# 书籍

1.大话数据结构,程杰,清华大学出版社, 2011.06 –文字讲解容易理解，但废话太多，阅读时可以忽略这些无用的废话，直奔主题和代码。部分章节参考。至少代码没有错，其他几本书代码有错。

1.数据结构与算法分析新视角,周幸妮,2016，文字讲解容易理解，推荐。部分代码有错。

1.数据结构(C,第2版)严蔚敏,人民邮电出版社2015 --讲解简洁，不太容易理解，代码逻辑严谨。部分章节、代码参考。

2.数据结构与算法(C,第3版)陈琳琳/李建林,清华大学出版社, 2015.07，文字讲解容易理解，但好几处文字错误、代码逻辑错误。部分文字错误已经用图片编辑软件修改。

4.2019版数据结构高分笔记(第7版)--在上述三本书的基础上做的讲解，相当于对上述三本书的辅助、辅导。

都不理想，可以几本书相互参考。

“数据结构与算法(陈琳琳)”有些地方代码错误、描述不清。

**网络资料**

<https://github.com/labuladong/fucking-algorithm>

<https://labuladong.gitbook.io/algo/>

<https://github.com/algorithm-visualizer/algorithm-visualizer>

<https://algorithm-visualizer.org/simple-recursive/cellular-automata>

# 链表

## 单向链表

#include <stdlib.h>

#include <stdio.h>

typedef struct list\_node {

int data;

struct list\_node \*next;

} Node;

Node \*insert(Node \*\*list, int data){

Node \*list\_head = \*list;

Node \*pre\_node = NULL;

Node \*cur\_node = NULL;

// init list head

if (list\_head == NULL){

list\_head = (Node\*)malloc(sizeof(Node));

memset(list\_head,0,sizeof(Node));

list\_head->next = NULL; //**头结点最好不要存放有效元素值**，否则后续的链表操作容易造成错误和麻烦。

\*list = list\_head;

printf("Init list successful.\n");

}

#if 1

// insert a node to the list tail

for( pre\_node = list\_head,cur\_node = list\_head->next; cur\_node != NULL; pre\_node = cur\_node,cur\_node = cur\_node->next);

cur\_node = (Node\*)malloc(sizeof(Node));

memset(cur\_node,0,sizeof(Node));

cur\_node->data = data;

pre\_node->next = cur\_node;

cur\_node->next = NULL;

#else

// insert a node after list head

cur\_node = (Node\*)malloc(sizeof(Node));

memset(cur\_node,sizeof(Node),0);

cur\_node->data = data;

cur\_node->next = list\_head->next;

list\_head->next = cur\_node;

#endif

printf("Insert node %d successful.\n", cur\_node->data);

return \*list;

}

void traverse(Node \*\*list){

Node \*list\_head = \*list;

Node \*cur\_node = NULL;

if (list\_head == NULL){

printf("No Node. Empty list.\n");

return;

}

for(cur\_node = list\_head->next; cur\_node != NULL; cur\_node = cur\_node->next){

printf("Node data: %d.\n", cur\_node->data);

}

}

Node \*delete(Node \*\*list, int data){

Node \*list\_head = \*list;

Node \*pre\_node = NULL;

Node \*cur\_node = NULL;

if (list\_head == NULL){

printf("Delete error, Empty list.\n");

}

for( pre\_node = list\_head,cur\_node = list\_head->next; cur\_node != NULL; pre\_node = cur\_node,cur\_node = cur\_node->next){

if(cur\_node->data == data){

pre\_node->next = cur\_node->next;

free(cur\_node);

cur\_node = NULL;

printf("Delete node %d successful.\n", data);

break;

}

}

return \*list;

}

void destory(Node \*\*list){

Node \*list\_head = \*list;

Node \*pre\_node = NULL;

Node \*cur\_node = NULL;

int data;

if (list\_head == NULL){

printf("List already empty.\n");

}

for( cur\_node = list\_head->next; cur\_node != NULL; cur\_node = list\_head->next){

data = cur\_node->data;

list\_head->next = cur\_node->next;

free(cur\_node);

cur\_node = NULL;

printf("Destory, delete node %d successful.\n", data);

}

free(list\_head);

list\_head = NULL;

\*list = NULL;

printf("Destory list successful.\n");

