# 时间复杂度

2017年2月25日　19:05

时间复杂度（执行次数）：关注最高项的阶数，忽略常数项、低阶项。
一般复杂度指的是空间复杂度

- 求解O(n)的方法：
  1. 用常数1取代加法常数
  2. 在修改后的运行次数函数中只保留最高阶项
  3. 如果最高阶项存在且不是1，则去除与这个项相乘的系数

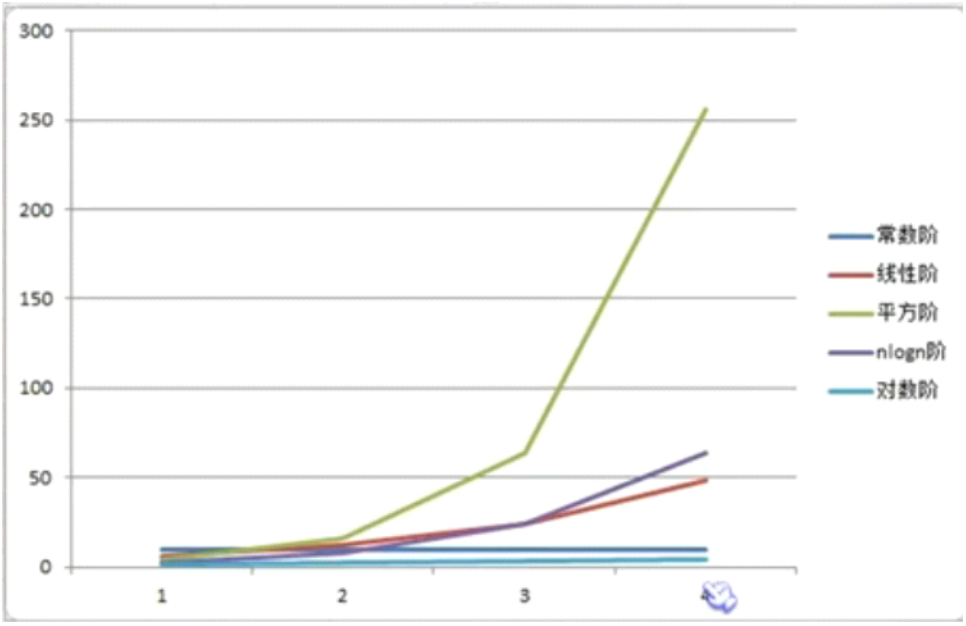
- 线性阶O(n): 一般含有非嵌套循环
- 平方阶O(n^2): 双层循环

```
for (i = 0; i < n; i++) {
    for (j = i; j < n; j++) {
        //...
    }
}
```

- 对数阶O( log2(n) )

```
int i = 1, n = 100;
while (i < n) {
    i = i * 2;
}
```

由于每次i*2之后，就举例n更近一步，假设有x个2相乘后大于或等于n，则会退出循环。
于是由2^x = n得到x = log(2)n，所以这个循环的时间复杂度为O(logn)。




常用的时间复杂度所耗费的时间从小到大依次是：O(1) < O(logn) < (n) < O(nlogn) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)

# 空间复杂度

2017年2月25日 20:00

# 指针

2017年2月26日 9:06

```c
#include <stdio.h>

int main(void)
{
    int *p; //p是一个变量名字，int *表示p变量只能存储int类型变量的地址
    int i = 10;
    int j;

    p=&i;   //p存放i的地址
    j = *p; //把p指向的对象的内容赋值给j
    printf("i=%d, j=%d, *p=%d\n, p=%d\n", i,j,*p, p);

    return 0;
}
```

```c
#include <stdio.h>

void f(int* i)
{
    *i = 100;   //i指向的对象的内容更改为100
}
int main(void)
{
    int i = 9;
    f(&i);
    printf("%d\n", i);
    return 0;
}
```

# 内存空间的申请和释放

2017年2月26日 16:14

C语言： malloc(), free()

一个malloc对应一个free。而不是一个字节free一次。当销毁栈的时候，只需要free(s->bottom)一次即可.

# 线性表

2017年2月25日 20:42

## 2.1.1 什么叫线性表

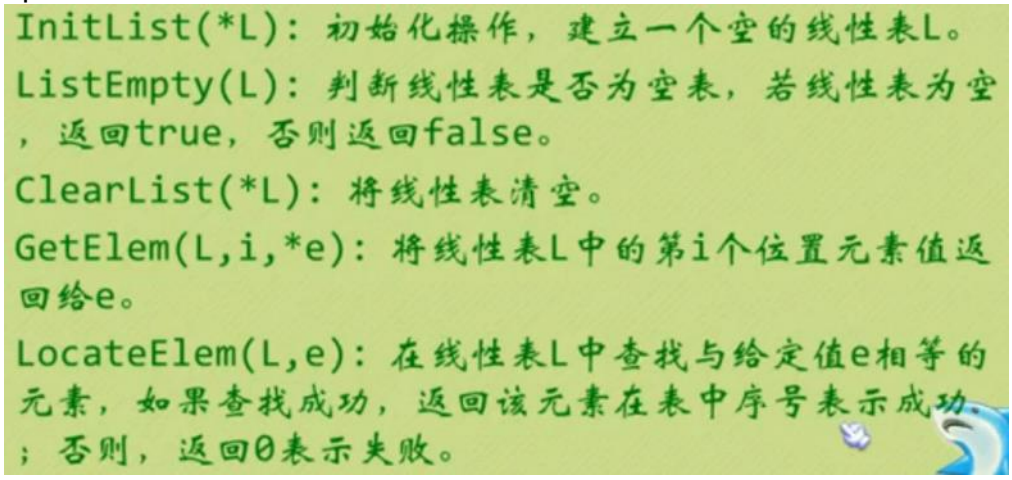
线性表数据结构具有以下特征：

- 有且只有一个“首元素”；
- 有且只有一个“末元素”；
- 除末元素之外，其余元素均有惟一的后继元素；
- 除首元素之外，其余元素均有惟一的前驱元素。

对于线性表，主要可进行以下操作：

- 添加结点；
- 插入结点；
- 删除结点；
- 查找结点；
- 遍历结点；
- 统计结点数。

```
ADT list
Data
    {a1, a2, …} //ai是ElemType类型的数据

Operation
```



ListInsert(*L,i,e): 在线性表L中第i个位置插入新元素e。

ListDelete(*L,i,*e): 删除线性表L中第i个位置元素，并用e返回其值。

ListLength(L): 返回线性表L的元素个数。

endADT

# 顺序存储结构

2017年2月25日　　20:25

- 在读、存操作，时间复杂度是O(1)
- 在插入和删除操作，时间复杂度是O(n)
- 适合在读取操作使用
- 缺点：易造成内存空间浪费

```
// 顺序存储结构封装
#define MAXSIZE 20  // 线性表最大存储长度
typedef int ElemType;
typedef struct{
    ElemType data[MAXSIZE];
    int length; // 线性表当前长度
} SqList;
```

LOC(ai+1)=LOC(ai)+c; //c是一个元素所占的存储单元字节数

A=A∪B

- ListLength()
- GetElem()
- LocateElm()
- ListInsert()

```
// La表示A集合，Lb表示B集合。
void unionL(List *La, list Lb)
{
    int La_len, Lb_len, i;

    ElemType e;
    La_len = ListLength(*La);
    Lb_len = ListLength(Lb);

    for( i=1; i <= Lb_len; i++ )
    {
        GetElem(Lb, i, &e);
        if( !LocateElem(*La, e) )
        {
            ListInsert(La, ++La_len, e);
        }
    }
}
```

//SqList.h

```h
#ifndef SQLIST_H
#define SQLIST_H
#include <iostream>
#include <string>
using namespace std;
#define MAXSIZE 100
#define ERROR 1
#define OK 0

class DATA
{
public:
    char key[15];   //结点关键字
    char name[20];
    int age;
};
class SqList
{
private:
    DATA listData[MAXSIZE];
    int listLength;
public:
    void SeqListInit(SqList* SL);
    int SeqListLength(SqList* SL);              //返回顺序表元素数量
    int SeqListAdd(SqList* SL, DATA data);      //添加元素到顺序表尾部
    int SeqListInsert(SqList* SL, int i, DATA data);
    int SeqListDelete(SqList* SL, int i);
    DATA* SeqListFindByNum(SqList* SL, int i);  //根据序号返回指向元素的指针
    int SeqListFindByKey(SqList* SL, char* key); //根据关键字查找
    int SeqListAll(SqList* SL);                 //遍历
};
#endif
```

//SqList.cpp

```cpp
#include <iostream>
#include "SqList.h"

void SqList::SeqListInit(SqList* SL)
{
    SL->listLength = 0;
}
int SqList::SeqListLength(SqList* SL)
{
    return SL->listLength;
}
int SqList::SeqListAdd(SqList* SL, DATA data)
{
    if (SL->listLength >= MAXSIZE)
    {
        cout << "顺序表已满！" << endl;
        return ERROR;
    }
    SL->listData[++SL->listLength]= data;
    return OK;
}
```

```cpp
int SqList::SeqListInsert(SqList* SL, int i, DATA data)
{
    int j;

    if (SL->listLength >= MAXSIZE)
    {
        cout << "顺序表已满！" << endl;
        return ERROR;
    }
    if (i<1 || i>SL->listLength + 1)
    {
        cout << "插入位置超出范围！" << endl;
        return ERROR;
    }
    for (j = SL->listLength - 1; j >= i + 1; j--)
    {
        SL->listData[j + 1] = SL->listData[j];
    }
    SL->listData[i] = data;
    ++SL->listLength;
    return OK;
}
```

```
int SqList::SeqListDelete(SqList* SL, int i)
{
    int j;

    if (SL->listLength <= 0)
    {
        cout << "空表！" << endl;
        return ERROR;
    }
    if (i<1 || i > SL->listLength)
    {
        cout << "删除序号超出范围！" << endl;
        return ERROR;
    }
    for (j = i; j <= SL->listLength - 1; j++)
    {
        SL->listData[j - 1] = SL->listData[j];
    }
    SL->listLength--;
}
DATA* SqList::SeqListFindByNum(SqList* SL, int i)
{
    if (i<1 || i > SL->listLength)
    {
        cout << "查找序号超出范围！" << endl;
        return NULL;
    }
    return &(SL->listData[i - 1]);  //数组下标比顺序表索引少1
}
int SqList::SeqListFindByKey(SqList* SL, char* key)
{
    int j;

    for (j = 0; j <= SL->listLength - 1; j++)
    {
        if (strcmp(SL->listData[j].key, key)==0)
        {
            return j + 1;
        }
    }
    return OK;
}
int SqList::SeqListAll(SqList* SL)
{
    int j;

    for (j = 0; j < SL->listLength; j++)
    {
        printf("(%s,%s,%d)\n", SL->listData[j].key, SL->listData[j].name, SL->listData[j].age);
    }
    return OK;
}
```

插入和删除操作时间复杂度：

最好情况，i是最后一个位置，O(1)

最坏情况，i是第一个位置，O(n)

平均复杂度O(n)

# 链式存储结构

2017年2月25日　　22:04

```
//节点数据域类型
typedef struct ElemType
{
    int age;
    char name[10];
}ElemType;
//节点类型
typedef struct Node
{
    ElemType data;
    struct Node* next;
}Node;
typedef struct Node* LinkList;
```

