[**纸上谈兵: 排序算法简介及其C实现**](http://www.cnblogs.com/vamei/archive/2013/03/12/2948847.html)

作者：Vamei 出处：http://www.cnblogs.com/vamei 欢迎转载，也请保留这段声明。谢谢！

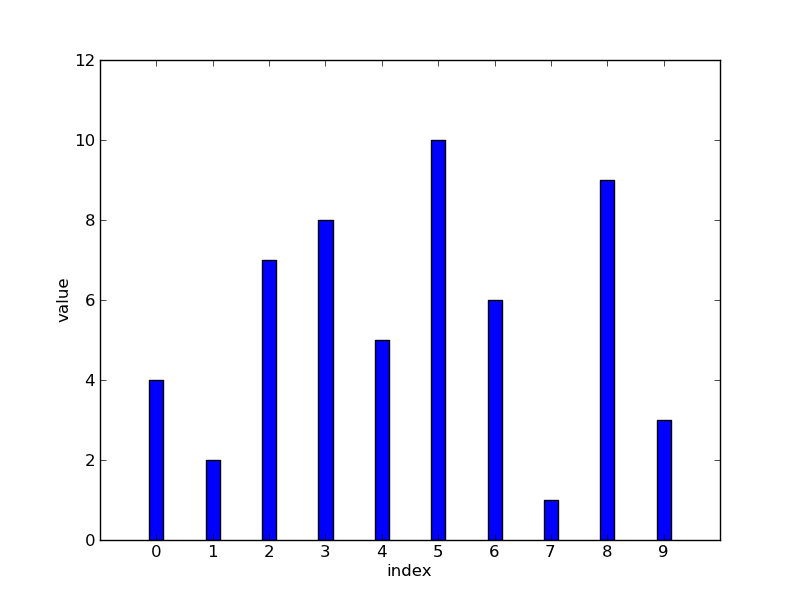
排序算法(Sorting Algorithm)是计算机算法的一个组成部分。

排序的目标是将一组数据 (即一个序列) 重新排列，排列后的数据符合从大到小 (或者从小到大) 的次序。这是古老但依然富有挑战的问题。Donald Knuth的经典之作《计算机程序设计艺术》(The Art of Computer Programming)的第三卷就专门用于讨论排序和查找。从无序到有序，有效的减小了系统的熵值，增加了系统的有序度。对于一个未知系统来说，有序是非常有用的先验知识。因此，排序算法很多时候构成了其他快速算法的基础，比如二分法就是基于有序序列的查找算法。直到今天，排序算法依然是计算机科学积极探索的一个方向。

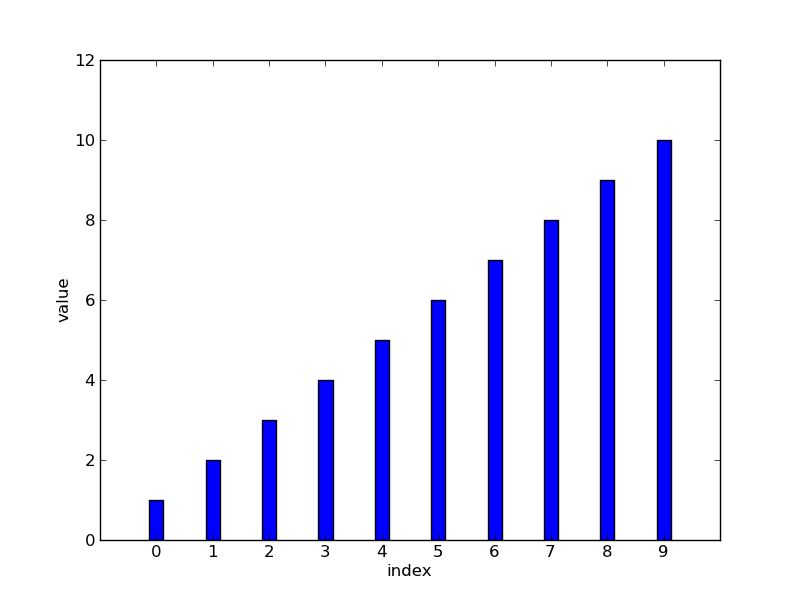
我在这里列出一些最常见的排序方法，并尝试使用C语言实现它们。一组数据存储为一个数组a，数组有n个元素。a[i]为数组中的一个元素，i为元素在数组中的位置 (index)。根据C的规定，数组下标从0开始。假设数组从左向右排列，下标为0的元素位于数组的最左边。

序列将最终排列成从小到大的顺序。下面函数中的参数ac是数组中元素的数目，也就是n。

(C语言的数组名都转成指针，传递给函数，所以需要传递数组中元素的数目ac给函数，详细见"Expert C Programming: Deep C Secrets"一书)



起始数列 （unsorted）



有序数列 (sorted)

下面的链接中，有相关算法的动画图例，强烈推荐同时阅读。

<http://www.sorting-algorithms.com/>

**冒泡排序 (Bubble Sort)**

对于一个已经排序好的序列，它的任意两个相邻元素，都应该满足a[i-1] <= a[i]的关系。冒泡排序相当暴力的实现了这一目标：不断扫描相邻元素，看它们是否违章。一旦违章，立即纠正。在冒泡排序时，计算机从右向左遍历数组，比较相邻的两个元素。如果两个元素的顺序是错的，那么sorry，请两位互换。如果两个元素的顺序是正确的，则不做交换。经过一次遍历，我们可以保证最小的元素(泡泡)处于最左边的位置。

然而，经过这么一趟，冒泡排序不能保证所有的元素已经按照次序排列好。我们需要再次从右向左遍历数组元素，进行冒泡排序。这一次遍历，我们不用考虑最左端的元素，因为该元素已经是最小的。遍历结束后，继续重复扫描…… 总共可能进行n-1次的遍历。

如果某次遍历过程中，没有发生交换，bingo，这个数组已经排序好，可以中止排序。如果起始时，最大的元素位于最左边，那么冒泡算法必须经过n-1次遍历才能将数组排列好，而不能提前完成排序。

[复制代码](javascript:void(0);)

/\*By Vamei\*/  
/\*swap the neighbors if out of order\*/

void bubble\_sort(int a[], int ac)

{

/\*use swap\*/

int i,j;

int sign;

for (j = 0; j < ac-1; j++) {

sign = 0;

for(i = ac-1; i > j; i--)

{

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

sign = 1;

swap(a+i, a+i-1);

}

}

if (sign == 0) break;

}

}

[复制代码](javascript:void(0);)

**插入排序 (Insertion Sort)**

假设在新生报到的时候，我们将新生按照身高排好队(也就是排序)。如果这时有一名学生加入，我们将该名学生加入到队尾。如果这名学生比前面的学生低，那么就让该学生和前面的学生交换位置。这名学生最终会换到应在的位置。这就是插入排序的基本原理。

对于起始数组来说，我们认为最初，有一名学生，也就是最左边的元素(i=0)，构成一个有序的队伍。

随后有第二个学生(i=1)加入队伍，第二名学生交换到应在的位置；随后第三个学生加入队伍，第三名学生交换到应在的位置…… 当n个学生都加入队伍时，我们的排序就完成了。

[复制代码](javascript:void(0);)

/\*By Vamei\*/

/\*insert the next element

into the sorted part\*/

void insert\_sort(int a[], int ac)

{

/\*use swap\*/

int i,j;

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

{

i = j-1;

while((i>=0) && (a[i+1] < a[i]))

{

swap(a+i+1, a+i);

i--;

}

}

}

[复制代码](javascript:void(0);)

**选择排序 (Selection Sort)**

排序的最终结果：任何一个元素都不大于位于它右边的元素 (a[i] <= a[j], if i <= j)。所以，在有序序列中，最小的元素排在最左的位置，第二小的元素排在i=1的位置…… 最大的元素排在最后。

选择排序是先找到起始数组中最小的元素，将它交换到i=0；然后寻找剩下元素中最小的元素，将它交换到i=1的位置…… 直到找到第二大的元素，将它交换到n-2的位置。这时，整个数组的排序完成。

[复制代码](javascript:void(0);)

/\*By Vamei\*/

/\*find the smallest of the rest,

then append to the sorted part\*/

void select\_sort(int a[], int ac)

{

/\*use swap\*/

int i,j;

int min\_idx;

for (j = 0; j < ac-1; j++)

{

min\_idx = j;

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

{

if (a[i] < a[min\_idx])

{

min\_idx = i;

}

}

swap(a+j, a+min\_idx);

}

}

[复制代码](javascript:void(0);)

**希尔排序 （Shell Sort）**

我们在冒泡排序中提到，最坏的情况发生在大的元素位于数组的起始。这些位于数组起始的大元素需要多次遍历，才能交换到队尾。这样的元素被称为乌龟(turtle)。

乌龟元素的原因在于，冒泡排序总是相邻的两个元素比较并交换。所以每次从右向左遍历，大元素只能向右移动一位。(小的元素位于队尾，被称为兔子(rabbit)元素，它们可以很快的交换到队首。)

希尔排序是以更大的间隔来比较和交换元素，这样，大的元素在交换的时候，可以向右移动不止一个位置，从而更快的移动乌龟元素。比如，可以将数组分为4个子数组（i=4k, i=4k+1, i=4k+2, i=4k+3），对每个子数组进行冒泡排序。比如子数组i=0，4，8，12...。此时，每次交换的间隔为4。

完成对四个子数组的排序后，数组的顺序并不一定能排列好。希尔排序会不断减小间隔，重新形成子数组，并对子数组冒泡排序…… 当间隔减小为1时，就相当于对整个数组进行了一次冒泡排序。随后，数组的顺序就排列好了。

希尔排序不止可以配合冒泡排序，还可以配合其他的排序方法完成。

[复制代码](javascript:void(0);)

/\*By Vamei\*/

/\*quickly sort the turtles at the tail of the array\*/

void shell\_sort(int a[], int ac)

{

int step;

int i,j;

int nsub;

int \*sub;

/\* initialize step \*/

step = 1;

while(step < ac) step = 3\*step + 1;

/\* when step becomes 1, it's equivalent to the bubble sort\*/

while(step > 1) {

/\* step will go down to 1 at most \*/

step = step/3 + 1;

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

/\* pick an element every step,

and combine into a sub-array \*/

nsub = (ac - i - 1)/step + 1;

sub = (int \*) malloc(sizeof(int)\*nsub);

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

sub[j] = a[i+j\*step];

}

/\* sort the sub-array by bubble sorting.

It could be other sorting methods \*/

bubble\_sort(sub, nsub);

/\* put back the sub-array\*/

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

a[i+j\*step] = sub[j];

}

/\* free sub-array \*/

free(sub);

}

}

}

[复制代码](javascript:void(0);)

Shell Sorting依赖于间隔(step)的选取。一个常见的选择是将本次间隔设置为上次间隔的1/1.3。见参考书籍。

**归并排序 (Merge Sort)**

如果我们要将一副扑克按照数字大小排序。此前已经有两个人分别将其中的一半排好顺序。那么我们可以将这两堆扑克向上放好，假设小的牌在上面。此时，我们将看到牌堆中最上的两张牌。

我们取两张牌中小的那张取出放在手中。两个牌堆中又是两张牌暴露在最上面，继续取小的那张放在手中…… 直到所有的牌都放入手中，那么整副牌就排好顺序了。这就是归并排序。

下面的实现中，使用递归：

[复制代码](javascript:void(0);)

/\*By Vamei\*/

/\*recursively merge two sorted arrays\*/

void merge\_sort(int \*a, int ac)

{

int i, j, k;

int ac1, ac2;

int \*ah1, \*ah2;

int \*container;

/\*base case\*/

if (ac <= 1) return;

/\*split the array into two\*/

ac1 = ac/2;

ac2 = ac - ac1;

ah1 = a + 0;

ah2 = a + ac1;

/\*recursion\*/

merge\_sort(ah1, ac1);

merge\_sort(ah2, ac2);

/\*merge\*/

i = 0;

j = 0;

k = 0;

container = (int \*) malloc(sizeof(int)\*ac);

while(i<ac1 && j<ac2) {

if (ah1[i] <= ah2[j]) {

container[k++] = ah1[i++];

}

else {

container[k++] = ah2[j++];

}

}

while (i < ac1) {

container[k++] = ah1[i++];

}

while (j < ac2) {

container[k++] = ah2[j++];

}

/\*copy back the sorted array\*/

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

a[i] = container[i];

}

/\*free space\*/

free(container);

}

[复制代码](javascript:void(0);)

**快速排序 (Quick Sort)**

我们依然考虑按照身高给学生排序。在快速排序中，我们随便挑出一个学生，以该学生的身高为参考(pivot)。然后让比该学生低的站在该学生的右边，剩下的站在该学生的左边。

很明显，所有的学生被分成了两组。该学生右边的学生的身高都大于该学生左边的学生的身高。

我们继续，在低身高学生组随便挑出一个学生，将低身高组的学生分为两组(很低和不那么低)。同样，将高学生组也分为两组(不那么高和很高)。

如此继续细分，直到分组中只有一个学生。当所有的分组中都只有一个学生时，则排序完成。

在下面的实现中，使用递归:

[复制代码](javascript:void(0);)

/\*By Vamei\*/

/\*select pivot, put elements (<= pivot) to the left\*/

void quick\_sort(int a[], int ac)

{

/\*use swap\*/

/\* pivot is a position,

all the elements before pivot is smaller or equal to pvalue \*/

int pivot;

/\* the position of the element to be tested against pivot \*/

int sample;

/\* select a pvalue.

Median is supposed to be a good choice, but that will itself take time.

here, the pvalue is selected in a very simple wayi: a[ac/2] \*/

/\* store pvalue at a[0] \*/

swap(a+0, a+ac/2);

pivot = 1;

/\* test each element \*/

for (sample=1; sample<ac; sample++) {

if (a[sample] < a[0]) {

swap(a+pivot, a+sample);

pivot++;

}

}

/\* swap an element (which <= pvalue) with a[0] \*/

swap(a+0,a+pivot-1);

/\* base case, if only two elements are in the array,

the above pass has already sorted the array \*/

if (ac<=2) return;

else {

/\* recursion \*/

quick\_sort(a, pivot);

quick\_sort(a+pivot, ac-pivot);

}

}

[复制代码](javascript:void(0);)

理想的pivot是采用分组元素中的中位数。然而寻找中位数的算法需要另行实现。也可以随机选取元素作为pivot，随机选取也需要另行实现。为了简便，我每次都采用中间位置的元素作为pivot。

**堆排序 (Heap Sort)**

堆(heap)是常见的数据结构。它是一个有优先级的队列。最常见的堆的实现是一个有限定操作的Complete Binary Tree。这个Complete Binary Tree保持堆的特性，也就是父节点(parent)大于子节点(children)。因此，堆的根节点是所有堆元素中最小的。堆定义有插入节点和删除根节点操作，这两个操作都保持堆的特性。

我们可以将无序数组构成一个堆，然后不断取出根节点，最终构成一个有序数组。

堆的更详细描述请阅读参考书目。

下面是堆的数据结构，以及插入节点和删除根节点操作。你可以很方便的构建堆，并取出根节点，构成有序数组。

[复制代码](javascript:void(0);)

/\* By Vamei

Use an big array to implement heap

DECLARE: int heap[MAXSIZE] in calling function

heap[0] : total nodes in the heap

for a node i, its children are i\*2 and i\*2+1 (if exists)

its parent is i/2 \*/

void insert(int new, int heap[])

{

int childIdx, parentIdx;

heap[0] = heap[0] + 1;

heap[heap[0]] = new;

/\* recover heap property \*/

percolate\_up(heap);

}

static void percolate\_up(int heap[]) {

int lightIdx, parentIdx;

lightIdx = heap[0];

parentIdx = lightIdx/2;

/\* lightIdx is root? && swap? \*/

while((parentIdx > 0) && (heap[lightIdx] < heap[parentIdx])) {

/\* swap \*/

swap(heap + lightIdx, heap + parentIdx);

lightIdx = parentIdx;

parentIdx = lightIdx/2;

}

}

int delete\_min(int heap[])

{

int min;

if (heap[0] < 1) {

/\* delete element from an empty heap \*/

printf("Error: delete\_min from an empty heap.");

exit(1);

}

/\* delete root

move the last leaf to the root \*/

min = heap[1];

swap(heap + 1, heap + heap[0]);

heap[0] -= 1;

/\* recover heap property \*/

percolate\_down(heap);

return min;

}

static void percolate\_down(int heap[]) {

int heavyIdx;

int childIdx1, childIdx2, minIdx;

int sign; /\* state variable, 1: swap; 0: no swap \*/

heavyIdx = 1;

do {

sign = 0;

childIdx1 = heavyIdx\*2;

childIdx2 = childIdx1 + 1;

if (childIdx1 > heap[0]) {

/\* both children are null \*/

break;

}

else if (childIdx2 > heap[0]) {

/\* right children is null \*/

minIdx = childIdx1;

}

else {

minIdx = (heap[childIdx1] < heap[childIdx2]) ?

childIdx1 : childIdx2;

}

if (heap[heavyIdx] > heap[minIdx]) {

/\* swap with child \*/

swap(heap + heavyIdx, heap + minIdx);

heavyIdx = minIdx;

sign = 1;

}

} while(sign == 1);

}

[复制代码](javascript:void(0);)

**总结**

除了上面的算法，还有诸如Bucket Sorting, Radix Sorting涉及。我会在未来实现了相关算法之后，补充到这篇文章中。相关算法的时间复杂度分析可以参考书目中找到。我自己也做了粗糙的分析。如果博客园能支持数学公式的显示，我就把自己的分析过程贴出来，用于引玉。

上面的各个代码是我自己写的，只进行了很简单的测试。如果有错漏，先谢谢你的指正。

最后，上文中用到的交换函数为：

[复制代码](javascript:void(0);)

/\* By Vamei \*/

/\* exchange the values pointed by pa and pb\*/

void swap(int \*pa, int \*pb)

{

int tmp;

tmp = \*pa;

\*pa = \*pb;

\*pb = tmp;

}

[**纸上谈兵: 表 (list)**](http://www.cnblogs.com/vamei/archive/2013/03/14/2958940.html)

作者：Vamei 出处：http://www.cnblogs.com/vamei 欢迎转载，也请保留这段声明。谢谢！

**表**

表(list)是常见的数据结构。从数学上来说，表是一个有序的元素集合。在C语言的内存中，表储存为分散的节点(node)。每个节点包含有一个元素，以及一个指向下一个(或者上一个)元素的指针。如下图所示:

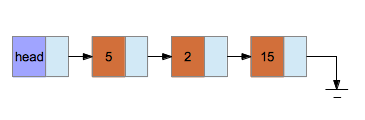
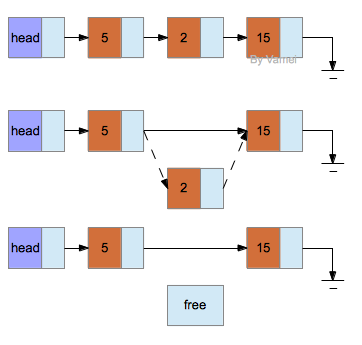


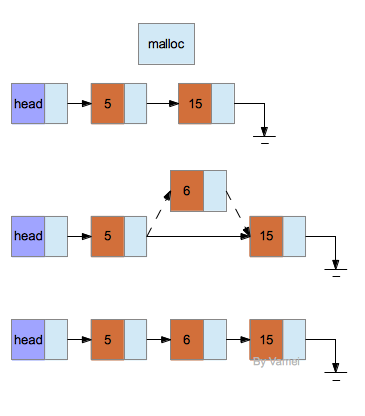
表: 橙色储存数据，蓝色储存指针

图中的表中有四个节点。第一个节点是头节点(head node)，这个节点不用于储存元素，只用于标明表的起始。头节点可以让我们方便的插入或者删除表的第一个元素。整个表中包含有三个元素(5， 2， 15)。每个节点都有一个指针，指向下一个节点。最后一个节点的指针为NULL，我们用“接地”来图示该指针。

表的功能与数组(array)很类似，数组也是有序的元素集合，但数组在内存中为一段连续内存，而表的每个节点占据的内存可以是离散的。在数组中，我们通过跳过固定的内存长度来寻找某个编号的元素。但在表中，我们必须沿着指针联系起的长链，遍历查询元素。此外，数组有固定的大小，表可以根据运行情况插入或者删除节点，动态的更改大小。表插入节点时需要从进程空间的堆中开辟内存空间，用以储存节点。删除节点可以将节点占据的内存归还给进程空间。

