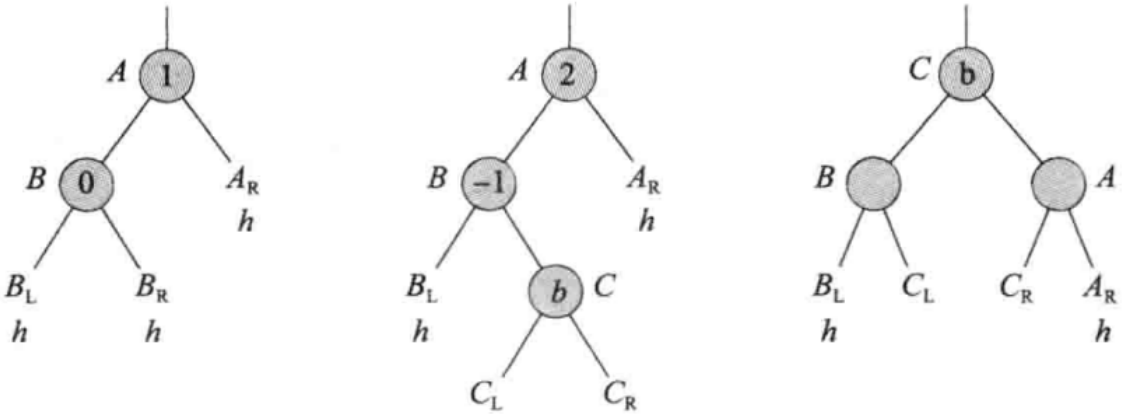


节点内的数字是平衡因子，子树名称下面的是子树高度

图 15-4　LL 旋转

步骤：以B为根节点，A为B右子树的根节点，BL`为B的左子树，BR为A的左子树

LR型不平衡



a）插入之前　　b）插入B_R之后　　c）LR旋转之后

$$b=0 \Rightarrow bf(B)=bf(A)=0 \text{ 旋转后}$$
$$b=1 \Rightarrow bf(B)=0,\ bf(A)=-1 \text{ 旋转后}$$
$$b=-1 \Rightarrow bf(B)=1,\ bf(A)=0 \text{ 旋转后}$$

2. R型不平衡

又可细分为RL型和RR型不平衡

与L型不平衡是镜像的

总的说来AVL搜索树的插入步骤可以归纳为

1. 沿着根节点开始的路径，根据新元素的关键字，去寻找新元素的插入位置。在此过程中，记录最新发现平衡因子为-1或1的节点，并令其为A节点。如果找到具有相同关键字的元素，那么插入失败，终止算法
2. 如果在步骤1种的A节点不存在，那么从根节点开始沿着原路径修改平衡因子，然后终止算法
3. 如果bf(A)=1并且将新节点插入A的右子树种或bf(A)=-1并且新节点插入到左子树，那么A的平衡因子是0。这种情况下，修改从A到新节点途中的平衡因子，然后终止算法
4. 确定A的不平衡类型并执行相应的旋转，并对新子树根节点至新插入节点的路径上的节点的平衡因子做相应的修改

AVL搜索树的删除

设q是被删除节点的父节点

- 如果q的新平衡因子是0，那么它的高度减1，这时需要改变它的父节点的平衡因子，也有可能要改变其他祖先节点的平衡因子
- 如果q的新平衡因子是1或-1，那么它的高度与删除前相同，且无需改变其祖先节点的平衡因子值
- 如果q的新平衡因子是-2或2，那么树在q处是不平衡的

从q到根节点的路径上，找到第一个平衡因子为-2或2的节点，记作A。如何恢复A的平衡，需要按类型而定

- 删除发生在A的左子树为L型，否则为R型

找到A最高的子树，进行旋转，若最高的子树为左子树，R-1,R0,R1

若最高子树为右子树，L-1,L0,L1

看课本即可

B-树

m叉搜索树

定义 15-2　*m* 叉搜索树（*m*-way search tree）可以是一棵空树。如果非空，它必须满足以下特征：

1）在相应的扩充搜索树中（即用外部节点替换空指针之后所得到的搜索树），每个内部节点最多可以有 m 个孩子以及 $1 \sim m-1$ 个元素（外部节点不含元素和孩子）。

2）每一个含有 p 个元素的节点都有 $p+1$ 个孩子。

3）对任意一个含有 p 个元素的节点，设 $k_1, \cdots, k_p$ 分别是这些元素的关键字。这些元素顺序排列，即 $k_1<k_2<\cdots<k_p$。设 $c_0, c_1, \cdots, c_p$ 是该节点的 $p+1$ 个孩子。在以 c_0 为根的子树中，元素的关键字小于 k_1；在以 C_p 为根的子树中，元素的关键字大于 k_p；在以 c_i 为根的子树中，元素的关键字大于 k_i 而小于 k_{i+1}，其中 $1 \leqslant i<p$。

