第一部分 预备知识

了解时间复杂度和算法分析、数据结构的概念目的等

第1章 数据结构和算法

1、数据结构就是一类数据的表示及其相关的操作。

2、一个算法如果能在所要求的资源限制（resource constraint）内将问题解决好，则称这个算法是有效率的（efficient）。

3、一个算法的代价（cost）是指这个算法消耗的资源量。

4、抽象数据类型（abstract data type，ADT）是指数据结构作为一个软件构建的实现。数据结构（data structure）是 ADT 的实现。

5、数据项有逻辑形式（logical form）和物理形式（physical form）两个方面。

6、算法（algorithm）有 5 条性质，正确性（correct）、具体步骤（concrete step）、确定性（no ambiguity）、有限性（finite）、可终止性（terminate）。

7、估算一个算法或者一个计算机程序效率的方法，称为算法分析（algorithm analysis）。

第3章 算法分析

1、渐近分析（asymptotic analysis）可以估算出当问题规模变大时，一种算法及实现它的程序的效率和开销。即是说当输入规模很大时，或者说达到极限（在微积分意义上）时，对一种算法进行的研究。

2、算法的增长率（growth rate）是指当输入的值增长时，算法代价的增长速率。表达式为 cn 的增长率称为线性增长率（linear growth rate）或者线性时间代价（linear time cost）。如果算法的运行时间函数中含有形如 n^2 的高次项，则称为二次增长率（quadratic growth rate），标有 2^n 的曲线属于指数增长率（exponential growth rate）。

3、算法运行时间的上限（upper bound）用来表示该算法可能有的最高增长率。如果某个算法的增长率上限（在最差情况下）是 $f(n)$，那么就说这种算法“最差情况下在集合 $O(f(n))$ 中”，或者直接说“最差情况下在 $O(f(n))$ 中”。例如，如果在最差情况下 $T(n)$ 增长速度与 n^2 相同，则称算法最差情况下在 $O(n^2)$ 中。

4、对于非负函数 $T(n)$，如果存在两个正常数 c 和 n_0，对任意 $n > n_0$，有 $T(n) \le cf(n)$，则称 $T(n)$ 在集合 $O(f(n))$ 中。

5、如果访问并检查数组中的一个元素需要时间c_s（顺序搜索法），如果待查元素在数组每个位置的出现概率均等，那么在平均情况下$T(n)=c_s\cdot\frac{n}{2}$。对于$n>1$，$c_s\cdot\frac{n}{2}\le c_s n$。所以根据定义，$T(n)$在$O(n)$中，$n_0=1$，$c=c_s$。

6、某一算法平均情况下$T(n)=c_1n^2+c_2n$，c_1、c_2为正数。若$n>1$，$c_1n^2+c_2n\le c_1n^2+c_2n^2\le(c_1+c_2)n^2$。因此取$c=c_1+c_2$，$n_0=1$，有$T(n)\le cn^2$。根据第二次定义，$T(n)$在$O(n^2)$中。

7、若在最佳、最差和平均情况下恒有$T(n)=c$，可以认为在这种情况下，$T(n)$在$O(c)$中。不过按照传统说法，运行时间上限为常数的算法在$O(1)$中。

8、$O(n)$在$O(n^2)$中，但$O(n^2)$不在$O(n)$中。

9、算法的下限用符号Ω来表示，读作“大欧米伽”（Omega）。若存在两个正常数c和n_0，对任意$n>n_0$，有$T(n)\ge cg(n)$，则称$T(n)$在集合$\Omega(g(n))$中。

10、假定$T(n)=c_1n^2+c_2n(c_1,c_2>0)$，则有$c_1n^2+c_2n\ge c_1n^2$。因此，取$c=c_1$，$n_0=1$，有$T(n)\ge cn^2$，根据定义，$T(n)$在$\Omega(n^2)$中。

11、如果一种算法既在$O(h(n))$中，又在$\Omega(h(n))$中，则称其为$\Theta(h(n))$。在Θ表示法中，不再说“在……中”，也就是说，若$f(n)$是$\Theta(g(n))$，则$g(n)$是$\Theta(f(n))$。

12、化简法则：（Ω、Θ类似）

①若$f(n)$在$O(g(n))$中，且$g(n)$在$O(h(n))$中，则$f(n)$在$O(h(n))$中。

②若$f(n)$在$O(kg(n))$中，对于任意常数$k>0$成立，则$f(n)$在$O(g(n))$中。

③若$f_1(n)$在$O(g_1(n))$中，且$f_2(n)$在$O(g_2(n))$中，则$f_1(n)+f_2(n)$在$O(\max(g_1(n),g_2(n)))$中。

④若$f_1(n)$在$O(g_1(n))$中，且$f_2(n)$在$O(g_2(n))$中，则$f_1(n)f_2(n)$在$O(g_1(n)g_2(n))$中。

13、简单的赋值语句：a=b；该语句执行时间为常数级的，即$\Theta(1)$。

14、sum = 0;

for (i=1; i<=n; i++)

sum += n; 时间代价为$\Theta(n)$。

15、sum = 0;

for (i=1; i<=n; i++)

for (j=1; j<=i; j++)

sum++;

for (k=0; k<n; k++)

A[k] = k; 总的运行时间为 $\Theta(c_1 + c_2 n + c_3 n^2)$，可化简为 $\Theta(n^2)$。

16、sum1 = 0;

for (i=1; i<=n; i++) // First double loop

for (j=1; j<=n; j++) // do n times

sum1++; 时间代价为 $\Theta(n^2)$。

sum2 = 0;

for (i=1; i<=n; i++) // Second double loop

for (j=1; j<=i; j++) // do i times

sum2++; 时间代价为 $\Theta(n^2)$。

17、sum1 = 0;

for (k=1; k<=n; k*=2) // Do log n times

for (j=1; j<=n; j++) // Do n times

sum1++; 时间代价为 $\Theta(n\log n)$。

sum2 = 0;

for (k=1; k<=n; k*=2) // Do log n times

for (j=1; j<=k; j++) // Do k times

sum2++; 时间代价为 $\Theta(n)$。

18、if 语句的最差情况时间代价是 then 和 else 语句中时间代价较大的那一个，switch 语句的最差情况代价是所有分支中开销最大的那一个。

19、递归的时间代价？

20、看书 P56-3.12 的时间复杂度

第二部分 基本数据结构

第 4 章 线性表、栈和队列

顺序表、链表的优缺点比较，简单的插入删除操作的实现及代码。

栈的入栈和出栈。

队列要会判断队头队尾指针位置，判断是否为空队列和队列插入元素的排序。

1、线性表是由称为元素（element）的数据项组成的一种有限且有序的序列。

2、线性表中不包含任何元素时，称之为空表（empty list）。当前存储的元素数目成为线性表的长度（length）。线性表的开始结点称为表头（head），结尾结点称为表尾（tail）。