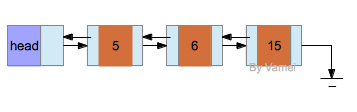


删除节点, free释放内存

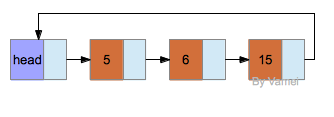


插入节点，malloc开辟内存

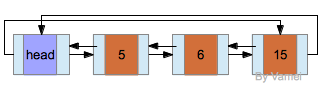
表有多种变种。上面的表中，指针指向是从前向后的，称为单向链表(linked list)。还有双向链表(double-linked list)，即每个节点增加一个指向前面一个元素的指针。以及循环链表(tabular list)，最后一个元素的指针并不为NULL，而是指向头节点。不同类型的链表有不同的应用场景。



双向链表



循环链表



双向循环链表

**单向链表的C实现**

一个数据结构的实现有两方面: 1. 数据结构的内存表达方式; 2. 定义在该数据结构上的操作。我们这里实现最简单的单向链表。表所支持的操作很灵活多样，我们这里定义一些最常见的操作。每个操作都写成一个函数。

[复制代码](javascript:void(0);)

/\* By Vamei \*/  
#include <stdio.h>

#include <stdlib.h>

typedef struct node \*LIST;   
typedef struct node \*position;  
  
/\* node，节点 \*/

struct node {

int element;

position next;

};

/\*   
 \* operations (stereotype)  
 \* 操作  
 \*/  
LIST init\_list(void);

void delete\_list(LIST);  
int is\_null(LIST);  
void insert\_node(position, int);

void delete\_node(LIST, position);  
position find\_last(LIST);

position find\_value(LIST, int);

position find\_previous(LIST, position);  
void print\_list(LIST);

/\* for testing purpose \*/

void main()

{

LIST L;

position np;

int i;

/\* elements to be put into the list \*/

int a[] = {1, 3, 5, 7, 9};

/\* initiate a list \*/

L = init\_list();

print\_list(L);

/\* insert nodes. Insert just after head node \*/

for (i=4; i>=0; i--) {

insert\_node(L, a[i]);

}

print\_list(L);

/\* delete first node with value 5 \*/

np = find\_value(L, 5);

delete\_node(L, np);

print\_list(L);

/\* delete list \*/

delete\_list(L);

/\* initiate a list \*/

L = init\_list();

print\_list(L);

/\* insert nodes. Insert just after head node \*/

for (i=4; i>=0; i--) {

insert\_node(L, a[i]);

}

print\_list(L);

/\* delete list \*/

delete\_list(L);

}

/\*

\* Traverse the list and print each element  
 \* 打印表

\*/

void print\_list(LIST L)

{

position np;

if(is\_null(L)) {

printf("Empty List\n\n");

return;

}

np = L;

while(np->next != NULL) {

np = np->next;

printf("%p: %d \n", np, np->element);

}

printf("\n");

}

/\*

\* Initialize a linked list. This list has a head node

\* head node doesn't store valid element value  
 \* 创建表

\*/

LIST init\_list(void)

{

LIST L;

L = (position) malloc(sizeof(struct node));

L->next = NULL;

return L;

}

/\*

\* Delete all nodes in a list  
 \* 删除表

\*/

void delete\_list(LIST L)

{

position np, next;

np = L;

do {

next = np->next;

free(np);

np = next;

} while(next != NULL);

}

/\*

\* if a list only has head node, then the list is null.  
 \* 判断表是否为空

\*/

int is\_null(LIST L)

{

return ((L->next)==NULL);

}

/\*

\* insert a node after position np  
 \* 在np节点之后，插入节点

\*/

void insert\_node(position np, int value)

{

position nodeAddr;

nodeAddr = (position) malloc(sizeof(struct node));

nodeAddr->element = value;

nodeAddr->next = np->next;

np->next = nodeAddr;

}

/\*

\* delete node at position np  
 \* 删除np节点

\*/

void delete\_node(LIST L, position np)

{

position previous, next;

next = np->next;

previous = find\_previous(L, np);

if(previous != NULL) {

previous->next = next;

free(np);

}

else {

printf("Error: np not in the list");

}

}

/\*  
 \* find the last node of the list  
 \* 寻找表的最后一个节点  
 \*/

position find\_last(LIST L)

{

position np;

np = L;

while(np->next != NULL) {

np = np->next;

}

return np;

}

/\*

\* This function serves for 2 purposes:

\* 1. find previous node

\* 2. return NULL if the position isn't in the list  
 \* 寻找npTarget节点前面的节点

\*/

position find\_previous(LIST L, position npTarget)

{

position np;

np = L;

while (np->next != NULL) {

if (np->next == npTarget) return np;

np = np->next;

}

return NULL;

}

/\*

\* find the first node with specific value  
 \* 查询

\*/

position find\_value(LIST L, int value)

{

position np;

np = L;

while (np->next != NULL) {

np = np->next;

if (np->element == value) return np;

}

return NULL;

}

[复制代码](javascript:void(0);)

在main()函数中，我们初始化表，然后插入(1, 3, 5, 7, 9）。又删除元素5。可以看到，节点零散的分布在内存中。删除节点操作不会影响其他节点的存储位置。

我们随后删除表，又重新创建表。可以看到，这次表占据内存的位置与第一次不同。

下面是main()函数的运行结果。

Empty List  
  
0x154d0b0: 1   
0x154d090: 3   
0x154d070: 5   
0x154d050: 7   
0x154d030: 9   
  
0x154d0b0: 1   
0x154d090: 3   
0x154d050: 7   
0x154d030: 9   
  
Empty List  
  
0x154d070: 1   
0x154d010: 3   
0x154d0b0: 5   
0x154d090: 7   
0x154d050: 9

**总结**

表: 内存中离散分布的有序节点

插入，删除节点

[**纸上谈兵: 栈 (stack)**](http://www.cnblogs.com/vamei/archive/2013/03/14/2960201.html)

作者：Vamei 出处：http://www.cnblogs.com/vamei 欢迎转载，也请保留这段声明。谢谢！

栈(stack)是简单的数据结构，但在计算机中使用广泛。它是有序的元素集合。栈最显著的特征是LIFO (Last In, First Out, 后进先出)。当我们往箱子里存放一叠书时，先存放的书在箱子下面，我们必须将后存放的书取出来，才能看到和拿出早先存放的书。



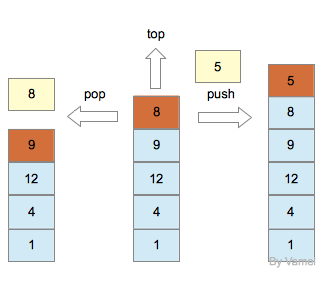
栈中的每个元素称为一个frame。而最上层元素称为top frame。栈只支持三个操作， pop, top, push。

pop取出栈中最上层元素(8)，栈的最上层元素变为早先进入的元素(9)。

top查看栈的最上层元素(8)。

push将一个新的元素(5)放在栈的最上层。

栈不支持其他操作。如果想取出元素12, 必须进行3次pop操作。



栈以及pop, push, top操作

栈最经典的计算机应用是函数调用。每个进程都会有一个栈，每个frame中记录了调用函数的参数，自动变量和返回地址。当该函数调用一个新的函数时，栈中会 push一个frame。当函数执行完毕返回时，该frame会pop，从而进入调用该函数的原函数，继续执行。详细请参阅[Linux从程序到进程](http://www.cnblogs.com/vamei/archive/2012/10/09/2715388.html)

实际使用的栈并不一定符合数据结构的栈。比如说，有的语言允许被调用函数查看非top frame的记录。这样的栈更类似于下面的经典游戏



**栈的C实现 (基于表)**

由于栈是限定了操作的有序的元素集合，所以我们既可以在数组的基础上来实现栈，也可以在表的基础上来实现栈。如果使用数组来实现栈，我们需要预留充足的空间供栈使用，并需要一个下标来记录最上层元素的位置。

我们这里使用单向链表来实现栈。我们可以利用介绍表(list)的文章中已经定义的操作来实现三个操作，但这里相对独立的重写了代码。

[复制代码](javascript:void(0);)

/\* By Vamei \*/

/\* use single-linked list to implement stack \*/

#include <stdio.h>

#include <stdlib.h>

typedef struct node \*position;

typedef int ElementTP;

// point to the head node of the list

typedef struct node \*STACK;

struct node {

ElementTP element;

position next;

};

STACK init\_stack(void);

void delete\_stack(STACK);

ElementTP top(STACK);

void push(STACK, ElementTP);

ElementTP pop(STACK);

int is\_null(STACK);

void main(void)

{

ElementTP a;

int i;

STACK sk;

sk = init\_stack();

push(sk, 1);

push(sk, 2);

push(sk, 8);

printf("Stack is null? %d\n", is\_null(sk));

for (i=0; i<3; i++) {

a = pop(sk);

printf("pop: %d\n", a);

}

printf("Stack is null? %d\n", is\_null(sk));

delete\_stack(sk);

}

/\*

\* initiate the stack

\* malloc the head node.

\* Head node doesn't store valid data

\* head->next is the top node

\*/

STACK init\_stack(void)

{

position np;

STACK sk;

np = (position) malloc(sizeof(struct node));

np->next = NULL; // sk->next is the top node

sk = np;

return sk;

}

/\* pop out all elements

\* and then delete head node

\*/

void delete\_stack(STACK sk)

{

while(!is\_null(sk)) {

pop(sk);

}

free(sk);

}

/\*

\* View the top frame

\*/

ElementTP top(STACK sk)

{

return (sk->next->element);

}

/\*

\* push a value into the stack

\*/

void push(STACK sk, ElementTP value)

{

position np, oldTop;

oldTop = sk->next;

np = (position) malloc(sizeof(struct node));

np->element = value;

np->next = sk->next;

sk->next = np;

}

/\*

\* pop out the top value

\*/

ElementTP pop(STACK sk)

{

ElementTP element;

position top, newTop;

if (is\_null(sk)) {

printf("pop() on an empty stack");

exit(1);

}

else {

top = sk->next;

element = top->element;

newTop = top->next;

sk->next = newTop;

free(top);

return element;

}

}

/\* check whether a stack is empty\*/

int is\_null(STACK sk)

{

return (sk->next == NULL);

}

[复制代码](javascript:void(0);)

输出结果:

Stack is null? 0  
pop: 8  
pop: 2  
pop: 1  
Stack is null? 1

**总结**

栈, LIFO

pop, push, top

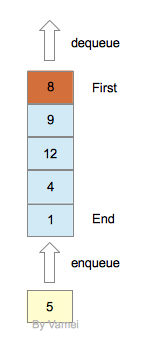
[**纸上谈兵: 队列 (queue)**](http://www.cnblogs.com/vamei/archive/2013/03/15/2961729.html)

作者：Vamei 出处：http://www.cnblogs.com/vamei 欢迎转载，也请保留这段声明。谢谢！

队列(queue)是一个简单而常见的数据结构。队列也是有序的元素集合。队列最大的特征是First In, First Out (FIFO，先进先出)，即先进入队列的元素，先被取出。这一点与[栈(stack)](http://www.cnblogs.com/vamei/archive/2013/03/14/2960201.html)形成有趣的对比。队列在生活中很常见，排队买票、排队等车…… 先到的人先得到服务并离开队列，后来的人加入到队列的最后。队列是比较公平的分配有限资源的方式，可以让队列的人以相似的等待时间获得服务。



队列支持两个操作，队首的元素离开队列(dequeue)，和新元素加入队尾(enqueue)。



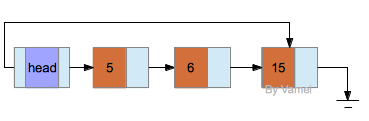
队列

队列在计算机中应用很广泛。一个经典的应用是消息队列(参考[Linux进程间通信](http://www.cnblogs.com/vamei/archive/2012/10/10/2715398.html))，实际上就是利用队列来分配有限的进程。还有FIFO文件(哦，你可以看到，这种文件叫做FIFO，肯定是和队列有关)，用以实现管道传输。再比如，我们将多个打印任务发送给打印机，打印机也是使用队列来安排任务的顺序。

**队列的C实现 (基于表)**

和栈相似，队列也可以有多种实现方式，这里是基于单链表的实现。

与[表(list)](http://www.cnblogs.com/vamei/archive/2013/03/14/2958940.html)中的实现方式略有不同的是，这里的head node有两个指针，一个(next)指向下一个元素，一个(end)指向队列的最后一个元素。这样做的目的是方便我们找到队尾，以方便的进行enqueue操作。我们依然可以使用之前定义的表，在需要找到队尾的时候遍历搜索到最后一个元素。



用于队列的表

下面是代码:

[复制代码](javascript:void(0);)

/\* By Vamei \*/

/\* use single-linked list to implement queue \*/

#include <stdio.h>

#include <stdlib.h>

typedef struct node \*position;

typedef int ElementTP;

// point to the head node of the list

typedef struct HeadNode \*QUEUE;

struct node {

ElementTP element;

position next;

};

/\*

\* CAUTIOUS: "HeadNode" is different from "node",

\* it's another struct

\* end: points to the last value in the queue

\*/

struct HeadNode {

ElementTP element;

position next;

position end;

};

/\*

\* Operations

\*/

QUEUE init\_queue(void);

void delete\_queue(QUEUE);

void enqueue(QUEUE, ElementTP);

ElementTP dequeue(QUEUE);

int is\_null(QUEUE);

/\*

\* Test

\*/

void main(void)

{

ElementTP a;

int i;

QUEUE qu;

qu = init\_queue();

enqueue(qu, 1);

enqueue(qu, 2);

enqueue(qu, 8);

printf("Queue is null? %d\n", is\_null(qu));

for (i=0; i<3; i++) {

a = dequeue(qu);

printf("dequeue: %d\n", a);

}

printf("Queue is null? %d\n", is\_null(qu));

delete\_queue(qu);

}

/\*

\* initiate the queue

\* malloc the head node.

\* Head node doesn't store valid data

\* head->next is the first node in the queue.

\*/

QUEUE init\_queue(void)

{

QUEUE hnp;

hnp = (QUEUE) malloc(sizeof(struct HeadNode));

hnp->next = NULL; // qu->next is the first node

hnp->end = NULL;

return hnp;

}

/\*

\* dequeue all elements

\* and then delete head node

\*/

void delete\_queue(QUEUE qu)

{

while(!is\_null(qu)) {

dequeue(qu);

}

free(qu);

}

/\*

\* enqueue a value to the end of the queue

\*/

void enqueue(QUEUE qu, ElementTP value)

{

position np, oldEnd;

oldEnd = qu->end;

np = (position) malloc(sizeof(struct node));

np->element = value;

np->next = NULL;

/\* if queue is empyt, then oldEnd is NULL \*/

if (oldEnd) {

oldEnd->next = np;

}

else {

qu->next = np;

}

qu->end = np;

}

/\*

\* dequeue the first value

\*/

ElementTP dequeue(QUEUE qu)

{

ElementTP element;

position first, newFirst;

if (is\_null(qu)) {

printf("dequeue() on an empty queue");

exit(1);

}

else {

first = qu->next;

element = first->element;

newFirst = first->next;

qu->next = newFirst;

free(first);

return element;

}

}

/\*

\* check: queue is empty?

\*/

int is\_null(QUEUE qu)

{

return (qu->next == NULL);

}

[复制代码](javascript:void(0);)

运行结果如下:

Queue is null? 0  
dequeue: 1  
dequeue: 2  
dequeue: 8  
Queue is null? 1

**总结**

队列，FIFO

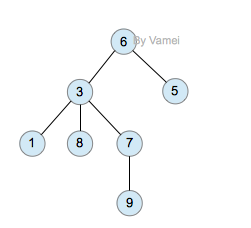
enqueue, dequeue

# [纸上谈兵: 树, 二叉树, 二叉搜索树](http://www.cnblogs.com/vamei/archive/2013/03/17/2962290.html)

作者：Vamei 出处：http://www.cnblogs.com/vamei 欢迎转载，也请保留这段声明。谢谢！

### 树的特征和定义

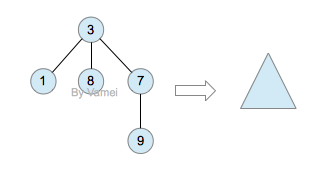
树(Tree)是元素的集合。我们先以比较直观的方式介绍树。下面的数据结构是一个树：



树有多个节点(node)，用以储存元素。某些节点之间存在一定的关系，用连线表示，连线称为边(edge)。边的上端节点称为父节点，下端称为子节点。树像是一个不断分叉的树根。

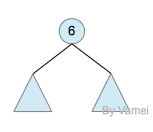
每个节点可以有多个子节点(children)，而该节点是相应子节点的父节点(parent)。比如说，3,5是6的子节点，6是3,5的父节点；1,8,7是3的子节点, 3是1,8,7的父节点。树有一个没有父节点的节点，称为根节点(root)，如图中的6。没有子节点的节点称为叶节点(leaf)，比如图中的1,8,9,5节点。从图中还可以看到，上面的树总共有4个层次，6位于第一层，9位于第四层。树中节点的最大层次被称为深度。也就是说，该树的深度(depth)为4。

如果我们从节点3开始向下看，而忽略其它部分。那么我们看到的是一个以节点3为根节点的树：



三角形代表一棵树

再进一步，如果我们定义孤立的一个节点也是一棵树的话，原来的树就可以表示为根节点和子树(subtree)的关系:



上述观察实际上给了我们一种严格的定义树的方法：

1. 树是元素的集合。

2. 该集合可以为空。这时树中没有元素，我们称树为空树 (empty tree)。

3. 如果该集合不为空，那么该集合有一个根节点，以及0个或者多个子树。根节点与它的子树的根节点用一个边(edge)相连。

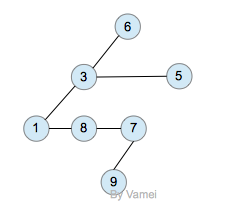
上面的第三点是以递归的方式来定义树，也就是在定义树的过程中使用了树自身(子树)。由于树的递归特征，许多树相关的操作也可以方便的使用递归实现。我们将在后面看到。

(上述定义来自"Data Structures and Algorithm Analysis in C, by Mark Allen Weiss"。 我觉得有一点不太严格的地方。如果说空树属于树，第三点应该是 “...以及0个和多个非空子树...” )

### 树的实现

树的示意图已经给出了树的一种内存实现方式: 每个节点储存元素和多个指向子节点的指针。然而，子节点数目是不确定的。一个父节点可能有大量的子节点，而另一个父节点可能只有一个子节点，而树的增删节点操作会让子节点的数目发生进一步的变化。这种不确定性就可能带来大量的内存相关操作，并且容易造成内存的浪费。

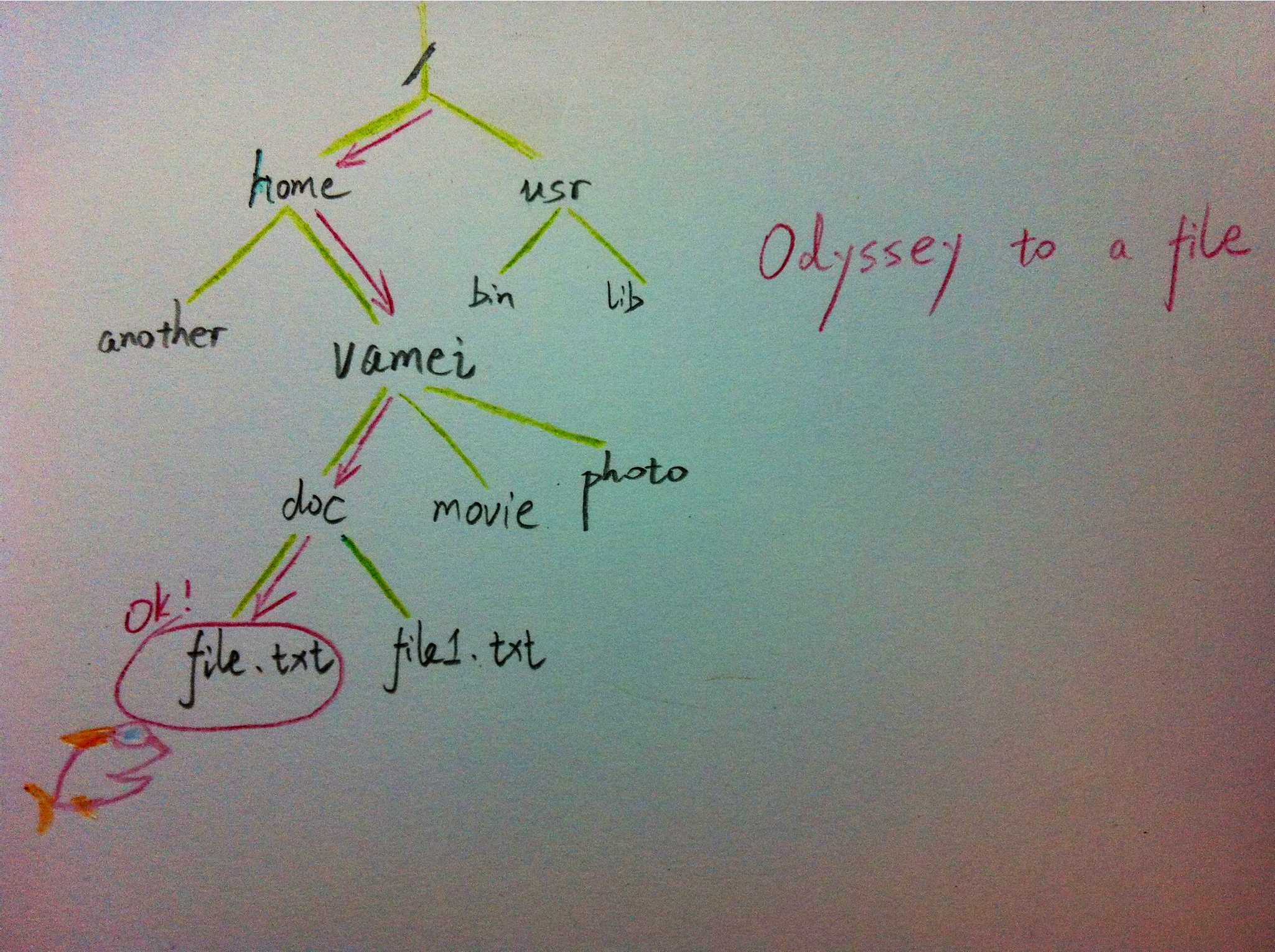
一种经典的实现方式如下:



树的内存实现

拥有同一父节点的两个节点互为兄弟节点(sibling)。上图的实现方式中，每个节点包含有一个指针指向第一个子节点，并有另一个指针指向它的下一个兄弟节点。这样，我们就可以用统一的、确定的结构来表示每个节点。

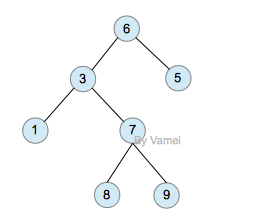
计算机的文件系统是树的结构，比如[Linux文件管理背景知识](http://www.cnblogs.com/vamei/archive/2012/09/09/2676792.html)中所介绍的。在UNIX的文件系统中，每个文件(文件夹同样是一种文件)，都可以看做是一个节点。非文件夹的文件被储存在叶节点。文件夹中有指向父节点和子节点的指针(在UNIX中，文件夹还包含一个指向自身的指针，这与我们上面见到的树有所区别)。在git中，也有类似的树状结构，用以表达整个文件系统的版本变化 (参考[版本管理三国志](http://www.cnblogs.com/vamei/archive/2013/02/21/2918069.html))。



文件树

### 二叉搜索树的C实现

二叉树(binary)是一种特殊的树。二叉树的每个节点最多只能有2个子节点：

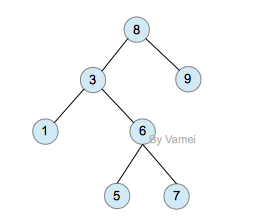


二叉树

由于二叉树的子节点数目确定，所以可以直接采用上图方式在内存中实现。每个节点有一个左子节点(left children)和右子节点(right children)。左子节点是左子树的根节点，右子节点是右子树的根节点。

如果我们给二叉树加一个额外的条件，就可以得到一种被称作二叉搜索树(binary search tree)的特殊二叉树。二叉搜索树要求：每个节点都不比它左子树的任意元素小，而且不比它的右子树的任意元素大。

(如果我们假设树中没有重复的元素，那么上述要求可以写成：每个节点比它左子树的任意节点大，而且比它右子树的任意节点小)



二叉搜索树，注意树中元素的大小

二叉搜索树可以方便的实现搜索算法。在搜索元素x的时候，我们可以将x和根节点比较:

1. 如果x等于根节点，那么找到x，停止搜索 (终止条件)

2. 如果x小于根节点，那么搜索左子树

3. 如果x大于根节点，那么搜索右子树

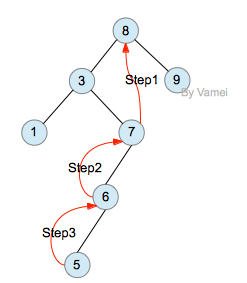
二叉搜索树所需要进行的操作次数最多与树的深度相等。n个节点的二叉搜索树的深度最多为n，最少为log(n)。

下面是用C语言实现的二叉搜索树，并有搜索，插入，删除，寻找最大最小节点的操作。每个节点中存有三个指针，一个指向父节点，一个指向左子节点，一个指向右子节点。

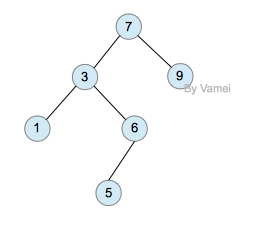
(这样的实现是为了方便。节点可以只保存有指向左右子节点的两个指针，并实现上述操作。)

删除节点相对比较复杂。删除节点后，有时需要进行一定的调整，以恢复二叉搜索树的性质(每个节点都不比它左子树的任意元素小，而且不比它的右子树的任意元素大)。

* 叶节点可以直接删除。
* 删除非叶节点时，比如下图中的节点8，我们可以删除左子树中最大的元素(或者右树中最大的元素)，用删除的节点来补充元素8产生的空缺。但该元素可能也不是叶节点，所以它所产生的空缺需要其他元素补充…… 直到最后删除一个叶节点。上述过程可以递归实现。



删除节点



删除节点后的二叉搜索树

[复制代码](javascript:void(0);)

/\* By Vamei \*/

/\* binary search tree \*/

#include <stdio.h>

#include <stdlib.h>

typedef struct node \*position;

typedef int ElementTP;

struct node {

position parent;

ElementTP element;

position lchild;

position rchild;

};

/\* pointer => root node of the tree \*/

typedef struct node \*TREE;

void print\_sorted\_tree(TREE);

position find\_min(TREE);

position find\_max(TREE);

position find\_value(TREE, ElementTP);

position insert\_value(TREE, ElementTP);

ElementTP delete\_node(position);

static int is\_root(position);

static int is\_leaf(position);

static ElementTP delete\_leaf(position);

static void insert\_node\_to\_nonempty\_tree(TREE, position);

void main(void)

{

TREE tr;

position np;

ElementTP element;

tr = NULL;

tr = insert\_value(tr, 18);

tr = insert\_value(tr, 5);

tr = insert\_value(tr, 2);

tr = insert\_value(tr, 8);

tr = insert\_value(tr, 81);

tr = insert\_value(tr, 101);

printf("Original:\n");

print\_sorted\_tree(tr);

np = find\_value(tr, 8);

if(np != NULL) {

delete\_node(np);

printf("After deletion:\n");

print\_sorted\_tree(tr);

}

}

/\*

\* print values of the tree in sorted order

\*/

void print\_sorted\_tree(TREE tr)

{

if (tr == NULL) return;

print\_sorted\_tree(tr->lchild);

printf("%d \n", tr->element);

print\_sorted\_tree(tr->rchild);

}

/\*

\* search for minimum value

\* traverse lchild

\*/

position find\_min(TREE tr)

{

position np;

np = tr;

if (np == NULL) return NULL;

while(np->lchild != NULL) {

np = np->lchild;

}

return np;

}

/\*

\* search for maximum value

\* traverse rchild

\*/

position find\_max(TREE tr)

{

position np;

np = tr;

if (np == NULL) return NULL;

while(np->rchild != NULL) {

np = np->rchild;

}

return np;

}

/\*

\* search for value

\*

\*/

position find\_value(TREE tr, ElementTP value)

{

if (tr == NULL) return NULL;

if (tr->element == value) {

return tr;

}

else if (value < tr->element) {

return find\_value(tr->lchild, value);

}

else {

return find\_value(tr->rchild, value);

}

}

/\*

\* delete node np

\*/

ElementTP delete\_node(position np)

{

position replace;

ElementTP element;

if (is\_leaf(np)) {

return delete\_leaf(np);

}

else {

/\* if a node is not a leaf, then we need to find a replacement \*/

replace = (np->lchild != NULL) ? find\_max(np->lchild) : find\_min(np->rchild);

element = np->element;

np->element = delete\_node(replace);

return element;

}

}

/\*

\* insert a value into the tree

\* return root address of the tree

\*/

position insert\_value(TREE tr, ElementTP value) {

position np;

/\* prepare the node \*/

np = (position) malloc(sizeof(struct node));

np->element = value;

np->parent = NULL;

np->lchild = NULL;

np->rchild = NULL;

if (tr == NULL) tr = np;

else {

insert\_node\_to\_nonempty\_tree(tr, np);

}

return tr;

}

//=============================================

/\*

\* np is root?

\*/

static int is\_root(position np)

{

return (np->parent == NULL);

}

/\*

\* np is leaf?

\*/

static int is\_leaf(position np)

{

return (np->lchild == NULL && np->rchild == NULL);

}

/\*

\* if an element is a leaf,

\* then it could be removed with no side effect.

\*/

static ElementTP delete\_leaf(position np)

{

ElementTP element;

position parent;

element = np->element;

parent = np->parent;

if(!is\_root(np)) {

if (parent->lchild == np) {

parent->lchild = NULL;

}

else {

parent->rchild = NULL;

}

}

free(np);

return element;

}

/\*

\* insert a node to a non-empty tree

\* called by insert\_value()

\*/

static void insert\_node\_to\_nonempty\_tree(TREE tr, position np)

{

/\* insert the node \*/

if(np->element <= tr->element) {

if (tr->lchild == NULL) {

/\* then tr->lchild is the proper place \*/

tr->lchild = np;

np->parent = tr;

return;

}

else {

insert\_node\_to\_nonempty\_tree(tr->lchild, np);

}

}

else if(np->element > tr->element) {

if (tr->rchild == NULL) {

tr->rchild = np;

np->parent = tr;

return;

}

else {

insert\_node\_to\_nonempty\_tree(tr->rchild, np);

}

}

}

[复制代码](javascript:void(0);)

运行结果:

Original:  
2   
5   
8   
18   
81   
101   
After deletion:  
2   
5   
18   
81   
101

上述实现中的删除比较复杂。有一种简单的替代操作，称为懒惰删除(lazy deletion)。在懒惰删除时，我们并不真正从二叉搜索树中删除该节点，而是将该节点标记为“已删除”。这样，我们只用找到元素并标记，就可以完成删除元素了。如果有相同的元素重新插入，我们可以将该节点找到，并取消删除标记。

懒惰删除的实现比较简单，可以尝试一下。树所占据的内存空间不会因为删除节点而减小。懒惰节点实际上是用内存空间换取操作的简便性。

### 总结

树, 二叉树, 二叉搜索树

二叉搜索树的删除

懒惰删除

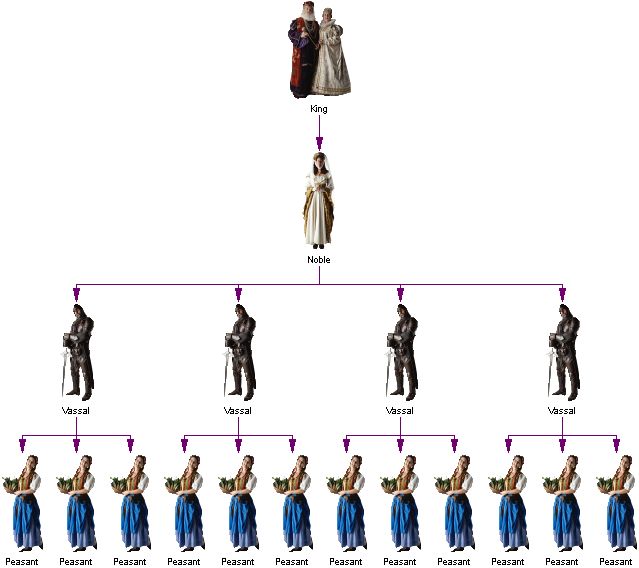
# [纸上谈兵: 堆 (heap)](http://www.cnblogs.com/vamei/archive/2013/03/20/2966612.html)

作者：Vamei 出处：http://www.cnblogs.com/vamei 欢迎转载，也请保留这段声明。谢谢！

堆(heap)又被为优先队列(priority queue)。尽管名为优先队列，但堆并不是队列。回忆一下，在[队列](http://www.cnblogs.com/vamei/archive/2013/03/15/2961729.html)中，我们可以进行的限定操作是dequeue和enqueue。dequeue是按照进入队列的先后顺序来取出元素。而在堆中，我们不是按照元素进入队列的先后顺序取出元素的，而是按照元素的优先级取出元素。

这就好像候机的时候，无论谁先到达候机厅，总是头等舱的乘客先登机，然后是商务舱的乘客，最后是经济舱的乘客。每个乘客都有头等舱、商务舱、经济舱三种个键值(key)中的一个。头等舱->商务舱->经济舱依次享有从高到低的优先级。

再比如，封建社会的等级制度，也是一个堆。在这个堆中，国王、贵族、骑士和农民是从高到低的优先级。



封建等级

Linux内核中的调度器(scheduler)会按照各个进程的优先级来安排CPU执行哪一个进程。计算机中通常有多个进程，每个进程有不同的优先级(该优先级的计算会综合多个因素，比如进程所需要耗费的时间，进程已经等待的时间，用户的优先级，用户设定的进程优先程度等等)。内核会找到优先级最高的进程，并执行。如果有优先级更高的进程被提交，那么调度器会转而安排该进程运行。优先级比较低的进程则会等待。“堆”是实现调度器的理想数据结构。

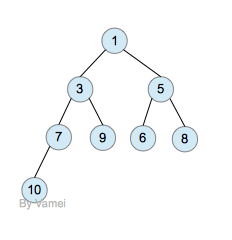
(Linux中可以使用nice命令来影响进程的优先级)

### 

### 堆的实现

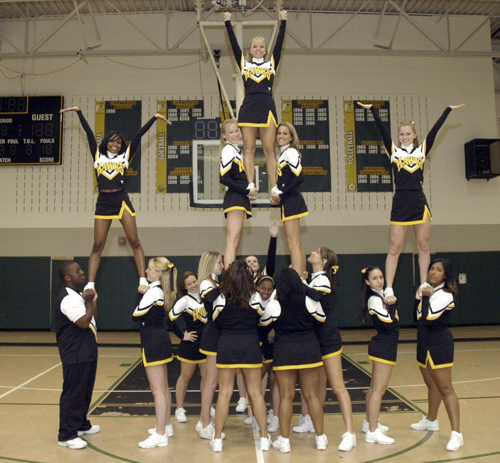
堆的一个经典的实现是完全二叉树(complete binary tree)。这样实现的堆成为二叉堆(binary heap)。

完全二叉树是增加了限定条件的二叉树。假设一个二叉树的深度为n。为了满足完全二叉树的要求，该二叉树的前n-1层必须填满，第n层也必须按照从左到右的顺序被填满，比如下图:



为了实现堆的操作，我们额外增加一个要求: 任意节点的优先级不小于它的子节点。如果在上图中，设定小的元素值享有高的优先级，那么上图就符合该要求。

这类似于“叠罗汉”。叠罗汉最重要的一点，就是让体重大的参与者站在最下面，让体重小的参与者站在上面 (体重小，优先级高)。为了让“堆”稳固，我们每次只允许最上面的参与者退出堆。也就是，每次取出的优先级最高的元素。

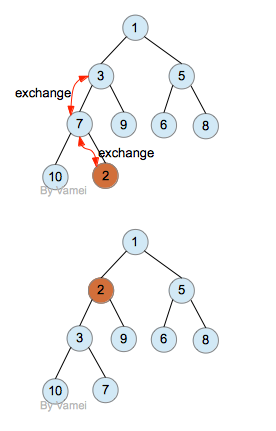


三个“叠罗汉”堆

我已经在[排序算法简介及其C实现](http://www.cnblogs.com/vamei/archive/2013/03/12/2948847.html)中实际使用了堆。堆的主要操作是插入和删除最小元素(元素值本身为优先级键值，小元素享有高优先级)。在插入或者删除操作之后，我们必须保持该实现应有的性质: 1. 完全二叉树 2. 每个节点值都小于或等于它的子节点。

在插入操作的时候，会破坏上述堆的性质，所以需要进行名为percolate\_up的操作，以进行恢复。新插入的节点new放在完全二叉树最后的位置，再和父节点比较。如果new节点比父节点小，那么交换两者。交换之后，继续和新的父节点比较…… 直到new节点不比父节点小，或者new节点成为根节点。这样得到的树，就恢复了堆的性质。

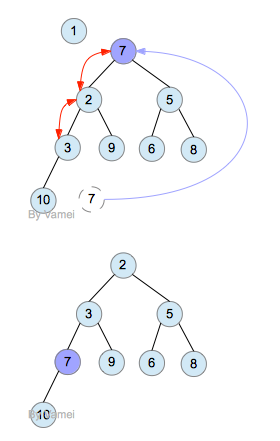
我们插入节点2:



插入

删除操作只能删除根节点。根节点删除后，我们会有两个子树，我们需要基于它们重构堆。进行percolate\_down的操作: 让最后一个节点last成为新的节点，从而构成一个新的二叉树。再将last节点不断的和子节点比较。如果last节点比两个子节点中小的那一个大，则和该子节点交换。直到last节点不大于任一子节点都小，或者last节点成为叶节点。

删除根节点1。如图:



删除根节点

下面是代码。与我们在[二叉搜索树](http://www.cnblogs.com/vamei/archive/2013/03/17/2962290.html)中使用表不同，我们这里使用数组来表示完全二叉树。数组下标为0的元素不用于储存节点，而用于记录完全二叉树中元素的总数。

[复制代码](javascript:void(0);)

/\* By Vamei

Use an big array to implement heap

DECLARE: int heap[MAXSIZE] in calling function

heap[0] : total nodes in the heap

for a node i, its children are i\*2 and i\*2+1 (if exists)

its parent is i/2 \*/

void insert(int new, int heap[])

{

int childIdx, parentIdx;

heap[0] = heap[0] + 1;

heap[heap[0]] = new;

/\* recover heap property \*/

percolate\_up(heap);

}

static void percolate\_up(int heap[]) {

int lightIdx, parentIdx;

lightIdx = heap[0];

parentIdx = lightIdx/2;

/\* lightIdx is root? && swap? \*/

while((parentIdx > 0) && (heap[lightIdx] < heap[parentIdx])) {

/\* swap \*/

swap(heap + lightIdx, heap + parentIdx);

lightIdx = parentIdx;

parentIdx = lightIdx/2;

}

}

int delete\_min(int heap[])

{

int min;

if (heap[0] < 1) {

/\* delete element from an empty heap \*/

printf("Error: delete\_min from an empty heap.");

exit(1);

}

/\* delete root

move the last leaf to the root \*/

min = heap[1];

swap(heap + 1, heap + heap[0]);

heap[0] -= 1;

/\* recover heap property \*/

percolate\_down(heap);

return min;

}

static void percolate\_down(int heap[]) {

int heavyIdx;

int childIdx1, childIdx2, minIdx;

int sign; /\* state variable, 1: swap; 0: no swap \*/

heavyIdx = 1;

do {

sign = 0;

childIdx1 = heavyIdx\*2;

childIdx2 = childIdx1 + 1;

if (childIdx1 > heap[0]) {

/\* both children are null \*/

break;

}

else if (childIdx2 > heap[0]) {

/\* right children is null \*/

minIdx = childIdx1;

}

else {

minIdx = (heap[childIdx1] < heap[childIdx2]) ?

childIdx1 : childIdx2;

}

if (heap[heavyIdx] > heap[minIdx]) {

/\* swap with child \*/

swap(heap + heavyIdx, heap + minIdx);

heavyIdx = minIdx;

sign = 1;

}

} while(sign == 1);

}

[复制代码](javascript:void(0);)

你可以尝试一下构建自己的main函数，测试相关的操作。

### 总结

堆，优先级

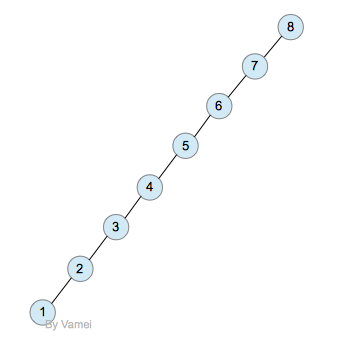
插入元素，删除最大优先级元素

# [纸上谈兵: AVL树](http://www.cnblogs.com/vamei/archive/2013/03/21/2964092.html)

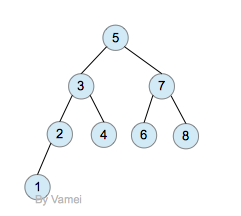
作者：Vamei 出处：http://www.cnblogs.com/vamei 欢迎转载，也请保留这段声明。谢谢！

### 二叉搜索树的深度与搜索效率

我们在[树, 二叉树, 二叉搜索树](http://www.cnblogs.com/vamei/archive/2013/03/17/2962290.html)中提到，一个有n个节点的二叉树，它的最小深度为log(n)，最大深度为n。比如下面两个二叉树:



深度为n的二叉树



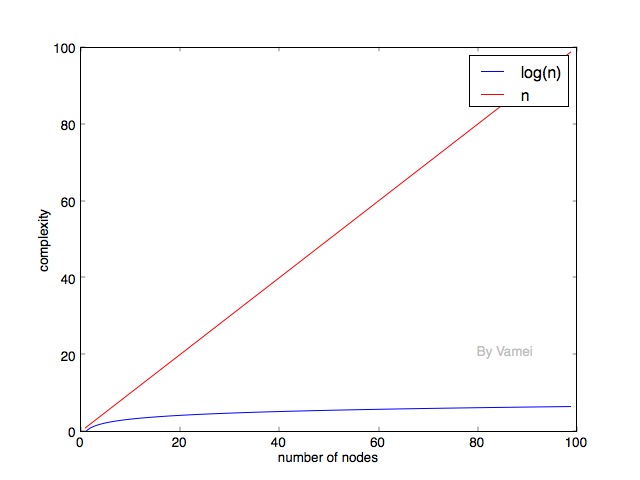
深度为log(n)的二叉树

这两个二叉树同时也是二叉搜索树(参考[树, 二叉树, 二叉搜索树](http://www.cnblogs.com/vamei/archive/2013/03/17/2962290.html))。注意，log以2为基底。log(n)是指深度的量级。根据我们对深度的定义，精确的最小深度为floor(log(n)+1)。

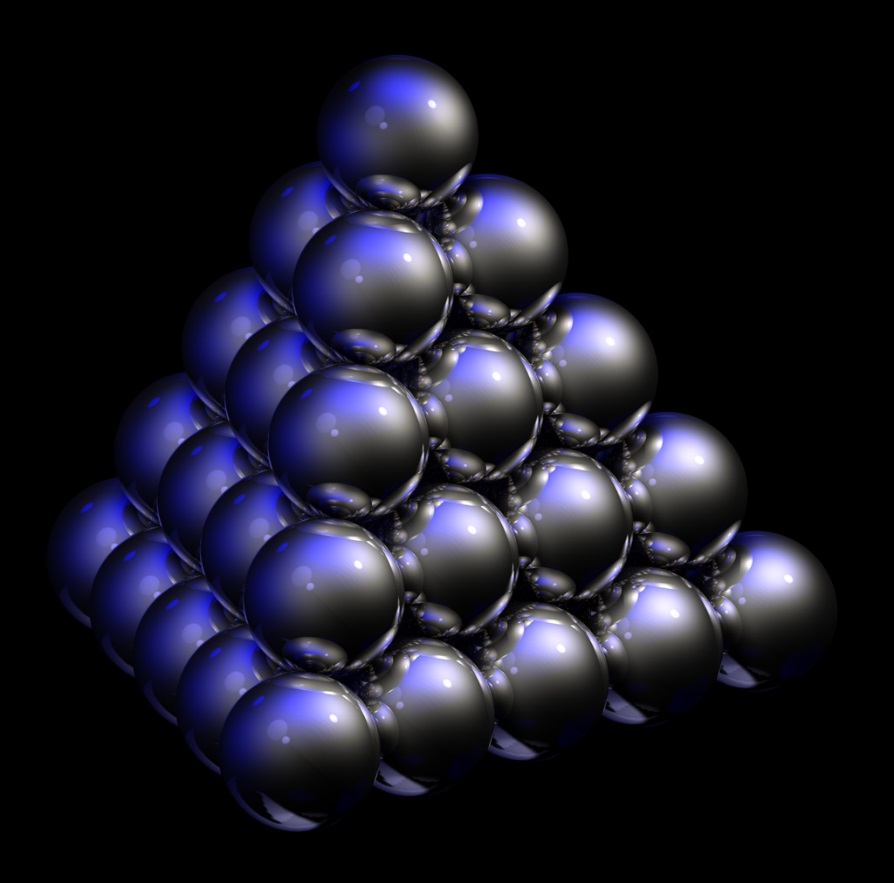
我们将处于同一深度的节点归为一层。如果除最后一层外的其他层都被节点填满时，二叉树有最小深度log(n)。

二叉搜索树的深度越小，那么搜索所需要的运算时间越小。一个深度为log(n)的二叉搜索树，搜索算法的时间复杂度也是log(n)。然而，我们在[二叉搜索树](http://www.cnblogs.com/vamei/archive/2013/03/17/2962290.html)中已经实现的插入和删除操作并不能让保持log(n)的深度。如果我们按照8，7，6，5，4，3，2，1的顺序插入节点，那么就是一个深度为n的二叉树。那么，搜索算法的时间复杂度为n。

n和log(n)的时间复杂度意味着什么呢？时间复杂度代表了完成算法所需要的运算次数。时间复杂度越小，算法的速度越快。



可以看到，随着元素的增加，log(n)的时间复杂度的增长要远小于n。所以，我们自然希望二叉搜索树能尽可能保持log(n)的深度。在上面深度为n的例子中，我们发现，每个节点只有左节点被填满。树的每一层都有很多空位。能不能尽可能减少每一层的空位呢？ (相应的，减少树的深度)



“紧致”的树

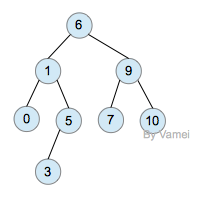
一种想法是先填满一层，再去填充下一层，这样就是一个完全二叉树(complete binary tree)。这样的二叉树实现插入算法会比较复杂。我们将介绍一种思路相似，但比较容易实现的树状数据结构——AVL树。

### AVL树

AVL树是根据它的发明者G. M. Adelson-Velskii和E. M. Landis命名的。它是一种特殊的二叉搜索树。AVL树要求: 任一节点的左子树深度和右子树深度相差不超过1

(空树的深度为0。注意，有的教材中，采用了不同的深度定义方法，所以空树的深度为-1)

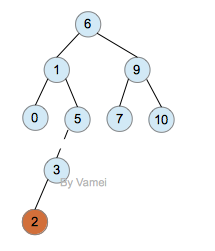
下面是AVL树:



AVL树

AVL树的特性让二叉搜索树的节点实现平衡(balance)：节点相对均匀分布，而不是偏向某一侧。因此，AVL树的搜索算法复杂度是log(n)的量级。

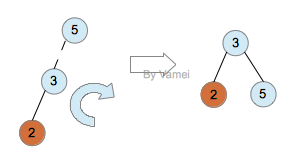
我们在[二叉搜索树](http://www.cnblogs.com/vamei/archive/2013/03/17/2962290.html)中定义的操作，除了插入，都可以用在AVL树上 (假设使用懒惰删除)。如果进行插入操作，有可能会破坏AVL树的性质，比如:



插入2: 破坏AVL树

观察节点5，它的左子树深度为2，右子树深度为0，所以左右两个子树深度相差为2，不再是AVL树。由于2的加入，从节点6，1，5，3到2的层数都增加1。6, 1, 5节点的AVL性质都被破坏。如果从节点2向上回溯，节点5是第一个被破坏的。从节点3开始的子树深度加1，这是造成6, 1, 5的AVL性质被破坏的本质原因。我们将5和3之间的路径画成虚线(就好像挂了重物，边被拉断一样)。

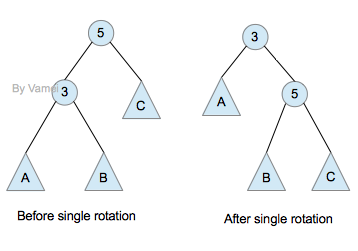
我们可以通过单旋照(single rotation)，调整以5为根节点的子树，来修正因为插入一个元素而引起的对AVL性质的破坏。如下:



Single rotation: 左侧超重，向右转

通过单旋转，3成为新的根节点，2,5称为3的左右子节点。子树重新成为AVL树。该子树的深度减小1，这将自动修正2带给节点6,1的“超负荷”。

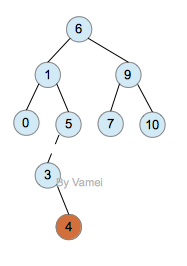
单旋转效果如下：

 向右单旋转

特别要注意的是，为了保持二叉树的性质，子树B过继给了节点5。

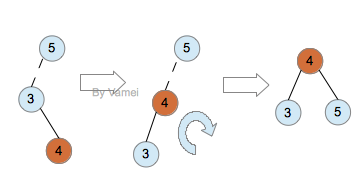
向左单旋转与之类似。作为练习，可以尝试绘制向左单旋转的示意图。

但如果插入的节点不是2，而是4，会是如何呢？



插入4

尝试单旋转，会发现无法解决问题。以5为根节点的子树向右单旋转后，树将以3为根节点，4,5为子节点。4比3大，却是3的左子节点，显然，这依然不符合二叉搜索树的性质。但基于和上面相似的原则(调整以5为根节点的树)，我们发现有一个简单的解决方式：

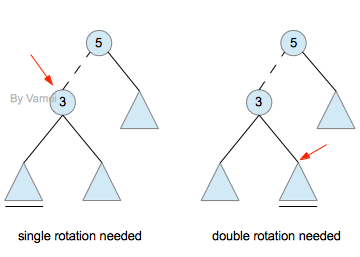


double rotation

上面的操作被称作双旋转(double rotation)。双旋转实际上是进行两次单旋转: 4为根节点的子树先进行一次向左的单旋转，然后将5为根节点的子树进行了一次向右的单旋转。这样恢复了树的ACL性质。

对于AVL树，可以证明，在新增一个节点时，总可以通过一次旋转恢复AVL树的性质。

当我们插入一个新的节点时，在哪里旋转？是用单旋转还是双旋转?



我们按照如下基本步骤进行：

1. 按照二叉搜索树的方式增加节点，新增节点称为一个叶节点。

2. 从新增节点开始，回溯到第一个失衡节点(5)。

    (如果回溯到根节点，还没有失衡节点，就说明该树已经符合AVL性质。)

3. 找到断的边(5->3)，并确定断弦的方向(5的左侧)

4. 以断边下端(3)为根节点，确定两个子树中的哪一个深度大(左子树还是右子树)。

    (这两棵子树的深度不可能相等，而且深度大的子树包含有新增节点。想想为什么)

5. 如果第2和第3步中的方向一致(都为左或者都为右)，需要单旋转以失衡节点为根节点的子树。

    否则，双旋转以失衡节点为根节点的子树。

下面是AVL树的插入算法实现如下:

[复制代码](javascript:void(0);)

/\* By Vamei \*/

/\* binary search tree \*/

#include <stdio.h>

#include <stdlib.h>

typedef struct node \*position;

typedef int ElementTP;

struct node {

int depth;

position parent;

ElementTP element;

position lchild;

position rchild;

};

/\* pointer => root node of the tree \*/

typedef struct node \*TREE;

position insert\_value(TREE, ElementTP);

int depth(TREE);

static void insert\_node\_to\_nonempty\_tree(TREE, position);

static void update\_root\_depth(TREE);

static TREE recover\_avl(TREE, position);

static int depth\_diff(TREE);

static position insert\_leaf(TREE, ElementTP);

static void insert\_node\_to\_nonempty\_tree(TREE, position);

static TREE left\_single\_rotate(TREE);

static TREE left\_double\_rotate(TREE);

static TREE right\_single\_rotate(TREE);

static TREE right\_double\_rotate(TREE);

void main(void)

{

TREE tr;

position np;

ElementTP element;

tr = NULL;

tr = insert\_value(tr, 18);

tr = insert\_value(tr, 5);

printf("root: %d\n", tr->element);

printf("depth: %d\n", depth(tr));

tr = insert\_value(tr, 2);

printf("root: %d\n", tr->element);

printf("depth: %d\n", depth(tr));

tr = insert\_value(tr, 4);

printf("root: %d\n", tr->element);

printf("depth: %d\n", depth(tr));

printf("root->lchild: %d\n", tr->lchild->element);

tr = insert\_value(tr, 3);

printf("root: %d\n", tr->element);

printf("depth: %d\n", depth(tr));

printf("root->lchild: %d\n", tr->lchild->element);

printf("root->lchild->lchild: %d\n", tr->lchild->lchild->element);

}

/\*

\* insert value

\*

\*/

position insert\_value(TREE tr, ElementTP value)

{

position new;

/\* insert a value to a binary search tree \*/

new = insert\_leaf(tr, value);

update\_root\_depth(new);

if (tr == NULL) {

tr = new;

}

else {

tr = recover\_avl(tr, new);

}

return tr;

}

/\*

\* get the depth of the tree

\* use this function to access depth

\*/

int depth(TREE tr) {

if (tr == NULL) {

return 0;

}

else {

return tr->depth;

}

}

//========================================

// static functions: for internal use

/\*

\* traverse the path from new node to root node

\* make one rotation, recover AVL and stop

\*/

static TREE recover\_avl(TREE tr, position np)

{

int myDiff;

while (np != NULL) {

update\_root\_depth(np);

myDiff = depth\_diff(np);

if (myDiff > 1 || myDiff < -1) {

if (myDiff > 1) {

/\* left rotate needed \*/

if(depth\_diff(np->rchild) > 0) {

np = left\_single\_rotate(np);

}

else {

np = left\_double\_rotate(np);

}

}

if (myDiff < -1) {

if(depth\_diff(np->lchild) < 0) {

np = right\_single\_rotate(np);

}

else {

np = right\_double\_rotate(np);

}

}

/\* if rotation changes root node \*/

if (np->parent == NULL) tr = np;

break;

}

np = np->parent;

}

return tr;

}

/\*

\* difference of rchild->depth and lchild->depth

\*/

static int depth\_diff(TREE tr)

{

if (tr == NULL) {

return 0;

}

else {

return depth(tr->rchild) - depth(tr->lchild);

}

}

/\*

\* left single rotation

\* return the new root

\*/

static TREE left\_single\_rotate(TREE tr)

{

TREE newRoot, parent;

parent = tr->parent;

newRoot = tr->rchild;

/\* detach & attach \*/

if (newRoot->lchild != NULL) newRoot->lchild->parent = tr;

tr->rchild = newRoot->lchild;

update\_root\_depth(tr);

/\* raise new root node \*/

newRoot->lchild = tr;

newRoot->parent = parent;

if (parent != NULL) {

if (parent->lchild == tr) {

parent->lchild = newRoot;

}

else {

parent->rchild = newRoot;

}

}

tr->parent = newRoot;

update\_root\_depth(newRoot);

return newRoot;

}

/\*

\* right single rotation

\* return the new root

\*/

static TREE right\_single\_rotate(TREE tr)

{

TREE newRoot, parent;

parent = tr->parent;

newRoot = tr->lchild;

/\* detach & attach \*/

if (newRoot->rchild != NULL) newRoot->rchild->parent = tr;

tr->lchild = newRoot->rchild;

update\_root\_depth(tr);

/\* raise new root node \*/

newRoot->rchild = tr;

newRoot->parent = parent;

if (parent != NULL) {

if (parent->lchild == tr) {

parent->lchild = newRoot;

}

else {

parent->rchild = newRoot;

}

}

tr->parent = newRoot;

update\_root\_depth(newRoot);

return newRoot;

}

/\*

\* left double rotation

\* return

\*/

static TREE left\_double\_rotate(TREE tr)

{

right\_single\_rotate(tr->rchild);

return left\_single\_rotate(tr);

}

/\*

\* right double rotation

\* return

\*/

static TREE right\_double\_rotate(TREE tr)

{

left\_single\_rotate(tr->lchild);

return right\_single\_rotate(tr);

}

/\*

\* update tr->depth

\* assume lchild->depth and rchild->depth are correct

\*/

static void update\_root\_depth(TREE tr)

{

int maxChildDepth;

int depLChild, depRChild;

if (tr==NULL) return;

else {

depLChild = depth(tr->lchild);

depRChild = depth(tr->rchild);

maxChildDepth = depLChild > depRChild ? depLChild : depRChild;

tr->depth = maxChildDepth + 1;

}

}

/\*

\* insert a new value into the tree as a leaf

\* return address of the new node

\*/

static position insert\_leaf(TREE tr, ElementTP value)

{

position np;

/\* prepare the node \*/

np = (position) malloc(sizeof(struct node));

np->element = value;

np->parent = NULL;

np->lchild = NULL;

np->rchild = NULL;

if (tr != NULL) {

insert\_node\_to\_nonempty\_tree(tr, np);

}

return np;

}

/\*

\* insert a node to a non-empty tree

\* called by insert\_value()

\*/

static void insert\_node\_to\_nonempty\_tree(TREE tr, position np)

{

/\* insert the node \*/

if(np->element <= tr->element) {

if (tr->lchild == NULL) {

/\* then tr->lchild is the proper place \*/

tr->lchild = np;

np->parent = tr;

return;

}

else {

insert\_node\_to\_nonempty\_tree(tr->lchild, np);

}

}

else if(np->element > tr->element) {

if (tr->rchild == NULL) {

tr->rchild = np;

np->parent = tr;

return;

}

else {

insert\_node\_to\_nonempty\_tree(tr->rchild, np);

}

}

}

[复制代码](javascript:void(0);)

输出如下：

root: 18  
depth: 2  
  
root: 5  
depth: 2

root: 5  
depth: 3  
root->lchild: 2  
  
depth: 3  
root->lchild: 3  
root->lchild->lchild: 2

(空行是额外加的)

总结:

AVL树: 平衡，深度相差不超过1

单旋转，双旋转

# [纸上谈兵: 伸展树 (splay tree)](http://www.cnblogs.com/vamei/archive/2013/03/24/2976545.html)

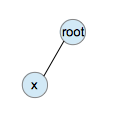
作者：Vamei 出处：http://www.cnblogs.com/vamei 欢迎转载，也请保留这段声明。谢谢！

我们讨论过，树的搜索效率与树的深度有关。二叉搜索树的深度可能为n，这种情况下，每次搜索的复杂度为n的量级。AVL树通过动态平衡树的深度，单次搜索的复杂度为log(n) (以上参考[纸上谈兵 AVL树](http://www.cnblogs.com/vamei/archive/2013/03/21/2964092.html))。我们下面看伸展树(splay tree)，它对于m次连续搜索操作有很好的效率。

伸展树会在一次搜索后，对树进行一些特殊的操作。这些操作的理念与AVL树有些类似，即通过旋转，来改变树节点的分布，并减小树的深度。但伸展树并没有AVL的平衡要求，任意节点的左右子树可以相差任意深度。与二叉搜索树类似，伸展树的单次搜索也可能需要n次操作。但伸展树可以保证，m次的连续搜索操作的复杂度为mlog(n)的量级，而不是mn量级。

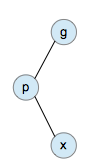
具体来说，在查询到目标节点后，伸展树会不断进行下面三种操作中的一个，直到目标节点成为根节点 （注意，祖父节点是指父节点的父节点）

1. zig: 当目标节点是根节点的左子节点或右子节点时，进行一次单旋转，将目标节点调整到根节点的位置。



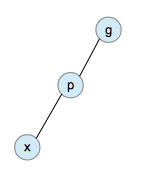
zig

2. zig-zag: 当目标节点、父节点和祖父节点成"zig-zag"构型时，进行一次双旋转，将目标节点调整到祖父节点的位置。



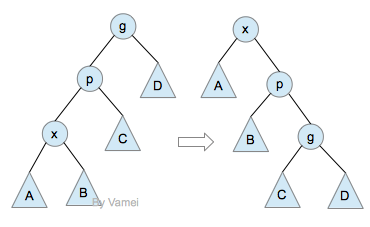
zig-zag

3. zig-zig：当目标节点、父节点和祖父节点成"zig-zig"构型时，进行一次zig-zig操作，将目标节点调整到祖父节点的位置。



zig-zig

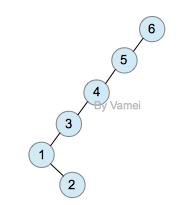
单旋转操作和双旋转操作见[AVL树](http://www.cnblogs.com/vamei/archive/2013/03/21/2964092.html)。下面是zig-zig操作的示意图:



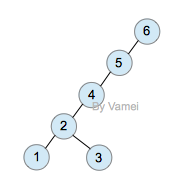
zig-zig operation

在伸展树中，zig-zig操作(基本上)取代了AVL树中的单旋转。通常来说，如果上面的树是失衡的，那么A、B子树很可能深度比较大。相对于单旋转(想一下单旋转的效果)，zig-zig可以将A、B子树放在比较高的位置，从而减小树总的深度。

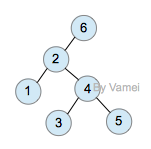
下面我们用一个具体的例子示范。我们将从树中搜索节点2：



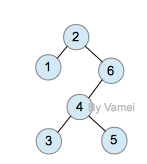
Original



zig-zag (double rotation)



zig-zig



zig (single rotation at root)