m叉搜索树的搜索、插入、删除看课本即可

m叉搜索树的高

一棵高度为h的m叉搜索树(不含外部节点)最少有h个元素(每层一个节点，每个节点包含一个元素)，最多有m^h-1个元素。

在高度为h的m叉搜索树中，元素个数在h到m^h-1之间，所以一棵n元素的m叉搜索树的高度是$log_m^{(n+1)}$~n

m阶B-树

m阶B-树是一棵m叉搜索树。如果B-树非空，那么相应的扩展树满足下面特征：

- 根节点至少有两个孩子
- 除根节点外，所有内部节点至少有$\lceil m/2 \rceil$个孩子
- 所有外部节点在同一层

B-树的高度

设T是一棵高度为h的m阶B-树。令d=$\lceil m/2 \rceil$，n是T的元素个数，则

1. $2d^{h-1}-1 \leq n \leq m^h-1$

2. $log_m^{n+1} \leq h \leq log_d^{\frac{n+1}{2}} + 1$

B-树的搜索

与m叉树相同

B-树的插入

在B-树种插入一个新元素，首先要在B-树种搜索关键字与之相同的元素。如果存在这样的元素，那么插入失败，因为在B-树的元素种不允许有重复的关键字。如果不存在这样的元素，比那可以将元素插入在搜索路径中所遇到的最后一个内部节点中。如果节点饱和，插入一个新元素时，需要分裂该节点

B-树的删除

1. 该元素位于叶子节点，直接删除
2. 该元素位于非叶子节点，转换为情况1

第16章 图

无权图的描述

邻接矩阵

一个n顶点图G=(V,E)的邻接矩阵是一个nxn矩阵，其中每个元素是0或1.

如果G是一个无向图，则其中元素定义如下：

$$A(i,j) = \begin{cases} 1 & \text{如果}(i,j) \in E\text{或}(j,i) \in E \\ 0 & \text{其他} \end{cases}$$

如果G是有向图，那么其中的元素定义如下：

$$A(i,j) = \begin{cases} 1 & \text{如果}(i,j) \in E \\ 0 & \text{其他} \end{cases}$$

一些结论:

- n个顶点的无向图，有A(i,i)=0
- 无向图的邻接矩阵是对称的,A(i,j)=A(j,i)
- 对于n顶点的无向图，$\sum_{j=1}^{n} A(i,j) = \sum_{j=1}^{n} A(j,i) = d_i$($d_i$是顶点的度)
- 有向图中，$\sum_{j=1}^{n} A(i,j) = d_i^{out} \quad \sum_{j=1}^{n} A(j,i) = d_i^{in}$

将邻接链表映射到数组

- 使用映射A(i,j)=1,当且仅当a(i,j)=true　将nxn的图映射到一个(n+1)(n+1)的布尔型数组 需要$(n+1)^2$个字节
- 映射A(i,j)=1，当且仅当a(i-1,j-1)=true　将nxn的图映射到一个nxn的布尔型数组，需要n^2个字节
- 进而还可以吧对角线的元素去掉
- 进而还可以只存储无向图的上三角或下三角部分　仅需$(n^2-n)/2$字节

使用邻接矩阵时，确定邻接于一个给定节点的集合需要用时$\Theta(n)$,增加一条边或删除一条边用时$\Theta(1)$

邻接链表

一个顶点i的邻接表是一个线性表，它包含所有邻接于顶点i的顶点，在一个图的邻接表描述中，图的每一个顶点都有一个邻接表，当邻接表为链表时，就是邻接链表

我们使用类型为链表的数组aList来描述所有链表，aList[i].firstNode指向顶点i的邻接表的第一个顶点。如果x指向链表aList[i]的一个顶点，那么(i,x->element)是图的一条边，其中element的数据类型是int

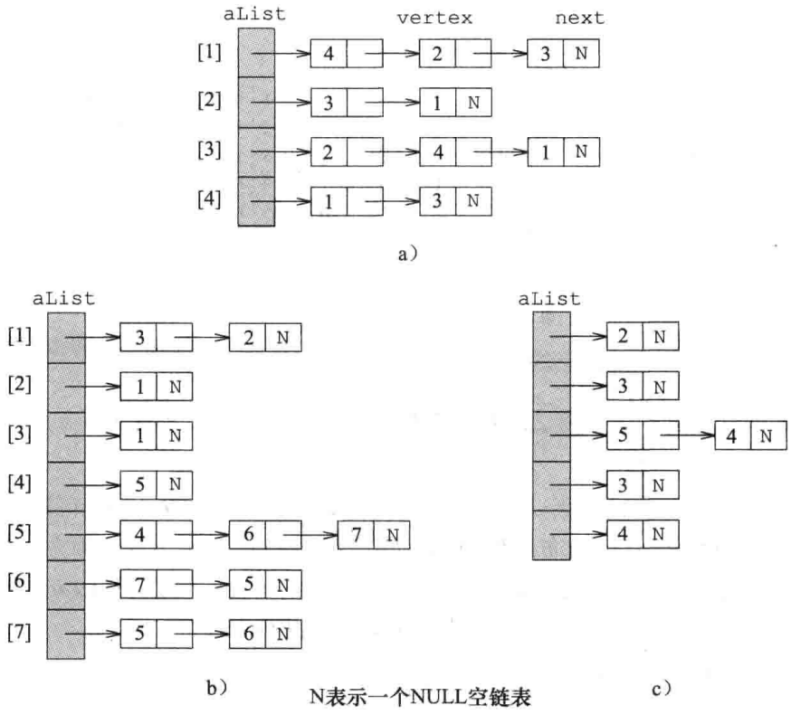


图 16-11　图 16-1 的邻接链表

空间复杂度分析

一个指针和一个整数各需4个字节，因此用一个邻接链表描述一个有n个顶点的图需要8(n+1)字节存储n+1个firstNode指针和aList链表的listSize域，4x2xm个字节存储m个链表节点，m是边数