}

**int main(int argc, char \* argv[])**

{

Node \*list = NULL;

int a[7] = {3,7,1,9,8,11,17};

int i;

for(i=0;i<7;i++){

insert(&list, a[i]); //注意:实参使用&list才能改变指针变量list

}

traverse(&list);

delete(&list, a[5]);

traverse(&list);

delete(&list, a[2]);

traverse(&list);

destory(&list);

traverse(&list);

}

## 双向链表

我写的双向链表操作,20181029

#include <stdlib.h>

#include <stdio.h>

typedef struct list\_DNode {

int data;

struct list\_DNode \*pre;

struct list\_DNode \*next;

} DNode;

DNode \*insert(DNode \*\*list, int data){

DNode \*list\_head = \*list;

DNode \*pre\_DNode = NULL;

DNode \*cur\_DNode = NULL;

// init list head

if (list\_head == NULL){

list\_head = (DNode\*)malloc(sizeof(DNode));

memset(list\_head,sizeof(DNode),0);

list\_head->pre = NULL;

list\_head->next = NULL;

\*list = list\_head;

printf("Init dual list successful.\n");

}

#if 1

// insert a DNode to the list tail

for( pre\_DNode = list\_head,cur\_DNode = list\_head->next; cur\_DNode != NULL; pre\_DNode = cur\_DNode,cur\_DNode = cur\_DNode->next);

cur\_DNode = (DNode\*)malloc(sizeof(DNode));

memset(cur\_DNode,sizeof(DNode),0);

cur\_DNode->data = data;

cur\_DNode->next = pre\_DNode->next; //修改四个指向：1.cur\_DNode->next，步骤1.2.修改cur\_DNode->next、cur\_DNode->pre

cur\_DNode->pre = pre\_DNode; // 2.cur\_DNode->pre

if(pre\_DNode->next!=NULL) pre\_DNode->next->pre = cur\_DNode; // 3.pre\_DNode->next->pre，步骤3.4.将cur\_DNode插入到链表中。如果插入第一个节点则pre\_DNode->next==NULL，此时不需要步骤3.，因此需要if判断。

pre\_DNode->next = cur\_DNode; // 4.pre\_DNode->next，注意先修改pre\_DNode->next->pre，然后在修改pre\_DNode->next。

#else

// insert a DNode after list head

cur\_DNode = (DNode\*)malloc(sizeof(DNode));

memset(cur\_DNode,sizeof(DNode),0);

cur\_DNode->data = data;

cur\_DNode->next = list\_head->next;

cur\_DNode->pre = list\_head;

if(list\_head->next!=NULL) list\_head->next->pre = cur\_DNode;

list\_head->next = cur\_DNode;

#endif

printf("Insert DNode %d successful.\n", cur\_DNode->data);

return \*list;

}

void traverse(DNode \*\*list){

DNode \*list\_head = \*list;

DNode \*cur\_DNode = NULL;

if (list\_head == NULL){

printf("No DNode. Empty list.\n");

return;

}

for(cur\_DNode = list\_head->next; cur\_DNode != NULL; cur\_DNode = cur\_DNode->next){

printf("DNode data: %d.\n", cur\_DNode->data);

}

}

DNode \*delete(DNode \*\*list, int data){

DNode \*list\_head = \*list;

DNode \*pre\_DNode = NULL;

DNode \*cur\_DNode = NULL;

if (list\_head == NULL){

printf("Delete error, Empty list.\n");

}

for( pre\_DNode = list\_head,cur\_DNode = list\_head->next; cur\_DNode != NULL; pre\_DNode = cur\_DNode,cur\_DNode = cur\_DNode->next){

if(cur\_DNode->data == data){

if(cur\_DNode->next!=NULL) cur\_DNode->next->pre = cur\_DNode->pre; //如果删除的是最后一个节点，则不需要这个步骤

pre\_DNode->next = cur\_DNode->next; //保存被删除节点的前后指针cur\_DNode->next、cur\_DNode->pre到相应的前后节点中，摘掉

free(cur\_DNode);

cur\_DNode = NULL;

printf("Delete DNode %d successful.\n", data);

break;

}

}

return \*list;

}

void destory(DNode \*\*list){

DNode \*list\_head = \*list;

DNode \*pre\_DNode = NULL;

DNode \*cur\_DNode = NULL;

int data;

if (list\_head == NULL){

printf("List already empty.\n");