3、线性表的实现有两种标准方法，顺序表（array based list 或 sequential list）和链表（linked list）。

4、顺序表的插入（从表头插入）、删除代码：

```
bool insert(const E& it) //在这里 curr 等于 0 的位置，listSize 等于最后一个元素的位置
{
    if (listSize < maxSize)
    {
        for(int i=listSize; i>curr; i--) // Shift Elems up
            listArray[i] = listArray[i-1]; // to make room
        listArray[fence] = it;
        listSize++; // Increment list size
        return true;
    }
    else // List is full
        return false;
}

E remove() //curr 所在的位置就是要删除的元素，因此此时 curr 不一定为 0
{
    Assert((curr>=0) && (curr<listSize), "Nothing to remove");
    E it = listArray[curr]; // Copy the Elem
    for(int i=curr; i<listSize-1; i++) // Shift them down
        listArray[i] = listArray[i+1];
    listSize--; // Decrement size
```

```
    return it;
}
```

5、利用指针实现的线性表通常称为链表（link list）。链表是动态的（dynamic），即能按需要为表中新的元素分配存储空间。链表是由一系列称为表的结点（node）的对象组成的。因为结点是一个独立的对象，所以它能很好地实现独立的结点类。建立结点类还有一个好处，就是它能被栈和队列的链接实现方式重用（reuse）。Link 类表示了结点的完整定义。因为在有这种结点建立的链表中，每个节点只有一个指向表中下一结点的指针，所以称为单链表（singly linked list）或单向表（one-way list）。

6、一个链表实现的关键设计是怎样表示栅栏。最合理的选择是用来指向当前元素，为了操作的便利性，一般让 curr 指针指向当前元素的前一个元素。

7、当链表为空，或者当前位置已经在线性表末端时，可以通过增加特殊的表头结点（header node）来解决。表头结点是表中的第一个结点，它与表中其他元素一样，只是值被忽略，不被当做表中的实际元素。

8、链表的插入、删除代码：

```
bool insert(const E& it)
{
    curr->next = new Link<E>(it, curr->next);
    if (tail == curr) tail = curr->next; // New tail
    rightcnt++;
    return true;
}

E remove()
{
    Assert(curr->next != NULL, "Nothing to remove");
    E it = curr->next->element; // Remember value
    Link<E>* ltemp = curr->next; // Remember link node
    curr->next = ltemp->next; // Remove from list
    if (tail == ltemp) tail = curr; // Reset tail
    delete ltemp; // Reclaim space
```

```
    cnt--; // Decrement the count
    return it;
}
```

9、空间表？

10、顺序表和链表的优缺点比较：

①顺序表

缺点：大小事先固定。空间需求为Ω(n)，还可能更多。

优点：对于表中的每一个元素都没有浪费空间。没有结构性开销，顺序表比链表有更高的空间效率。

②链表

优点：只要存在可用的内存空间分配，链表中元素的个数就没有限制。空间需求为Θ(n)。

缺点：链表需要在每个结点上附加一个指针，其结构性开销占据了整个存储空间的一大部分。

11、作为一般规律，当线性表元素数目变化较大或者未知时，最好使用链表实现；而如果用户事先知道线性表的大致长度，使用顺序表的空间效率会更高。

12、双链表（doubly linked list）可以从一个表结点出发，在线性表中随意访问它的前驱结点和后继结点。双链表存储了两个指针，一个指向它的后继结点，另一个指向它的前驱结点。除了使用头结点外，双链表额外在表尾部加入了一个尾结点（tailer node）。与头结点类似，它不包含任何数据信息，并且总是在链表中存在。

13、双链表的插入、删除代码：

```
void insert(const E& it)
{
    curr->next = curr->next->prev = new Link<E>(it, curr, curr->next);
    //Link(const Elem& e, Link* prevp, Link* nextp)
    cnt++;
}

E remove()
{
    Assert(curr->next != tail, "Nothing to remove");
    E it = curr->next->element; // Remember value
```

```
        Link<E>* ltemp = curr->next; // Remember link node
        ltemp->next->prev = curr;
        curr->next = ltemp->next; // Remove from list
        delete ltemp; // Reclaim space
        cnt--; // Removed from right
        return it;
    }
```

14、双链表与单链表相比唯一的缺点就是使用更多的空间。双链表的每一个结点需要两个指针，它的结构性开销是单链表的两倍。

15、栈（stack）是限定仅在一端进行插入或删除操作的线性表。栈也称为“LIFO”线性表，意思是“后进先出”（Last In First Out）。LIFO 的原则意味着栈存储和删除元素的顺序与元素到达的顺序相反。习惯上称栈的可访问元素为栈顶（top）元素，元素插入栈称为入栈（push），删除元素时称为出栈（pop）。就线性表而言，实现栈的方法多种多样，书中主要介绍顺序栈（array based stack）和链式栈（linked stack），其分别类似于顺序表和链表。

16、成员 top 表示当前所在的位置值，也指当前栈中的元素数目。而当前位置总是在栈顶。

17、顺序栈的入栈、出栈代码：

```
    void push(const E& it)
    { // Put "it" on stack
        Assert(top != maxSize, "Stack is full");
        listArray[top++] = it; //首先把一个值插入到栈顶位置，然后把 top 加 1
    }

    E pop()
    { // Pop top element
        Assert(top != 0, "Stack is empty");
        return listArray[--top]; //首先把 top 减 1，然后删除栈顶元素
    }
```

18、链式栈不需要表头结点，其中唯一一个数据成员是 top，它是一个指向链式栈第一个结点（栈顶）的指针。

19、链式栈的入栈、出栈代码：

```
void push(const E& it)
{ // Put "it" on stack
    top = new Link<E>(it, top);
    size++;
}

E pop()
{ // Remove "it" from stack
    Assert(top != NULL, "Stack is empty");
    E it = top->element;
    Link<E>* ltemp = top->next;
    delete top;
    top = ltemp;
    size--;
    return it;
}
```

20、顺序栈与链式栈的比较：初始时顺序栈必须说明一个固定长度，当栈不够满时，一些空间将被浪费掉。链式栈的长度可变，但是对于每个元素都需要一个链接域，从而产生了结构性开销。

21、递归的实现：子程序调用是通过把有关子程序的必要信息（包括返回地址、参数、局部变量）存储到一个栈中实现的，这块信息称为活动记录（activation record）；子程序调用时再把活动记录压入栈。每次从子程序中返回时，就从栈中弹出一个活动记录。

22、队列（queue）也是一种受限制的线性表，队列元素只能从队尾插入（称为入队操作，enqueue），从队首删除（称为出队操作，dequeue）。队列也称为“FIFO”线性表，即“先进先出”（First In First Out）。本书介绍队列的两种实现方法：顺序队列和链式队列。