**头指针**：指向第一个结点的指针，头指针是必要元素

**头结点**：为了操作的统一而设立的，一般数据域为空

如果是无头节点的链表，头指针指向第一个非空节点

如果是含头节点的链表，头指针指向头节点



例如：p是指向第i个元素的指针，p->data表示数据域，p->next表示指针域

- 定义链表结构

```c
#include <stdio.h>
#include <stdlib.h>
using namespace std;

typedef struct DataType
{
    int age;
    char name[10];
}ElemType;

typedef struct Node
{
    ElemType data;      //数据域可以一个ElemType类型的变量
    struct Node *next;  //指针域指向另一结点
} NODE, *PNODE; // NODE等价于struct Node, PNODE等价于struct Node *


PNODE CreateListTail(void); //尾插法建立单链表
bool isEmpty(PNODE pHead);
int listLength(PNODE pHead);
```

# 两种结构性能比较

2017年2月26日 15:50

一、时间性能

查找：

顺序结构O(1)

单链表O(n)

插入/删除：

顺序结构O(n)

单链表O(1)

二、空间性能

顺序结构低

# 单链表

2017年2月26日　　17:56

## 建立单链表

一、头插法

从一个空表开始，生成新结点。把新加进的元素放在表头后的第一个位置

1. 新结点的next指向头节点之后
2. 表头的next指向新节点

```
/*
 * n:链表节点数
 * (*list):指向链表的头节点的指针
 */
void CreateLinkHead(LinkList *list, int n)
{
    int j;
    LinkList p;

    srand(time(0)); //初始化随机数种子，需要添加“stdlib.h","time.h"
    *list = (LinkList)malloc(sizeof(Node));
    (*list)->next = NULL;    //刚开始建立空表，头节点为空

    for (j = 0; j < n; j++)
    {
        p = (LinkList)malloc(sizeof(Node)); //生成新节点
        p->data.age = rand() % 100 + 1; //1~100随机数
        p->next = (*list)->next;
        (*list)->next = p;
    }
}
```

缺点：输入“abc",输出"cba",倒序了

二、尾插法

新加入的元素放在表尾

1. 表尾节点的next指向新节点
2. 表尾节点更新为这个新节点

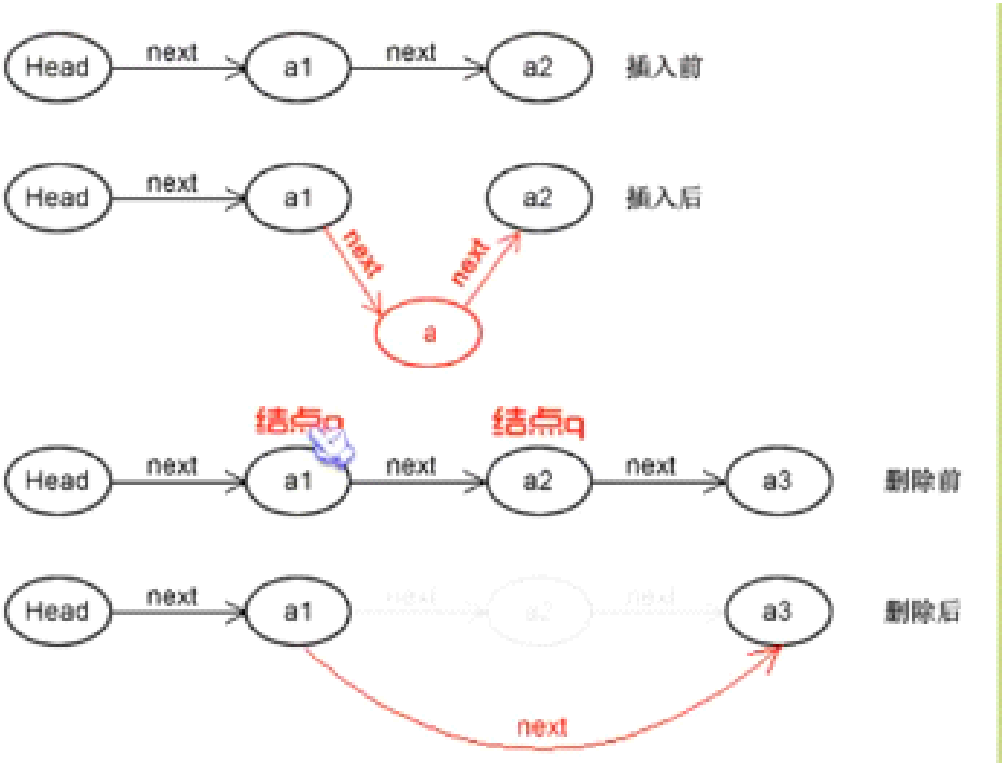
```
PNODE CreateListTail(void)
{
    int j, len, value;
    PNODE pNew, pTail, pHead;

    pHead = (PNODE)malloc(sizeof(Node));

    if (pHead == NULL)
    {
        printf("分配失败，程序终止！\n");
        exit(-1);
    }
    pTail = pHead;  //刚开始建立空表时，r指向头节点
    pTail->next = NULL;

    printf("请输入您要生成的链表节点个数：\n");
    scanf_s("%d", &len);

    for (j = 0; j < len; j++)
    {
        printf("请输入第%d个节点的值：", j + 1);
        scanf_s("%d", &value);

        pNew = (PNODE)malloc(sizeof(Node)); //生成新节点
        if (pNew == NULL)
        {
            printf("分配失败，程序终止！\n");
            exit(-1);
        }
        pNew->data.age = value;
        pTail->next = pNew;
        pTail = pNew;
        pTail->next = NULL;
    }
    return pHead;
}
```

- 遍历链表

```
void listTraverse(PNODE pHead)
{
    PNODE p;

    p = pHead->next;
    while (p != NULL) //如果链表不为空
    {
        printf("%d\n", p->data);
        p = p->next;
    }//退出循环的条件：p指向最后一个元素，此事p的指针域为NULL
}
```

- 获取链表中第i个结点

```c
int GetElem(LinkList list, int i, ElemType* e)
{
    int j;
    LinkList p;

    j = 1;
    p = list->next; //从链表（含头结点）的第1个结点开始查找

    while (p && j<i) //若p为空，意味着指向了最后一个结点的指针域
    {
        p = p->next;
        ++j;
    }
    if (!p || j>i)
    {
        return ERROR;
    }
    *e = p->data;
    return OK;
}
```




- 在链表的第i个位置后面插入结点(s)

```
newP->next = p->next;
p->next = newP;
```

```c
void listInsert(PNODE pHead, int pos, ElemType val)
{
    PNODE p, newP;
    int i, length;

    p = pHead->next;
    length=listLength(pHead);

    if (pos<1 || pos>length + 1)
    {
        printf("插入位置不合法！\n");
        exit(-1);
    }

    while (p != NULL && i<pos-1)  //找寻待插入的位置，
    {
        p = p->next;
        ++i;
    }

    newP = (PNODE)malloc(sizeof(NODE));
    if (newP == NULL)
    {
        printf("动态内存分配失败！\n");
        exit(-1);
    }
    // 找到的条件是i==pos
    newP->data = val;
    newP->next = p->next;
    p->next = newP;
}
```

- 删除第i个位置的结点

```
tmp = p->next;
p->next = tmp->next;
```

```c
void listDelete(PNODE pHead)
{
    int j,i;
    PNODE p,tmp;

    j = 1;
    p = pHead;
    tmp = NULL;

    printf("请输入你要删除的结点序号：");
    scanf_s("%d",&i);
    if (p == NULL)
    {
        printf("此链表为空！\n");
        exit(-1);
    }
    while ( p!=NULL && j < i)
    {
        p = p->next;
        ++j;
    }
    if (j>i)
    {
        printf("超出范围！\n");
        exit(-1);
    }
    // j==i时找到
    tmp = p->next;
    p->next = tmp->next;
    free(tmp);
}
```

- 判断链表是否为空

```c
bool isEmpty(PNODE pHead)
{
    if (pHead->next == NULL)
        return true;
    else
        return false;
}
```

- 获取链表的长度

```c
int listLength(PNODE pHead)
{
    int length=0;
    // 注意pHead是头指针，它的指针域才是存放第一个结点
    PNODE tmp = pHead->next;

    while (tmp != NULL)
    {
        length++;
        tmp = tmp->next;
    }
    free(tmp);
    return length;
}
```

## 排序

```c
void listSort(PNODE pHead)
{
    int i, j, tmp, length;
    PNODE p, q;

    length = listLength(pHead);
    for (i = 0, p = pHead->next; i < length - 1; i++, p = p->next)
    {
        for (j = i + 1, q = p->next; j < length; j++, q = q->next)
        {
            if (p->data.age > q->data.age)
            {
                tmp = p->data.age;
                p->data.age = q->data.age;
                q->data.age = tmp;
            }
        }
    }
}
```

## 整表删除

1. 第一个节点赋值给P，
2. 循环执行：下一个节点赋值给q,释放p，将q赋值给p

```c
int ClearList(LinkList *list)
{
    LinkList p, q;

    p = (*list)->next; //初始化p指向链表的第一个节点

    while (p)
    {
        q = p->next; //记录p的下一个节点
        free(p);
        p = q;
    }

    (*list)->next = NULL; //删除完毕，头指针的指针域置空
    return OK;
}
```

# 循环链表

2017年2月26日　　16:34

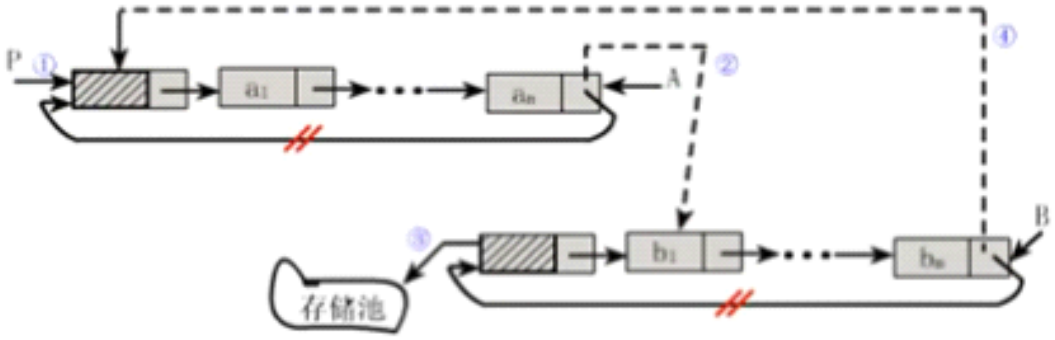
约瑟夫问题：

由第1个人开始报数，每报数到第3人该人就必须自杀，然后再由下一个重新从1开始报数

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node
{
    int data;
    struct Node* next;
}Node;

/*
*n: 节点数
*p: 指向当前结点的指针
*s: 指向新节点的指针
*head: 头节点
*/
Node* create(int n)
{
    Node *p, *head,*s;
    int i;

    i = 1;
    head = (Node*)malloc(sizeof Node);
    p = head;
    s = NULL;

    if (n != 0) //节点数不为零
    {
        while (i <= n)
        {
            s = (Node*)malloc(sizeof(Node)); //创建新节点
            s->data = i;
            i++;
            p->next = s;
            p = s;      // 更新当前节点的指针
        }
        s->next = head->next; //最后一次生成的节点的next指向第一个节点
```

```
    }

    free(head);
    return s->next; //返回第一个节点的地址
}


int main(void)
{
    int n = 10; //节点数
    int m = 3;   //从当前节点开始报数，数到3的那个节点删除
    int i;
    Node* p = create(n);
    Node* tmp;

    while (p != p->next)
    {
        for (i = 1; i < m-1; i++)
        {
            p = p->next; //在这里p指向第2个节点
        }
        printf("%d\n",p->next->data); //要删除的节点的值