}

for( cur\_DNode = list\_head->next; cur\_DNode != NULL; cur\_DNode = list\_head->next){

data = cur\_DNode->data;

list\_head->next = cur\_DNode->next;

if(cur\_DNode->next!=NULL) cur\_DNode->next->pre = cur\_DNode->pre;

free(cur\_DNode);

cur\_DNode = NULL;

printf("Destory, delete DNode %d successful.\n", data);

}

free(list\_head);

list\_head = NULL;

\*list = NULL;

printf("Destory dual list successful.\n");

}

**int main(int argc, char \* argv[])**

{

DNode \*list = NULL;

int a[7] = {3,7,1,9,8,11,17};

int i;

for(i=0;i<7;i++){

insert(&list, a[i]); //注意:实参使用&list才能改变指针变量list

}

traverse(&list);

delete(&list, a[5]);

traverse(&list);

delete(&list, a[2]);

traverse(&list);

destory(&list);

traverse(&list);

}

## 参考

参考Linux内核、lk。



## 链表面试题

<http://www.360doc.com/content/10/0112/16/722458_13353508.shtml>

**①链表反转**

单向链表的反转是一个经常被问到的一个面试题，也是一个非常基础的问题。比如一个链表是这样的： 1->2->3->4->5，通过反转后成为5->4->3->2->1。

网上流传的c++笔试题汇总

(1)已知链表的头结点head,写一个函数把这个链表逆序(Intel)

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

Node \* ReverseList(Node \*head){ //链表逆序,  
Node \*p1 = head;  
Node \*p2 = p1->next;  
Node \*p3 = p2->next;

if (head == NULL || head->next == NULL) return head;

p1->next = NULL;  
while ( p3 != NULL ){  
 p2->next = p1;  
 p1 = p2;  
 p2 = p3;  
 p3 = p3->next;  
}  
p2->next = p1;  
head = p2;  
return head;

}

将思路用文字描述出来。

三个节点p1、p2、p3。p3用于保存下一个节点，用于遍历。第一趟，修改p1、p2之间的指针为反向，然后将p1、p2、p3整体向后移动一个节点，新的p1、p2、p3是原来的p2、p3、p3->next节点，循环，逐个节点往后移动、逐个地将新的p1、p2指向修改为反向。

或者，遍历时利用一个辅助指针，存储遍历过程中当前指针指向的下一个元素，然后将当前节点元素的指针反转后，利用已经存储的指针往后面继续遍历。源代码如下：

typedef struct linka {

int data;

linka\* next;

};

void reverse(linka\*& head) {

linka \*pre, \*cur, \*ne;

if(head ==NULL) return;

pre=head;

cur=head->next;

while(cur){

ne = cur->next;

cur->next = pre;

pre = cur;

cur = ne;

}

head->next = NULL;

head = pre;

}

还有一种利用递归的方法。这种方法的基本思想是在反转当前节点之前先调用递归函数反转后续节点。

不过这个方法有一个缺点，就是在反转后的最后一个结点会形成一个环，所以必须将函数的返回的节点的next域置为NULL。因为要改变head指针，所以我用了引用。算法的源代码如下：

linka\* reverse(linka\* p,linka\* head){

if(p == NULL || p->next == NULL){

head=p;

return p;

}else{

linka\* tmp = reverse(p->next,head);

tmp->next = p;

return p;

}

}

交换两个变量值

temp=a;

a=b;

b=temp; //首尾相连，构成一个环

**②链表合并**

(2)已知两个链表head1和head2各自有序，请把它们合并成一个链表依然有序。(保留所有结点，即便大小相同)

Node \* Merge(Node \*head1 , Node \*head2){

Node \*head = NULL; //链表头指针

Node \*pcurrent = NULL; //动态移动指针

Node \*p1 = NULL;

Node \*p2 = NULL;

if ( head1 == NULL) return head2;  
 if ( head2 == NULL) return head1;

if ( head1->data < head2->data ){ //先找出头结点。有些情况下，头结点尾结点需要单独处理。  
 head = head1;  
 p1 = head1->next;  
 p2 = head2;  
 }else{  
 head = head2;  
 p2 = head2->next;  
 p1 = head1;  
 }  
 pcurrent = head;

while ( p1 != NULL && p2 != NULL){  
 if ( p1->data <= p2->data ){  
 pcurrent->next = p1;  
 pcurrent = p1; //写成pcurrent = pcurrent->next更好  
 p1 = p1->next;  
 }else{  
 pcurrent->next = p2;  
 pcurrent = p2; //写成pcurrent = pcurrent->next更好  
 p2 = p2->next;  
 }  
 }

if ( p1 != NULL ) pcurrent->next = p1;  
 if ( p2 != NULL ) pcurrent->next = p2;

return head;