23、顺序队列中“移动队列”问题可以通过假定数组是循环的来解决，即允许队列直接从数组中编号最高的位置延续到编号最低的位置。使用取模操作可以很容易实现。

24、区分满队列和空队列的方法：一种是记录队列中元素的个数，或者至少用一个布尔变量来指示是否为空。另一种是设置数组的大小为 n+1，但是只需存储 n 个元素。本书采用后者。

25、front 表示当前队首元素的位置，rear 表示当前队尾元素的位置，size 用来控制队列的循环（它是取模操作符的基数），且数组大小实际要比队列允许的最大长度大 1。在这种实现方法中，队首

存放在数组中编号较低的位置，队尾存放在数组中编号较高的位置。因此，enqueue 增加 rear 指针的值（对 size 取模），dequeue 增加 front 指针的值（对 size 取模）。即是说 rear 从 0 开始，front 从 1 开始，且 0 号位置不存储元素。

26、顺序队列的入队、出队代码：

```
void enqueue(const E& it)
{ // Put "it" in queue
    Assert(((rear+2) % maxSize) != front, "Queue is full");
    rear = (rear+1) % maxSize; // Circular increment
    listArray[rear] = it;
}

E dequeue()
{ // Take element out
    Assert(length() != 0, "Queue is empty");
    E it = listArray[front];
    front = (front+1) % maxSize; // Circular increment
    return it;
}
```

27、链式队列（linked queue）中成员 front 和 rear 分别是指向队首元素和队尾元素的指针，可以使用一个头结点简化操作。初始化时，front 和 rear 同时指向头结点，之后 front 总是指向头结点，而 rear 指向队列的尾结点。enqueue 只是简单地把新元素放在链表尾部，然后修改 rear 指针指向新的链表结点。dequeue 只是简单地删除队列中的第一个结点，并修改 front 指针。

28、链式队列的入队、出队代码：

```
void enqueue(const E& it)
{ // Put element on rear
    rear->next = new Link<E>(it, NULL);
    rear = rear->next;
    size++;
}
```

```
E dequeue()
{ // Remove element from front
    Assert(size != 0, "Queue is empty");
    E it = front->next->element; // Store dequeued value
    Link<E>* ltemp = front->next; // Hold dequeued link
    front->next = ltemp->next; // Advance front
    if (rear == ltemp) rear = front; // Dequeue last element
    delete ltemp; // Delete link
    size --;
    return it; // Return element value
}
```

29、队列插入元素的排序？

30、字典？

第 5 章 二叉树

二叉树的概念、三种遍历方式、满和完全二叉树叶节点和分支节点关系。
二叉检索树的构建、判断、删除和添加，堆的构建、添加删除一个数据的操作。
Huffman 树的构建、叶子节点和终总结点关系。

1、一棵二叉树（binary tree）由结点（node）的有限集合组成，这个集合或者为空（empty），或者由一个根结点（root）及两棵不相交的二叉树组成，这两棵二叉树分别称为这个根结点的左子树（left subtree）和右子树（right subtree）。（不相交是指它们没有公共结点。）这两棵子树的根结点称为此二叉树根结点的子节点（children）。从一个结点到它的两个子结点都有边（edge）相连，这个结点称为其子结点的父结点（parent）。

2、如果一棵树的一串结点 $n_1,n_2,...,n_k$ 有如下关系：结点 n_i 是 n_{i+1} 的父结点 $(1 \le i < k)$，就把 $n_1,n_2,...,n_k$ 称为一条由 n_1 到 n_k 的路径（path）。这条路径的长度（length）是 $k-1$。如果有一条路径从结点 R 至结点 M，那么 R 就称为 M 的祖先（ancestor），M 则称为 R 的子孙（descendant）。因此，所有结点都是根结点的子孙，而根结点是它们的祖先。

3、结点 M 的深度（depth）就是从根结点到 M 的路径长度。树的高度（height）等于最深结点的深度加 1。任何深度为 d 的结点的层数（level）都为 d。根结点的层数为 0，深度也为 0。

4、满二叉树（full binary tree）的每一个结点或者是一个分支结点，并恰好有两个非空子结点；或者是叶结点。完全二叉树（complete binary tree）有严格的形状要求：从根结点起每一层从左到右

填充。一棵高度为 d 的完全二叉树除了 $d-1$ 层以外，每一层都是满的。底层叶结点集中在左边的若干位置上。

5、堆数据结构是完全二叉树，Huffman 编码树是满二叉树。

6、满二叉树定理：非空满二叉树的叶结点数等于其分支结点数加 1。

7、一棵非空二叉树空子树的数目等于其结点数目加 1。

8、按照一定顺序访问二叉树的结点，称为一次遍历或遍历（traversal）。对每个结点都进行一次访问并将其列出，称为二叉树结点的枚举（enumeration）。

9、（根左右）先访问结点再访问其子结点，称为前序遍历（preorder traversal）。前序遍历打印的第一个结点是根结点，接下来打印所有左子树的结点，最后打印的是右子树的结点。

```
template <typename Elem>
void preorder(BinNode<Elem>* root)
{
    if (root == NULL) return; // Empty subtree, do nothing
    visit(root); // Perform desired action
    preorder(root->left());
    preorder(root->right());
}
```

10、（左右根）先访问结点的子结点，再访问该结点，称为后序遍历（postorder traversal）。

11、（左根右）中序遍历（inorder traversal）则先访问左子结点（包括整棵子树），然后访问该结点，最后访问右子结点（包括整棵子树）。

12、数组实现完全二叉树：r 表示结点的下标，它的范围在 0 到 $n-1$ 之间。n 表示二叉树结点的总数。$parent(r)=\frac{r-1}{2}$，当$r\neq 0$时；$leftchild(r)=2r+1$，当$2r+1<n$时；$rightchild(r)=2r+2$，当$2r+2<n$时；$leftsibling(r)=r-1$，当r为偶数时；$rightsibling(r)=r+1$，当r为奇数并且$r+1<n$时。

13、二叉检索树（Binary Search Tree，BST；也可译为“二叉查找树”、“二叉排序树”等）。对于二叉检索树的任何一个结点，设其值为 K，则该结点左子树中任意一个结点的值都小于 K；该结点右子树中任意一个结点的值都大于或等于 K。二叉检索树的特点是，如果按照中序遍历将各个结点打印出来，就会得到由小到大排列的结点。

14、

```
bool find(const Key& K, Elem& it) const
    { return findhelp(root, K, it); }

template <typename Key, typename Elem>
bool findhelp(BNode<Key, Elem>* root,
const Key& K, Elem& e) const
```

```
{
    if (root == NULL) return false; // Empty tree
    else if (K < root->key())
    return findhelp(root->left(), K, e); // Check left
    else if (K > root->key())
    return findhelp(root->right(), K, e); // Check right
    else { e = root->val(); return true; } // Found it
}
```