        //删除节点
        tmp = p->next;
        p->next = tmp->next;
        free(tmp);
        p = p->next;       //更新p指向下一次开始报数的节点
    }
    // 仅剩一个节点
    printf("%d\n",p->data);
    return 0;
}
```

- 空链表的判断 p==p->next
- 合并两个单链表




两个单循环链表的链接操作示意图

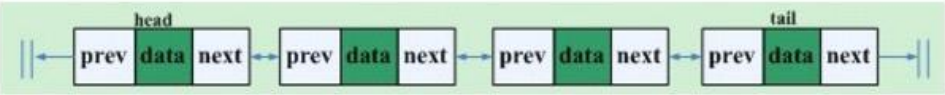
```
int Connect(LinkList LA, LinkList LB) // A-B的形式
{
    LinkList tmp;

    tmp = LA->next; //保存A表的头节点
    LA->next = LB->next->next;
    free(LB->next); //释放B表的头节点
    LB->next = tmp; //新表的表尾连接到原先A的表头节点

    return OK;
}
```

? 判断单链表中是否有环

方法一：p每次走1步，q每次走2步，如果一段时间后q==p,则存在环

# 双向链表

2017年2月26日　　22:14

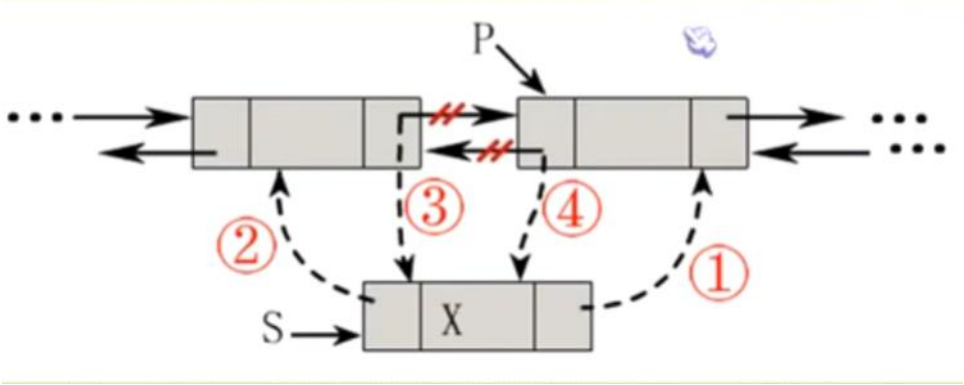



双向链表的结构：头指针（*head），尾指针（*tail）

每个节点的结构：数据(data)、前指针(*prev/ *prior)和后指针(*next)

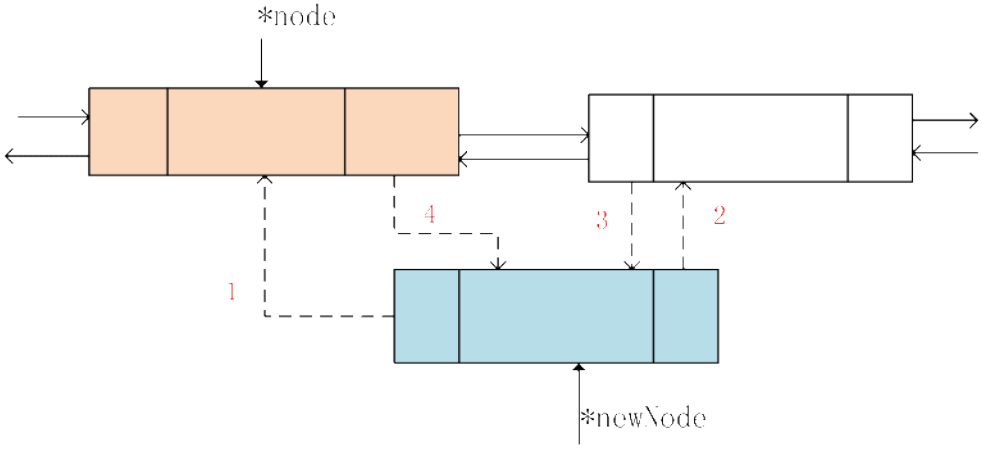
## 插入元素

- 前插




```
1. x->next = p;
2. x->prior = p->prior;
3. p->prior->next = s;
4. p->prior = s;
```

- 后插

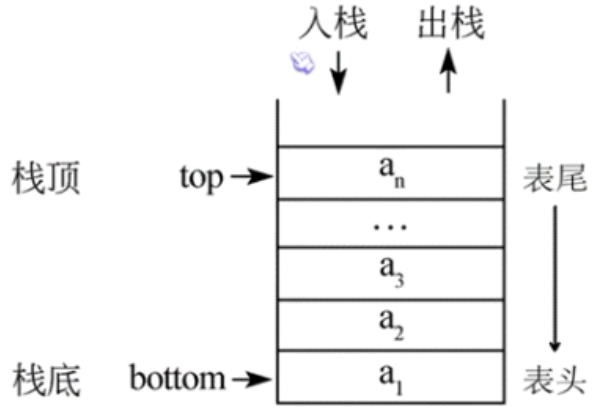



```
newNode->prior=node;
newNode->next=node->next;
node->next->prior=newNode;
node->next=newNode;
```

# 栈的顺序存储结构（常用）

2017年2月27日 14:59

栈只能在表尾（栈顶top）进行操作




- 插入push，入栈

```
void stackPush(SQSTACK *s, ELEMTYPE e)
{
    //如果栈满，增加空间
    if (((s->top) - (s->bottom)) >= s->stackSize)
    {
        //重新分配空间
        s->bottom = (ELEMTYPE*)realloc(s->bottom, (s->stackSize + SIZE_INCREMENT)*sizeof(ELEMTYPE));
        if (s->bottom == NULL)
        {
            exit(-1);
        }

        s->top = s->bottom + s->stackSize;  //设置栈顶
        s->stackSize += SIZE_INCREMENT;     //重新设置栈的容量
    }

    //先入栈再移动栈顶指针
    *(s->top) = e;
    s->top++;
    //入栈操作完毕，top指针指向一个空单元
}
```

- 出栈pop

```
void stackPop(SQSTACK *s, ELEMTYPE *e)
{
    if (s->bottom == s->top) //如果栈空
    {
        printf("error：栈空！\n");
        exit(-1);
    }
    else
    {
        s->top = s->top -1;
        *e = *(s->top); //top所指向的元素的值，存放在e指向的存储空间
    }
}
```

- 清空

```
void clearStack(SQSTACK *s)
{
    while (s->top != s->bottom)
    {
        s->top--;
    }
}
```

- 销毁

```
void destroyStack(SQSTACK *s)
{
    // int i;
    // for (i = 0; i < s->stackSize; i++)
    // {
    //   /*一个malloc对应一个free。而不是一个字节free一次。
    //      所以你只需要free一次即可，放在for里面就错了*/
    //      s->bottom++;
    // }
    free(s->bottom); //释放bottom指针指向的整个栈空间
    s->bottom = NULL;
    s->top = NULL;
    s->stackSize = 0;
}
```

- 当前栈的容量

```
STATUS stackLen(SQSTACK s)
{
    return (s.top - s.bottom);
}
```

- 遍历

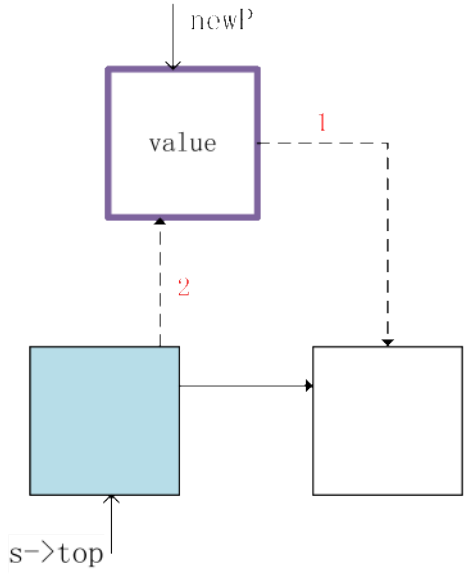
```
void stackTraverse(SQSTACK *s)
{
    int i = 0;
    ELEMTYPE *p = s->bottom;

    if (p == s->top)
    {
        printf("栈空！\n");
    }
    while (p != s->top)
    {
        printf("第%d个元素： ", i);
        printf("height = %d, width = %d, length = %d\n",
            p->heigth, p->width, p->length);
        ++p;
        ++i;
    }
}
```

# 栈的链式存储结构

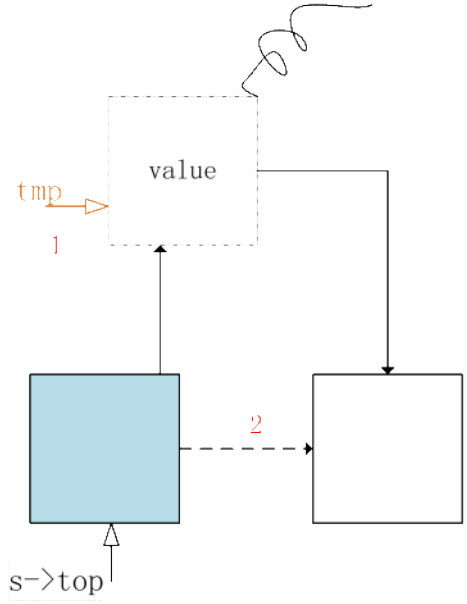
2017年2月28日　　12:41

栈顶放在单链表的表头，*top和*head合并

链栈的push操作相当于单链表中的后插入操作；




链表的pop操作相当于单链表中的删除操作



# 逆波兰表达式的堆栈实现

2017年2月28日 13:50

遇到数字，进栈

遇到操作符，将数据弹出栈

运算的到结果后，再入栈

。。。

正常表达式转换成逆波兰表达式