}

(3)已知两个链表head1和head2各自有序，请把它们合并成一个链表依然有序，这次要求用递归方法进行。(Autodesk)  
答案：  
Node \* MergeRecursive(Node \*head1 , Node \*head2){  
 Node \*head = NULL;

if ( head1 == NULL ) return head2;  
 if ( head2 == NULL) return head1;

if ( head1->data < head2->data ){  
 head = head1;  
 head->next = MergeRecursive(head1->next,head2);  
 }else{  
 head = head2;  
 head->next = MergeRecursive(head1,head2->next);  
 }

return head;  
}  
  
**42. 写一个函数找出一个整数数组中，第二大的数 (microsoft)**  
答案：  
const int MINNUMBER = -32767;  
int find\_sec\_max( int data[] , int count){  
 int maxnumber = data[0];  
 int sec\_max = MINNUMBER;  
 for ( int i = 1; i < count; i++){  
 if ( data[i] > maxnumber ){  
 sec\_max = maxnumber;  
 maxnumber = data[i];  
 }else{  
 if ( data[i] > sec\_max )  
 sec\_max = data[i];  
 }  
 }  
 return sec\_max;  
}  
  
**43. 写一个在一个字符串(n)中寻找一个子串(m)第一个位置的函数。**  
KMP算法效率最好，时间复杂度是O(n+m)。  
  
**45. 如何判断一个单链表是有环的？(注意不能用标志位，最多只能用两个额外指针)**  
struct node { char val; node\* next;}  
bool check(const node\* head){} //return false:无环,true:有环  
一种O(n)的办法就是(搞两个指针，一个每次递增一步，一个每次递增两步，如果有环的话两者必然重合，反之亦然)：  
bool check(const node\* head){  
 if(head==NULL) return false;  
 node \*low=head, \*fast=head->next;  
 while(fast!=NULL && fast->next!=NULL){  
 low=low->next;  
 fast=fast->next->next;  
 if(low==fast) return true;  
 }  
 return false;  
}

**③删除链表中的重复元素**

给定一个排序链表，删除所有含有重复数字的节点，只保留原始链表中 没有重复出现 的数字。例:

输入: 1->2->3->3->4->4->5

输出: 1->2->5

void rm\_duplicate(Node \*\*list){

Node \*list\_head = \*list;

Node \*list\_cur = NULL;

Node \*it\_pre = NULL;

Node \*it = NULL;

list\_cur = list\_head->next;

it\_pre = list\_cur;

it = it\_pre->next;

while(list\_cur != NULL){

while(it != NULL){

if (list\_cur->data == it->data) {

it\_pre->next = it->next;

printf("Remove duplicate node %d successful.\n", it->data);

free(it);

it = it\_pre->next;

} else {

it\_pre = it;

it = it->next;

}

}

list\_cur = list\_cur->next;

it\_pre = list\_cur;

if(it\_pre!=NULL) it = it\_pre->next;

}

}

删除两个链表中的重复元素

给定两个单链表pA，pB，pA中间没有重复的数字，pB也没有重复的数字，要求删除pA和pB中所有重复数字的结点。

void rm\_duplicate(Node \*lista, Node \*listb){

Node\* lista\_head = lista;

Node\* listb\_head = listb;

Node\* lista\_cur = NULL;

Node\* listb\_cur = NULL;

Node\* listb\_pre = NULL;

if(lista\_head == NULL || listb\_head == NULL) return;

for(lista\_cur = lista\_head->next; lista\_cur != NULL; lista\_cur = lista\_cur->next){

listb\_pre = listb\_head;

listb\_cur = listb\_pre->next;

do {

if(lista\_cur->data == listb\_cur->data){

listb\_pre->next = listb\_cur->next;

printf("Duplicated, Delete node %d from middle or tail successful.\n", listb\_cur->data);

free(listb\_cur);

listb\_cur = listb\_pre->next;

break;