15、

```
void insert(const Key& k, const Elem& it)
    {
      root = inserthelp(root, k, it);
      nodecount++;
    }

    template <typename Key, typename Elem>
    BNode<Key, Elem>* inserthelp(BNode<Key, Elem>* root, const Key& k, const Elem& it)
    {
        if (root == NULL)
        return new BNode<Key, Elem>(k, it, NULL, NULL);
        if (k < root->key())
        root->setLeft(inserthelp(root->left(), k, it));
        else
        root->setRight(inserthelp(root->right(), k, it));
        return root; // Return tree with node inserted
    }
```

16、

```
template <typename Key, typename Elem>
BNode<Key, Elem>* deletemin(BNode<Key, Elem>* root, BNode<Key, Elem>*& S)
{
    if (root->left() == NULL)
    { // Found min
        S = root;
        return root->right();
    }
    else
    { // Continue left
        root->setLeft(deletemin(root->left(), S));
        return root;
    }
}
```

16、

```
bool remove(const Key& K, Elem& it)
{
    BNode<Key,Elem>* t = NULL;
    root = removehelp(root, K, t);
    if (t == NULL) return false; // Nothing found
        it = t->val();
    nodecount--;
    delete t;
    return true;
    }
```

```
template <typename Key, typename Elem>
BNode<Key, Elem>* removehelp(BNode<Key, Elem>* root, const Key& K,
BNode<Key, Elem>*& R)
{
    if (root == NULL) return NULL; // K is not in tree
    else if (K < root->key())
        root->setLeft(removehelp(root->left(), K, R));
    else if (K > root->key())
        root->setRight(removehelp(root->right(), K, R));
    else
    { // Found: remove it
        BNode<Key, Elem>* temp;
        R = root;
        if (root->left() == NULL) // Only a right child
            root = root->right(); // so point to right
        else if (root->right() == NULL) // Only a left child
            root = root->left(); // so point to left
        else
        { // Both children are non-empty
            root->setRight(deletemin(root->right(), temp));
            Elem te = root->val();
            root->setVal(temp->val());
            temp->setVal(te);
            R = temp;
        }
    }
    return root;
}
```

18、一些按照重要性或优先级来组织的对象称为优先队列（priority queue）。按优先级组织的 BST，其平均情况下插入和删除操作的总时间代价为 $\Theta(n\log n)$。

19、堆（heap）由两条性质来定义，首先，它是一棵完全二叉树，用数组来实现。其次，堆中存储的数据是局部有序的。最大堆（max heap）的性质是，任意一个结点存储的值都大于或等于其任意一个子结点存储的值，即根结点存储着该树所有结点中的最大值。最小堆（min heap）的性质是，任意一个结点存储的值都小于或等于其任意一个子结点存储的值，即根结点存储着该树所有结点中的最小值。

20、调用堆的 insert 函数，插入 n 个值得时间代价为 $\Theta(n\log n)$。

```
void insert(const Elem& it)
{
    Assert(n < maxsize, "Heap is full");
    int curr = n++;
    Heap[curr] = it; // Start at end of heap
    // Now sift up until curr’s parent > curr
    while ((curr!=0) &&(Comp::prior(Heap[curr], Heap[parent(curr)])))
    {
    swap(Heap, curr, parent(curr));
    curr = parent(curr);
    }
}
```

21、buildheap 的时间代价为 $\Theta(n)$。

```
void buildHeap() // Heapify contents of Heap
```

```
{ for (int i=n/2-1; i>=0; i--) siftdown(i); }

void siftdown(int pos)
{
    while (!isLeaf(pos))
    { // Stop if pos is a leaf
        int j = leftchild(pos); int rc = rightchild(pos);
        if ((rc < n) && Comp::prior(Heap[rc], Heap[j]))
        j = rc; // Set j to greater child’s value
        if (Comp::prior(Heap[pos], Heap[j])) return; // Done
        swap(Heap, pos, j);
        pos = j; // Move down
    }
}
```

22、把最大值（根结点）删除后，把堆中最后一个位置上的元素（数组中实际的最后一个元素）移动到根位置上，再用 siftdown 进行一次排序。则删除最大元素的平均时间代价与最差时间代价为 $\Theta(\log n)$。

23、最大堆在查找一个任意值得效率不高，只适合查找最大值。

24、Huffman 编码树（Huffman coding tree）或简称 Huffman 树（Huffman tree）是满二叉树，其每个叶结点对应一个字母，叶结点的权重就是它对应字母出现的频率。其目的在于按照最小外部路径权重（minimum external path weight）建立一棵树。一个叶结点的加权路径长度（weighted path length）定义为权重乘以深度。具有最小外部路径权重的二叉树就是，对于给定的叶结点集合，具有加权路径长度之和最小的二叉树。权重大的叶结点的深度小，因而它相对总路径长度的花费最小。因此，如果其他叶结点的权重小，就被推到树的较深处。

25、建立 Huffman 树的过程：首先，创建 n 个初始的 Huffman 树，每棵树只包含单一的叶结点，叶结点记录对应的字母。将这 n 棵树按照权重大小顺序排位一列。接着，拿走前两棵树（权重最小的两棵树），再把它们标记为 Huffman 树的叶结点，把这两个叶结点标记为一个分支结点的两个子结点，而这个结点的权重即为两个叶结点的权重之和。把所得的新树放回序列中的适当位置，使得权重的顺序保持为升序。重复上述步骤，直至序列中只剩下一个元素，则 Huffman 树建立完毕。

26、当 Huffman 树构造完成，从根结点开始，分别把“0”或者“1”标于树的每条边上。“0”对应于连接左子结点的那条边，“1”则对应于连接右子结点的那条边。字母的 Huffman 编码就是从根结点到对应于该字母叶结点路径的二进制代码。

第 6 章 树

树的概念、遍历方法、五种表示方法（父指针表示法、子节点表示法…..）、K 叉树概念

1、一棵树 T 是由一个或一个以上结点组成的有限集，其中有一个特定的结点 R 称为 T 的根结点。结点的出度（out degree）定义为该结点的子结点数目。森林（forest）定义为一棵或更多棵树的集合。结点为 n 的树必然有 n-1 条边，因为除了根以外的每个结点都有一条边连接其父结点。

2、除了先序和后序跟二叉树遍历方式一样，中序遍历对树不具有自然的定义，可以给出一些很随意的定义。例如，先中根遍历最左子树，然后访问根结点，再依次中根遍历其余子树。

3、父指针（parent pointer）表示法或许是实现树的最简单方法，其对每个结点只保存一个指针域，指向其父结点。可以很方便确认两个结点是否在同一棵树中。并查算法（UNION/FIND，也称并查集）用一棵树代表一个集合。父指针数组表示法，每个对应于结点数组中某个位置的结点，存储其值及一个指向父结点的指针。为了简明起见，父指针表示为父结点在数组中位置的下标值。