当e远远小于n^2时，邻接链表比邻接矩阵需要更少的空间

时间复杂度分析

确定邻接于顶点i的顶点需要用时$\Theta(\text{邻接于顶点}i\text{的顶点数})$

插入或删除一条边(i,j) 对于无向图用时$\Theta(d_i + d_j)$

对于有向图用时$\Theta(d_i^{out})$

邻接数组

用二维不规则数组存储图

比邻接链表少用4m字节，因为不需要next指针域

渐进时间复杂度与邻接链表法相同，但实际上大部分图的操作，邻接数组要比邻接链表快

加权图的描述

将无权图的描述进行简单扩充即可得到加权图的描述。用成本邻接矩阵C描述加权图。C(i,j)表示边(i,j)的权值，使用方法和邻接矩阵的使用方法意义。在这种方法中，需要给不存在的边赋一个值，一般是一个很大的值，在实现代码中，用noEdge代表这个值

链表的元素有两个域vertex和weight，就可以从无权图的邻接链表得到加权图的邻接链表。

图的遍历

广度优先搜索

这种从一个顶点开始，搜索所有可到达顶点的方法叫做**广度优先搜索**。这种搜索方法可使用队列实现

```
virtual void bfs(int v,int reach[],int label)
{//广度优先搜索。reach[i]用来标记从顶点v所有可到达的顶点
    arrayQueue<int> q(10);
    reach[v]=label;
    q.push(v);
    while(!q.empty())
    {
        int w=q.front;
        q.pop();
        vertexIterator<T> *iw=iterator(w);
        int u;
        while((u=iw->next())!=0)
```

```
            if(reach[u]==0)
            {
                q.push(u);
                reach[u]=label;
            }
        delete iw;
    }
}
```

为邻接矩阵法定制的BFS代码

```
void bfs(int v,int reach[],int label)
{
    arrayQueue<int> q(10);
    reach[v]=label;
    q.push(v);
    while(!q.empty())
    {
        //从队列中删除一个有标记的顶点
        int w=q.front();
        q.pop();
        for(int u=1;u<=n;u++)
        {
            if(a[w][u]!=noedge&&reach[w][u]==0)
            {
                q.push[u];
                reach[w][u]=label;
            }
        }
    }
}
```

时间复杂度分析

如果顶点v邻接的顶点有s个

则O(sn)

为邻接链表法定制的BFS代码

```
void bfs(int v,int reach[],int label)
{
    arrayQueue<int> q(10);
    reach[v]=label;
    q.push(v);
    while(!q.empty())
```

```
    {
        int w=q.front();
        q.pop();
        for(chainNode<int>*u =aList[u].firstNode;u!=NULL;u=u->next)
        {
            if(reach[u->element]==0)
            {
                q.push(u->element);
                reach[u->element]=label;
            }
        }
    }
}
```

时间复杂度分析

O($\sum_i d_i^{out}$)

深度优先搜索

与二叉树的前序遍历很相似

```
void dfs(int v,int reach[],int label)
{
    graph<T>::reach=reach;
    graph<T>::label=label;
    rDfs(v);
}

void rDfs(v)
{
    reach[v]=label;
    vertexIterator<T> *iv=iterator(v);
    int u;
    while((u=iv->next())!=0)
    {
        if(reach[u]==0)
            rDfs(u);
    }
    delete iv;
}
```

复杂度分析

DFS和BFS具有相同的时间和空间复杂性

应用

寻找一条路径

用深度优先搜索或广度优先搜索，寻找一条从源点sourceNode到终点theDestination的路径。注意要记录这条路径的每个节点

前序方法dfs

```
int *findPath(int theSource,int theDestination)
{
    int n=numberOfVertices;
    path=new int [n+1];
    path[1]=theSource;
    length=1;
    destination=theDestination;
    reach=new int [n+1];
    for(int i=1;i<=n;i++)
        reach[i]=0;
    if(theSource==theDestination||rFindPath(theSource))
        path[0]=length-1;
    else
    {
        delete[] path;
        path=NULL;
    }
    delete reach;
    return path;
}
bool rFindPath(int s)
{
    reach[s]=1;
    vertexIterator<T>* is=iterator(s);
    int u;
    while((u=is->next())!=0)
    {
        if(reach[u]==0)
            path[++length]=u;
        if(u==destination||rFindPath(u))
            return true;
        length--;
    }
    delete is;
```

```
    return false;
}
```

FindPath首先初始化 graph的静态数据成员:destination, path,length和 reach。算法实际上调用了保护性方法 graph::rFindPath，这个方法在路径不存在时，返回false。

方法graph::rFindPath对DFS 做了两点修改:

1)一到达路径终点，rFindPath就停止。

2)rFindPath把从源点theSource 到当前顶点u 的路径上的顶点都记录在数组 path 中。

rFind寻找的不是最短路径，如果需要找到最短路径，需要用BFS代替DFS

连通图及其构成

从任意一个顶点开始BFS或DFS，然后检验是否所有顶点都被标记为已到达顶点，从而可以判断一个无向图是否连通。连通的概念仅是对无向图定义的

```
bool connected()
{
    if(directed())
        throw undefinedMethod("graph::connected() not defined for directed graph");
    int n=numberOfVertices;
    reach=new [n+1];
    for(int i=1;i<=n;i++)
        reach[i]=0;
     dfs(1,reach,1);
    for(int i=1;i<=n;i++)
    {
        if(reach[i]==0)
            return false;
    }
    return true;
}
```