上面的第一次查询需要n次操作。然而经过一次查询后，2节点成为了根节点，树的深度大减小。整体上看，树的大部分节点深度都减小。此后对各个节点的查询将更有效率。

伸展树的另一个好处是将最近搜索的节点放在最容易搜索的根节点的位置。在许多应用环境中，比如网络应用中，某些固定内容会被大量重复访问(比如江南style的MV)。伸展树可以让这种重复搜索以很高的效率完成。

### 伸展树的C实现

[复制代码](javascript:void(0);)

/\* By Vamei \*/

/\* Splay Tree \*/

#include <stdio.h>

#include <stdlib.h>

typedef struct node \*position;

typedef int ElementTP;

struct node {

position parent;

ElementTP element;

position lchild;

position rchild;

};

/\* pointer => root node of the tree \*/

typedef struct node \*TREE;

TREE find\_value(TREE, ElementTP);

position insert\_value(TREE, ElementTP);

static void splay\_tree(TREE, position);

static position search\_value(TREE, ElementTP);

static void with\_grandpa(TREE, position);

static void insert\_node\_to\_nonempty\_tree(TREE, position);

static TREE left\_single\_rotate(TREE);

static TREE left\_double\_rotate(TREE);

static TREE right\_single\_rotate(TREE);

static TREE right\_double\_rotate(TREE);

static TREE left\_zig\_zig(TREE);

static TREE right\_zig\_zig(TREE);

void main(void)

{

TREE tr;

tr = NULL;

tr = insert\_value(tr, 6);

tr = insert\_value(tr, 5);

tr = insert\_value(tr, 4);

tr = insert\_value(tr, 3);

tr = insert\_value(tr, 1);

tr = insert\_value(tr, 2);

tr = find\_value(tr, 2);

printf("%d\n", tr->rchild->lchild->element);

}

/\*

\* insert a value into the tree

\* return root address of the tree

\*/

position insert\_value(TREE tr, ElementTP value)

{

position np;

/\* prepare the node \*/

np = (position) malloc(sizeof(struct node));

np->element = value;

np->parent = NULL;

np->lchild = NULL;

np->rchild = NULL;

if (tr == NULL) tr = np;

else {

insert\_node\_to\_nonempty\_tree(tr, np);

}

return tr;

}

/\*

\*

\* return NUll if not found

\*/

TREE find\_value(TREE tr, ElementTP value)

{

position np;

np = search\_value(tr, value);

if (np != NULL && np != tr) {

splay\_tree(tr, np);

}

return np;

}

/\*

\* splaying the tree after search

\*/

static void splay\_tree(TREE tr, position np)

{

while (tr->lchild != np && tr->rchild != np) {

with\_grandpa(tr, np);

}

if (tr->lchild == np) {

right\_single\_rotate(tr);

}

else if (tr->rchild == np) {

left\_single\_rotate(tr);

}

}

/\*

\* dealing cases with grandparent node

\*/

static void with\_grandpa(TREE tr, position np)

{

position parent, grandPa;

int i,j;

parent = np->parent;

grandPa = parent->parent;

i = (grandPa->lchild == parent) ? -1 : 1;

j = (parent->lchild == np) ? -1 : 1;

if (i == -1 && j == 1) {

right\_double\_rotate(grandPa);

}

else if (i == 1 && j == -1) {

left\_double\_rotate(grandPa);

}

else if (i == -1 && j == -1) {

right\_zig\_zig(grandPa);

}

else {

left\_zig\_zig(grandPa);

}

}

/\*

\* search for value

\*/

static position search\_value(TREE tr, ElementTP value)

{

if (tr == NULL) return NULL;

if (tr->element == value) {

return tr;

}

else if (value < tr->element) {

return search\_value(tr->lchild, value);

}

else {

return search\_value(tr->rchild, value);

}

}

/\*

\* left single rotation

\* return the new root

\*/

static TREE left\_single\_rotate(TREE tr)

{

TREE newRoot, parent;

parent = tr->parent;

newRoot = tr->rchild;

/\* detach & attach \*/

if (newRoot->lchild != NULL) newRoot->lchild->parent = tr;

tr->rchild = newRoot->lchild;

/\* raise new root node \*/

newRoot->lchild = tr;

newRoot->parent = parent;

if (parent != NULL) {

if (parent->lchild == tr) {

parent->lchild = newRoot;

}

else {

parent->rchild = newRoot;

}

}

tr->parent = newRoot;

return newRoot;

}

/\*

\* right single rotation

\* return the new root

\*/

static TREE right\_single\_rotate(TREE tr)

{

TREE newRoot, parent;

parent = tr->parent;

newRoot = tr->lchild;

/\* detach & attach \*/

if (newRoot->rchild != NULL) newRoot->rchild->parent = tr;

tr->lchild = newRoot->rchild;

/\* raise new root node \*/

newRoot->rchild = tr;

newRoot->parent = parent;

if (parent != NULL) {

if (parent->lchild == tr) {

parent->lchild = newRoot;

}

else {

parent->rchild = newRoot;

}

}

tr->parent = newRoot;

return newRoot;

}

/\*

\* left double rotation

\* return

\*/

static TREE left\_double\_rotate(TREE tr)

{

right\_single\_rotate(tr->rchild);

return left\_single\_rotate(tr);

}

/\*

\* right double rotation

\* return

\*/

static TREE right\_double\_rotate(TREE tr)

{

left\_single\_rotate(tr->lchild);

return right\_single\_rotate(tr);

}

/\*

\* insert a node to a non-empty tree

\* called by insert\_value()

\*/

static void insert\_node\_to\_nonempty\_tree(TREE tr, position np)

{

/\* insert the node \*/

if(np->element <= tr->element) {

if (tr->lchild == NULL) {

/\* then tr->lchild is the proper place \*/

tr->lchild = np;

np->parent = tr;

return;

}

else {

insert\_node\_to\_nonempty\_tree(tr->lchild, np);

}

}

else if(np->element > tr->element) {

if (tr->rchild == NULL) {

tr->rchild = np;

np->parent = tr;

return;

}

else {

insert\_node\_to\_nonempty\_tree(tr->rchild, np);

}

}

}

/\*  
 \* right zig-zig operation  
 \*/

static TREE right\_zig\_zig(TREE tr)

{

position parent,middle,newRoot;

parent = tr->parent;

middle = tr->lchild;

newRoot = tr->lchild->lchild;

tr->lchild = middle->rchild;

if (middle->rchild != NULL) middle->rchild->parent = tr;

middle->rchild = tr;

tr->parent = middle;

middle->lchild = newRoot->rchild;

if (newRoot->rchild != NULL) newRoot->rchild->parent = middle;

newRoot->rchild = middle;

middle->parent = newRoot;

newRoot->parent = parent;

if (parent != NULL) {

if (parent->lchild == tr) {

parent->lchild = newRoot;

}

else {

parent->rchild = newRoot;

}

}

return newRoot;

}

/\*  
 \* left zig-zig operation  
 \*/

static TREE left\_zig\_zig(TREE tr)

{

position parent,middle,newRoot;

parent = tr->parent;

middle = tr->rchild;

newRoot = tr->rchild->rchild;

tr->rchild = middle->lchild;

if (middle->lchild != NULL) middle->lchild->parent = tr;

middle->lchild = tr;

tr->parent = middle;

middle->rchild = newRoot->lchild;

if (newRoot->lchild != NULL) newRoot->lchild->parent = middle;

newRoot->lchild = middle;

middle->parent = newRoot;

newRoot->parent = parent;

if (parent != NULL) {

if (parent->rchild == tr) {

parent->rchild = newRoot;

}

else {

parent->lchild = newRoot;

}

}

return newRoot;

}

[复制代码](javascript:void(0);)

运行结果:

4

### 总结

splay tree, m operations: mlog(n)

zig-zig

# [纸上谈兵: 哈希表 (hash table)](http://www.cnblogs.com/vamei/archive/2013/03/24/2970339.html)

作者：Vamei 出处：http://www.cnblogs.com/vamei 欢迎转载，也请保留这段声明。谢谢！

### HASH

哈希表(hash table)是从一个集合A到另一个集合B的映射(mapping)。映射是一种对应关系，而且集合A的某个元素只能对应集合B中的一个元素。但反过来，集合B中的一个元素可能对应多个集合A中的元素。如果B中的元素只能对应A中的一个元素，这样的映射被称为一一映射。这样的对应关系在现实生活中很常见，比如：

A  -> B

人 -> 身份证号

日期 -> 星座

上面两个映射中，人 -> 身份证号是一一映射的关系。在哈希表中，上述对应过程称为hashing。A中元素a对应B中元素b，a被称为键值(key)，b被称为a的hash值(hash value)。



 韦小宝的hash值

映射在数学上相当于一个函数f(x):A->B。比如 f(x) = 3x + 2。哈希表的核心是一个哈希函数(hash function)，这个函数规定了集合A中的元素如何对应到集合B中的元素。比如：

A: 三位整数    hash(x) = x % 10    B: 一位整数

104                               4

876                               6

192                               2

上述对应中，哈希函数表示为hash(x) = x % 10。也就是说，给一个三位数，我们取它的最后一位作为该三位数的hash值。

哈希表在计算机科学中应用广泛。比如：

Ethernet中的FCS：参看[小喇叭开始广播 (以太网与WiFi协议)](http://www.cnblogs.com/vamei/archive/2012/11/27/2790065.html)

IP协议中的checksum：参看[我尽力 (IP协议详解)](http://www.cnblogs.com/vamei/archive/2012/12/02/2796988.html)

git中的hash值：参看[版本管理三国志](http://www.cnblogs.com/vamei/archive/2013/02/21/2918069.html)

上述应用中，我们用一个hash值来代表键值。比如在git中，文件内容为键值，并用SHA算法作为hash function，将文件内容对应为固定长度的字符串(hash值)。如果文件内容发生变化，那么所对应的字符串就会发生变化。git通过比较较短的hash值，就可以知道文件内容是否发生变动。

再比如计算机的登陆密码，一般是一串字符。然而，为了安全起见，计算机不会直接保存该字符串，而是保存该字符串的hash值(使用MD5、SHA或者其他算法作为hash函数)。当用户下次登陆的时候，输入密码字符串。如果该密码字符串的hash值与保存的hash值一致，那么就认为用户输入了正确的密码。这样，就算黑客闯入了数据库中的密码记录，他能看到的也只是密码的hash值。上面所使用的hash函数有很好的单向性：很难从hash值去推测键值。因此，黑客无法获知用户的密码。

(之前有报道多家网站用户密码泄露的时间，就是因为这些网站存储明文密码，而不是hash值，见[多家网站卷入CSDN泄密事件 明文密码成争议焦点](http://www.chinanews.com/it/2011/12-23/3553242.shtml))

注意，hash只要求从A到B的对应为一个映射，它并没有限定该对应关系为一一映射。因此会有这样的可能：两个不同的键值对应同一个hash值。这种情况叫做hash碰撞(hash collision)。比如网络协议中的checksum就可能出现这种状况，即所要校验的内容与原文并不同，但与原文生成的checksum(hash值)相同。再比如，MD5算法常用来计算密码的hash值。已经有实验表明，MD5算法有可能发生碰撞，也就是不同的明文密码生成相同的hash值，这将给系统带来很大的安全漏洞。(参考[hash collision](http://en.wikipedia.org/wiki/MD5#Collision_vulnerabilities)）

### HASH与搜索

hash表被广泛的用于搜索。设定集合A为搜索对象，集合B为存储位置，利用hash函数将搜索对象与存储位置对应起来。这样，我们就可以通过一次hash，将对象所在位置找到。一种常见的情形是，将集合B设定在数组下标。由于数组可以根据数组下标进行随机存取(random access，算法复杂度为1)，所以搜索操作将取决于hash函数的复杂程度。

比如我们以人名(字符串)为键值，以数组下标为hash值。每个数组元素中存储有一个指针，指向记录 (有人名和电话号码)。

下面是一个简单的hash函数:

[复制代码](javascript:void(0);)

#define HASHSIZE 1007

/\* By Vamei  
 \* hash function  
 \*/

int hash(char \*p)

{

int value=0;

while((\*p) != '\0') {

value = value + (int) (\*p); // convert char to int, and sum

p++;

}

return (value % HASHSIZE); // won's exceed HASHSIZE

}

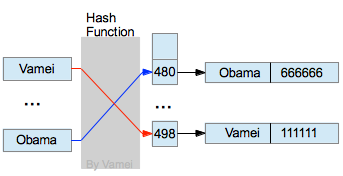
[复制代码](javascript:void(0);)

hash value of "Vamei": 498

hash value of "Obama": 480

我们可以建立一个HASHSIZE大小的数组records，用于储存记录。HASHSIZE被选择为质数，以便hash值能更加均匀的分布。在搜索"Vamei"的记录时，可以经过hash，得到hash值498，再直接读取records[498]，就可以读取记录了。

(666666是Obama的电话号码，111111是Vamei的电话号码。纯属杜撰，请勿当真)



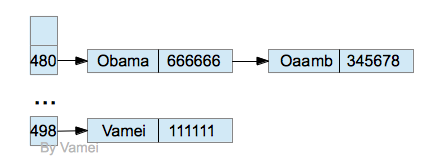
hash搜索

如果不采用hash，而只是在一个数组中搜索的话，我们需要依次访问每个记录，直到找到目标记录，算法复杂度为n。我们可以考虑一下为什么会有这样的差别。数组虽然可以随机读取，但数组下标是随机的，它与元素值没有任何关系，所以我们要逐次访问各个元素。通过hash函数，我们限定了每个下标位置可能存储的元素。这样，我们利用键值和hash函数，就可以具备相当的先验知识，来选择适当的下标进行搜索。在没有hash碰撞的前提下，我们只需要选择一次，就可以保证该下标指向的元素是我们想要的元素。

### 冲突

hash函数需要解决hash冲突的问题。比如，上面的hash函数中，"Obama"和"Oaamb"有相同的hash值，发生冲突。我们如何解决呢？

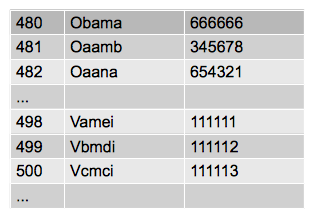
一个方案是将发生冲突的记录用链表储存起来，让hash值指向该链表，这叫做open hashing:



open hashing

我们在搜索的时候，先根据hash值找到链表，再根据key值遍历搜索链表，直到找到记录。我们可以用其他数据结构代替链表。

open hashing需要使用指针。我们有时候想要避免使用指针，以保持随机存储的优势，所以采用closed hashing的方式来解决冲突。



closed hashing

这种情况下，我们将记录放入数组。当有冲突出现的时候，我们将冲突记录放在数组中依然闲置的位置，比如图中Obama被插入后，随后的Oaamb也被hash到480位置。但由于480被占据，Oaamb探测到下一个闲置位置(通过将hash值加1)，并记录。

closed hashing的关键在如何探测下一个位置。上面是将hash值加1。但也可以有其它的方式。概括的说，在第i次的时候，我们应该探测POSITION(i)=(h(x) + f(i)) % HASHSIZE的位置。上面将hash值加1的方式，就相当于设定f(i) = 1。当我们在搜索的时候，就可以利用POSITION(i)，依次探测记录可能出现的位置，直到找到记录。

(f(i)的选择会带来不同的结果，这里不再深入)

如果数组比较满，那么closed hashing需要进行许多次探测才能找到空位。这样将大大减小插入和搜索的效率。这种情况下，需要增大HASHSIZE，并将原来的记录放入到新的比较大的数组中。这样的操作称为rehashing。

### 总结

hash表，搜索

hash冲突, open hashing, closed hashing

# [纸上谈兵: 数学归纳法, 递归, 栈](http://www.cnblogs.com/vamei/archive/2013/03/30/2989930.html)

作者：Vamei 出处：http://www.cnblogs.com/vamei 欢迎转载，也请保留这段声明。谢谢！

### 数学归纳法

数学归纳法(mathematical induction)是一种数学证明方法，常用于证明命题(命题是对某个现象的描述)在自然数范围内成立。随着现代数学的发展，自然数范围内的证明实际上构成了许多其他领域(比如数学分析)的基础，所以数学归纳法对于整个数学体系至关重要。

数学归纳法本身非常简单。如果我们想要证明某个命题对于自然数n都成立，那么:

第一步 证明命题对于n = 1成立。

第二步 假设命题对于n成立，n为任意自然数，证明在此假设下，命题对于n+1成立。

命题得证

想一下上面的两个步骤。它们实际上意味着，命题对于n = 1成立 -> 命题对于n = 2成立 -> 命题对于n = 3成立……直到无穷。因此，命题对于任意自然数都成立。这就好像多米诺骨牌，我们确定n的倒下会导致n + 1的倒下，然后推倒第一块骨牌，就能保证任意骨牌的倒下。



我们来看一下使用数学归纳法来证明高斯求和公式:

http://images.cnitblog.com/blog/413416/201303/30004412-b91c6657684442298605f91d3a296072.png

n为任意自然数。

(这个公式据说是高斯小学时想出来的。老师惩罚全班同学，必须算出1到100的累加，才能回家。于是高斯想出了上面的方法。天才都是被逼出来的么？)

我们的命题是: 高斯求和公式对于任意自然数n都成立。

下面为数学归纳法的证明步骤:

第一步 n = 1，等式左边(1的累加)为1，右边(右边公式代入n=1)也为1，等式两边相等，等式成立，因此命题对于 n = 1 成立。

第二步 假设上述公式对于任意n成立， 即1到n的累加为n\*(n+1)/2

    那么，对于n+1，等式的左边(从1到n+1的累加)等于n\*(n+1)/2 + (n+1)，即(n+1)\*(n+2)/2，

                  等式的右边的n用n+1代替，成为(n+1)\*(n+2)/2，

    等式两边相等，等式成立。因此，当假设命题对于n成立时，命题对于n+1成立。

因此，命题得证。

### 递归

递归(recursion)是计算机中的重要概念，它是指一个计算机程序调用其自身。为了保证计算机不陷入死循环，递归要求程序有一个能够达到的终止条件(base case)。比如下面的程序，是用于计算高斯求和公式:

[复制代码](javascript:void(0);)

/\*

\* Gauss summation

\*/

int f(n)

{

if (n == 1) {

return 1; // base case

}

else {

return f(n-1) + n; // induction

}

}

[复制代码](javascript:void(0);)

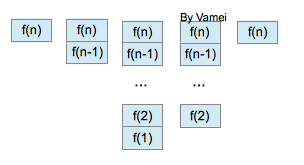
在程序中规定了f(1)的值，以及f(n)和f(n-1)的关系。这正是数学归纳法思想的体现。想要得到f(n)，必须计算f(n-1)；想要f(n-1)，必须计算f(n-2)……直到f(1)。由于我们已经知道了f(1)的值，我们就可以填补前面所有的空缺，最终返回f(n)的值。

递归是数学归纳法在计算机中的程序实现。使用递归设计程序的时候，我们设置base case，并假设我们会获得n-1的结果，并实现n的结果。这就好像数学归纳法，我们只关注初始和衔接，而不需要关注具体的每一步。

### 栈

递归是用[栈(stack)数据结构](http://www.cnblogs.com/vamei/archive/2013/03/14/2960201.html)实现的。正如我们上面所说的，计算f(n)，需要f(n-1)；计算f(n-1)，需要f(n-2)……。我们在寻找到f(1)之前，会有许多空缺: f(n-1)的值什么？ f(n-2)的值是什么? …… f(2)的值是什么？f(1)的值是什么? 我们的第一个问题是f(n)是什么，结果，这个问题引出下一个问题，再下一个问题…… 每个问题的解答都依赖于下一个问题，直到我们找到第一个可以回答的问题: f(1)的值是什么?