4、子结点表（list of children）表示法，其中每个分支结点都存储其子结点形成的一个链表。最左侧的一列数标明数组的下标，值（val）列存储结点值，父结点（par）列存储父结点索引（或父指针）。对于分支结点，最后一列值存储指向子结点链表的指针。链表中的每个表项都存储指向结点的某个子结点的指针（或目标的下标）。子结点表表示法在数组中存储树的结点，每个结点包括结点值、一个父指针（或索引）及一个指向子节点链表的指针，链表中子结点的顺序从左到右。每个链表表项（element）都包含指向一个子结点的指针。这样，结点的最左子结点可以由链表的第一个表项直接找到。

5、左子结点/右兄弟结点表示法，每个结点都存储结点的值，以及指向父结点、最左子结点和右兄弟结点的指针。再痛一个数组中，把其中一棵树添加为另一棵树的子树，只需要简单设置三个指针值即可。

6、动态结点表示法（dynamic node implementation），子结点指针存储在固定长度的数组中，对于每个结点，在第一个域中存储其值，在第二个域中存储子结点指针数组的长度。

7、动态左子结点/右兄弟结点表示法，左子结点是树中结点的最左子结点，右子结点是结点原来的右兄弟结点。可以将森林转化为单一的二叉树，每个结点存储指向其左子结点和右兄弟结点的指针。为了进行转化，假定树的根结点为兄弟结点。

8、K 叉树（K-ary tree）的内部结点恰好有 K 个子结点。这样，二叉树就是 2-ary 树，与树不同的是，K 叉树的结点有 K 个子结点，子结点数目是固定的。注意当 K 变大时，空（NULL）指针的潜在数目会增加，并且叶结点与分支结点在所需要的空间大小上的差异也会更加显著。相应地，满（full）K 叉树和完全（complete）K 叉树与满二叉树和完全二叉树是类似的。

第三部分 排序与检索

排序

内排序和外排序分别有哪些

每个排序的特点、具体排序过程、时间复杂度和空间复杂度

（外排序简单了解）

排序算法	最差时间复杂度	平均时间复杂度	空间复杂度	特点	代码

插入排序 Insertion Sort （内排序）	$\Theta(n^2)$	$\Theta(n^2)$	O(1)	inssort 处理第 i 条记录的情况，记录关键码的值设为 X。当 X 比它上面的记录的关键码值小时，就向上移动该记录。直到遇到一个关键码比它小或者与它相等的值，本次插入才完成，因为再往前就一定都比它小了。	template <typename Elem, typename Comp>
					void inssort(Elem A[], int n)
					{
					for (int i=1; i<n; i++)
					for (int j=i; (j>0) && (Comp::prior(A[j], A[j-1])); j--)
					swap(A, j, j-1);
					}

冒泡排序 Bubble Sort （内排序）	$\Theta(n^2)$	$\Theta(n^2)$	O(1)	如果低序号的关键码值比高序号的关键码值大，则将二者交换顺序。一旦遇到一个最小关键码值，这个过程将使它像个“气泡”似地被推到数组的顶部。第二次再重复调用上面的过程。	template <typename Elem, typename Comp>
					void bubsort(Elem A[], int n)
					{
					for (int i=0; i<n-1; i++)
					for (int j=n-1; j>i; j--)
					if (Comp::prior(A[j], A[j-1]))

					swap(A, j, j-1); }

选择排序
Selection Sort
（内排序）
	$\Theta(n^2)$	$\Theta(n^2)$	O(1)	选择排序的第 i 次是“选择”数组中第 i 小的记录，并把该记录放到数组的第 i 个位置上。其本质也是冒泡排序，但交换次数比冒泡排序少。	template <typename Elem, typename Comp>
					void selsort(Elem A[], int n)
					{
					for (int i=0; i<n-1; i++)
					{
					int lowindex = i;
					for (int j=n-1; j>i; j--)
					if (Comp::prior(A[j], A[lowindex]))
					lowindex = j;
					swap(A, i, lowindex);
					}
					}

Shell 排序
Shellsort
（内排序）
	$\Theta(n^s)$
	$1 < s < 2$
		$\Theta(n\log n)$	O(1)	Shell 排序将序列分成多个子序列，然后分别对子序列进行排序，最后把子序列组合起来。在执行每一次循环时，Shell 排序把序列分为互不相连的子序列，并使各个子序列中的元素在整个数组中的间距相同。对子序列采用插入排序的方式。	template <typename Elem, typename Comp>
					void inssort2(Elem A[], int n, int incr)
					{
					for (int i=incr; i<n; i+=incr)
					for (int j=i; (j>=incr) && (Comp::prior(A[j], A[j-incr])); j-=incr)
					swap(A, j, j-incr);
					}
					template <typename Elem, typename Comp>
					void shellsort(Elem A[], int n)
					{
					for (int i=n/2; i>2; i/=2)
					for (int j=0; j<i; j++)
					inssort2<Elem,Comp>(&A[j], n-j, i);
					inssort2<Elem,Comp>(A, n, 1);
					}

归并排序
Mergesort
（内排序）
	$\Theta(n\log n)$	$\Theta(n\log n)$	O(1)	在排序问题上，分治思想体现在把待排序的列表分成片段，先处理各个片段，再通过某种方式把片段重组。例如给出 8 个待排序的数，归并排序将递归地把线性表分成 8 个只有一个元素的	template <typename Elem, typename Comp>
					void mergesort(Elem A[], Elem temp[], int left, int right)
					{
					if (left == right) return;
					int mid = (left+right)/2;

				子线性表，然后对子表进行重组。再进行第一轮归并，变为 4 个长度为 2 的子线性表；再进行第二轮归并，形成 2 个长度为 4 的子线性表；最终对剩下两个子线性表进行归并，最终形成一个排好序的线性表。当输入数据存储在链表中时，把两个链表进行归并只需要把待归并链表中的第一个结点取出来，然后连接到结果链表的尾部。当使用归并算法为一个数组排序时，使用两个数组轮换进行排序，避免每一次归并都是用一个新数组。把排序好的子数组首先复制到辅助数组中，然后再把它们归并回原数组。	mergesort<Elem,Comp>(A, temp, left, mid);
					mergesort<Elem,Comp>(A, temp, mid+1, right);
					for (int i=left; i<=right; i++)
					temp[i] = A[i];
					int i1 = left; int i2 = mid + 1;
					for (int curr=left; curr<=right; curr++)
					{
					if (i1 == mid+1)
					A[curr] = temp[i2++];
					else if (i2 > right)
					A[curr] = temp[i1++];
					else if (Comp::prior(temp[i1], temp[i2]))
					A[curr] = temp[i1++];
					else
					A[curr] = temp[i2++];
					}
					}