```
a+b         to   a b +
a+(b-c)     to   a b c - +
a+(b-c)*d   to   a b c - d * +
a+d*(b-c)   to   a d b c - * +
```

# 链队列

2017年2月28日　　15:42

链队列是一种只允许在队尾进行插入，在对首进行删除操作的特殊线性表。

“先进先出”


1．创建队列

（1）创建头结点

（2）头指针和尾指针指向这个生成的头结点

# 优先队列

2017年4月20日　　10:24

**堆有序**： 二叉树的每个节点都>=子节点（根节点是对有序的二叉树中的最大节点）
**二叉堆**：用堆有序的完全二叉树排序的元素，并在数组中按层级存储（不使用arr[0])
**堆的有序化**：打破堆的状态，再遍历堆并按要求恢复堆的状态

位置k的结点：父节点[k/2], 子节点[2k],[2k+1]

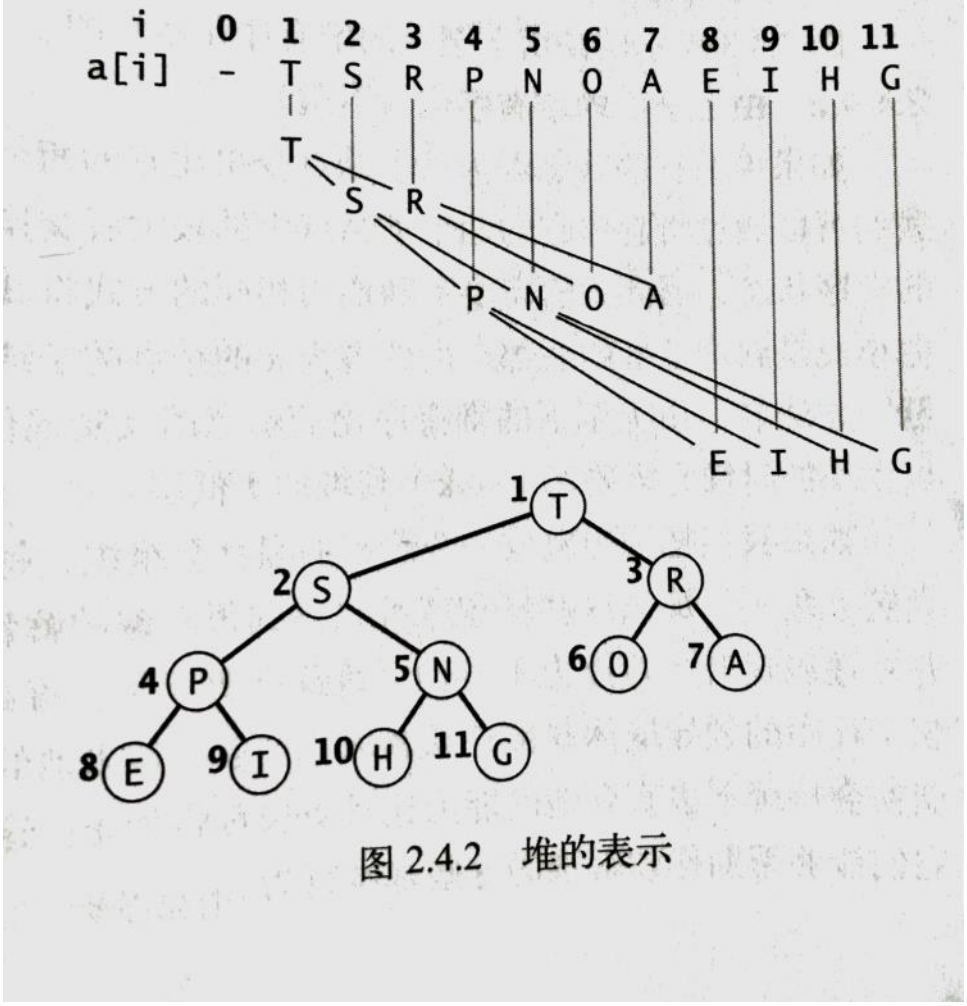


图 2.4.2　堆的表示

由下至上的堆有序化-上浮：某个结点优先级上升时　void swim(T a, int k)
由上至下的堆有序化-下沉：某个结点优先级下降时　void sink(T a, int k)

“队列：队尾插入，队首删除”
删除最大元素 (2lgN次比较)：从顶端删除最大元素，减小堆大小，将最后一个元素放到顶端并下沉
插入元素 (lgN+1次比较)：添加到数组末尾，增加堆大小，上浮新插入的元素

# 递归思想

2017年2月28日 19:18

递归：函数调用自己

每个递归定义必须至少有一个停止条件，当满足这个条件时递归不再进行。

```c
int Fib(int i)
{
    if (i < 2) //递归停止条件
        return ((i == 0) ? 0 : 1);
    else
        return Fib(i - 1) + Fib(i - 2);
}

int main(void)
{
    /*迭代*/
    //int i,n;
    //int a[40];
    //a[0] = 0;
    //a[1] = 1;
    //printf("%d %d ", a[0], a[1]);
    //
    //for (i = 2; i < 40; i++)
    //{
    //  a[i] = a[i - 1] + a[i - 2];
    //  printf("%d ", a[i]);
    //}

    /*递归*/
    int i;
    for (i = 0; i < 40; i++)
    {
        printf("%d ", Fib(i));
    }
    return 0;
}
```

当不确定循环次数时，采用递归思想方便

1. 将输入的任意长度的字符串反向输出

```c
//递归函数
void RecPrint()
{
    char c;
    scanf_s("%c", &c);
    if (c != '#')
    {
        //用户输入未结束，继续接收用户数据
        RecPrint();
    }
    /*当最后一次输入‘#’时不满足第一个if条件，
    返回到上一次调用的地方，继续执行后面的代码*/
    if (c != '#')
    {
        printf("%c", c);
    }

}

int main(void)
{
    /*递归*/
    RecPrint();
    return 0;
}
```

# 汉诺塔问题

2017年2月28日 20:26

将X上的64个圆盘移动到Z上，每次移动保证小盘在上，大盘在下。
X-----Y-----Z

步骤一：将X上的1~63个盘子借助Z移到Y上
步骤二：将最大的第64个盘子移到Z上
步骤三：将Y上的63个盘子借助X移到Z上

# BF算法

2017年3月1日 11:13

S主串，T子串

S[1]和T[1]比较，若相等，比较S[2]和T[2]

若不相等，比较S[1]和T[2]

…

# KMP算法

2017年3月1日 14:21

问题是由模式串决定的，而不是目标串
数组的0号元素存放字符串长度

next数组：指导模式串下一次改用第几号元素进行匹配。
如果j=0：表示两个字符串的指针都右移一个

# 低位优先-键索引计数法

2017年4月25日 星期二 19:35

适用：固定长度的字符串

思路：

1. 计算每个键出现的频率，count[a[i].key()+1] ++
2. 将频率转换为索引，count[r+1] += count[r]
3. 数据分类，将所有元素移动到辅助数组aux[]中进行排序，aux[count[a[i].key()]++] = a[i]
4. 回写 a[i]=aux[i]

基于R个字符编码的字母表，含有N个键长为W的字符串键，

访问次数~7WN+3WR，

额外空间~N+R

为一张用于排序的索引表是很容易的（请见图 5.1.3）。

```
for (i = 0; i < N; i++)
  count[a[i].key() + 1]++;
```

| | | count[] | | | | | |
|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 |
| 总是0 | | 0 | 0 | 0 | 0 | 0 | 0 |
| Anderson | 2 | 0 | 0 | 0 | **1** | 0 | 0 |
| Brown | 3 | 0 | 0 | 0 | 1 | **1** | 0 |
| Davis | 3 | 0 | 0 | 0 | 1 | **2** | 0 |
| Garcia | 4 | 0 | 0 | 0 | 1 | 2 | **1** |
| Harris | 1 | 0 | **0** | **1** | 1 | 2 | 1 |
| Jackson | 3 | 0 | 0 | 1 | 1 | **3** | 1 |
| Johnson | 4 | 0 | 0 | 1 | 1 | 3 | **2** |
| Jones | 3 | 0 | 0 | 1 | 1 | **4** | 2 |
| Martin | 1 | 0 | 0 | **2** | 1 | 4 | 2 |
| Martinez | 2 | 0 | 0 | 2 | **2** | 4 | 2 |
| Miller | 2 | 0 | 0 | 2 | **3** | 4 | 2 |
| Moore | 1 | 0 | 0 | **3** | 3 | 4 | 2 |
| Robinson | 2 | 0 | 0 | 3 | **4** | 4 | 2 |
| Smith | 4 | 0 | 0 | 3 | 4 | 4 | **3** |
| Taylor | 3 | 0 | 0 | 3 | 4 | **5** | 3 |
| Thomas | 4 | 0 | 0 | 3 | 4 | 5 | **4** |
| Thompson | 4 | 0 | 0 | 3 | 4 | 5 | **5** |
| White | 2 | 0 | 0 | 3 | **5** | 5 | 5 |
| Williams | 3 | 0 | 0 | 3 | 5 | **6** | 5 |
| Wilson | 4 | 0 | 0 | 3 | 5 | 6 | **6** |
| | | 0 | 0 | 3 | 5 | 6 | 6 |

第三组的总人数

图 5.1.2　计算出现频率

```
for (int r = 0; r < R; r++)
  count[r+1] += count[r];
```

| | count[] | | | | | |
|---|---|---|---|---|---|---|
| r | 0 | 1 | 2 | 3 | 4 | 5 |
| 总是0 | | | | | | |
| 0 | 0 | 0 | 3 | 5 | 6 | 6 |
| 1 | 0 | **0** | 3 | 5 | 6 | 6 |
| 2 | 0 | 0 | **3** | 5 | 6 | 6 |
| 3 | 0 | 0 | 3 | **8** | 6 | 6 |
| 4 | 0 | 0 | 3 | 8 | **14** | 6 |
| 5 | 0 | 0 | 3 | 8 | 14 | **20** |
| | 0 | 0 | 3 | 8 | 14 | 20 |

组号小于3的总人数（第三组在输出中的起始索引）

图 5.1.3　将频率转换为起始索引

### 5.1.1.3　数据分类

nt[4] 加 2，因为
示例中 count[1]

在这个示例中，
中的起始位置为
于每个键值 r，
count[] 转化

