标注：蓝色字体是红色字体下的一个小概念。

时间复杂度

Algorithm（给人看的，可以用伪代码pseudocode表示） 不等于 Program（给机器运行的）

分析algorithms的两大元素:

Correctness

Efficiency: running time(time complexity)

Worst-case running time: the maximum number of steps the algorithm takes over all possible inputs

Upper Bound (of an Algorithm):

The number of operations sufficient to solve a problem by an algorithm

The worst-case complexity of an algorithm

Good algorithms :small upper bounds(上界越小越好)

Lower Bound (of a Problem):

The worst-case number of operations necessary to solve the problem by any algorithmLarger lower bounds is better (in terms of analysis)//相较于上界，下界越大越好

Upper bound = lower bound—>optimal algorithms

The Big-O Notation(用于计算时间复杂度,上界函数)：

f(n) = O(g(n)) ,if f(n) <=c g(n) for all n>= n0 for some constants c, n0

f(n) grows slower or same as g(n)

Consecutive statements: Complexity of {S1; S2;} = max(complexity of S1, complexity

of S2)

Loops: Complexity = complexity of statements inside the loop \*number of times the loop is executed

If-then-else: Complexity of if (A) then B else C = max(complexity of A+B,complexity of A+C)

Function calls:Complexity of function calls should be analysed separately from the calling routine

Big-Omega（下界函数）:

f(n) = Ω(g(n)) if f(n)>=c g(n) ,for all >= n0, for some constants c, n0

Big-Theta（Optimal最优函数）:

f(n) = Θ(g(n)) if f(n) = O(g(n)) and f(n) =Ω(g(n))

数据结构

Algorithms + Data Structures = Programs

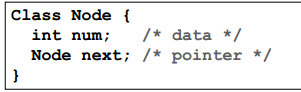
Common data structure:

1.array数组：Direct access – A[i] in O(1) time

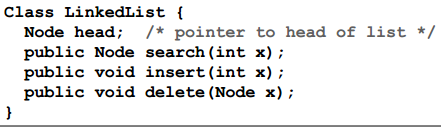
2.linked list链表：



一个链表中结点的声明：

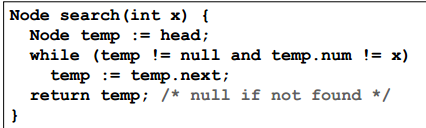


一个链表的声明（包含链表头，三个操作——搜索，插入，删除，这三个操作具体实现见下）：

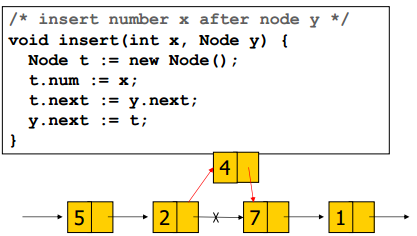


由此可见，链表不同于数组，是不可以直接访问的（direct access）

搜索：



插入（在y结点之后插入t）：



删除：



双向链表Doubly linked list：除了指向下一个结点的指针之后，还增加了一个指向上一个结点previous的指针

循环链表Circular linked list:最后一个结点node的next指针指向head链表头

3.stack栈

last-in-first-out (LIFO)

Push(v)压栈:将元素v压入栈中/

pop()弹栈：将栈顶的元素弹出栈外

Top()返回栈顶的元素

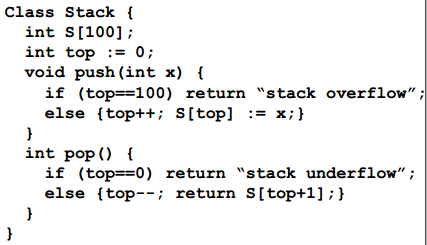
4.queue队列

first-in-first-out (FIFO)Enqueue(v): insert v into queue Q

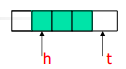
Dequeue(): return and remove first element from Q

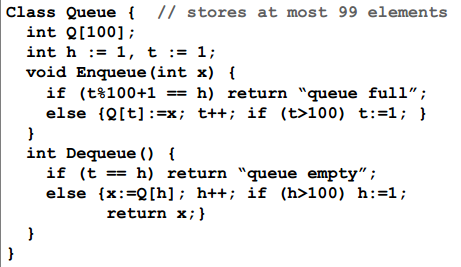
3&4都是一种抽象的数据结构，而1&2是具体的数据结构。3&4可以由1&2实现。

e.g实现一个栈stack 



实现一个队列：



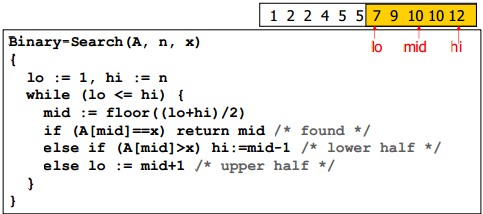


排序和搜索

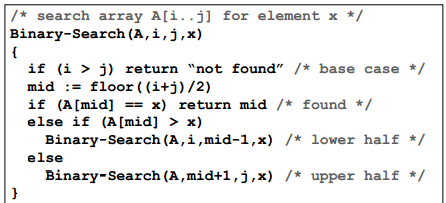
二分查找Binary search：

在一个有序的数列中查找一个数，与中间数进行比较，根据比较大小，选择数的前半部分或者后半部分作为新的查找数列，进入下一次查找。这将需要查找的数列一步一步地划分成小数列，也就是将一个大的任务划分成一个个的子任务，符合递归调用的思想。