我们用栈来保存我们在探索过程中的疑问。C语言中，函数的调用已经是用栈记录离场情境和返回地址。递归是函数对自身的调用，所以很自然的，递归用栈来保存我们的“疑问” 。



我们假设栈向下增长。首先，我们调用f(100)，那么当执行到

return f(n-1) + n;

f(100)暂停执行，并记录当前的状态，比如n的值，当前执行到的位置。随后调用f(99)，栈增加一个frame，直到调用f(98) ... 栈不断增长，直到f(1)。f(1)得到结果1，并返回给f(2)。f(1)栈frame删除，转移到f(2)frame情境中继续执行

return f(n-1) + n;

然后返回给f(3) ... 直到f(99)返回给f(100)，并执行

return f(n-1) + n;

返回f(100)的值，得到结果。

上述过程是C编译器自动完成的。在实现递归算法时，也可以自行手动实现栈。这样可以得到更好的运行效率。

### 总结

数学归纳法

递归

栈

[**纸上谈兵: 左倾堆 (leftist heap)**](http://www.cnblogs.com/vamei/archive/2013/04/19/2978555.html)

作者：Vamei 出处：http://www.cnblogs.com/vamei 欢迎转载，也请保留这段声明。谢谢！

我们之前讲解了[堆(heap)](http://www.cnblogs.com/vamei/archive/2013/03/20/2966612.html)的概念。堆是一个优先队列。每次从堆中取出的元素都是堆中优先级最高的元素。

在之前的文章中，我们基于完全二叉树(complete binary tree)实现了堆，这样的堆叫做二叉堆(binary heap)。binary heap有一个基本要求: 每个节点的优先级大于两个子节点的优先级。在这一要求下，堆的根节点始终是堆的元素中优先级最高的元素。此外，我们实现了delete\_min()操作，从堆中取出元素；insert()操作，向堆中插入元素。

现在，我们考虑下面的问题: 如何合并(merge)两个堆呢？ 一个方案是从第一个堆中不断取出一个元素，并插入到第二个堆中。这样，我们需要量级为n的操作。我们下面要实现更有效率的合并。

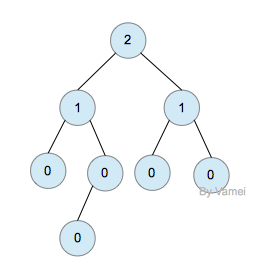
**左倾堆 (Leftist Heap)**

左倾堆基于[二叉树](http://www.cnblogs.com/vamei/archive/2013/03/17/2962290.html)(binary tree)。左倾堆的节点满足堆的基本要求，即(要求1)每个节点的优先级大于子节点的优先级。与[二叉堆](http://www.cnblogs.com/vamei/archive/2013/03/20/2966612.html)不同，左倾堆并不是完全二叉树。二叉堆是非常平衡的树结构，它的每一层都被填满(除了最下面一层)。左倾堆则是维持一种不平衡的结构: 它的左子树节点往往比右子树有更多的节点。



不平衡

左倾堆的每个节点有一个附加信息，即null path length (npl)。npl是从一个节点到一个最近的不满节点的路径长度(不满节点:两个子节点至少有一个为NULL)。一个叶节点的npl为0，一个NULL节点的npl为-1。



各个节点的npl (这里显示的不是元素值)

根据npl的定义，我们有推论1: 一个节点的npl等于子节点npl中最小值加1: npl(node) = min(npl(lchild), npl(rchild)) + 1

有了npl的概念，我们可以完整的定义左倾堆。左倾堆是一个符合下面要求的二叉树:

* 要求1: 每个节点的优先级大于子节点的优先级。
* 要求2: 对于任意节点的左右两个子节点，右子节点的npl不大于左子节点的npl。

**左倾堆的性质**

从上面的要求1和2可以知道，左倾堆的任意子树也是一个左倾堆。

由于左倾堆的特征，左倾堆的右侧路径(right path)较短。右侧路径是指我们从根节点开始，不断前往右子节点所构成的路径。对于一个左倾堆来说，右侧路径上节点数不大于任意其他路径上的节点数，否则，将违反左倾堆的要求2。

我们还可以证明推论2，如果一个左倾堆的右侧路径上有r个节点，那么该左倾堆将至少有2r-1个节点。我们采用归纳法证明:

* r = 1, 右侧路径上有一个节点，所以至少有21-1个节点
* 假设任意r, 左倾堆至少有2r-1节点。那么对于一个右侧路径节点数为r+1的左倾堆来说，根节点的右子树的右侧路径有r个节点。根节点的左子树的右侧路径至少有r个节点。根据假设，该左倾堆将包括:
  + 右子树:至少有2r-1个节点
  + 左子树: 至少有2r-1个节点
  + 1个根节点
* 因此，对于r+1，整个左倾堆至少有2r+1-1个节点。证明完成

换句话说，一个n节点的的左倾堆，它的右侧路径最多有log(n+1)个节点。如果对右侧路径进行操作，其复杂度将是log(n)量级。

我们将沿着右侧路径进行左倾堆的合并操作。合并采用递归。合并如下:

1. (base case) 如果一个空左倾堆与一个非空左倾堆合并，返回非空左倾堆
2. 如果两个左倾堆都非空，那么比较两个根节点。取较小的根节点为新的根节点(满足要求1)，合并较小根节点堆的右子堆与较大根节点堆。
3. 如果右子堆npl > 左子堆npl，互换右子堆与左子堆。
4. 更新根节点的npl = 右子堆npl + 1

上面的合并算法调用了合并操作自身，所以是递归。由于我们沿着右侧路径递归，所以复杂度是log(n)量级。

**左倾堆的实现**

上面可以看到，左倾堆可以相对高效的实现合并(merge)操作。

其他的堆操作，比如insert, delete\_min都可以在merge基础上实现：

* 插入(insert): 将一个单节点左倾堆(新增节点)与一个已有左倾堆合并。
* 删除(delete\_min): 删除根节点，将剩余的左右子堆合并。

[复制代码](javascript:void(0);)

/\* By Vamei \*/

/\*

\* leftist heap

\* bassed on binary tree

\*/

#include <stdio.h>

#include <stdlib.h>

typedef struct node \*position;

typedef int ElementTP;

struct node {

ElementTP element;

int npl;

position lchild;

position rchild;

};

typedef struct node \*LHEAP;

LHEAP insert(ElementTP, LHEAP);

ElementTP find\_min(LHEAP);

LHEAP delete\_min(LHEAP);

LHEAP merge(LHEAP, LHEAP);

static LHEAP merge1(LHEAP, LHEAP);

static LHEAP swap\_children(LHEAP);

int main(void)

{

LHEAP h1=NULL;

LHEAP h2=NULL;

h1 = insert(7, h1);

h1 = insert(3, h1);

h1 = insert(5, h1);

h2 = insert(2, h2);

h2 = insert(4, h2);

h2 = insert(8, h2);

h1 = merge(h1, h2);

printf("minimum: %d\n", find\_min(h1));

return 0;

}

/\*

\* insert:

\* merge a single-node leftist heap with a leftist heap

\* \*/

LHEAP insert(ElementTP value, LHEAP h)

{

LHEAP single;

single = (position) malloc(sizeof(struct node));

// initialze

single->element = value;

single->lchild = NULL;

single->rchild = NULL;

return merge(single, h);

}

/\*

\* find\_min:

\* return root value in the tree

\* \*/

ElementTP find\_min(LHEAP h)

{

if(h != NULL) return h->element;

else exit(1);

}

/\*

\* delete\_min:

\* remove root, then merge two subheaps

\* \*/

LHEAP delete\_min(LHEAP h)

{

LHEAP l,r;

l = h->lchild;

r = h->rchild;

free(h);

return merge(l, r);

}

/\*

\* merge two leftist heaps

\* \*/

LHEAP merge(LHEAP h1, LHEAP h2)

{

// if one heap is null, return the other

if(h1 == NULL) return h2;

if(h2 == NULL) return h1;

// if both are not null

if (h1->element < h2->element) {

return merge1(h1, h2);

}

else {

return merge1(h2, h1);

}

}

// h1->element < h2->element

static LHEAP merge1(LHEAP h1, LHEAP h2)

{

if (h1->lchild == NULL) {

/\* h1 is a single node, npl is 0 \*/

h1->lchild = h2;

/\* rchild is NULL, npl of h1 is still 0 \*/

}

else {

// left is not NULL

// merge h2 to right

// swap if necessary

h1->rchild = merge(h1->rchild, h2);

if(h1->lchild->npl < h1->rchild->npl) {

swap\_children(h1);

}

h1->npl = h1->rchild->npl + 1; // update npl

}

return h1;

}

// swap: keep leftist property

static LHEAP swap\_children(LHEAP h)

{

LHEAP tmp;

tmp = h->lchild;

h->lchild = h->rchild;

h->rchild = tmp;

}

[复制代码](javascript:void(0);)

**总结**

左倾堆利用不平衡的节点分布，让右侧路径保持比较短的状态，从而提高合并的效率。

在合并过程，通过左右互换，来恢复左倾堆的性质。

# [纸上谈兵: 图 (graph)](http://www.cnblogs.com/vamei/p/3113912.html)

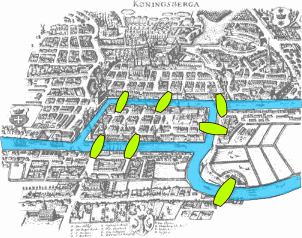
作者：Vamei 出处：http://www.cnblogs.com/vamei 欢迎转载，也请保留这段声明。谢谢！

图(graph)是一种比较松散的数据结构。它有一些节点(vertice)，在某些节点之间，由边(edge)相连。节点的概念在树中也出现过，我们通常在节点中储存数据。边表示两个节点之间的存在关系。在树中，我们用边来表示子节点和父节点的归属关系。树是一种特殊的图，但限制性更强一些。

这样的一种数据结构是很常见的。比如计算机网络，就是由许多节点(计算机或者路由器)以及节点之间的边(网线)构成的。城市的道路系统，也是由节点(路口)和边(道路)构成的图。地铁系统也可以理解为图，地铁站可以认为是节点。基于图有许多经典的算法，比如求图中两个节点的最短路径，求最小伸展树等。

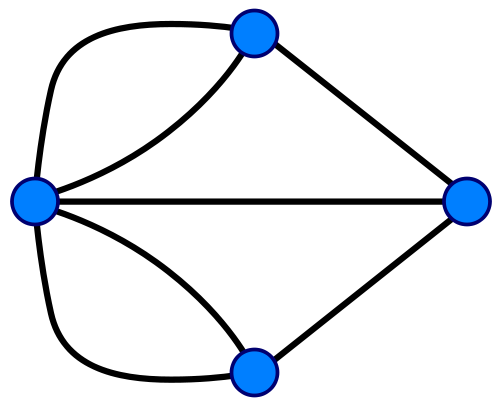
图的经典研究是柯尼斯堡七桥问题(Seven Bridges of Königsberg)。柯尼斯堡是现今的加里宁格勒，城市中有一条河流过，河中有两个小岛。有七座桥桥连接河的两岸和两个小岛。送信员总想知道，有没有一个办法，能不重复的走过7个桥呢？

(这个问题在许多奥数教材中称为"一笔画"问题)



欧拉时代的柯尼斯堡地图

柯尼斯堡的可以看作由7个边和4个节点构成的一个图:



这个问题最终被欧拉巧妙的解决。七桥问题也启发了一门新的数学学科——图论(graph theory)的诞生。欧拉的基本思路是，如果某个节点不是起点或者终点，那么连接它的边的数目必须为偶数个(从一个桥进入，再从另一个桥离开)。对于柯尼斯堡的七桥，由于4个节点都为奇数个桥，而最多只能有两个节点为起点和终点，所以不可能一次走完。

### 图的定义

严格的说，图G=(V,E)是由节点的集合V和边的集合E构成的。一个图的所有节点构成一个集合V。一个边可以表示为(v1,v2)，其中v1,v2∈V，即两个节点。如果(v1,v2)有序，即(v1,v2)与(v2,v1)不同，那么图是有向的(directed)。有序的边可以理解为单行道，只能沿一个方向行进。如果(v1,v2)无序，那么图是无向的(undirected)。无序的边可以理解成双向都可以行进的道路。一个无序的边可以看作连接相同节点的两个反向的有序边，所以无向图可以理解为有向图的一种特殊情况。

(七桥问题中的图是无向的。城市中的公交线路可以是无向的，比如存在单向环线)

图的一个路径(path)是图的一系列节点w1,w2,...,wn，且对于1≤i<n，有(wi,wi+1)∈E。也就是说，路径是一系列的边连接而成，路径的两端为两个节点。路径上边的总数称为路径的长度。乘坐地铁时，我们会在选择某个路径，来从A站到达B站。这样的路径可能有不止一条，我们往往会根据路径的长度以及沿线的拥挤状况，来选择一条最佳的路线。如果存在一条长度大于0的路径，该路径的两端为同一节点，那么认为该图中存在环路(cycle)。很明显，上海的地铁系统中存在环路。



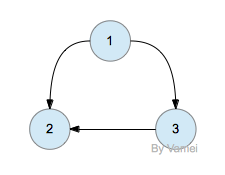
找到一条环路

如果从每个节点，到任意一个其它的节点，都有一条路径的话，那么图是连通的(connected)。对于一个有向图来说，这样的连通称为强连通(strongly connected)。如果一个有向图不满足强连通的条件，但将它的所有边都改为双向的，此时的无向图是连通的，那么认为该有向图是弱连通(weakly connected)。

如果将有火车站的城市认为是节点，铁路是连接城市的边，这样的图可能是不连通的。比如北京和费城，北京有铁路通往上海，费城有铁路通往纽约，但北京和费城之间没有路径相连。

### 图的实现

一种简单的实现图的方法是使用二维数组。让数组a的每一行为一个节点，该行的不同元素表示该节点与其他节点的连接关系。如果(u,v)∈E，那么a[u][v]记为1，否则为0。比如下面的一个包含三个节点的图:

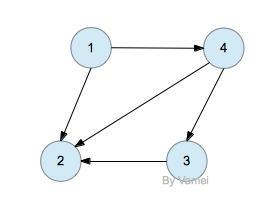


可以简单表示为

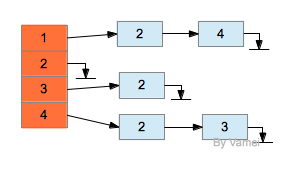
|  |  |  |  |
| --- | --- | --- | --- |
| a | 1 | 2 | 3 |
| 1 | 0 | 1 | 1 |
| 2 | 0 | 0 | 0 |
| 3 | 0 | 1 | 0 |

这种实现方式所占据的空间为O(|V|2)，|V|为节点总数。所需内存随着节点增加而迅速增多。如果边不是很密集，那么很多数组元素记为0，只有稀疏的一些数组元素记为1，所以并不是很经济。

更经济的实现方式是使用邻接表(adjacency list)，即记录每个节点所有的相邻节点。对于节点m，我们建立一个链表。对于任意节点k，如果有(m,k)∈E，就将该节点放入到对应节点m的链表中。邻接表是实现图的标准方式。比如下面的图，



可以用如下的数据结构实现：



左侧为一个数组，每个数组元素代表一个节点，且指向一个链表。该链表包含有该数组元素所有的相邻元素。

总体上看，邻接表可以分为两部分。邻接表所占据的总空间为O(|V|+|E|)。数组部分储存节点信息，占据|V|)的空间，即节点的总数。链表存储边的信息，占据|E|的空间，即边的总数。在一些复杂的问题中，定点和边还可能有其他的附加信息，我们可以将这些附加信息储存在相应的节点或者边的位置。

下面为具体的C代码:

[复制代码](javascript:void(0);)

/\* By Vamei \*/

#include <stdio.h>

#include <stdlib.h>

#define NUM\_V 5

typedef struct node \*position;

/\* node \*/

struct node {

int element;

position next;

};

/\*

\* operations (stereotype)

\*/

void insert\_edge(position, int, int);

void print\_graph(position graph, int nv);

/\* for testing purpose \*/

void main()

{

struct node graph[NUM\_V];

int i;

// initialize the vertices

for(i=1; i<NUM\_V; i++) {

(graph+i)->element = i;

(graph+i)->next = NULL;

}

// insert edges

insert\_edge(graph,1,2);

insert\_edge(graph,1,4);

insert\_edge(graph,3,2);

insert\_edge(graph,4,2);

insert\_edge(graph,4,3);

print\_graph(graph,NUM\_V);

}

/\* print the graph \*/

void print\_graph(position graph, int nv) {

int i;

position p;

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

p = (graph + i)->next;

printf("From %3d: ", i);

while(p != NULL) {

printf("%d->%d; ", i, p->element);

p = p->next;

}

printf("\n");

}

}

/\*

\* insert an edge

\*/

void insert\_edge(position graph,int from, int to)

{

position np;

position nodeAddr;

np = graph + from;

nodeAddr = (position) malloc(sizeof(struct node));

nodeAddr->element = to;

nodeAddr->next = np->next;

np->next = nodeAddr;

}

[复制代码](javascript:void(0);)

运行结果:

From   1: 1->4; 1->2;   
From   2:   
From   3: 3->2;   
From   4: 4->3; 4->2;

上面的实现主要基于链表，可参考[纸上谈兵: 表 (list)](http://www.cnblogs.com/vamei/archive/2013/03/14/2958940.html) 。

### 总结

图是一种很简单的数据结构。图的组织方式比较松散，自由度比较大，但也造成比较高的算法复杂度。我将在以后介绍一些图的经典算法。

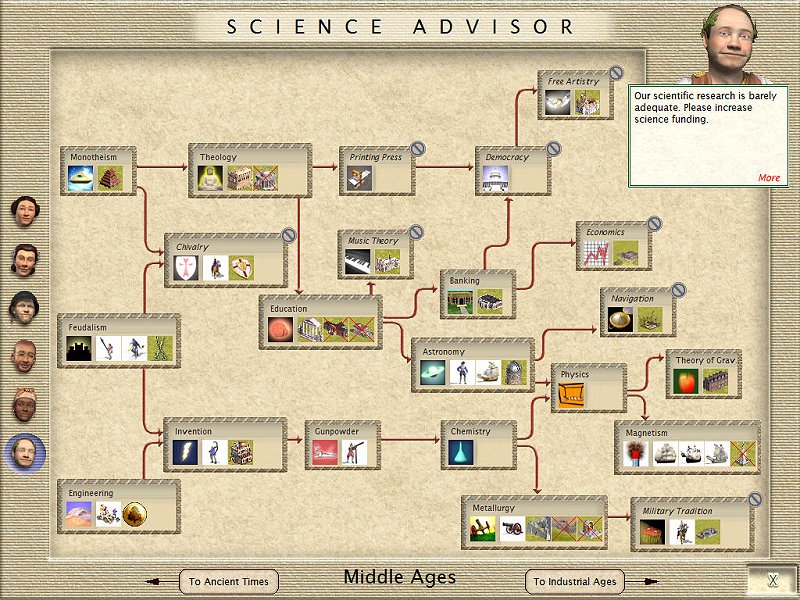
[**纸上谈兵: 拓扑排序**](http://www.cnblogs.com/vamei/p/3232432.html)

作者：Vamei 出处：http://www.cnblogs.com/vamei 欢迎转载，也请保留这段声明。谢谢！

《文明》是一款风靡20多年的回合制策略游戏，由Sid Meier开发。《文明》结构宏大，内容丰富，玩法多样，游戏性强，称得上是历史上最伟大的游戏。在文明中，你可以选择某个文明的，从部落开始发展，在接下来的几千年的历史中，发展科技、开荒拓野、发动战争等等。游戏在保持自由度的前提下，对各个社会文明的发展顺序有很好的仿真性，让玩家仿佛置身于历史长河，坐观文明的起落。美国的一些大学的历史系甚至于使用该游戏作为教学工具。

(作为《文明》的忠实爱好者，多少个昼夜耗费在一张张地图上啊。好吧，是为了学习历史。)

**“科技树”**



《文明》中的科技树

游戏有一个“科技树”系统，即你可以按照该图所示的顺序来发展科技。“科技树”是这样一个概念: 为了研发某个科技，我们必须已经掌握了该科技的所有前提科技(prerequisite)。科技树中有一些箭头，从A科技指向B科技，那么A就是B的前提科技。比如，根据上面的科技树，为了研发物理(Physics)，我们必须已经掌握了化学(Chemistry)和天文学(Astronomy)。而为了研发化学(Chemistry)，我们又必须掌握了火药(Gunpowder)。如果一个科技没有其它科技指向，那么就不需要任何前提科技就可以研发，比如图中的封建主义(Feudalism)。如果同一时间只能研发一项科技，那么玩家的科技研发就是一个序列，比如封建主义，工程(Engineering)，发明(Invention)，火药，一神教(monotheism)，骑士制度(Chivalry)…… 有些序列是不合法的，比如工程，发明，火药……，在研发的“发明”时，并没有满足“封建主义”的前提条件。

一个有趣的问题是，如何找到合法的序列呢？当我们在设计计算机玩家时，很可能需要解决这样一个问题。

图(graph)中的拓扑排序算法(Topological Sort)可以给出一个合法序列。虽然在游戏中被称为“科技树”，但“科技树”并不符合数据结构中的树结构。在数据结构中，树的每个节点只能由一个父节点，整个树只有一个根节点。因此，“科技树”是一个不折不扣的图结构。此外，该树是一个有向的无环(acyclic)图。图中不存在环 (cycle, 环是一个长度大于0的路径，其两端为同一节点)。

**拓扑排序**

拓扑排序利用了一个事实，即在一个无环网络中，有一些节点是没有箭头指入的，比如科技树中的一神教、封建主义、工程。它们可以作为序列的起点。

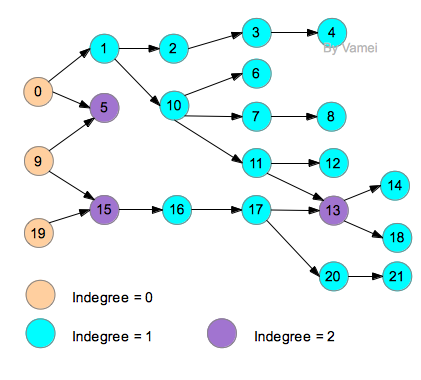
* 选择没有箭头指入的节点中的任一个，可以作为合法序列的起点，放入序列。
* 当我们将某个节点放入序列后，去掉该节点以及从该节点指出的所有箭头。在新的图中，选择没有箭头指向的节点，作为序列中的下一个点。
* 重复上面的过程，直到所有的节点都被放入序列。