```
for (int i = 0; i < N; i++)
   aux[count[a[i].key()]++] = a[i];
```

count[]

| i | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 0 | 0 | 3 | 8 | 14 |
| 1 | 0 | 4 | 8 | 14 |
| 2 | 0 | 4 | **9** | 14 |
| 3 | 0 | 4 | **10** | 14 |
| 4 | 0 | 4 | 10 | 15 |
| 5 | 1 | 4 | 10 | 15 |
| 6 | 1 | 4 | **11** | 15 |
| 7 | 1 | 4 | 11 | 16 |
| 8 | 1 | 4 | **12** | 16 |
| 9 | 2 | 4 | 12 | 16 |
| 10 | 2 | 5 | 12 | 16 |
| 11 | 2 | 6 | 12 | 16 |
| 12 | 3 | 6 | 12 | 16 |
| 13 | 3 | 7 | 12 | 16 |
| 14 | 3 | 7 | 12 | 17 |
| 15 | 3 | 7 | **13** | 17 |
| 16 | 3 | 7 | 13 | 18 |
| 17 | 3 | 7 | 13 | 19 |
| 18 | 3 | 8 | 13 | 19 |
| 19 | 3 | 8 | **14** | 19 |
|  | 3 | 8 | 14 | 20 |
|  | 3 | 8 | 14 | 20 |

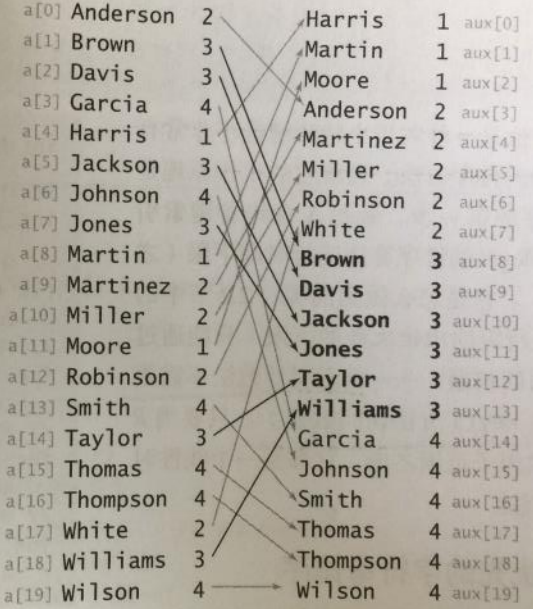



图 5.1.4　将数据分类（键为 3 的条目均突出显示）

# 高位优先-键索引计数法

2017年4月25日 星期二 21:22

适用：不定长字符串表

思路：

1. 计算每个键出现的频率，count[ charAt(a[i], d)+2] ++
2. 将频率转换为索引，`count[r+1] += count[r]`
3. 数据分类，将所有元素移动到辅助数组aux[]中进行排序，`aux[count[charAt(a[i],d)+1]++] = a[i]`
4. 回写 a[i]=aux[i-low]
5. 递归调用 `MSDsort(a, low+count[r], low+count[r+1]-1, d+1)`

平均查找字符个数：N$log_R N$

平均访问次数：8N+3R ~ 7WN+3WR ，W是字符串平均长度

令人无法承受的时间和空间。在仔细研究它的性能特点之前，我们要先讨论三个在任何应用中必须解决的重要的问题（这些问题曾在第 2 章中讨论过）。


图 5.1.11　高位优先的字符串排序：sort(a, 0, 14, 0) 的顶层轨迹

# 三向字符串快速排序

2017年4月26日 星期三 10:21

适用：重复键、有较长公共前缀的键、取值范围较小的键、小数组

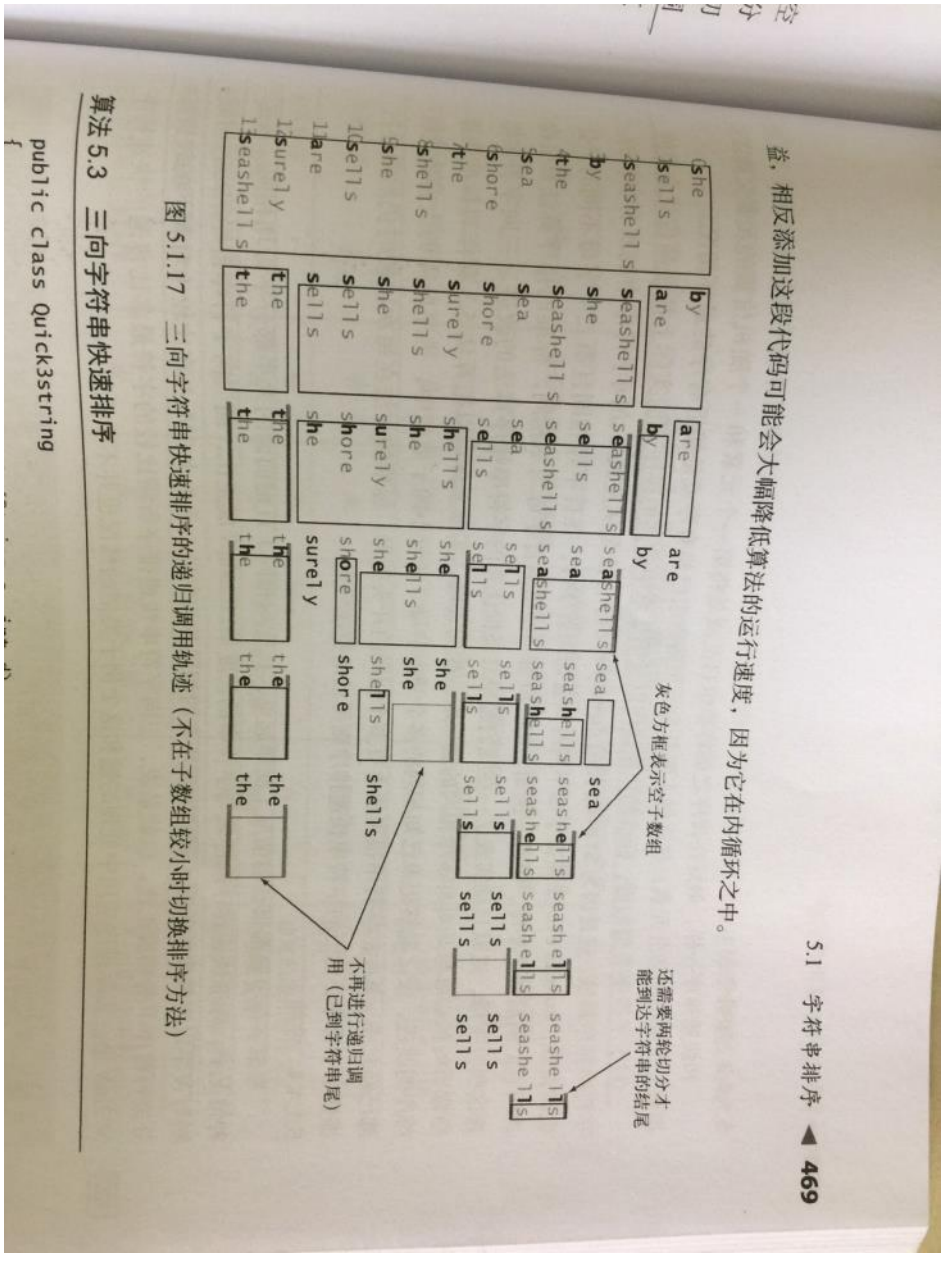



图 5.1.17 三向字符串快速排序的递归调用轨迹（不在子数组较小时切换排序方法）

**算法 5.3 三向字符串快速排序**

```
public class Quick3string
{
```

# 单词查找树

2017年4月27日 13:50

```
for (i = 0; i < N; i++)
  count[a[i].key() + 1]++;
```

count[]

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 总是0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Anderson 2 | 0 | 0 | 0 | **1** | 0 | 0 |
| Brown 3 | 0 | 0 | 0 | **1** | **1** | 0 |
| Davis 3 | 0 | 0 | 0 | **1** | **2** | 0 |
| Garcia 4 | 0 | 0 | 0 | **1** | **2** | **1** |
| Harris 1 | 0 | 0 | **1** | **1** | **2** | **1** |
| Jackson 3 | 0 | 0 | **1** | **1** | **3** | **1** |
| Johnson 4 | 0 | 0 | **1** | **1** | **3** | **2** |
| Jones 3 | 0 | 0 | **1** | **1** | **4** | **2** |
| Martin 1 | 0 | 0 | **2** | **1** | **4** | **2** |
| Martinez 2 | 0 | 0 | **2** | **2** | **4** | **2** |
| Miller 2 | 0 | 0 | **2** | **3** | **4** | **2** |
| Moore 1 | 0 | 0 | **3** | **3** | **4** | **2** |
| Robinson 2 | 0 | 0 | **3** | **4** | **4** | **2** |
| Smith 4 | 0 | 0 | **3** | **4** | **4** | **3** |
| Taylor 3 | 0 | 0 | **3** | **4** | **5** | **3** |
| Thomas 4 | 0 | 0 | **3** | **4** | **5** | **4** |
| Thompson 4 | 0 | 0 | **3** | **4** | **5** | **5** |
| White 2 | 0 | 0 | **3** | **5** | **5** | **5** |
| Williams 3 | 0 | 0 | **3** | **5** | **6** | **5** |
| Wilson 4 | 0 | 0 | **3** | **5** | **6** | **6** |

第三组的总人数

图 5.1.2 计算出现频率

### 5.1.1.3 数据分类

```
for (int r = 0; r < R; r++)
  count[r+1] += count[r];
```

count[]

| r | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 3 | 5 | 6 | 6 |
| 1 | 0 | 0 | **0** | **3** | 5 | 6 | 6 |
| 2 | 0 | 0 | 0 | **3** | **5** | 6 | 6 |
| 3 | 0 | 0 | 0 | 3 | **8** | **6** | 6 |
| 4 | 0 | 0 | 0 | 3 | 8 | **14** | **6** |
| 5 | 0 | 0 | 0 | 3 | 8 | 14 | **20** |

组号小于3的总人数（第三组在输出中的起始索引）

图 5.1.3 将频率转换为起始索引

nt[4] 加 2，因为
示例中 count[1]

在这个示例中，
中的起始位置为
于每个键值 r，
count[] 转化

```
for (int i = 0; i < N; i++)
   aux[count[a[i].key()]++] = a[i];
```

count[]

| i | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 0 | 0 | 3 | 8 | 14 |
| 1 | 0 | 4 | 8 | 14 |
| 2 | 0 | 4 | **9** | 14 |
| 3 | 0 | 4 | **10** | 14 |
| 4 | 0 | 4 | 10 | 15 |
| 5 | 1 | 4 | 10 | 15 |
| 6 | 1 | 4 | **11** | 15 |
| 7 | 1 | 4 | 11 | 16 |
| 8 | 1 | 4 | **12** | 16 |
| 9 | 2 | 4 | 12 | 16 |
| 10 | 2 | 5 | 12 | 16 |
| 11 | 2 | 6 | 12 | 16 |
| 12 | 3 | 6 | 12 | 16 |
| 13 | 3 | 7 | 12 | 16 |
| 14 | 3 | 7 | 12 | 17 |
| 15 | 3 | 7 | **13** | 17 |
| 16 | 3 | 7 | 13 | 18 |
| 17 | 3 | 7 | 13 | 19 |
| 18 | 3 | 8 | 13 | 19 |
| 19 | 3 | 8 | **14** | 19 |
| | 3 | 8 | 14 | 20 |
| | 3 | 8 | 14 | 20 |

| | | |
|---|---|---|
| a[0] | Anderson | 2 |
| a[1] | Brown | 3 |
| a[2] | Davis | 3 |
| a[3] | Garcia | 4 |
| a[4] | Harris | 1 |
| a[5] | Jackson | 3 |
| a[6] | Johnson | 4 |
| a[7] | Jones | 3 |
| a[8] | Martin | 1 |
| a[9] | Martinez | 2 |
| a[10] | Miller | 2 |
| a[11] | Moore | 1 |
| a[12] | Robinson | 2 |
| a[13] | Smith | 4 |
| a[14] | Taylor | 3 |
| a[15] | Thomas | 4 |
| a[16] | Thompson | 4 |
| a[17] | White | 2 |
| a[18] | Williams | 3 |
| a[19] | Wilson | 4 |

| | | |
|---|---|---|
| Harris | 1 | aux[0] |
| Martin | 1 | aux[1] |
| Moore | 1 | aux[2] |
| Anderson | 2 | aux[3] |
| Martinez | 2 | aux[4] |
| Miller | 2 | aux[5] |
| Robinson | 2 | aux[6] |
| White | 2 | aux[7] |
| **Brown** | **3** | aux[8] |
| **Davis** | **3** | aux[9] |
| **Jackson** | **3** | aux[10] |
| **Jones** | **3** | aux[11] |
| **Taylor** | **3** | aux[12] |
| **Williams** | **3** | aux[13] |
| Garcia | 4 | aux[14] |
| Johnson | 4 | aux[15] |
| Smith | 4 | aux[16] |
| Thomas | 4 | aux[17] |
| Thompson | 4 | aux[18] |
| Wilson | 4 | aux[19] |

图 5.1.4　将数据分类（键为 3 的条目均突出显示）

# 树的存储结构

2017年3月1日　　14:52

树的定义：

1. 由节点和边组成
2. 每个节点有1个父节点，但可能有多个子节点
3. 但有一个节点例外，该节点没有父节点，此节点称为根节点

深度：从根节点到最底层节点的层数

叶子节点：没有子节点的节点

度：节点拥有的子树的数目

树的度：各节点的度的最大值

- 树的顺序结构

```c
#include <stdio.h>

#define MAX_TREE_SIZE 100
typedef int ELEMTYPE;

//child node
typedef struct ChildNode
{
    int child;                //孩子节点的下标
    struct ChildNode *next; //指向下一个孩子节点的指针
} *Childptr;

//表头结构
typedef struct
{
    ELEMTYPE data;           //节点数据域
    int parent;              //双亲节点的下标
    Childptr firstChild;     //指向第一个孩子节点的指针
} CTBox;