在一个无向图中，从一个顶点i可到达的顶点集合 C与连接 C的任意两个顶点的边称为**连通构件(connected component)。**

构件标记问题(component-labeling problem)是指对无向图的顶点进行标记，使得2个顶点具有相同的标记，当且仅
当它们属于同一构件。

```
int labelComponents(int c[])
```

```
{
    if(directed())
        throw undefinedMethod("graph::labelComponents() not defined for directed graph");
    int n=numberOfVertices;
    for(int i=1;i<=n;i++)
        c[i]=0;//令所有顶点都是非构件
    label=0;
    for(int i=1;i<=n;i++)
    {
        if(c[i]==0)
        {
            label++;
            bfs(i,c,label);
        }
    }
    return label;
}
```

生成树

在一个具有n个顶点的连通无向图或网中，如果从任一个顶点开始进行 BFS，那么从定
理16-1可知，所有顶点都将被加上标记。在graph::bfs(见序16-4)的内层while 循环中正好有 n-1个顶点到达。在该循环中，当到达一个新顶点时，到达的边是(w,u)。这样得
到的边集有 n-1条边，且它包含一条从v到其他每个顶点的路径，这条路径构成了一个连通
子图，该子图即为G的生成树。
广度优先生成树(breadth-first spanning tree)是按BFS所得到的生成树。

用DFS方法得到的生成树叫做**深度优先生成树(depth-first spanningtree)**

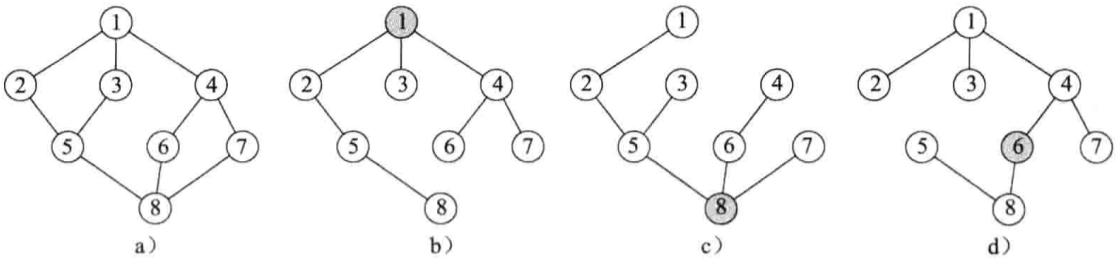


图 16-20　图及其一些广度优先生成树

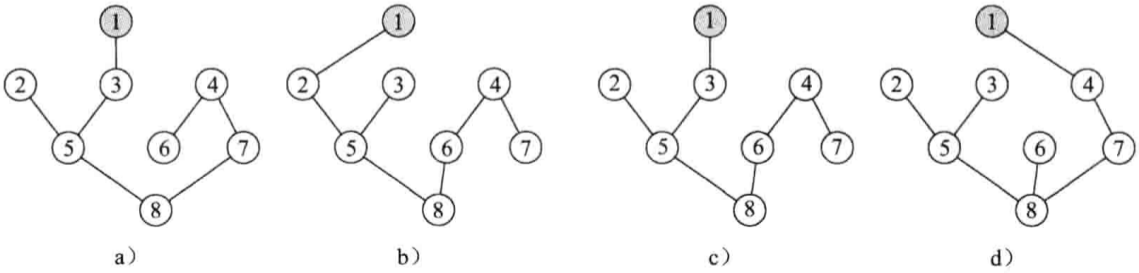


图 16-21　图 16-20a 的一些深度优先生成树

第17章 贪婪算法

在**贪婪算法**中，我们要逐步构造一个最优解。每一步，我们都要在一定的标准下，作出一个最优决策，且这个决策在后续的步骤中都不可更改。做出决策所依据的标准叫做**贪婪准则**

在有些情况下，贪婪算法都能得到最优解，但也有只能得到近似最优解的情况，通常不会与最优解偏差过大

货箱装载问题

有一艘大船准备用来装载货物。所有待装货物都装在货箱中且所有货箱的大小都一样，但货箱的重量都各不相同。设第i 个货箱的重量为wi（1≤i≤n），而货船的最大载重量为c。

目的：是在货船上装入最多的货箱。

贪婪准则：从剩下的货箱中，选择重量最小的货箱

```
void containerLoading(container *c,int capacity,int
numberOfContainers,int *x)
{
    int n=numberOfContainers;
    heapSort(c,numberOfContainers);
    for(int i=1;i<=n,i++)
    {
        x[i]=0;
    }
```

```
    for(int i=1;i<=n&&c[i].weight<=capacity;i++)
    {
        x[c[i].id]=1;
        capacity-=c[i].weight;//剩余容量
    }
}
```

时间复杂度O(nlogn)