} else {

listb\_pre = listb\_cur;

listb\_cur = listb\_cur->next;

}

} while(listb\_cur != NULL);

}

}

对于头结点存放有效元素值的情况，如果是listb头结点相等，那么需要特殊考虑。头结点不存放有效元素就不需要这样的考虑了。**所以头结点最好不要存放有效元素值**。

void rm\_duplicate(Node \*lista, Node \*listb){

Node\* lista\_head = lista;

Node\* listb\_head = listb;

Node\* lista\_cur = NULL;

Node\* listb\_cur = NULL;

Node\* listb\_pre = NULL;

if(lista\_head == NULL || listb\_head == NULL) return;

for(lista\_cur = lista\_head; lista\_cur != NULL; lista\_cur = lista\_cur->next){

listb\_cur = listb\_head;

do {

if (lista\_cur->data == listb\_cur->data) {

listb\_head = listb\_cur->next;

listb\_pre = listb\_cur;

listb\_cur = listb\_cur->next;

printf("Duplicated, Delete node %d from head successful.\n", listb\_pre->data);

free(listb\_pre);

break;

}

} while(listb\_cur != NULL);

}

//如果是listb中间和尾结点相等，操作方法与头结点不存放有效元素值相同。

{//代码，删除中间和尾结点相等的元素}

}

# 常用排序算法

参考：大话数据结构,程杰,清华大学出版社, 2011.06

数据结构与算法分析新视角,周幸妮,2016

## 交换排序

### (1)冒泡排序(Bubble Sort)

冒泡排序：指定数值小的元素在前、数值大的元素在后，元素之间两两比较，逆序则交换两个元素的位置，。

第一趟，全部n个元素两两相互比较，从arr[0]开始，arr[0]、arr[1]比较，逆序则交换位置，arr[1]、arr[2]比较，逆序则交换位置，…，最后，arr[n-2]、arr[n-1]比较，逆序则交换位置；一趟完成后，最大元素就冒泡到最后的位置arr[n-1]了。

第二趟，前n-1个元素两两相互比较，从arr[0]开始，arr[0]、arr[1]比较，逆序则交换位置，arr[1]、arr[2]比较，逆序则交换位置，…，最后，arr[n-3]、arr[n-2]比较，逆序则交换位置；一趟完成后，次大元素就冒泡到最后的位置arr[n-2]了。

第三趟，…

void BubbleSort(){

int i,j,temp;

int change=0;//检测一次排序中是否发生交换，若无交换，则序列有序，仅需一趟排序

for(i=n-1;i>=0&&change==0;i--){

change=1;

for(j=0;j<i;j++){

if(a[j]>a[j+1]){

change=0;

temp=a[j];

a[j]=a[j+1];

a[j+1]=temp;

}

}

}

}

时间复杂度：O(n2)

代码参考：《数据结构与算法分析新视角,周幸妮,2016》- 7.3交换排序、《大话数据结构.程杰》- 9.3 冒泡排序

### (2)快速排序(Quick Sort)

快速排序是对冒泡排序的一种改进，基本思路是将待排序序列分成两部分，其中一部分的记录都比另一部分的记录小，随后分别对这两部分再分成两部分，使一部分的记录都小于另一部分，如此反复最终使整个序列有序。

参考：《数据结构与算法分析新视角,周幸妮,2016》- 7.3交换排序

QuickSort(array[100],100,0,99);

void QuickSort(int \*arr, int n, int low, int high){

int pivotloc;

if(low<high){

pivotloc=Partition\_1(arr,low,high);

QuickSort(arr,n,low,pivotloc-1);

QuickSort(arr,n,pivotloc+1,high);

}

}

int Partition\_1(int arr[], int low, int high){

int pivotloc=arr[low], temp;

while(low<high){

while(low<high && arr[high]>=pivotloc)

--high;

if(arr[high]<pivotloc){ //判断是low==high、还是arr[high]<pivotloc引起while终止，low==high引起while终止

temp=arr[low]; //不需要交换、直接跳出外层while，arr[high]<pivotloc引起while终止才需要交换。

arr[low]=arr[high];

arr[high]=temp;

}

while(low<high && arr[low]<=pivotloc)