快速排序 Quicksort（内排序）	$\Theta(n^2)$	$\Theta(n\log n)$	$O(\log n)$ ~ $O(n)$	快速排序以一种更为有效的方式实现了“分治法”（divide and conquer）的思想。它首先选择一个轴值（pivot），假设输入的数组中有 k 个小于轴值的结点，于是这些结点被放在数组最左边的 k 个位置上，而大于轴值的结点被放在数组最右边的 n-k 个位置上。对于一个未排序的数组来说，先找到(i+j)/2 位置的轴值，再将该轴值与数组最后一个数（j 位置）交换，再调用函数 partition 进行划分实现，目的是找到 k 值，找到 k 值之后，将 k 与数组最后一个元素（即之前找到的轴值）交换，再进行(i,k-1)和(k+1,j)之间的快速排序(递归)。	template <typename Elem, typename Comp>
					void qsort(Elem A[], int i, int j)
					{
					if (j <= i) return;
					int pivotindex = findpivot(A, i, j);
					swap(A, pivotindex, j);
					int k = partition<Elem,Comp>(A, i-1, j, A[j]);
					swap(A, k, j);
					qsort<Elem,Comp>(A, i, k-1);
					qsort<Elem,Comp>(A, k+1, j);
					}
					
					template <typename Elem>
					inline int findpivot(Elem A[], int i, int j)
					{ return (i+j)/2; }

					template <typename Elem,
					typename Comp>
					inline int partition(Elem A[],
					int l, int r, Elem& pivot)
					{
					do
					{
					while
					(Comp::prior(A[++l], pivot));
					while ((l < r) &&
					Comp::prior(pivot, A[--r]));
					swap(A, l, r);
					} while (l < r);
					return l;
					}

堆排序
Heapsort
（内排序）
	$\Theta(n\log n)$	$\Theta(n\log n)$	O(1)	先把数组转化成为一个满足堆定义的序列，然后把堆顶最大元素取出来，再把剩下数值排成堆，并取出堆顶数值，如此下去直到堆为空。假设 n 个元素存储在数组中的 0 到 n-1 位置上，把堆顶元素取出来，并再度调整时，应该把它置于数组的第 n-1 个位置，这时堆中元素的数目为 n-1 个。取出新的最大值，并把这个第二大值放到数组的第 n-2 个位置。依次移除剩下的元素之后，就排出了一个由小到大排列的数组。这就是要用最大值堆而不用最小值堆的原因。	template <typename Elem,
					typename Comp>
					void heapsort(Elem A[], int n)
					{
					Elem maxval;
					heap<Elem,Comp> H(A, n,
					n);
					for (int i=0; i<n; i++)
					maxval = H.removefirst();
					}

基数排序
Radix Sort
（内排序）
	$\Theta(nk+rk)$
	n 为数据个数
	r 为基数
	k 为位数
		$\Theta(nk+rk)$	$O(n+r)$	首先根据个位数的数值，在走访数值时将它们分配至编号 0 到 9 的桶子中；接着再进行一次分配，这次是根据十位数来分配；这时候整个数列已经排序完毕；如果排序的对象有三位数以上，则持续进行以上的动作直至最高位数为止。	template <typename Elem,
					typename getKey>
					void radix(Elem A[], Elem B[],
					int n, int k, int r, int cnt[])
					{
					int j;
					for (int i=0, rtoi=1; i<k; i++,
					rtoi*=r)
					{
					for (j=0; j<r; j++)
					cnt[j] = 0;
					for (j=0; j<n; j++)

					cnt[(getKey::key(A[j])/rtoi)%r]++;
					for (j=1; j<r; j++)
					cnt[j] = cnt[j-1] + cnt[j];
					for (j=n-1; j>=0; j--)
					B[--cnt[(getKey::key(A[j])/rtoi)%r]] = A[j];
					for (j=0; j<n; j++)
					A[j] = B[j];
					}
					}

三种简单排序算法的渐近复杂度比较

	插入排序 Insertion	冒泡排序 Bubble	选择排序 Selection
比较情况：Comparisons			
最佳情况 Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
平均情况 Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
最差情况 Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
交换情况：Swaps			
最佳情况 Best Case	0	0	$\Theta(n)$
平均情况 Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
最差情况 Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

检索

散列方法、散列函数冲突解决方法

（会哈希函数和使用开放定址法的二次探测再散列方法解决冲突）

1、根据关键码值直接访问表，可以通过一些计算，把关键码值映射到数组中的位置来访问记录，这个过程称为散列（hashing）。

2、把关键码值映射到位置的函数称为散列函数（hash function），通常用 h 表示。存放记录的数组称为散列表（hash table），用 HT 表示。散列表中的一个位置称为一个槽（slot）。散列表 HT 中槽的数目用变量 M 表示，槽从 0 到 M-1 编号。设计散列系统的目的是使得对于任何关键码值 K 和某个散列函数 h，i=h(K)是表中满足 0≤h(K)＜M 的一个槽，并且记录在 HT[i]存储的关键码值与 K 相等。

3、散列方法通常不适用于允许多条记录有相同关键码值的应用程序。散列方法一般不适用于范围检索。

4、需要设计一个散列函数，允许把记录存储在一个更小的表中，由于可能的关键码值范围超过表

长，至少有些槽会被多个关键码值映射到。对于一个散列函数 h 和两个关键码值k_1、k_2，如果有$\mathrm{h}(k_1)=\beta=\mathrm{h}(k_2)$，其中$\beta$是表中的一个槽，那么就说$k_1$和$k_2$对于$\beta$在散列函数下有冲突（collision）。

5、冲突解决技术可以分为两类：开散列方法（open hashing，也称为单链方法：separate chaining）和闭散列方法（closed hashing，也称为开地址方法：open addressing）。这两种方法的不同之处与冲突记录的存储有关，开散列方法把冲突记录存储在表外，比散列方法把冲突记录存储在表中另一个槽内。

6、开散列方法的最简单形式把散列表中的每个槽定义为一个链表的表头，散列到一个槽的所有记录都放到这个槽的链表内。这个表中的每一个槽存储一条记录，以及一个指向链表其余部分的指针。槽链接的链表中的记录可以按照多种方式排列：按照插入次序排列、按照关键码值次序排列，或者按照访问频率次序排列。

7、闭散列方法把所有记录直接存储到散列表中。每条关键码值标记为k_R，记录R有一个基槽，就是$\mathrm{h}(k_R)$，即由散列函数计算出来的槽。如果要插入一条记录R，而另一条记录已经占据了R的基槽，那么就把R存储在表的其他槽内，由冲突解决策略确定应该是哪个槽。