0/1背包问题

在0/1背包问题中，需对容量为c 的背包进行装载。从n 个物品中选取装入背包的物品，每件物品i 的重量为wi ，价值为pi 。

可行的背包装载：背包中物品的总重量不能超过背包的容量

约束条件为$\sum wixi <= c和xi \in [0,1](1 \leq i \leq n)$。

最佳装载是指所装入的物品价值最高，即取得最大值。

贪婪准则：从剩余的物品中，选出可以装入背包的价值最大的物品。

利用这种规则，价值最大的物品首先被装入（假设有足够容量），然后是下一个价值最大的物品，如此继续下去。

这种策略不能保证得到最优解。

拓扑排序

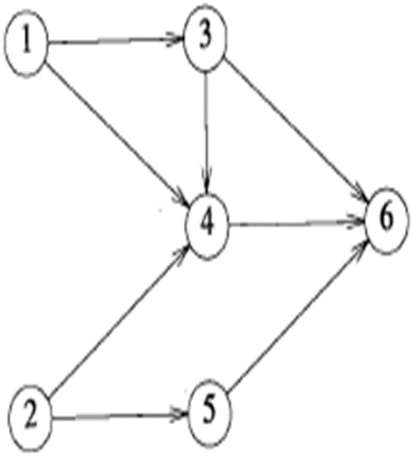
一个复杂的工程通常可以分解成一组简单任务(活动)的集合，完成这些简单任务意味着整个工程的完成。

任务之间具有先后关系。

顶点活动网络(AOV—Activity on vertex network)：任务的集合以及任务的先后顺序

顶点：表示任务(活动)

有向边(i, j)：表示任务间先后关系——任务j 开始前任务i 必须完成。



拓扑序列(Topological orders/topological sequences):

满足：对于在任务的有向图中的任一边 (i,j)，在序列中任务*i* 在任务*j* 的前面

拓扑排序(Topological Sorting):

根据任务的有向图建立拓扑序列的过程

```
bool topologicalOrder(int *theOrder)
{ //求有向图中顶点的拓扑序列;如果找到了一个拓扑序列，则返回true，此时，在theOrder[0:n-1]中记录拓扑序列;如果不存在拓扑序列，则返回false
      ……//确定图是有向图
int n=numberofVertices();
//计算入度
int *inDegree = new int [n+1];
fill(indegree+1, indegree+n+1, 0); //初始化
for (i=1; i<=n; i++) {// i的出边
vertexIterator<T> *ii=iterator(i);
int u ;
while ((u=ii->next())!=0) {
inDegree[u]++;}
}

//把入度为 0 的顶点压入栈
arrayStack<int> stack;
for (i = 1; i <= n; i++)
if (!inDegree[i]) stack.push(i);
// 生成拓扑序列
j = 0; // 数组theOrder 的索引
while (!stack.empty()) // 从堆栈中选择
    {int nextVertex= stack.top(); // 从栈中提取下一个顶点
stack.pop();
theOrder[j++] = nextVertex;
//更新nextVertex邻接到的顶点的入度
```

```
vertexIterator<T> *inextVertex =iterator(nextVertex);
int u;
while (u= inextVertex->next())!=0)
{ inDegree[u]--;
  If   (inDegree[u]==0)    stack.push(u);
}
}

return (j == n);
}
```

•使用(耗费)邻接矩阵描述：

$\Theta(n^2)$

•使用邻接链表描述:

$\Theta(n+e)$

单源最短路径(Dijkstra算法)

对于给定的源顶点sourceVertex，需找出从它到图中其他任意顶点(称为目的)的最短路径。

假设:

边的长度(耗费) >= 0.

没有路径的长度< 0.

贪婪准则：从一条最短路径还没有到达的顶点中，选择一个可以产生最短路径的目的顶点，也就是按路径长度的递增顺序产生最短路径

使用数组predecessor，令predecessor[i]是从源顶点到达顶点i前面的哪个顶点

便于按长度递增顺序产生最短路径，我们定义distanceFromSource[i]是在已生成的最短路径商扩展一条最短边而从到达顶点i时这条最短边的长度

算法思想:

1. 初始化distanceFromSource,predecessor数组，确定从源顶点可以直接到达的顶点以及其距离，并将其放入newReachableVertices数组内
2. 从newReachableVertices数组中找到下一个最小可以到达的顶点v
3. 检验可以通过v顶点可以到达的顶点距离是否比distanceFromSource短，如果是，更新数组，对于没有访问过的顶点，要将其插入newReachableVertices数组内

4. 重复2，3步，直到没有可以访问的新顶点

```
void shortestPaths(int sourceVertex,T *distanceFromSource,int *
predecessor)
{
    if（souceVertex<1||sourceVertex>n)
        throw illegalParameterValue("Invalid source vertex");
    graphChain<int> newReachableVertices;
    //初始化
    for(int i=1;i<=n;i++)
    {
        distanceFromSource[i]=a[sourceVertex][i];
        if(distanceFromSource[i]==noEdge)
            predecessor[i]=-1;
        else
        {
            predecessor[i]=sourceVertex;
            newReachableVertices.insert(0,i);
        }
    }
    distanceFromSource[sourceVertex]=0;
    predecessor[sourceVertex]=0;//源顶点没有前驱
    //更新distanceFromSource和predecessor
    while（! newReachableVertices.empty())
    {//还存在更多的路径
        //寻找distanceFromSource值最小的，还未到达的顶点v
        chain<int>::iterator
iNewReachableVertices=newReachableVertices.begin();
        chain<int>::iterator theEnd=newReachableVertices.end();
        int v=*iNewReachableVertices;
        iNewReachableVertices++;
        while(iNewReachableVertices!=theEnd)
        {
            int w=*iNewReachableVertices;
            iNewReachableVertices++;
            if(distanceFromSource[w]<distanceFromSource[v])
                v=w;
        }
        //下一条最短路径是达到顶点v
        //从newReachableVertices删除顶点v，然后更新
distanceFromSource
        newReachableVertices.eraseElement(v);
        for(int j=1;j<=n;j++)
        {
```

```
            if(a[v][j]!=noEdge&&
(predecessor[j]==-1||distanceFromSource[j]>distanceFromSource[v
]+a[v][j]))
            {
                distanceFromSource[j]=distanceFromSource[v]+a[v]
[j];
                if(predecessor[j]==-1)
                    newReachableVertices(0,j);
                predecessor[j]=v;
            }
          }
      }
}
```