//tree
typedef struct
{
    CTBox nodes[MAX_TREE_SIZE]; //节点数组
    int root;                   //根节点位置
    int num;                    //树的现有节点数目
};
```

# 二叉树

2017年3月1日 15:44

二叉树的度<=2
二叉树是有序树

- 满二叉树：所有叶子节点都在最底层，满二叉树一定是完全二叉树
- 完全二叉树：编号为j的节点与同样深度的满二叉树种编号为j的结点位置完全相同
  (1)叶子节点只能出现在最下两层
  (2)最下层的叶子一定集中在左部连续位置
  (3)倒数第二层，若有叶子节点，一定都在右部连续位置
  (4)如果结点度为1，该节点只有左孩子
  (5)同样节点的二叉树，完全二叉树的深度最小

**一般二叉树性质：**

1. 在非空二叉树的k层上，至多有$2^k$个节点(k>=0)
2. 高度为k的二叉树中,最多有$2^{k+1}$-1个节点(k>=0)
3. 对于任何一棵非空的二叉树,如果叶节点个数为$n_0$，度数为2的节点个数为$n_2$，则有: $n0 = n_2 + 1$

**完全二叉树性质:**

1. 具有n个节点的完全二叉树的高度k为$[\log_2 n]$
2. 对于具有n个节点的完全二叉树,如果按照从上(根节点)到下(叶节点)和从左到右的顺序对二叉树中的所有节点从0开始到n-1进行编号,则对于任意的下标为k的节点，有：

- 如果k=0,则它是根节点，它没有父节点；如果k>0,则它的父节点的下标为[(i-1)/2];
- 如果2k+1 <= n-1,则下标为k的节点的左子结点的下标为2k+1;否则,下标为k的节点没有左子结点.
- 如果2k+2 <= n-1,则下标为k的节点的右子节点的下标为2k+2;否则,下标为k的节点没有右子节点

二叉树的链式存储结构

```
typedef int ELEMTYPE;
typedef struct BNode
{
    ELEMTYPE data;
    struct BNode *lchild;
    struct BNode *rchild;
} BNODE, *BNODEPTR;
```

- 二叉树的遍历

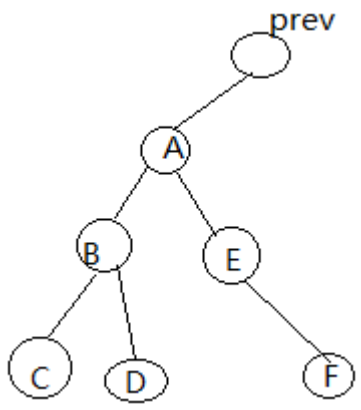
  先序遍历：根-左子树-右子树

  中序遍历:左子树-根-右子树

  后序遍历：左子树-右子树-根

# 线索二叉树

2017年3月2日　　11:17

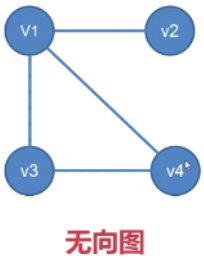



```c
#include <stdio.h>

typedef char ELEMTYPE;
/*线索存储标志位
 *Link(0)：表示指向左右孩子的指针
 *Clue(1)：表示指向前驱后继的线索
 */
typedef enum{ LINK, CLUE } PointerTag;
typedef struct ClueBTnode
{
    ELEMTYPE data;
    struct ClueBTnode *Lchild;
    struct ClueBTnode *Rchild;
    PointerTag Ltag;
    PointerTag Rtag;
} CLUE_BT_NODE;

/*全局变量*/
extern CLUE_BT_NODE *prev; //指向前驱结点


/*公共接口函数*/
CLUE_BT_NODE* CreateCBTree(CLUE_BT_NODE* BTptr);        //生成线索二叉树
void midOrderClueing(CLUE_BT_NODE* BTptr);              //中序遍历线索化
```

# 图的基本概念

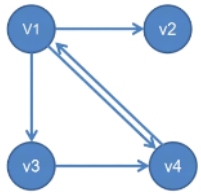
2017年4月15日 19:19

无向图

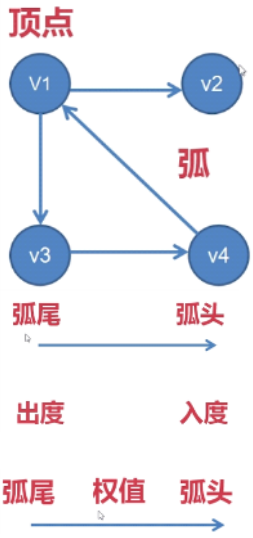



无向图

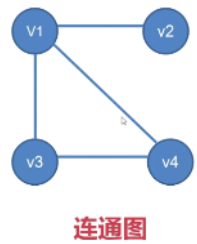
有向图




有向图

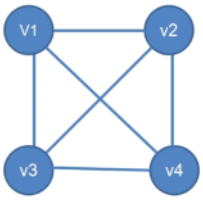



1. 连通图：任何一个顶点都有通往其他顶点的路径，即图中不存在出度和入度都为0的顶点




连通图

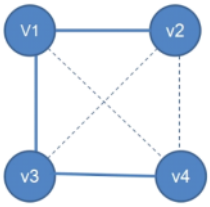
2. 完全图：任何一个顶点与其他顶点都有直接的路径，边数=n(n-1)/2




边数=n(n-1)/2

完全图

3．生成树：所有顶点+最少连接这些顶点的边， 边数=n-1




边数=n-1

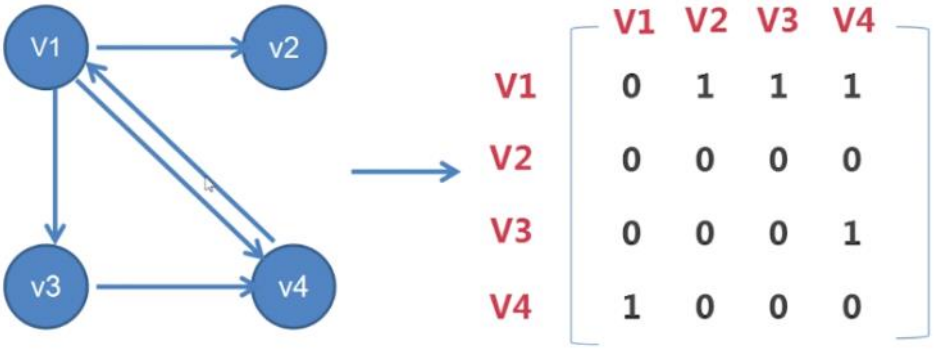
生成树

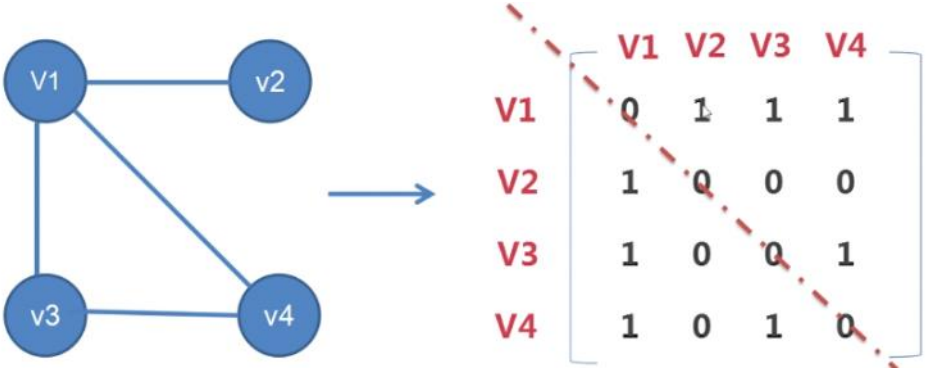
# 图的存储结构

2017年4月15日 19:30

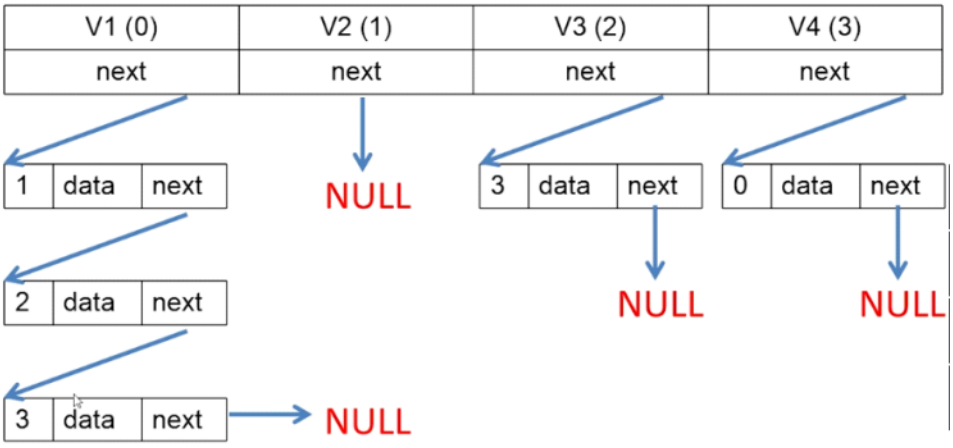
- 邻接矩阵（数组）
  - 顶点： 顶点索引+顶点数据
  - 弧（有弧记为“1” ）

```
struct Node
{
    NodeIndex;
    NodeData;
}
struct Graph
{
    NodeArray;
    AdjacentArray;
}
```




- 邻接表（有向图，链表）
  - 顶点：顶点索引+出弧链表头指针+顶点数据
  - 弧：弧头顶点索引+弧数据+下一条弧指针




- 十字链表（有向图，链表）

```
struct Node
{
    顶点索引；
    顶点数据；
    第一条入弧结点指针；
    第一条出弧结点指针；
}
struct Arc
{
    弧尾顶点索引；
    弧头顶点索引；
    指向下一条弧头相同的弧的指针；
    指向下一条弧尾相同的弧的指针；
    弧数据；
}
struct Map
{
    顶点数组；
}
```

- 邻接多重表（无向图，链表）

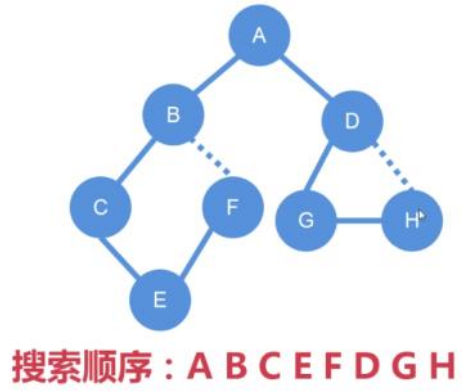
```
struct Node
{
    顶点索引；
    顶点数据；
    第一条边节点指针；
}
struct Edge
```

```
{
    A顶点索引；
    B顶点索引；
    与A顶点相连的下一条边的指针；
    与B顶点相连的下一条边的指针；
}
struct Map
{
    顶点数组；
}
```

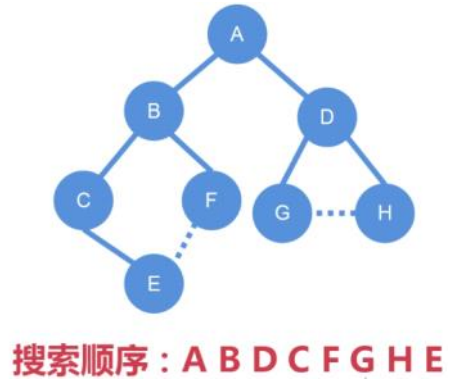
# 图遍历

2017年4月15日　　20:01

深度优先搜索（类似先序遍历）




广度优先搜索



# 最小生成树

2017年4月15日 20:05

Prim算法

# 总结

2017年4月16日　　15:33

| 各种常用排序算法 | | | | | | |
|---|---|---|---|---|---|---|
| 类别 | 排序方法 | 时间复杂度 | | | 空间复杂度 | 稳定性 |
| | | 平均情况 | 最好情况 | 最坏情况 | 辅助存储 | |
| 插入排序 | 直接插入 | $O(n^2)$ | $O(n)$ | $O(n^2)$ | $O(1)$ | 稳定 |
| | shell排序 | $O(n^{1.3})$ | $O(n)$ | $O(n^2)$ | $O(1)$ | 不稳定 |
| 选择排序 | 直接选择 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 不稳定 |
| | 堆排序 | $O(nlog_2n)$ | $O(nlog_2n)$ | $O(nlog_2n)$ | $O(1)$ | 不稳定 |
| 交换排序 | 冒泡排序 | $O(n^2)$ | $O(n)$ | $O(n^2)$ | $O(1)$ | 稳定 |
| | 快速排序 | $O(nlog_2n)$ | $O(nlog_2n)$ | $O(n^2)$ | $O(nlog_2n)$ | 不稳定 |
| 归并排序 | | $O(nlog_2n)$ | $O(nlog_2n)$ | $O(nlog_2n)$ | $O(1)$ | 稳定 |
| 基数排序 | | $O(d(r+n))$ | $O(d(n+rd))$ | $O(d(r+n))$ | $O(rd+n)$ | 稳定 |
| 注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数 | | | | | | |

# 冒泡排序

2017年9月20日 星期三 21:17

**每趟排序过程中，从数组的尾部开始每个元素与它前一个元素进行比较，如果比之小，则交换两者顺序。**
**第二趟排序从剩下的前n-1个元素中进行**

**// 递增排序，每次将大元素下沉到数组尾部arr[n]**