8、桶式散列，一种实现闭散列的方法，即把散列表中的槽分成多个桶，把散列表中的M个槽分成B个桶，每个桶中包含M/B个槽。散列函数把每一条记录分配到某个桶的第一个槽中。如果这个槽已经被占用，那么就顺序地沿着桶查找，直到找到一个空槽，如果一个桶全部被占满了，那么就把这条记录存储在表后具有无限容量的溢出桶中。所有的桶共享同一个溢出桶。

9、桶式散列的一个简单变体是把关键码值散列到散列表的某些槽中。如果基槽已经被占用，那么在寻找一个用来放置记录的空槽时，就把记录压向桶的后面。如果已经到达桶底，那么冲突解决例程就到桶的上面来继续查找空槽。例如，假定桶中包含 8 条记录，第一个桶由第 0 个槽到第 7 个槽组成。如果一条记录被散列到第 5 个槽，冲突解决过程就会按照 5、6、7、0、1、2、3、4 的顺序检查各个槽是否为空，试图把记录插入槽中。如果桶中所有的槽都满了，那么就把记录放到溢出桶中。这种方法的优点是减少了初始冲突，这是因为所有的槽都可以是基槽，而不仅仅是桶中的第一个槽。

10、在插入期间，冲突解决策略的目标就是当记录的基槽已经被占用时，在散列表中找到一个空槽。可以把任何冲突解决方法都看成产生一组有可能放置记录的散列表的槽。这组槽中的第一个就是关键码的基槽。如果基槽已经被占用，冲突解决策略就会到达这个组中的下一个槽。如果这个槽也被占用了，那么就必须找到另一个槽，以此类推。这组槽就称为冲突解决策略产生的探查序列（probe sequence），而且探查序列是由称为探查函数的函数 p 产生的。

11、桶式散列也是线性探查的冲突解决技术的一个例子。清除基本聚集的另一种技术称为二次探查（quadratic probing），对于函数$\mathrm{p}(K,i)=c_1i^2+c_2i+c_3$在某种程度上就是二次探查函数，最简单的变体就是$\mathrm{p}(K,i)=i^2$。对于一个长度$M=101$的散列表，假定关键码$k_1$、$k_2$，$\mathrm{h}(k_1)=30$，$\mathrm{h}(k_2)=29$，$k_1$的探查序列是 30、31、34、39，$k_2$的探查序列是 29、30、33、38，这两个关键码的探查序列此后就立即分开了。

第四部分 高级数据结构

第 11 章 图

图的基本术语、邻接矩阵和表的表示方法、
DFS、BFS、拓扑排序、最短路径、最小支撑树、
Prim 算法、Kruskal 算法（全部深入了解）
矩阵表示法、计算存储位置

1、图可以用$G=(V,E)$来表示，每个图中都包括一个顶点集合V和一个边集合E，其中E中的每条边都是V中某一对顶点之间的连接。顶点总数记为$|V|$，边的总数记为$|E|$，$|E|$的取值范围是从 0 到$|V|^2-|V|$。边数较少的图称为稀疏图，边数较多的图称为密集图 u，包括所有可能边的图称为完全图。

2、如果图的边限定为从一个顶点指向另一个顶点，则称这个图为有向图。如果图中的边没有方向性，则称之为无向图。如果图中各顶点均带有标号，则称之为标号图。

3、一条边所连接的两个顶点称之为相邻的，这两个顶点互称为邻接点。连接一对邻接点U、V的边被称为与顶点U、V相关联的边，记作(U,V)。每条边都可能附带一个值，称为权。边上标有权的图称为带权图。

4、图的邻接矩阵是一个$|V|\times|V|$数组。假设$|V|=n$，各顶点依次记为$v_0,v_1,...,v_{n-1}$，则相邻矩阵的第i行包括所有以v_i为起点的边。如果从v_i到v_j存在一条边，则对第i行的第j个元素进行标记，否则不做标记。相邻矩阵的空间代价都为$\Theta(|V|^2)$。

5、图的邻接表是一个以链表为元素的数组。这个数组中包含$|V|$个元素，其中第i个元素存储的是一个指针，它指向顶点v_i的边构成的链表。这个链表通过存储顶点v_i的邻接点来表示对应的边。邻接表的空间代价为$\Theta(|V|+|E|)$。

6、两种图的表示法都可以用来存储有向图和无向图。无向图中连接某两个顶点U和V的边可以用两条有向边来代替：一条从U到V，一条从V到U。

7、如果使用邻接表，则查看它的所有$|V|$条可能的边，其时间代价就是$\Theta(|V|+|E|)$；而邻接矩阵的时间代价为$\Theta(|V|^2)$，因此相邻矩阵在图的算法中常常带来更高的时间代价。

8、图的遍历算法通常为图的每个顶点保留一个标志位（mark bit）。在算法开始时，图的所有顶点的标志位清零。在遍历过程中，当某个顶点被访问时，就会设置其标志位。如果在遍历过程中遇到某个顶点被设置了标志位，就不再访问它。遍历算法一结束，就可以通过检查标志位数组来确定是否已处理了所有顶点。如果还有未标记的顶点，可以从某个未被标记的顶点开始继续遍历。

9、深度优先搜索（depth-first search，DFS），在搜索过程中，每当访问某个顶点V时，DFS 就会递归地访问它的所有未被访问的相邻顶点。同样，DFS 把所有从顶点V出去的边存入栈中。从栈顶弹出一条边，根据这条边找到顶点V的一个相邻顶点，这个顶点就是下一个要访问的顶点，对这个顶点重复对顶点V的操作。这样做的结果就是沿着图的一个分支一直处理下去，完成这个分支后再回溯处理下一个分支，以此类推。深度优先搜索过程中将会产生一条深度优先搜索树，这个树的结构由遍历过程中所有连接某一新的（未被访问的）顶点的边组成。在有向图中，DFS 对每一条边

处理一次；在无向图中，DFS 对每一条边都从两个方向处理，时间代价为$\Theta(|V|+|E|)$。

10、
```
void DFS(Graph* G, int v)
    {
        PreVisit(G, v);
        G->setMark(v, VISITED);
        for (int w=G->first(v); w<G->n(); w = G->next(v,w))
            if (G->getMark(w) == UNVISITED)
                DFS(G, w);
        PostVisit(G, v);
    }
```
11、广度优先搜索（breadth-first search，BFS），用队列代替栈，在进一步深入访问其他顶点之前，检查起点的所有相邻顶点。如果把图结构看成一个树结构，而且图的起点作为树的根节点，则 BFS 将从顶至底逐层地对各个结点进行访问（树的层次遍历）。