对于每个节点，我们可以使用一个量入度(indegree)，用来表示有多少个箭头指入该节点。当某个节点被删除时，图发生变化，我们需要更新图中节点的入度。

为了方便，我将“科技树”中的节点编号，为了符合C语言中的传统，编号从0开始。下面是对应关系:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 一神教 | 神学 | 印刷术 | 民主 | 自由艺术 | 骑士制度 | 音乐理论 | 银行 | 经济学 | 封建主义 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| 教育 | 天文学 | 航海术 | 物理 | 重力理论 | 发明 | 火药 | 化学 | 磁学 | 工程 |
| 20 | 21 |  |  |  |  |  |  |  |  |
| 冶金 | 军事传统 |  |  |  |  |  |  |  |  |

我们根据编号，绘制上述图(graph)。我同时用三种颜色，来表示不同点的入度(indegree):



我们使用邻接表的数据结构(参考[纸上谈兵: 图](http://www.cnblogs.com/vamei/p/3113912.html))，来实现图。构建代码如下:

[复制代码](javascript:void(0);)

/\* By Vamei \*/

#include <stdio.h>

#include <stdlib.h>

#define NUM\_V 22

typedef struct node \*position;

/\* node \*/

struct node {

int element;

position next;

};

/\*

\* operations (stereotype)

\*/

void insert\_edge(position, int, int);

void print\_graph(position graph, int nv);

/\* for testing purpose \*/

void main()

{

struct node graph[NUM\_V];

int i;

// initialize the vertices

for(i=0; i<NUM\_V; i++) {

(graph+i)->element = i;

(graph+i)->next = NULL;

}

// insert edges

insert\_edge(graph,0,1);

insert\_edge(graph,0,5);

insert\_edge(graph,1,2);

insert\_edge(graph,1,10);

insert\_edge(graph,2,3);

insert\_edge(graph,3,4);

insert\_edge(graph,7,8);

insert\_edge(graph,9,5);

insert\_edge(graph,9,15);

insert\_edge(graph,10,6);

insert\_edge(graph,10,7);

insert\_edge(graph,10,11);

insert\_edge(graph,11,12);

insert\_edge(graph,11,13);

insert\_edge(graph,13,14);

insert\_edge(graph,13,18);

insert\_edge(graph,15,16);

insert\_edge(graph,16,17);

insert\_edge(graph,17,13);

insert\_edge(graph,17,20);

insert\_edge(graph,19,15);

insert\_edge(graph,20,21);

print\_graph(graph,NUM\_V);

}

/\* print the graph \*/

void print\_graph(position graph, int nv) {

int i;

position p;

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

p = (graph + i)->next;

printf("From %3d: ", i);

while(p != NULL) {

printf("%d->%d; ", i, p->element);

p = p->next;

}

printf("\n");

}

}

/\*

\* insert an edge

\*/

void insert\_edge(position graph,int from, int to)

{

position np;

position nodeAddr;

np = graph + from;

nodeAddr = (position) malloc(sizeof(struct node));

nodeAddr->element = to;

nodeAddr->next = np->next;

np->next = nodeAddr;

}

[复制代码](javascript:void(0);)

我们可以根据上面建立的图，来获得各个节点的入度。使用一个数组indeg来储存各个节点的入度，即indeg[i] = indgree of ithnode。下面的init\_indeg()用于初始化各节点的入度:

[复制代码](javascript:void(0);)

// according to the graph, initialize the indegree at each vertice

void init\_indeg(position graph, int nv, int indeg[]) {

int i;

position p;

// initialize

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

indeg[i] = 0;

}

// assimilate the graph

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

p = (graph + i)->next;

while(p != NULL) {

(indeg[p->element])++;

p = p->next;

}

}

}

[复制代码](javascript:void(0);)

正如我们之前叙述的，我们需要找到入度为0的节点，并将这些节点放入序列。find\_next()就是用于寻找下一个入度为0的节点。找到对应节点后，返回该节点，并将该节点的入度更新为-1:

[复制代码](javascript:void(0);)

/\* find the vertice with 0 indegree\*/

int find\_next(int indeg[], int nv) {

int next;

int i;

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

if(indeg[i] == 0) break;

}

indeg[i] = -1;

return i;

}

[复制代码](javascript:void(0);)

当我们取出一个节点放入序列时，从该节点出发，指向的节点的入度会减1，我们用update\_indeg()表示该操作:

[复制代码](javascript:void(0);)

// update indeg when ver is removed

void update\_indeg(position graph, int nv, int indeg[], int ver) {

position p;

p = (graph + ver)->next;

while(p != NULL) {

(indeg[p->element])--;

p = p->next;

}

}

[复制代码](javascript:void(0);)

有了上面的准备，我们可以寻找序列。使用数组seq来记录序列中的节点。下面的get\_seq()用于获得拓扑排序序列:

[复制代码](javascript:void(0);)

// return the sequence

void get\_seq(position graph, int nv, int indeg[], int seq[]){

int i;

int ver;

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

ver = find\_next(indeg, nv);

seq[i] = ver;

update\_indeg(graph, nv, indeg, ver);

}

}

[复制代码](javascript:void(0);)

综合上面的叙述，我们可以使用下面代码，来实现拓扑排序:

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

/\* By Vamei \*/

#include <stdio.h>

#include <stdlib.h>

#define NUM\_V 22

typedef struct node \*position;

/\* node \*/

struct node {

int element;

position next;

};

/\*

\* operations (stereotype)

\*/

void insert\_edge(position, int, int);

void print\_graph(position, int);

void init\_indeg(position, int , int \*);

void update\_indeg(position, int, int \*, int);

void get\_seq(position, int, int \*, int \*);

int find\_next(int \*, int);

/\* for testing purpose \*/

void main()

{

struct node graph[NUM\_V];

int indeg[NUM\_V];

int seq[NUM\_V];

int i;

// initialize the graph

for(i=0; i<NUM\_V; i++) {

(graph+i)->element = i;

(graph+i)->next = NULL;

}

insert\_edge(graph,0,1);

insert\_edge(graph,0,5);

insert\_edge(graph,1,2);

insert\_edge(graph,1,10);

insert\_edge(graph,2,3);

insert\_edge(graph,3,4);

insert\_edge(graph,7,8);

insert\_edge(graph,9,5);

insert\_edge(graph,9,15);

insert\_edge(graph,10,6);

insert\_edge(graph,10,7);

insert\_edge(graph,10,11);

insert\_edge(graph,11,12);

insert\_edge(graph,11,13);

insert\_edge(graph,13,14);

insert\_edge(graph,13,18);

insert\_edge(graph,15,16);

insert\_edge(graph,16,17);

insert\_edge(graph,17,13);

insert\_edge(graph,17,20);

insert\_edge(graph,19,15);

insert\_edge(graph,20,21);

print\_graph(graph,NUM\_V);

init\_indeg(graph,NUM\_V,indeg);

get\_seq(graph, NUM\_V, indeg, seq);

for (i=0; i<NUM\_V; i++) {

printf("%d,", seq[i]);

}

}

void print\_graph(position graph, int nv) {

int i;

position p;

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

p = (graph + i)->next;

printf("From %3d: ", i);

while(p != NULL) {

printf("%d->%d; ", i, p->element);

p = p->next;

}

printf("\n");

}

}

/\*

\* insert an edge

\*/

void insert\_edge(position graph,int from, int to)

{

position np;

position nodeAddr;

np = graph + from;

nodeAddr = (position) malloc(sizeof(struct node));

nodeAddr->element = to;

nodeAddr->next = np->next;

np->next = nodeAddr;

}

void init\_indeg(position graph, int nv, int indeg[]) {

int i;

position p;

// initialize

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

indeg[i] = 0;

}

// update

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

p = (graph + i)->next;

while(p != NULL) {

(indeg[p->element])++;

p = p->next;

}

}

}

// update indeg when ver is removed

void update\_indeg(position graph, int nv, int indeg[], int ver) {

position p;

p = (graph + ver)->next;

while(p != NULL) {

(indeg[p->element])--;

p = p->next;

}

}

/\* find the vertice with 0 indegree\*/

int find\_next(int indeg[], int nv) {

int next;

int i;

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

if(indeg[i] == 0) break;

}

indeg[i] = -1;

return i;

}

// return the sequence

void get\_seq(position graph, int nv, int indeg[], int seq[]){

int i;

int ver;

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

ver = find\_next(indeg, nv);

seq[i] = ver;

update\_indeg(graph, nv, indeg, ver);

}

}

[复制代码](javascript:void(0);)

上面算法中有一个遍历所有节点的for循环，而循环中的find\_next()函数也会遍历所有的节点。因此，算法复杂度为O(|V|2)。

在find\_next()中，我们将放入序列的节点入度记为-1。find\_next()每次会遍历所有的节点，没有必要遍历入度为-1的节点。为了改善算法复杂度，我们可以使用一个队列(或者栈)来记录入度为0的元素。我们每次从队列中取出一个元素放入拓扑序列，并更新相邻元素的入度。如果该元素的相邻元素的入度变为0，那么将它们放入队列中。可以证明，这样的改造后，算法复杂度为O(|V|+|E|)

**问题拓展**

通过上面的算法，我们可以获得一个合法的科技发展序列。我随后想到一个问题，如何输出所有的科技发展序列呢？

这个问题的关键在于，某个时刻，可能同时有多个节点的入度同时为0。它们中的任何一个都可以成为下一个元素。为此，我们需要记录所有的可能性。

(我想到的，是复制入度数组和序列数组，以储存不同的可能性。不知道大家有什么更好的想法呢？)

**总结**

图，拓扑排序

# [“不给力啊，老湿！”：RSA加密与破解](http://www.cnblogs.com/vamei/p/3480994.html)

作者：Vamei 出处：http://www.cnblogs.com/vamei 欢迎转载，也请保留这段声明。谢谢！

加密和解密是自古就有技术了。经常看到侦探电影的桥段，勇敢又机智的主角，拿着一长串毫无意义的数字苦恼，忽然灵光一闪，翻出一本厚书，将第一个数字对应页码数，第二个数字对应行数，第三个数字对应那一行的某个词。数字变成了一串非常有意义的话：

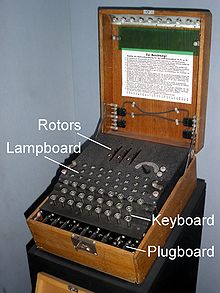
Eat the beancurd with the peanut. Taste like the ham.

主角喜极而泣……

这种加密方法是将原来的某种信息按照某个规律打乱。某种打乱的方式就叫做密钥(cipher code)。发出信息的人根据密钥来给信息加密，而接收信息的人利用相同的密钥，来给信息解密。就好像一个带锁的盒子。发送信息的人将信息放到盒子里，用钥匙锁上。而接受信息的人则用相同的钥匙打开。加密和解密用的是同一个密钥，这种加密称为对称加密（symmetric encryption）。



如果一对一的话，那么两人需要交换一个密钥。一对多的话，比如总部和多个特工的通信，依然可以使用同一套密钥。但这种情况下，对手偷到一个密钥的话，就知道所有交流的信息了。二战中盟军的情报战成果，很多都来自于破获这种对称加密的密钥。



二战中德军的传奇加密机：Enigma

为了更安全，总部需要给每个特工都设计一个不同的密钥。如果是FBI这样庞大的机构，恐怕很难维护这么多的密钥。在现代社会，每个人的信用卡信息都需要加密。一一设计密钥的话，银行怕是要跪了。

对称加密的薄弱之处在于给了太多人的钥匙。如果只给特工锁，而总部保有钥匙，那就容易了。特工将信息用锁锁到盒子里，谁也打不开，除非到总部用唯一的一把钥匙打开。只是这样的话，特工每次出门都要带上许多锁，太容易被识破身份了。总部老大想了想，干脆就把造锁的技术公开了。特工，或者任何其它人，可以就地取材，按照图纸造锁，但无法根据图纸造出钥匙。钥匙只有总部的那一把。



上面的关键是锁和钥匙工艺不同。知道了锁，并不能知道钥匙。这样，银行可以将“造锁”的方法公布给所有用户。每个用户可以用锁来加密自己的信用卡信息。即使被别人窃听到，也不用担心：只有银行才有钥匙呢！这样一种加密算法叫做非对称加密(asymmetric encryption)。非对称加密的经典算法是RSA算法。它来自于数论与计算机计数的奇妙结合。

为了了解RSA加密，请听一个卧底的自白：

### RSA加密

我是潜伏在龙凤大酒楼的卧底。想让下面信息以加密的方式发送到总部：

A CHEF HIDE A BED

厨子藏起来了一张床！这是如此的重要，需要立即通知总部。千万重要的是，不能让反革命的厨子知道。

第一步是转码，也就是将英文转换成某个对应的数字。这个对应很容易建立，比如：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G | H | I |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

将上面的信息转码，获得下面的数字序列：

A CHEF HIDE A BED

1 3856 8945 1 254

这串数字完全没有什么秘密可言。厨子发现了这串数字之后，很容易根据数字顺序，对应字母表猜出来。

为了和狡猾的厨子斗智斗勇，我们需要对这串数字进一步加密。使用总部发给我们的锁，两个数字：3和10。我们分为两步处理。

第一步是求乘方。第一个数字是3，也就是说，总部指示我们，求上面数字串的3次方：

原字符串: 1   3   8   5   6   8   9   4   5   1   2   5   4

三次乘方: 1  27 512 125 216 512 729  64 125   1   8 125  64

第二步是求余数。第二个上锁的数字是10，将上面每个三次乘方除以10，获得其余数：

余数： 1 7 2 5 6 2 9 4 5 1 8 5 4

将这串数字发回总部。中途被厨子偷看到，但一时不能了解其中的意思。如果还是像刚才一样对应字母表的话，信息是：

AGBEFBIDEAHED

这串字母完全不包含正常的单词。

信息到了总部。总部开始用神奇的钥匙来解读。这个钥匙是3。(偷偷告诉你的，别告诉厨子。)

（这里钥匙不小心和之前锁中的一个数字相同。这只是巧合。）

解锁过程也是两步。第一步求钥匙次的乘方，即3次方。第二步求它们除以10（锁之一）的余数。

加密信息：1   7   2   5   6   2   9   4   5   1   8   5   4

三次乘方：1 343   8 125 216   8 729  64 125   1 512 125  64 (这里用的是钥匙的“3”)

除十得余：1   3   8   5   6   8   9   4   5   1   2   5   4

正是我们发送的信息。对应字母表，总部可以立即知道原来的信息。

### 特工练习

再次强调，为了演示方便，选用了简单的锁和钥匙。锁和钥匙只是凑巧相同。为此，我们做一个小练习。

练习：总部新公布出来的锁是2987(次乘方)和3937(为除数)。

1) 作为特工，用上面的算法为信息加密(你可能需要一些编程来计算，尝试用[Python的数学计算功能](http://www.cnblogs.com/vamei/archive/2012/05/29/2524376.html)？)。

猜到钥匙是什么了呢？不是上面两个数字中的任何一个，而是143！

2) 作为值班人员，验证143是钥匙，可以解密信息。

为了简便，你可以只检验一个简单的信息，比如“IE”。

下面是我根据这个练习写的一个Python小程序。这里的转码用的是ASCII编码标准，而不是上面的A对应1，B对应2。

[复制代码](javascript:void(0);)

# By Vamei

#==== Agent ========

# coding covert: string to number

# By ASCII convention

def convert(original):

return map(ord, original)

# the input is a list of integers

def encrypt(input\_list):

f = lambda x: (x\*\*2987)%3937

return map(f, input\_list)

#==== Headquarter =====

# the input is the result of the encrypt function

def decrypt(encrypted\_list):

f = lambda x: (x\*\*143)%3937

return map(f, encrypted\_list)

# convert numbers back to a string

def inv\_convert(decrypt\_list):

f = lambda x: str(unichr(x))

result = map(f, decrypt\_list)

return "".join(result)

# Test

message = "Go to hell!"

secret = encrypt(convert(message))

print(secret)

public = inv\_convert(decrypt(secret))

print(public)

[复制代码](javascript:void(0);)

### 费马与欧拉

发觉自己被愚弄了，厨子很生气，后果很严重。厨子发奋看了书，知道了这个加密方法叫RSA，是三为发明人 R. Rivest, A. Shamir和L. Adelman名字首字母合起来的。RSA算法是1977年发明的。全称是RSA Public Key System。这个"Public Key"是公共密钥，也就是我们上面说的锁。再读下去，厨子大窘。这个1977年的，现代计算机加密的RSA算法，居然源于17世纪。

**1. 费马小定律**

RSA的原理借助了数论中的“欧拉定理”(Euler's theorem)。17世纪的费马首先给出一个该定理的特殊形式，即“费马小定理”：

p是一个正的质数，a是任意一个不能被p整除的整数。那么，ap−1−1能被p整除。

我们并不需要太深入了解费马小定理，因为等下就会看到这个定理的“升级版”。但这个定理依然很美妙，它优美的得到乘方和整除的某种特殊关系。使用一个例子来说明它。比如p=7，a=3。那么费马小定律表示，37−1−1可以被7整除。

事实上，上面的数字计算得到36−1=728，它确实可以被7整除。

练习：尝试一个其它的例子，比如p=5，a=4，验证费马小定律是否成立。

\*\*\* 数学小贴士：

1) 除 (divide)，商和余数：两个整数相除，有一个为整数的商，和一个余数。比如10/3=3,余1。我们用一个特别的方式记录这一叙述:

10≡1(mod3)

也可以写成另一种方式：

[10]3=[1]3

这一表述方式与“10除以3，得3余1”这样的方式并没有什么区别。但采用标准的数学方式更容易和别人交流。

如果我们知道：

[a]n=[b]n

那么存在某个整数t，且：

a=nt+b

2) 整除 (divisible)：当一个整数a除以另一个整数b，余数为0时，那么我们说a可以被b整除。比如说，4可以被2整除。即

[4]2=[0]2

3) 质数 (prime number)：一个质数是只能被±1和这个数自身整除的整数（不包括±1）。比如2，3，5，7，11，13等等。

\*\*\*\*\*\*

费马是一名律师，也是一名业余数学家。他对数学贡献很大，堪称“业余数学家之王”。比如他和帕斯卡的通信算是概率论的开端。还有“费马大定理”，或者称为“费马猜想”。费马有在书边写注释的习惯。他在页边角写下了费马猜想，并说：

我发现了一个美妙的证明，但由于空白太小而没有写下来。

费马自己的证明没有再被发现。“费马猜想”的证明是300多年后，以现代数学为工具证得的，而这些数学工具在费马的时代是不存在的。这导致现代的数学家怀疑费马是不是在吹牛。费马小定理是费马的另一个定理。在费马那里，也还是个猜想。证明要等到欧拉。



程序员们：注释要完整啊！

**2. 欧拉定律**

时间流过一百年。欧拉是18世纪的瑞典数学家。这位数学巨人写了75本数学专著，几乎把当时所有的数学领域都征服了一遍。欧拉后来被叶卡捷琳娜二世邀请到俄国。据说，无神论者狄徳罗造访俄国，他宣称上帝并不存在，靠雄辩击败了整个俄国宫廷。欧拉曾醉心神学，对上帝很虔诚。欧拉看不下去了，上前说，“先生，eiπ+1=0，所以上帝存在。请回答！” 狄徳罗败给这个问题，灰溜溜的走了。

（这个传说的可信度不高，因为狄徳罗本人也是一位颇有造诣的数学家。）



欧拉定理(Euler's theorem)是欧拉在证明费马小定理的过程中，发现的一个适用性更广的定理。

首先定义一个函数，叫做欧拉Phi函数，即ϕ(n)，其中，n是一个正整数。

ϕ(n)=总数(从1到n−1，与n互质的整数)

比如5，那么1，2，3，4，都与5互质。与5互质的数有4个。ϕ(5)=4

再比如6，与1，5互质，与2，3，4并不互质。因此，ϕ(6)=2

对于一个质数p来说，它和1, 2, 3, ..., p - 1都互质，所以ϕ(p)=p−1。比如ϕ(7)=6,ϕ(11)=10

\*\*\* “互质”的数学小贴士：

1) 因子 (factor)：每个整数都可以写成质数相乘的形式，每个这样的质数称为该整数的一个因子。

2) 互质 (relative prime)：如果两个整数没有公共因子，这两个质数互质。

\*\*\*\*\*\*

欧拉定理叙述如下：

如果n是一个正整数，a是任意一个非0整数，且n和a互质。那么，aϕ(n)−1可以被n整除。  (1)