复杂性分析 $O(n^2)$

最小成本生成树

在n个顶点的无向网格G中，每棵生成树都刚好有n-1条边，现在的问题是如何选择n-1条边使它形成G的最小生成树

限制条件：所有的边构成一个生成树。

优化函数：子集中所有边的权值之和。

最小生成树

具有*n*个顶点的无向(连通)网络G的每个生成树刚好具有*n*-1条边。

最小耗费生成树问题是用某种方法选择n-1条边使它们形成*G*的**最小生成树**。

三种求解该问题的贪婪思想：

Kruscal算法

算法思想：

开始，初始化含有 n个顶点 0条边的森林.

Kruskal算法所使用的贪婪准则是：从剩下的边中选择一条**不会产生环路**的**具有最小耗费**的**边**加入已选择的边的集合中。

Kruskal算法分*e*步(*e*是网络中边的数目)。

按耗费递增的顺序来考虑这*e*条边，每次考虑一条边。

当考虑某条边时，若将其加入到已选边的集合中会出现环路，则将其抛弃，否则，将它选入。

```
伪代码：
初始化所选边集T=空集
E为网格中的边集
while(E!=空集&&|T|!=n-1)
{
    从E中找出耗费最小的边(U,V);
    if(T加入(u,v)后不成环) 加入(u,v);
}
if(|T|=n-1) T是最小耗费生成树
else 网格不是连通的，不能找到生成树
```

数据结构选择：边集E 小根堆，被选边集用数组来表示

复杂度分析 设有n个顶点，e条边

使用并查集

初始化：O(n)

find操作的次数最多为2*e*，Unite操作的次数最多为*n*-1(若网络是连通的，则刚好是*n*-1次)。

比O(n+e)稍大一点。

使用边的最小堆，按耗费递增的顺序来考虑*e*条边：O(eloge).

$O(n + eloge)$

Prim算法

贪婪准则：从剩余的边中，选择一条成本最小的边，并且把它加入已选的边集中形成一棵树

```
伪代码：
令T是已入选的边集，初始化T=空集
令TV是已在树中的顶点集，TV={1}
令E是网格的边集
while(E!=空&&|T|!=n-1)
{
令(u,v)是一条成本最小的边，且u∈TV,v不属于TV
if(没有这样的边) break;
E=E-{u,v};
把边{u,v}加入T
把顶点v加入TV
}
if(|T|=n-1) T是一棵最小生成树
```

```
14  else 网格不连通，没有最小生成树
```

Solin算法

Kruscal算法和Prim算法的结合，从多个点并发生成最小生成树

算法思想.

从含有 n个顶点的森林开始.

每一步中为森林中的每棵树选择一条边，这条边刚好有一个顶点在树中且边的代价最小。将所选择的边加入要创建的生成树中。

一个森林中的两棵树可选择同一条边。

当有多条边具有相同的耗费时，两棵树可选择与它们相连的不同的边。

丢弃重复的边和构成环路的边。

直到仅剩下一棵树或没有剩余的边可供选择时算法终止。

第18章 分而治之

算法思想：

1. 将问题分解成两个或多个更小的问题
2. 分别解决小问题
3. 把各小问题的解答组合起来，即可得到原来问题的解

归并排序

算法思想：把n个元素按非递减顺序排列。若n为1，则算法终止；否则，将序列划分为k个子序列，先对每一个子序列排序，然后将有序子序列归并为一个序列。

二路划分：将n个元素的序列仅仅划分为两个子序列

几种划分思想：

- 把前n-1个元素放到第一个子序列中，最后一个元素放入第二个字序列中，在第一个子序列排序后，使用插入函数将两个子序列归并。这就是插入排序的递归形式时间复杂度$O(n^2)$
- 将最大的元素放入第一个子序列中，剩下n-1个元素放入第二个子序列中，第二个子序列排序后，将第一个子序列放到第二个子序列后。如果使用冒泡函数来寻找关键字的最大值，那么这种划分就是冒泡排序的递归形式；如果使用Max函数来寻找关键字的最大值，这种划分就是选择排序的递归形式。两种方法的时间复杂度都是$O(n^2)$

- 平衡分割法的情况：

  A:含有**n/k** 个元素

  B:其余的元素

  递归地使用分而治之方法对*A*和*B*进行排序

  将排好序的*A*和*B*归并为一个集合。

可以证明，在两个较小实例大小接近相等时，算法的时间复杂度最小，最好、最坏、平均复杂度均为$\Theta(nlogn)$

二路归并排序的一种迭代算法是这样的：

- 首先将每两个相邻的大小为1的子序列归并
- 然后将每两个相邻的大小为2的子序列归并
- 如此反复，直到只剩下一个有序序列

  从a归并到b，从b归并到a，其实消除了从b到a的复制过程

```
template<class T>
    void mergeSort(T a[],int n)
{//使用归并排序对a[0:n-1]排序
    T *b=new T[n];
    int segmentSize=1;
    while(segmentSize<n)
    {
        mergePass(a,b,n,segmentSize);//从a归并到b
        segmentSize++;
        mergePass(b,a,n,segmentSize);//从b归并到a
        segmentSize++;
    }
    delete[] b;
}
```