12、
```
void BFS(Graph* G, int start, Queue<int>* Q)
    {
        int v, w;
        Q->enqueue(start);
        G->setMark(start, VISITED);
        while (Q->length() != 0)
          {
            v = Q->dequeue();
            PreVisit(G, v);
            for (w=G->first(v); w<G->n(); w = G->next(v,w))
                if (G->getMark(w) == UNVISITED)
                {
                    G->setMark(w, VISITED);
                    Q->enqueue(w);
                }
        }
    }
```
13、拓扑排序（topological sort），将一个有向无环图（DAG）中所有顶点在不违反前置依赖条件规定的基础上排成线性序列的过程。可以通过对图进行深度优先搜索来寻找拓扑序列，也可以用队列代替递归来实现拓扑排序。首先访问所有的边，计算指向每个顶点的边数（即计算每个顶点的前置条件数目）。将所有没有前置条件的顶点放入队列，然后开始处理队列。当从队列中取出一个顶点时，把它打印出来，同时将其所有相邻顶点（那些以顶点V作为前置条件的顶点）的前置条件计数减 1.当某个相邻顶点的前置条件计数为 0 时，就将其放入队列。如果还有顶点没有被打印，而队列已经为空，则图中必然包含回路（即不可能在不违反任何前置条件的情况下为这些任务安排一个合理的顺序）。

14、基于 DFS 算法实现的拓扑排序（递归）：
```
void topsort(Graph* G)
{
    int i;
    for (i=0; i<G->n(); i++)
        G->setMark(i, UNVISITED);
    for (i=0; i<G->n(); i++)
        if (G->getMark(i) == UNVISITED)
            tophelp(G, i);
}
```

```
void tophelp(Graph* G, int v)
{
    G->setMark(v, VISITED);
    for (int w=G->first(v); w<G->n(); w =
G->next(v,w))
        if (G->getMark(w) == UNVISITED)
            tophelp(G, w);
    printout(v);
}
```

14、基于队列实现的拓扑排序：

```
void topsort(Graph* G, Queue<int>* Q)
{
    int Count[G->n()];
    int v, w;
    for (v=0; v<G->n(); v++) Count[v] = 0;
    for (v=0; v<G->n(); v++)
    for (w=G->first(v); w<G->n(); w =
G->next(v,w))
    Count[w]++;
    for (v=0; v<G->n(); v++)
        if (Count[v] == 0)
            Q->enqueue(v);
    while (Q->length() != 0)
    {
        v = Q->dequeue();
        printout(v);
        for (w=G->first(v); w<G->n(); w =
G->next(v,w))
        {
            Count[w]--;
            if (Count[w] == 0)
                Q->enqueue(w);
        }
    }
}
```

15、单源最短路径（single-source shortest paths），在图G中给定一个顶点S，找出从顶点S到G中所有其他顶点的最短路径。也许只需要找到顶点S与顶点T之间的最短路径，但是在最差情况下，在寻找从顶点S与顶点T的最短路径时，也会同时寻找从顶点S到其他任何一个顶点的最短路径。对于无权图（或图中所有边的权相等），通过简单的广度优先搜索就可以找到单源最短路径。Dijkstra 算法可以寻找当各边权不相等时的最短路径，起始顶点为顶点 A，除了顶点 A 以外，其余的各个顶点的最短路径长度初始值均赋初值为∞。处理完顶点 A 后，它的相邻顶点的最短路径长度估计值被更新为到顶点 A 的直接距离。处理完顶点 C（离顶点 A 最近的顶点）后，顶点 B、顶点 E 的估计值被更新为相应的经过顶点 C 的最短路径长度。其余顶点按照顶点 B、顶点 D、顶点 E 的顺序处理。

16、最小支撑树（minimum-cost spanning tree，MST）问题的输入是一个连通的无向图G，图中的每一条边都有附带的长度或权值。MST 是一个包括图G中的所有顶点及其一部分边的图，这些边是图G所有边集合的子集，且要满足这个子集中所有边的权之和为所有子集中最小的且子集中的边能保证图是连通的。MST 中没有回路，MST 是一个有$|V|-1$条边的自由树结构。

17、Prim 算法，从图中任意一个顶点 N 开始，初始化 MST 为 N。选出与顶点 N 相关联的边中权最小的一条边，设其连接顶点 N 与另一个顶点 M。把顶点 M 和边(N,M)加入到 MST 中。接下来，选出与顶点 N 或顶点 M 相关联的边中权值最小的一条边，设其连接到另一个新顶点，将这条边和新顶点添加到 MST 中，反复进行这样的处理，每一步都选出一条边来扩展 MST，这条边是连接当前已经在 MST 中的某个顶点与一个不在 MST 中的顶点组成的边集合中代价最小的那条边。Prim 算法与 Dijkstra 算法的区别主要在于 Prim 算法不是要寻找下一个离起始顶点最近的顶点，而是下一个与 MST 中某个顶点距离最近的顶点。

18、

```
void Prim(Graph* G, int* D, int s)
    {
        int V[G->n()];
        int i, w;
        for (i=0; i<G->n(); i++)
```

```
        {
            int v = minVertex(G, D);
            G->setMark(v, VISITED);
            if (v != s)
                AddEdgetoMST(V[v], v); // Add edge to MST
            if (D[v] == INFINITY) return; // Unreachable vertices
            for (w=G->first(v); w<G->n(); w = G->next(v,w))
                if (D[w] > G->weight(v,w))
                {
                    D[w] = G->weight(v,w); // Update distance
                    V[w] = v; // Where it came from
                }
        }
    }
```

19、Kruskal 算法，它首先将顶点集合分为$|V|$个等价类，每个等价类中包含一个顶点。然后按照权的大小顺序处理每一条边。如果一条边连接属于两个不同等价类的顶点，那么就把这条边添加到 MST 中，并把这两个等价类合并为一个等价类。反复执行这个过程，直到最后只剩下一个等价类。用最小值堆来实现按照权的大小顺序处理每一条边，用树结构的基于父指针表示法的并查算法来确定两个顶点是否属于同一个等价类，算法的总时间代价为$\Theta(|E|\log|E|)$。

20、class KruskElem

```
//KruskElem 类用于在最小值堆中存储边
{
public:
    int from, to, distance;
    KruskElem() { from = -1; to = -1; distance =
-1; }
    KruskElem(int f, int t, int d)
    { from = f; to = t; distance = d; }
};

void Kruskel(Graph* G)
{
    ParPtrTree A(G->n());
    KruskElem E[G->e()];
    int i;
    int edgecnt = 0;
    for (i=0; i<G->n(); i++)
    for (int w=G->first(i); w<G->n(); w =
G->next(i,w))
    {
        E[edgecnt].distance = G->weight(i, w);
        E[edgecnt].from = i;
        E[edgecnt++].to = w;
    }
    heap<KruskElem, Comp> H(E, edgecnt,
edgecnt);
    int numMST = G->n();
    for (i=0; numMST>1; i++)
    {
        KruskElem temp;
        temp = H.removefirst();
        int v = temp.from; int u = temp.to;
        if (A.differ(v, u))
        {
            A.UNION(v, u);
            AddEdgetoMST(temp.from,
        temp.to);
            numMST--;
        }
    }
}
```