++low;

if(arr[low]>pivotloc){ //判断是low==high、还是arr[low]>pivotloc引起while终止，low==high引起while终止

temp=arr[low]; //不需要交换、直接跳出外层while，arr[low]>pivotloc引起while终止才需要交换。

arr[low]=arr[high];

arr[high]=temp;

}

}

}

时间复杂度：O(nlogn)

代码参考：《数据结构与算法分析新视角,周幸妮,2016》- 7.3交换排序

## 选择排序

### (3)简单选择排序(Selection Sort)

选择排序的基本思想是每一趟在所有记录中选择最小的记录放入有序序列中，如此反复就能够实现无序序列的有序化。

相对于冒泡排序来说，简单选择排序效率更高一些，因为冒泡排序每一轮的每一次比较后，如果发现前面的数比后面的数大，就要立即进行数据交换，而选择排序则每一轮最多进行一次数据交换。冒泡排序数据交换次数平均要比选择排序多。参考：《数据结构与算法分析新视角,周幸妮,2016》- 7.4选择排序

选择排序：根据位置选择元素(指定位置，选择元素)。指定位置arr[0]存放最小元素、指定位置arr[1]存放次小元素、指定位置arr[2]存放再次小元素、…。

指定位置arr[0]存放最小元素，第一趟，从数组中去查找最小元素，找到最小元素后，将其放到位置arr[0]；

指定位置arr[1]存放次小元素，第二趟，从数组中去查找次小元素，找到次小元素后，将其放到位置arr[1]；

第三趟，…

代码：

void SelectionSort(int \*a, int n){

int i,j,loc,temp;

for(i=0;i<n-1;i++)

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

if(a[j]<a[i])

loc=j; //记录某一趟比较中最小元素所在位置

}

if(i!=loc){ //最后只进行一次位置交换

temp=a[i];

a[i]=a[loc];

a[loc]=a[i];

}

}

}

时间复杂度：O(n2)

代码参考《大话数据结构.程杰》- 9.4 简单选择排序

### (4)堆排序(Heap Sort)

未完

## 插入排序

### (5)直接插入排序(Insertion Sort)

代码参考：《大话数据结构.程杰》- 9.5直接插入排序、《数据结构与算法分析新视角,周幸妮,2016》- 7.2插入排序

void InsertSort(int \*a, int n){

int i,j,temp;

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

if(a[i]<a[i-1]){

temp=a[i]; //哨兵

for(j=i-1;temp<a[j]&&j>=0;--j) //查找插入位置

a[j+1]=a[j];

a[j+1]=temp;

}

}

}

时间复杂度：O(n2)

### (6)希尔排序(Shell Sort)

代码参考：《数据结构与算法分析新视角,周幸妮,2016》- 7.2插入排序、《大话数据结构.程杰》- 9.6希尔排序、

《数据结构(C,第2版)严蔚敏》-8.2.3 希尔排序

void ShellSort(int \*a, int n){

int k,dlta[3]={5,3,1};

for(k=0;k<3;++k)

ShellInsert(arr,n,dlta[k]);

}

void ShellInsert(int \*arr,int n, int dk){

int i,j,k,temp;

for(k=0;k<dk;k++){

for(i=k+dk;i<n;i=i+dk){

if(arr[i]<arr[i-dk]){

temp=arr[i];

for(j=i-dk;j>=0&&temp<arr[j];j=j-dk)

arr[j+dk]=arr[j];

arr[j+dk]=temp;

}

}

}

}

时间复杂度：O(nlog2n)

## 归并排序

### (7)归并排序(Merge Sort)

参考：《数据结构(C,第2版)严蔚敏》-8.5归并排序、《大话数据结构.程杰》-9.8归并排序

void Merge(int \*r, int \*t, int low, int mid, int high){//将有序序列r[low…mid]和t[mid+1...high]归并为有序序列r[low...high]

int i=low,j=mid+1,k=low;

while(i<=mid&&j<=high)

if(r[i]<r[j])

t[k++]=r[i++];

else

t[k++]=r[j++];

while(i<=mid)

t[k++]=r[i++];

while(j<=high)

t[k++]=r[j++];

}

void MSort(int \*r,int \*t, int low, int high){//r[low...high]归并后放入t[low...high]

int s[NUM];

if(low==high)

t[low]=r[low];

else{

mid=(low+high)/2;

MSort(r,s,low,mid);

MSort(r,s,mid+1,high);

Merge(s,t,low,mid,high);