由于质数p有ϕ(p)=p−1。因此，从欧拉定理可以推出费马小定理。我们可以只使用欧拉定理，把费马小定理抛到脑后了。我们用一个例子简单的检验欧拉定理。如果n是6，那么ϕ(6)=2。让a是11，和6互质。112−1为120，确实可以被n，也就是6整除，符合欧拉定理。

数学中还有一个关于Phi函数的推论：

m和n是互质的正整数。那么，ϕ(mn)=ϕ(m)ϕ(n)        (2)

### ****RSA西游记****

下面我们要进入实质的证明。除了上面的(1)和(2)推论，还需要提前说明一个问题，即：

[ab]n=[a]n[b]n        (3)

证明：假设a和b除以n的余数为c1,c2。a和b可以写成a=nt1+c1,b=nt2+c2。那么，ab=n2t1t2+nt1c2+nt2c1+c1c2。因此ab除以n的余数为c1c2。即[ab]n=[a]n[b]n。

根据此可以推论，[am]n=[a]mn。

演一出叫做“西游记”的大戏，选角开始：

先选择两个质数p和q，分别是沙和尚和白龙马。让n=pq，n是唐僧。一路向西，唐僧靠的是沙和尚和白龙马出力：一个背行李，一个驮人。

而k=ϕ(n)=(p−1)(q−1)。这里使用了(2)以及“质数p的Phi函数值为p-1”。k是八戒，也就是Phi(唐僧)，就是唐僧的一个跟屁虫。

选择任意d，并保证它与k互质。d是观音。观音姐姐在高老庄，真的是把八戒给“质”了一把。

取整数e，使得[de]k=[1]k。也就是说de=kt+1，t为某一整数。e是悟空，横行无忌。

我们记得公开的用来上锁的两个数字，它们分别是悟空e和唐僧n。悟空威力大，负责乘方。唐僧太唠叨：一切妖怪见到它，就变成了余数。悟空和唐僧合作，就把世界搞乱了。

总部的观音姐姐d看不下去了。观音姐姐威力也大，也是乘方。再逼着唐僧重新唠叨。世界就恢复了。

善哉，善哉！



我们看一下这一魔幻大片“西游记”的现实主义原理。根据欧拉定理(1)，对于任意z，如果z与n互质，那么：

[zϕ(n)]n=[zk]n=[1]n

因此，

[zde]n=[zkt+1]n=[(zk)tz]n=[z]n

上面主要使用了de=kt+1以及(3)。也就是说：

[zde]n=[z]n

根据(3)的推论，有

([ze]n)d=[z]n

妖怪z，经过e和d的各一道，又变回了妖！上面过程中，悟空e和观音d忙得不亦乐乎，唐僧n就在一旁边唠叨边打酱油了。

这一等式，也正是我们加密又解密的过程 (加密: 悟空次方 + 唐僧唠叨。解密: 观音次方 + 唐僧唠叨)。悟空和唐僧是公钥，扔出去亮相。观音是私钥，偷偷藏起来，必要的时候才出来。

(上面都默认余数是最小正余数，也就是说，10除以3的余数为1，而不是4。尽管4也可以算是10的余数，即[4]3=[10]3。)



姐姐，饶了我吧。

3和8两个妖怪见到唐僧5，都被唠叨成了余数3。这样就观音姐姐就算法力无边，还是没法还原。为了让唐僧求余的时候，不会把数字弄混了，RSA算法要求所有妖怪z小于唐僧n。为了对足够多的字符转码加密，n必须大过最大的妖怪。

但唐僧n大更重要的原因是要保护马仔。想破解，必须找到观音。回顾我们选择角色的过程。我们可以这样破解：唐僧n是公开的，1) 先找到它的隐藏手下沙和尚和白龙马。2) 沙和尚和白龙马知道了，那么二师兄k就保不住了。3) de = kt + 1，即找到一个e，可以让de - 1被k整除。观音姐姐就找到了。

上面的整个破解过程中，最困难的是第一步，即找到两个隐藏的打手。通常，p和q都会选的非常大，比如说200位。这导致唐僧n也非常大，有400位。寻找一个400位数字的质数分解并不容易，我们要做的除法运算次数大约为10400−−−−√/2。这是10199次除法运算！天河2号每秒浮点运算是1016级别。那么，找到隐藏打手的工作，大约需要10174年……。这个活，看来只能佛祖干了。

练习 如果唐僧不够大的话，马仔就危险了。想想之前的厨子，知道悟空是3，唐僧是10。隐藏打手是谁？ 八戒呢？ 观音呢？

总之，带头大哥不够“罩”的话，团伙就要被一窝端了。



### 总结

正如我在“[数学与编程](http://www.cnblogs.com/vamei/p/3472301.html#2838197)”中提到的，数学可以是程序员军火库中有力的武器。加密、解密这一事关IT安全的大课题，却和数论这一纯粹数学学科发生奇妙的关系。RSA算法的数学基础在于欧拉定理。这一诞生了几百年没有什么实用性的数学理论，却在网络时代，找到自己的栖身之处。

RSA算法是非对称算法。公开的加密方式，私有的解密方式。RSA安全的关键在于很难对一个大的整数进行因子分解。下一次，如果看到RSA被破解之类的消息，卧底必须大喊一声：“不给力呀，老湿！”

这篇文章已经充分的准备了数学，但(1), (2), (3)还可以深挖下去。如果这篇文章收获够多“赞”，就准备写一个续篇。讨论一下欧拉定理的证明，以及一些特殊情况下的RSA破解。

[**纸上谈兵: 最短路径与贪婪**](http://www.cnblogs.com/vamei/p/3604629.html)

作者：Vamei 出处：http://www.cnblogs.com/vamei 欢迎转载，也请保留这段声明。谢谢!

[图](http://www.cnblogs.com/vamei/p/3113912.html)是由节点和连接节点的边构成的。节点之间可以由路径，即边的序列。根据路径，可以从一点到达另一点。在一个复杂的图中，图中两点可以存在许多路径。最短路径讨论了一个非常简单的图论问题，图中从A点到B点 ，那条路径耗费最短？



这个问题又异常复杂，因为网络的构成状况可能很复杂。

一个最简单的思路，是找出所有可能的从A到B的路径，再通过比较，来寻找最短路径。然而，这并没有将问题简化多少。因为搜索从A到B的路径，这本身就是很复杂的事情。而我们在搜索所有路径的过程中，有许多路径已经绕了很远，完全没有搜索的必要。比如从上海到纽约的路线，完全没有必要先从上海飞到南极，再从南极飞到纽约，尽管这一路径也是一条可行的路径。

所以，我们需要这样一个算法：它可以搜索路径，并当已知路径包括最短路径时，即停止搜索。我们先以无权网络为例，看一个可行的最短路径算法。

**无权网络**

无权网络(unweighted network)是相对于加权网络的，这里的“权”是权重。每条边的耗费相同，都为1。路径的总耗费即为路径上边的总数。

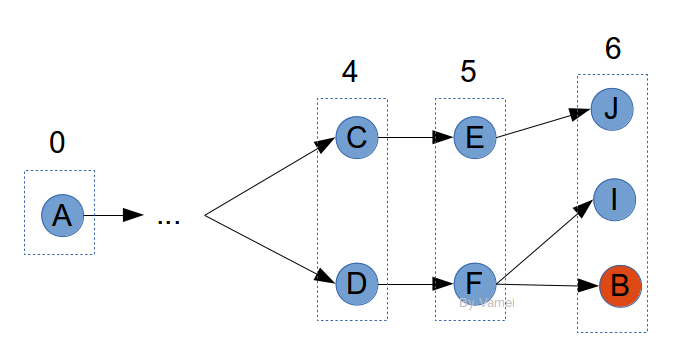
我们用“甩鞭子”的方式，来寻找最短路径。鞭子的长度代表路径的距离。



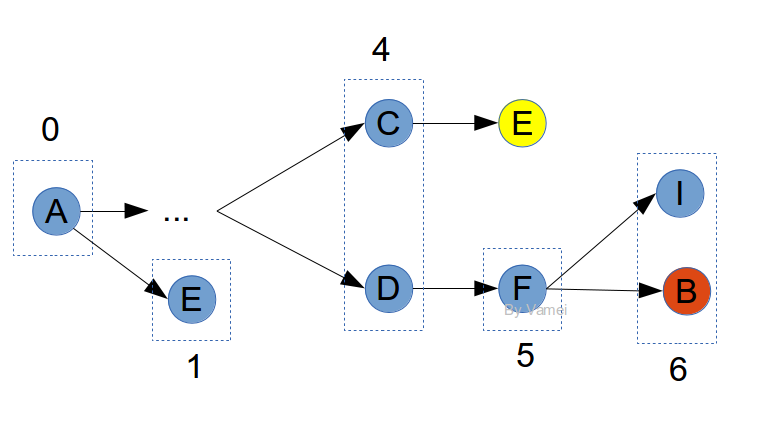
手拿一个特定长度的鞭子，站在A点。甩出鞭子，能打到一些点。比如C和D。

将鞭子的长度增加1。再甩出鞭子。此时，从C或D出发，寻找距离为1的邻接点，即E和F。这些点到A点的距离，为此时鞭子的长度。

记录点E和F，并记录它们的上游节点。比如E(C)， F(D)。我们同样可以记录此时该点到A的距离，比如5。



如果要记录节点E时，发现它已经出现在之前的记录中，这说明曾经有更短的距离到E。此时，不将E放入记录中。毕竟，我们感兴趣的是最短路径。如下图中的E：



黄色的E不被记录

最初的鞭子长度为0，站在A点，只能打到A点自身。当我们不断增加鞭子长度，第一次可以打到B时，那么此时鞭子的长度，就是从A到B的最短距离。循着我们的记录，倒推上游的节点，就可以找出整个最短路径。

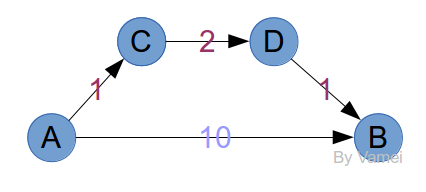
我们的记录本是个很有意思的东西。某个点放入记录时，此时的距离，都是A点到该点的最短路径。根据记录，我们可以反推出记录中任何一点的最短路径。这就好像真诚对待每个人。这能保证，当你遇到真爱时，你已经是在真诚相待了。实际上，记录将所有节点分割成两个世界：记录内的，已知最短距离的；记录外的，未知的。

**加权网络**

在加权网络中(weighted network)，每条边有各自的权重。当我们选择某个路径时，总耗费为路径上所有边的权重之和。

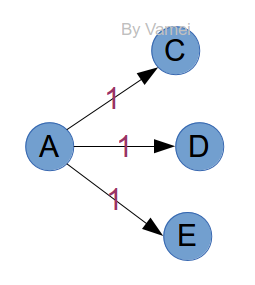
加权网络在生活中很常见，比如从北京到上海，可以坐火车，也可以坐飞机。但两种选择耗费的时间并不同。再比如，我们打出租车和坐公交车，都可以到市区，但车资也有所不同。在计算机网络中，由于硬件性能不同，连接的传输速度也有所差异。加权网络正适用于以上场景。无权网络是加权网络的一个特例。

这个问题看起来和无权网络颇为类似。但如果套用上面的方法，我们会发现，记录中的节点并不一定是最短距离。我们看下面的例子：

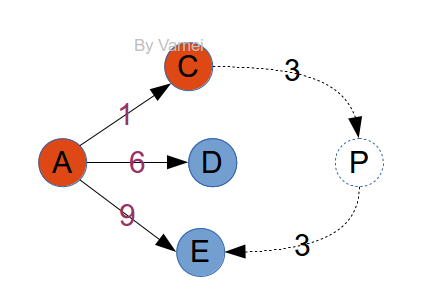


很明显，最短路径是A->C->D->B，因为它的总耗费只有4。按照上面的方法，我们先将A放入记录。从A出发，有B和C两个如果将B和C同时放入记录，那么记录中的B并不符合最短距离的要求。

那么，为什么无权网络可行呢？假设某次记录时，鞭子长度为5，那么这次记录点的邻接点，必然是距离为6的点。如果这些邻接点没有出现过，那么6就是它们的最短距离。所有第一次出现的邻接点，都将加入到下次的记录中。比如下面的例子，C/D/E是到达A的邻接点，它们到A的最短距离必然都是1。



对于加权网络来说，即使知道了邻接点，也无法判断它们是否符合最短距离。在记录C/D/E时，我们无法判断未来是否存在如下图虚线的连接，导致A的邻接点E并不是下一步的最短距离点：



但情况并没有我们想的那么糟糕。仔细观察，我们发现，虽然无法一次判定所有的邻接点为下一步的最短距离点，但我们可以确定点C已经处在从A出发的最短距离状态。A到C的其它可能性，比如途径D和E，必然导致更大的成本。

也就是说，邻接点中，有一个达到了最短距离点，即邻接点中，到达A距离最短的点，比如上面的C。我们可以安全的把C改为已知点。A和C都是已知点，点P成为新的邻接点。P到A得距离为4。

出于上面的观察，我们可以将节点分为三种：

* 已知点：已知到达A最短距离的点。“我是成功人士。”
* 邻接点：有从记录点出发的边，直接相邻的点。“和成功人士接触，也有成功的机会哦。”
* 未知点：“还早得很。”

最初的已知点只有A。已知点的直接下游节点为邻接点。对于邻接点，我们需要独立的记录它们。我们要记录的有：

* 当前情况下，从A点出发到达该邻接点的最短距离。比如对于上面的点D，为6。
* 此最短距离下的上游节点。对于上面的点D来说，为A。

每次，我们将邻接点中最短距离最小的点X转为已知点，并将该点的直接下游节点，改为邻接点。我们需要计算从A出发，经由X，到达这些新增邻接点的距离：新距离 = X最短距离 + QX边的权重。此时有两种情况，

* 如果下游节点Q还不是邻接点，那么直接加入，Q最短距离 = 新距离，Q上游节点为X。
* 如果下游节点Q已经是邻接点，记录在册的上游节点为Y，最短距离为y。如果新距离小于y，那么最小距离改为新距离，上游节点也改为X。否则保持原记录不变。

我们还用上面的图，探索A到E的路径：

第一步

|  |  |  |  |
| --- | --- | --- | --- |
|  | 状态 | 已知距离 | 上游 |
| A | 已知 | 0 | A |
| C | 邻接 | 1 | A |
| D | 邻接 | 6 | A |
| E | 邻接 | 9 | A |
| P | 未知 | 无穷 |  |

第二步

|  |  |  |  |
| --- | --- | --- | --- |
|  | 状态 | 已知距离 | 上游 |
| A | 已知 | 0 | A |
| C | 已知 | 1 | A |
| D | 邻接 | 6 | A |
| E | 邻接 | 9 | A |
| P | 邻接 | 4 | C |

第二步

|  |  |  |  |
| --- | --- | --- | --- |
|  | 状态 | 已知距离 | 上游 |
| A | 已知 | 0 | A |
| C | 已知 | 1 | A |
| D | 邻接 | 6 | A |
| E | 邻接 | 7 | P |
| P | 已知 | 4 | C |

第三步

|  |  |  |  |
| --- | --- | --- | --- |
|  | 状态 | 已知距离 | 上游 |
| A | 已知 | 0 | A |
| C | 已知 | 1 | A |
| D | 已知 | 6 | A |
| E | 邻接 | 7 | P |
| P | 已知 | 4 | C |

最后，E成为已知。倒退，可以知道路径为E, P, C, A。正过来，就是从A到E的最短路径了。

上面的算法是经典的Dijkstra算法。本质上，每个邻接点记录的，是基于已知点的情况下，最好的选择，也就是所谓的“贪婪算法”(greedy algorithm)。当我们贪婪时，我们的决定是临时的，并没有做出最终的决定。转换某个点成为已知点后，我们增加了新的可能性，贪婪再次起作用。根据对比。随后，某个邻接点成为新的“贪无可贪”的点，即经由其它任意邻接点，到达该点都只会造成更高的成本； 经由未知点到达该点更不可能，因为未知点还没有开放，必然需要经过现有的邻接点到达，只会更加绕远。好吧，该点再也没有贪婪的动力，就被扔到“成功人士”里，成为已知点。成功学不断传染，最后感染到目标节点B，我们就找到了B的最短路径。

**实现**

理解了上面的原理，算法的实现是小菜一碟。我们借用[图 (graph)](http://www.cnblogs.com/vamei/p/3113912.html)中的数据结构，略微修改，构建加权图。

我们将上面的表格做成数组records，用于记录路径探索的信息。

重新给点A,C,D,E,P命名，为0, 1, 2, 3, 4。

代码如下：

[复制代码](javascript:void(0);)

/\* By Vamei \*/

#include <stdio.h>

#include <stdlib.h>

#define NUM\_V 5

#define INFINITY 10000

typedef struct node \*position;

typedef struct record \*label;

/\* node \*/

struct node {

int element;

position next;

int weight;

};

/\* table element, keep record \*/

struct record {

int status;

int distance;

int previous;

};

/\*

\* operations (stereotype)

\*/

void insert\_edge(position, int, int, int);

void print\_graph(position, int);

int new\_neighbors(position, label, int, int);

void shortest\_path(position, label, int, int, int);

/\* for testing purpose \*/

void main()

{

struct node graph[NUM\_V];

struct record records[NUM\_V];

int i;

// initialize the vertices

for(i=0; i<NUM\_V; i++) {

(graph+i)->element = i;

(graph+i)->next = NULL;

(graph+i)->weight = -1;

}

// insert edges

insert\_edge(graph,0,1,1);

insert\_edge(graph,0,2,6);

insert\_edge(graph,0,3,9);

insert\_edge(graph,1,4,3);

insert\_edge(graph,4,3,3);

print\_graph(graph,NUM\_V);

// initialize the book

for (i=0; i<NUM\_V; i++){

(records+i)->status = -1;

(records+i)->distance = INFINITY;

(records+i)->previous = -1;

}

shortest\_path(graph, records, NUM\_V, 0, 3);

//

}

void shortest\_path(position graph, label records, int nv, int start, int end) {

int current;

(records+start)->status = 1;

(records+start)->distance = 0;

(records+start)->previous = 0;

current = start;

while(current != end) {

current = new\_neighbors(graph, records, nv, current);

}

while(current != start) {

printf("%d <- ", current);

current = (records+current)->previous;

}

printf("%d\n", current);

}

int new\_neighbors(position graph, label records, int nv, int current) {

int newDist;

int v;

int i;

int d;

position p;

// update the current known

(records + current)->status = 1;

// UPDATE new neighbors

p = (graph+current)->next;

while(p != NULL) {

v = p->element;

(records + v)->status = 0;

newDist = p->weight + (records + current)->distance;

if ((records + v)->distance > newDist) {

(records + v)->distance = newDist;

(records + v)->previous = current;

}

p = p->next;

}

// find the next known

d = INFINITY;

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

if ((records + i)->status==0 && (records + i)->distance < d){

d = (records + i)->distance;

v = i;

}

}

return v;

}

/\* print the graph \*/

void print\_graph(position graph, int nv) {

int i;

position p;

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

p = (graph + i)->next;

printf("From %3d: ", i);

while(p != NULL) {

printf("%d->%d; w:%d ", i, p->element, p->weight);

p = p->next;

}

printf("\n");

}

}

/\*

\* insert an edge

\* with weight

\*/

void insert\_edge(position graph,int from, int to, int weight)

{

position np;

position nodeAddr;

np = graph + from;

nodeAddr = (position) malloc(sizeof(struct node));

nodeAddr->element = to;

nodeAddr->next = np->next;

nodeAddr->weight = weight;

np->next = nodeAddr;

}

[复制代码](javascript:void(0);)

运行结果如下：

From   0: 0->3; w:9 0->2; w:6 0->1; w:1

From   1: 1->4; w:3

From   2:

From   3:

From   4: 4->3; w:3

3 <- 4 <- 1 <- 0

即从0到1到4到3，也就是从A到C到P到E，是我们的最短路径。

上面的算法中，最坏情况是目标节点最后成为已知点，即要寻找O(|V|)。而每个已知点是通过寻找O(|V|)个节点的最小值得到的。最后，打印出最短的路径过程中，需要倒退，最多可能有O|E|，也就是说，算法复杂度为O(|V|2+|E|)。

上面的records为一个数组，用于记录路径探索信息。我们可以用一个优先队列来代替它，将已知的节点移除优先队列。这样可以达到更好的运算效率。

练习: 自行设计一个加权网络，寻找最短路径。

**总结**

最短路径是寻找最优解的算法。在复杂的网络中，简单的实现方式无法运行，必须求助于精心设计的算法，比如这里的Dijkstra算法。利用贪婪的思想，我们不断的优化结果，直到找到最优解。