```
function bubbleSort(arr) {
    var len = arr.length;
    for (var i = 0; i < len; i++) {
        for (var j = 0; j < len - 1 - i; j++) {
            if (arr[j] > arr[j+1]) {        //相邻元素两两对比
                var temp = arr[j+1];        //元素交换
                arr[j+1] = arr[j];
                arr[j] = temp;
            }
        }
    }
    return arr;
}
```

# 选择排序

2017年4月16日　　14:04

选择排序原理：

1. 找到数组中最小的元素，将它和数组中的第一个元素交换位置；
2. 在剩下的元素中找到最小的元素，将它与数组中的第二个元素交换位置；
3. 。。。

（不断选择剩余元素中的最小者）

比较：N*N/2次

交换：N次

使用场景：待排序列长度较短。

```
void Selection::sort(Comparable a[])
{
    int len = sizeof(a) / sizeof(a[0]);
    for (int i = 0; i < len; i++)
    {//将a[i]与a[i+1..N]中最小的元素交换

        int min = i;
        for (int j = i + 1; j < len; j++)
        {//找到最小元素
            if (less(a[j], a[min])) min = j;
        }
        exch(a, min, i);
    }
}
```

```
function selectionSort(arr) {
    var len = arr.length;
    var minIndex, temp;

    for (var i = 0; i < len-1; i++) {
        minIndex = i;
        for (var j = i + 1; j < len; j++) {
            if (arr[j] < arr[minIndex]) {     //寻找最小的数
                minIndex = j;                 //将最小数的索引保存
            }
        }
        temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
```

```
    return arr;
}
```

# 快速排序

2017年4月18日　　17:12

思想：是属于“分治”排序。切分数组，使得比基准值小的元素放在它左侧，比他大的放在右侧

- 比较：2NlgN ， 最坏要N*N/2

适合：大数组排序

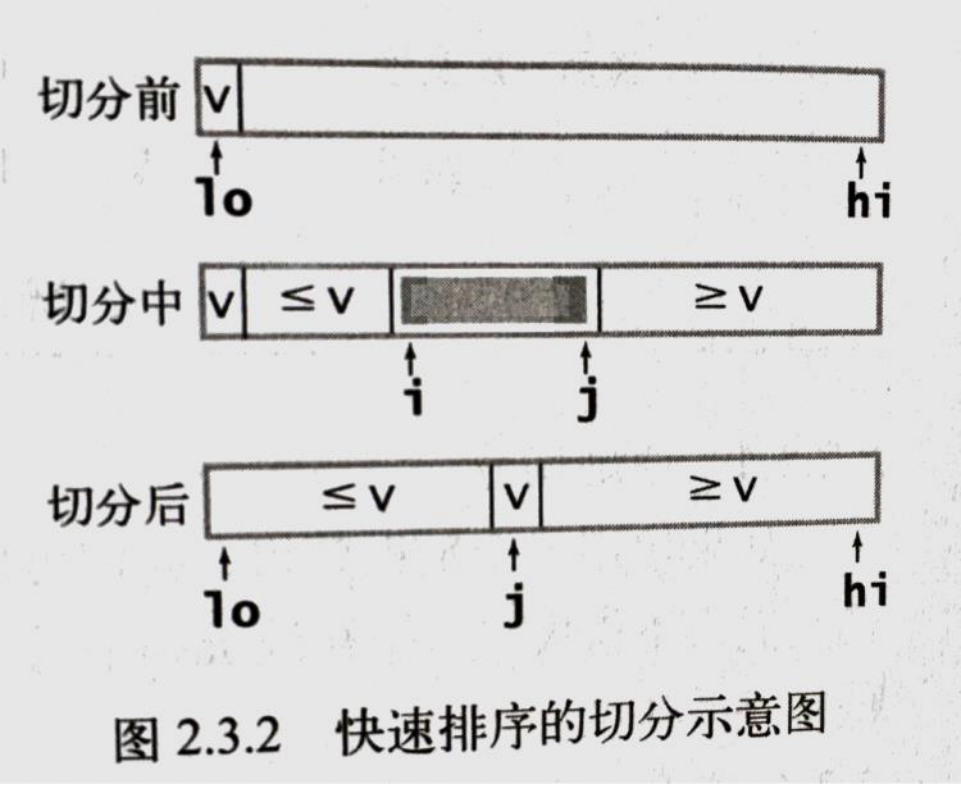



图 2.3.2　快速排序的切分示意图

Math.floor()向下取整
pivot是基准值，截取数组最中间的元素为轴心。

```
var quickSort = function(arr) {
    if (arr.length <= 1) { return arr; }
    var pivotIndex = Math.floor(arr.length / 2);
    var pivot = arr.splice(pivotIndex, 1)[0];
    var left = [];
    var right = [];
    for (var i = 0; i < arr.length; i++){
        if (arr[i] < pivot) {
            left.push(arr[i]);
        } else {
            right.push(arr[i]);
        }
    }
    return quickSort(left).concat([pivot], quickSort(right));
};
```

# 插入排序（直接、二分、希尔）

2017年4月16日 15:55

从第一个元素开始，该元素可以认为已经被排序；
取出下一个元素，在已经排序的元素序列中从后向前扫描；
如果该元素（已排序）大于待插入的元素，将已排序元素移到下一位置；
重复步骤3，直到找到已排序的元素小于或者等于新元素的位置；
将新元素插入到该位置后；
重复步骤2~5。

比较： N*N/4
交换： N*N/4

最好情况
比较： N-1
交换： 0

最坏情况
比较： N*N/2
交换： N*N/2

使用场景（升序排序）：基本有序序列，待插入元素值较大。适合小数组排序。

```
void Insertion::sort(Comparable a[])
{
    int len = sizeof(a) / sizeof(a[0]);
    for (int i = 1; i < len; i++)
    {//将a[i]插入到a[i-1] 、a[i-2]...中

        for (int j = i; j > 0; j--)
        {//内层循环将a[j]与前面的所有元素进行比较，若比前面元素小则进行交换

            if (less(a[j], a[j - 1]))
                exch(a, j, j - 1);
        }
    }
}
```

```
function insertSort(arr) {
    for(let i=1; i<arr.length; i++) {
        let key = arr[i]; //待插入元素
        let j = i-1;
        while(key<arr[j] && j>=0) { //挪位
            arr[j+1] = a[j];
            j--;
        } // 退出循环时j指向了真正需要插入的位置的前一个位置
        arr[j+1] = key;
    }
    return arr;
}
```

### 二分查找插入排序

从第一个元素开始，该元素可以认为已经被排序；

取出下一个元素，在已经排序的元素序列中二分查找到第一个比它大的数的位置；

将新元素插入到该位置后；

```
function binaryInsertSort(arr) {
    for(let i=1; i<arr.length; i++) {
        let key = arr[i];

        //在已排序列中找到一个比待插元素大的元素，并记录它的索引
        let left = 0;
        let right = i-1;
        while(left<=right) {
            let mid = parseInt((left+right)/2);
            if(key<arr[mid]) {
                right = mid-1;
            } else { // key>=arr[mid]
                left = mid+1;
            }
        } //退出循环时left==right

        //挪位
        let j=i-1;
        while(j>=left) {
            arr[j+1] = arr[j];
            j--;
        }
        arr[left] = key;
    }
    return arr;
}
```

## 希尔排序

交换不相邻（间距为h）的元素以对数组的局部进行排序，最终使用插入排序将局部有序的数组排序。（h有序数组）　—即将插入排序的比较间隔量从1改为h

适用于：大型序列排序

```
template <typename T>
void shell_sort<T>::sort(T &a)
{
    int N = a.size();
    int h = 1;
    while (h < N / 3)
    {//设置h的增量
        h = h * 3 + 1;
    }
    while (h >= 1)
    {//将数组变为h有序

        for (int i = h; i < N; i++)
        {//将a[i]插入到a[i-h],a[i-h*2], a[i-h*3]...之中

            for (int j = i; j >= h && (a[j]< a[j - 1]); j--)
            {
                swap(a[j], a[j - h]);
            }
        }
        h = h / 3;
    }
}
```

```
function shellSort(arr) {
    let gap = 1;
    while(gap<arr.length/5) {
        gap = gap*5 +1;
    }

    while(gap>0) {
        for(let i=gap; i<arr.length; i++) {
            let key = arr[i];
            let j = i-1;
            while(j>=0 && key<arr[j]) {
                arr[j+gap] = arr[j];
                j = j-gap;
            }
            arr[j+gap] = key;
        }
        gap = Math.floor(gap/5);
    }
    return arr;
}
```

# 归并排序

2017年4月16日 17:07

思想：该算法是采用分治法,将已有序的子序列合并，得到完全有序的序列

归并过程为：比较a[i]和a[j]的大小，若a[i]≤a[j]，则将第一个有序表中的元素a[i]复制到r[k]中，并令i和k分别加上1；否则将第二个有序表中的元素a[j]复制到r[k]中，并令j和k分别加上1，如此循环下去，直到其中一个有序表取完，然后再将另一个有序表中剩余的元素复制到r中从下标k到下标t的单元。

自顶向下归并排序：

- 比较： NlgN
- 访问次数：6NlgN

```
template <typename T>
void merge_sort<T>::sort_up_to_down(T &a, int low, int high)
{
    if (low >= high)
    {
        return;
    }
    int mid = (low + high) / 2;
    sort_up_to_down(a, low, mid);
    sort_up_to_down(a, mid + 1, high);
    merge(a, low, mid, high);
}
```

自底向上归并排序：适合链表组织的数据

- 比较：0.5NlgN~NlgN
- 访问：6NlgN

```
template <typename T>
void merge_sort<T>::sort_down_to_up(T &a)
{
    int N = a.size();
    int high;

    for (int subSize = 1; subSize < N; subSize = 2*subSize) //subSize子数组大小
    {
        for (int subIndex = 0; subIndex < N - subSize; subIndex = subIndex + 2 * subSize) //subIndex子数组索引
        {
            if ((subIndex + 2 * subSize - 1) < (N - 1))
                high = subIndex + 2 * subSize - 1;
            else
                high = N - 1;
            merge(a, subIndex, subIndex + subSize - 1, high);
        }
    }
}
```

屏幕剪辑的捕获时间：2017/4/25 13:44

```
void merge_sort<T>::merge(T &a, int low, int mid, int high)
{
    int i, j;
    static T aux;

    /*for (i = mid+1; i > low; i--) aux[i-1] = a[i-1];
    for (j = mid; j < high; j++) aux[high + mid - j] = a[j+1];
    for (int k = low; k <= high; k++)
    {
        if (aux[j] < aux[i])
        {
            a[k] = aux[j--];
        }
        else
        {
            a[k] = aux[i++];
        }
    }*/