}

}

void MergeSort(int r[NUM]){

MSort(r[NUM],r[NUM],0,NUM-1);

}

时间复杂度：O(nlog2n)

## 分配排序

### (8)桶排序(Bucket Sort)

代码参考：《数据结构与算法分析新视角,周幸妮,2016》- 7.6分配排序 - 7.6.1桶排序

### (9)基数排序(Radix Sort)

代码参考：《数据结构与算法分析新视角,周幸妮,2016》- 7.6分配排序 - 7.6.2基数排序

《数据结构(C,第2版)严蔚敏》-8.6基数排序

**基数排序详解**--<https://blog.csdn.net/u011948899/article/details/78027838>

基数排序分为两类：

第一类：最低位优先法，简称LSD法：先从最低位开始排序，再对次低位排序，直到对最高位排序后得到一个有序序列；具体过程如下图所示：

初始数组序列为：15,25,105,78,34,21,32,41，按照个位数大小依次入桶；

将桶中数依次倒出，对于同一个桶中的数按先进先出顺序倒出，结果为：21,41,32,34,15,25,105,78，再按十位数大小依次入桶；

将桶中数依次倒出，结果为：105,15,21,25,32,34,41,78，再按百位上数大小依次入桶，没有百位的数则按百位为0入桶；

将桶中数倒出，结果为：15,21,25,32,34,41,78,105

Java实现代码如下：

public void radixSort(int[] A,int n){

int max = A[0];

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

if(max < A[i])

max = A[i];

}

double d = Math.pow(10, String.valueOf(max).length());

int k = 1;

int[][] t = new int[10][n]; //桶

int[] num = new int[n]; //记录每个桶中存入数的个数

while(k < d){

for(int a : A){

int m = (a / k) % 10;

t[m][num[m]] = a;

num[m]++;

}

int c = 0;

for(int i = 0; i < n; i++){

if(num[i] != 0){

for(int j = 0;j < num[i];j++){

A[c++] = t[i][j];

}

}

num[i] = 0;

}

k = k \* 10;

}

}

时间复杂度：O(d(n+rd))

第二类：最高位优先法，简称MSD法：先从最高位开始排序，再逐个对各分组按次高位进行子排序，循环直到最低位。

仍以序列：15,25,105,78,34,21,32,41为例，从最高位百位依次入桶，只有105有百位，其他百位按0算；检测每个桶中的数据。当桶中的元素个数多于1个的时候，要对这个桶递归进行下一位的分组。

Java代码实现：

public class MSDSort {

public int[] sort(int[] A, int n){

int max = A[0];

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

if(max < A[i])

max = A[i];

}

int maxL = String.valueOf(max).length(); //获取数组中最长元素长度

int k = new Double(Math.pow(10, maxL - 1)).intValue();

int[][] t = new int[10][n]; //桶

int[] num = new int[n]; //记录每个桶中存入数的个数

for(int a : A){ //按最高位入桶

int m = (a / k) % 10;

t[m][num[m]] = a;

num[m]++;

}

int c = 0;

for(int i = 0; i < n; i++){

if(num[i] == 1){ //如果桶中只有一个数则直接取出

A[c++] = t[i][0];

}else if(num[i] > 1){ //如果桶中不止一个数，则另存如数组B递归

int[] B = new int[num[i]];

for(int j = 0;j < num[i];j++){

B[j] = t[i][j];

sort(B,num[i]); //递归方法

}

}

}

return A;

}

public static void main(String[] args) {

RadixSort r = new RadixSort();

int[] A = {12,1,23,123,34};

r.sort(A, A.length);

for(int a : A){

System.out.println(a);

}

}

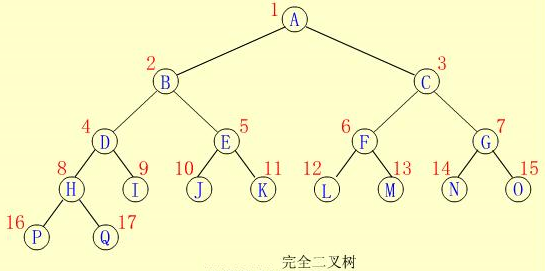
}

时间复杂度：O(d(n+rd))

## KMP

# 树

## 二叉树



二叉树的遍历操作，有时候直观看不出来，一定要画图，对照图片分析。

# 图