为了完成排序代码，需要函数mergePass，这个函数仅用于确定归并子序列的左右边界。实际归并由函数merge完成

```
template <class T>
    void mergePass(T x[],T y[],int n,int segmentSize)
{//从x到y归并相邻的数据段
    int i=0;
    while(i<=n-2*segmentSize)
    {
        merge(x,y,i,i+segmentSize-1,i+2*segmentSize-1);
        i+=segmentSize*2;
```

```
    }
  //少于两个满数据段
    if(i+segmentSize<n)
        merge(x,y,i,i+segmentSize-1,n-1);//剩余两个数据段
    else//剩余一个数据段
        for(int j=i;j<n;j++)
        {
            y[j]=x[j];
        }
}
```

```
template<class T>
    void merge(T c[],T d[],int startOfFirst,int endOfFirst,int endOfSecond)
{//把两个数据段从c归并到d
    int first=startOfFirst;
    second=endOfFist+1;
    result=startOfFirst;//用于归并数据段的索引
    while((first<=endOfFirst)&&(second<=endOfSecond))
        if(c[first]<=c[second])
            d[result++]=c[first++];
    else
        d[result++]=c[second++];
    //归并剩余元素
    if(first>endOfFirst)
        for(int q=second;q<=endOfsecond;q++)
            d[result++]=c[q];
    else
        for(int q=first;q<=endOfFirst;q++)
            d[result++]=c[q];
}
```

自然归并排序

首先从左至右扫描序列元素，如果位置i的元素比位置i+1大，则位置i就是一个分割点，以此确定输入序列中以及存在的有序段。

然后归并这些有序段，直到剩下一个有序段

最好情况 序列已经有序 $O(n)$

最坏情况 被认为与直接归并排序相同 $O(nlogn)$

只有在输入序列确实有很少的有序段时，才建议使用自然归并排序

快速排序

把n个元素划分为三段：左段left,中间段middle和右段right。中段仅有一个元素。左段的元素不大于中间段的元素，右段的元素都不小于中间段的元素，因此可以对left和right独立排序，并且排序后不需要归并。middle的元素为**支点(pivot)**或**分割元素(partitioning element)**

简单描述:

- 从a[0:n-1]中选择一个元素作为支点，组成中间段
- 把剩余元素分为左段left和右段right。使左段元素关键字不大于支点，右段元素关键字不小于支点
- 对左段进行快速排序
- 对右段进行快速排序

快速排序函数quickSort把数组a的最大元素移动到数组的最右端，然后调用递归函数quickSort执行排序。

驱动程序

```
template<class T>
    void quickSort(T a[],int n)
{
    if(n<=1) return;
    int max=indexOfMax(a,n);
    swap(a[n-1],a[max]);
    quickSort(a,0,n-2);
}
```

递归快速排序函数

```
template<class T>
    void quickSort(T a[],int leftEnd,int rightEnd)
{//对a[leftEnd:rightEnd] 排序
    if(leftEnd>=rightEnd) return;

    int leftCursor=leftEnd,
    rightCursor=rightEnd+1;
    T pivot=a[leftEnd];
    //将位于左侧不小于支点的元素或位于右侧不大于支点的元素交换
    while(true)
    {
        do
        {//寻找左侧不小于pivot的元素
```

```
            leftCursor++;
        }while(leftCursor<pivot);
        do
        {//寻找右侧不大于pivot的元素
            rightCursor--;
        }while(rightCursor>pivot);
        if(leftCursor>rightCursor) break;
        swap(a[leftCursor],a[rightCursor]);
    }
    a[leftEnd]=a[rightCursor];
    a[rightCursor]=pivot;

    quickSort(a,0,rightCursor-1);//对左侧的数段进行排序
    quickSort(a,rightCursor+1,rightEnd);//对右侧的数段进行排序
}
```

时间复杂度

最坏情况下，left段总是空 $\Theta(n^2)$

最好情况下，right段和left段元素个数大致相同 $\Theta(nlogn)$

平均复杂度也是 $\Theta(nlogn)$

三值取中快速排序

在三元素a[leftEnd],a[rightEnd],a[(leftEnd+rightEnd)/2]中选取大小居中的元素作为支点元素

各种排序算法的时间复杂度

方法	最坏复杂性	平均复杂性
冒泡排序	n^2	n^2
基数排序	n	n
插入排序	n^2	n^2
选择排序	n^2	n^2
堆排序	nlogn	nlogn
归并排序	nlogn	nlogn
快速排序	n^2	nlogn

选择

从n元素数组a[0：n-1]中找出第k小的元素

思路：

对a进行排序，a[n-k]就是第k小的元素，使用快速排序的话 平均复杂度为$\Theta(nlogn)$

通过修改上面的快速排序的代码，可以获得较快的求解方法

预处理程序

```
template<class T>
    T quickSort(T a[],int n,int k)
{
    if(k<1||k>n)
    throw illegalParameterValue("k must be between 1 and n");
    int max=indexOfMax(a,n);
    swap(a[n-1],a[max]);
    return select(a,0,n-1,k);
}
```