    //思路二：
    i = low;
    j = mid + 1;
    for (int k = low; k <= high; k++) aux[k] = a[k];
    for (int k = low; k <= high; k++)
    {
        if (i > mid)                a[k] = aux[j++];    //左侧用尽，取右侧元素
        else if (j > high)          a[k] = aux[i++];    //右侧用尽，取左侧元素
        else if (aux[j] < aux[i])   a[k] = aux[j++];    //右侧的当前元素<左侧当前元素，取右侧当前元素
        else                        a[k] = aux[i++];    //左侧的当前元素<右侧当前元素，取左侧当前元素
    }
}
```

屏幕剪辑的捕获时间：2017/4/25 13:45

归并排序一种渐进最优的基于比较排序的算法！

# 堆排序

2017年4月20日 星期四 09:56

1. 从数组的n/2开始至左用FixDown()函数构造有序堆
2. 循环将最大元素a[0]和a[i](I不断减小，直到堆大小为1为止)交换并FixDown()修复堆，直到堆变空。

```
//堆排序
void HeapSort(char a[], int size)
{
    for (int i = size / 2 - 1; i >= 0; i--) //构造最大堆
        MaxHeapFixDown(a, i, size);
    for (int i = size - 1; i >= 1; i--)     //调整堆
    {
        swap(a[i], a[0]);                   //传入数组从0开始存储元素
        MaxHeapFixDown(a, 0, i);
    }
}
//堆有序化--下沉
void MaxHeapFixDown(char a[], int i, int size)
{
    int j = 2 * i + 1;  //指向左孩子，“+1”是因为堆的第一个元素从下标1开始存储
    int temp = a[i];
    while (j<size)
    {
        if (j + 1<size && a[j]<a[j + 1])
            ++j;
        if (temp>a[j])  //找到安置点
            break;
        else {
            a[i] = a[j];
            i = j;
            j = 2 * i + 1;
        }
    }
    a[i] = temp;
}
```

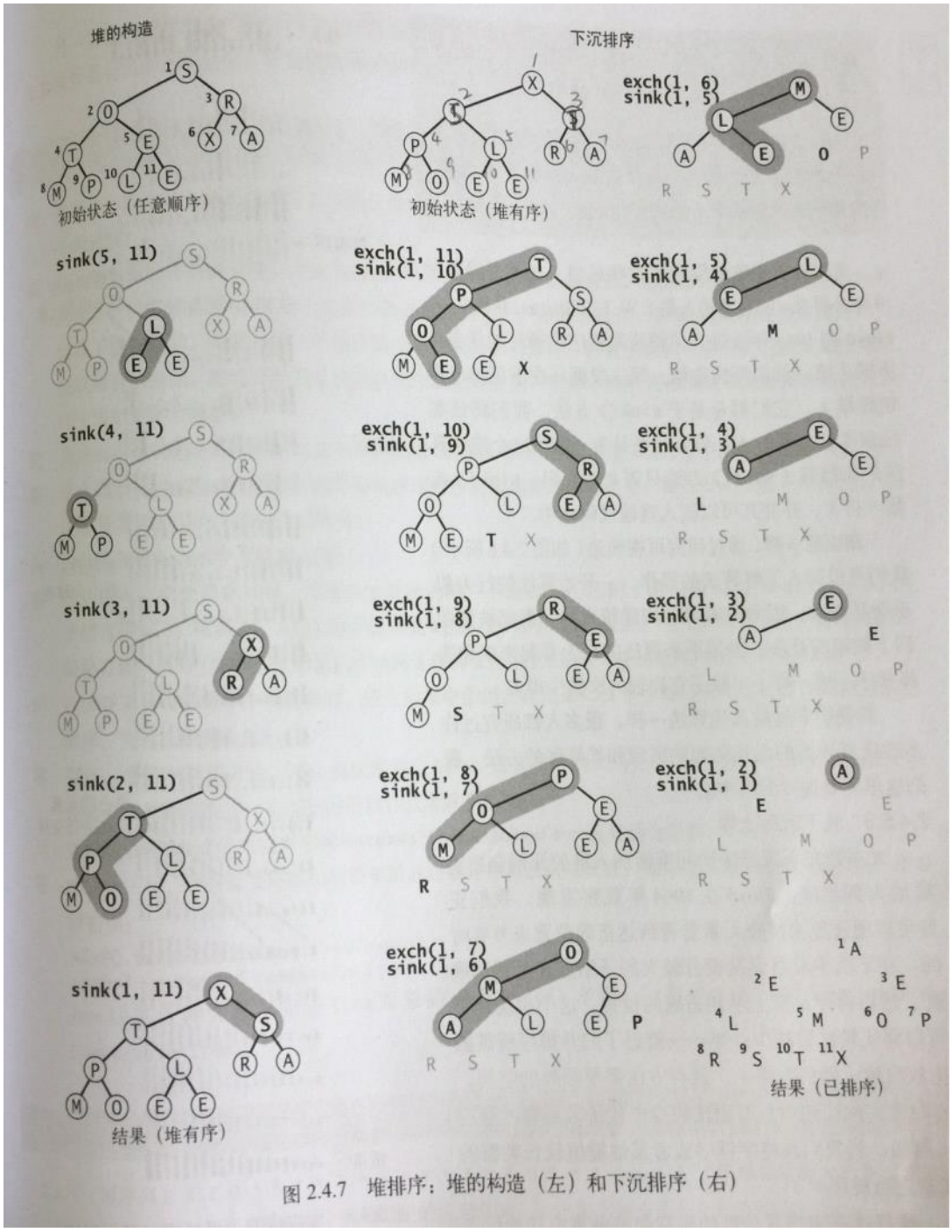
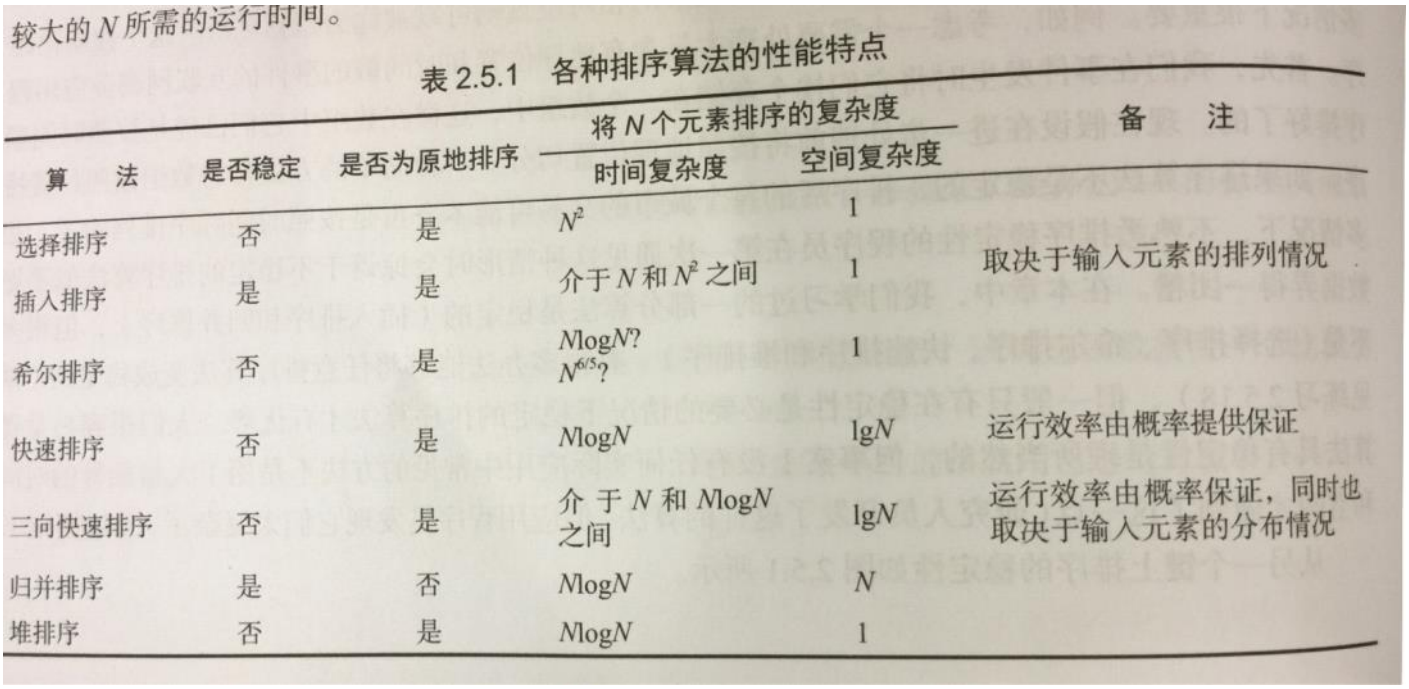



图 2.4.7　堆排序：堆的构造（左）和下沉排序（右）

# 各种排序比较

2017年4月22日 星期六 10:25



较大的 $N$ 所需的运行时间。

表 2.5.1 各种排序算法的性能特点

| 算法 | 是否稳定 | 是否为原地排序 | 将 $N$ 个元素排序的复杂度 | | 备注 |
|---|---|---|---|---|---|
| | | | 时间复杂度 | 空间复杂度 | |
| 选择排序 | 否 | 是 | $N^2$ | 1 | |
| 插入排序 | 是 | 是 | 介于 $N$ 和 $N^2$ 之间 | 1 | 取决于输入元素的排列情况 |
| 希尔排序 | 否 | 是 | $N\log N$? $N^{6/5}$? | 1 | |
| 快速排序 | 否 | 是 | $N\log N$ | $\lg N$ | 运行效率由概率提供保证 |
| 三向快速排序 | 否 | 是 | 介于 $N$ 和 $N\log N$ 之间 | $\lg N$ | 运行效率由概率保证，同时也取决于输入元素的分布情况 |
| 归并排序 | 是 | 否 | $N\log N$ | $N$ | |
| 堆排序 | 否 | 是 | $N\log N$ | 1 | |

快速排序是最快的通用排序算法

# 二分查找（有序表）

2017年4月22日 16:56

最多需比较lgN+1次

向空表插入N个键，最坏需要访问N*N 次

# 顺序查找（无序表）

2017年4月22日 16:57

未命中的查找和插入，需要比较：N次
命中的查找，最坏需要比较：N次

向空表插入N个不同键，需比较N*N/2

# 二叉查找树BST

2017年4月22日 17:08

BST:每个结点的键都大于左子树中任意结点的键，而小于右子树中任意结点的键

查找，平均比较次数：2lnN
插入操作，平均比较次数：2lnN

# 符号表的各种查找实现

2017年4月22日 17:01

表 3.1.11 符号表的各种实现的优缺点

| 使用的数据结构 | 实现 | 优点 | 缺点 |
| --- | --- | --- | --- |
| 链表（顺序查找） | SequentialSearchST | 适用于小型问题 | 对于大型符号表很慢 |
| 有序数组（二分查找） | BinarySearchST | 最优的查找效率和空间需求，能够进行有序性相关的操作 | 插入操作很慢 |
| 二叉查找树 | BST | 实现简单，能够进行有序性相关的操作 | 没有性能上界的保证<br>链接需要额外的空间 |
| 平衡二叉查找树 | RedBlackBST | 最优的查找和插入效率，能够进行有序性相关的操作 | 链接需要额外的空间 |
| 散列表 | SeparateChainHashST<br>LinearProbingHashST | 能够快速地查找和插入常见类型的数据 | 需要计算每种类型的数据的散列<br>无法进行有序性相关的操作<br>链接和空结点需要额外的空间 |