寻找第k小的元素的递归函数

```
template<class T>
T select(T a[],int leftEnd,int rightEnd,int k)
{//寻找第k大的元素
    if(leftEnd>=rightEnd) return a[leftEnd];

    int leftCursor=leftEnd,
    rightCursor=rightEnd+1;
    T pivot=a[leftEnd];
    //将位于左侧不小于支点的元素或位于右侧不大于支点的元素交换
    while(true)
    {
        do
        {//寻找左侧不小于pivot的元素
            leftCursor++;
        }while(leftCursor<pivot);
        do
        {//寻找右侧不大于pivot的元素
            rightCursor--;
        }while(rightCursor>pivot);
        if(leftCursor>rightCursor) break;//交换的一对元素没有找到
        swap(a[leftCursor],a[rightCursor]);
    }
```

```
    if(rightCursor-leftEnd+1==k)
        return pivot;
    a[leftEnd]=a[rightCursor];
    a[rightCursor]=pivot;
    //对一个数据段调用递归
    if(rightCursor-leftEnd+1<k)
    {
        return select(a,rightCursor+1,rightEnd,k-rightCursor+leftEnd-1);//在右段中寻找第k-支点是第几大的
    }
    else return select(a,leftEnd,rightCursor-1,k)//左段中寻找第k大的点
}
```

第19章 动态规划

在动态规划中，要考察一系列决策，已确定最优决策序列是否包含最优决策子序列。

当最优决策包含最优决策子序列时，可以建立**动态规划递归方程**，它可以帮助我们高效的解决问题

0/1背包问题

在0/1背包问题中，需对容量为c 的背包进行装载。从n 个物品中选取装入背包的物品，每件物品i 的重量为wi ，价值为pi

假设

$$f(i,y)表示剩余容量为y，剩余物品为i,i+1,\ldots,n的背包问题的最优解的值$$

$$f(n,y)=\begin{cases}p_n & \mathrm{y}\geq w_n\\ 0 & 0\leq y<w_n\end{cases}$$

$$f(i,y)=\begin{cases}max(f(i+1,y),f(i+1,y-w_i)+p_i & \mathrm{y}\geq w_i\\ f(i+1,y) & 0\leq y<w_i\end{cases}$$

无论第一次的选择是什么，接下来的选择一定是当前状态下的最优解，我们称此为**最优原则(principle of optimality)**

所有顶点对最短路径

1. **问题**

在n个顶点的有向图G中，寻找每一对顶点之间的最短路径，即对于每对顶点(i,j)，需要寻找从i到j的最短路径及从j到i的最短路径，对于无向图，这两条路径是一条。
对一个n个顶点的图，需寻找p =n(n-1)条最短路径。

2. **动态规划公式**

假定图可以由权值为负的边，但不能由带权长度为负值的环路。因此，每一对顶点(i,j)总有一条不含环路的最短路径

假设G有n个顶点，c(i,j,k)表示从顶点i到j的一条最短路径长度，其中间顶点的编号不大于k。

$$C(i,j,0)=\begin{cases}a[i][j] & (a[i][j]\text{是邻接耗费矩阵})\\ 0 & \mathrm{i}=j\\ +\infty & \mathrm{no}Edge\end{cases}$$

$$C(i,j,k)=\begin{cases}c(i,j,k-1) & \text{该路径中不含顶点}k\\ min(c(i,j,k-1),c(i,k,k-1)+c(k,j,k-1)) & \text{该路径中含顶点}k\end{cases}$$

3. **Floyd算法伪代码**

```
//初始化c
for(int i=1;i<=n;i++)
    for(int j=1;j<=n;j++)
        c[i][j][0]=a[i][j];
for (int k = 1; k <= n; k++)
   for (int i = 1; i <= n; i++)
      for (int j = 1; j <= n; j++)
         if (c[i][k][k-1] + c[k][j][k-1]< c[i][j][k-1] )
                     c[i][j][k] = c[i][k][k-1] + c[k][j][k-1]
          else     c[i][j][k] = c[i][j][k-1]

```

若用c(i,j) 代替c(i,j,k) ,最后所得的c(*i*,*j*) 之值将等于c(*i*,*j*,*n*) 值

c和kay值的计算

kay：从i到j的最短路径中最大的k值

```
template<class T>
    void allPairs(T **c,int **kay)
{
    for(int i=1;i<=n;i++)
    {
        for(int j=1;j<=n;j++)
        {
            c[i][j]=a[i][j];
            kay[i][j]=0;
        }
    }
```

```
    for(int i=1;i<=n;i++)
        c[i][i]=0;
    for(int k=1;k<=n;k++)
        for(int i=1;i<=n;i++)
            for(int j=1;j<=n;j++)
            {
             if(c[i][k]!=noEdge&&c[k][j]!=noEdge&&(c[i][j]=noEdge||c[i][k]+c[k][j]<c[i][j]))
                 c[i][j]=c[i][k]+c[k][j];
                k[i][j]=k;
            }
}
```

输出最短路径

```
template<class T>
    void outputPath(T **c,int **kay,T noEdge,int i,int j)
{
    if(c[i][j]==noEdge)
        cout<<"There is no path from "<<i<<" to "<<j<<endl;
    else
    {
        cout<<"The path is"<<i<<" ";
        outputPath(kay,i,j);
        cout<<endl;
    }
}
void outputPath(int **kay,int i,int j)
{
    if(i==j)
        return;
    if(k[i][j]==0)
        cout<<j<<" ";
    else
    {
        outputPath(kay,i,k[i][j]);
        outputPath(kay,kay[i][j],j);
    }
}
```