非递归：



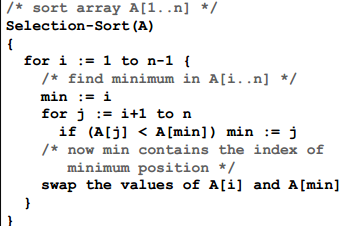
递归：递归的出口是所查找的数列为空（i>j）或者查找到的时候(A[mid]==x)



时间复杂度是O(log n)

选择排序Selection Sort：

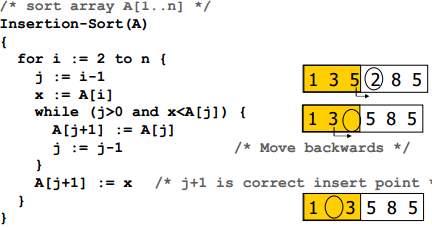
每次循环只做一次交换，选择出最值



时间复杂度：O(n^2)

插入排序insert sort：

每次循环，将下一个待排的数插入到已经排好的有序数列之中



Benefits from partially-sorted input

时间复杂度：O(n^2)

Binary insertion sort: 插入排序的升级版，在插入待排序的数时候，进行二分查找，而不是顺序查找。比较次数是O(nlogn)，但是时间复杂度是 O(n ^2)，主要是移动数据的原因

递归经典例子：汉诺塔

T(n)=2T(n-1)+1



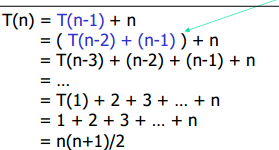
当我们需要吧A柱上n个盘子放到C柱时，我们可以先将n-1个盘子放到B柱上，再将A柱最底部的盘子放到C柱子上，再将n-1个盘子从B柱放到C柱。因此，将n个从A柱到C柱盘子的任务分解成n-1个盘子从A到B到C的子任务和一个盘子到从A到C的子任务。所以，递推公式是：T(n)=2T(n-1)+1

计算递归函数的三种方法：

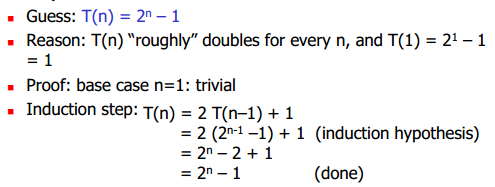
1.interation迭代法

递推到最初可知的值

e.g

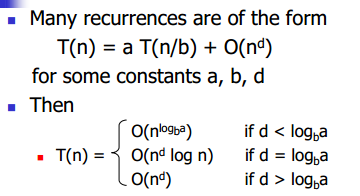


2.induction归纳法

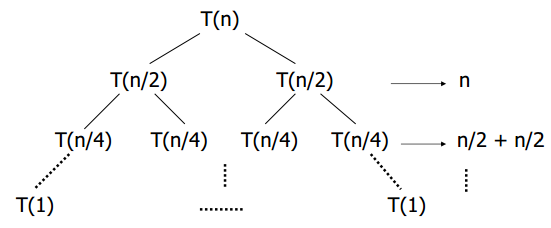


3.master theorem

套公式



递归树Recursion Tree可以帮助计算时间复杂度





Divide and Conquer

Principle of Divide and Conquer:

1.partition a problem into different parts

2.solve the subproblems

3.combine the solutions to form the overall solution

e.g. Find Max and Min

[divide]Divide the n number into two halves, s1 and S2

[Solve]For each half, find the max and min

[Combine]max(s)=max(max(S1),max(s2)), min(S)=min(min(S1),min(S2))

Max(S[1..n])

{

If(n==1) //base case, size=1

{

Max=S[1];min=s[1];

}else if(n==2) //base,size=2

{

If(S[1]>S[2]) {max=s[1];min=s[2];}

Else {max=s[2];min=s[1];}

}else

{

Divide S into S1 and S2, each with half of elements

(max1,min1)=MaxMin(S1) //recursion

(max2,min1)=MaxMin(S2)

If(max1>max2)Max=max1

Else max=max2

If(min1<min2)min=min1

Else min=min2

}

Return (max,min)

}

T(n)=2T(n/2)+2, T(2)=1,T(1)=0

T(n)=3n/2-2

Merge Sort

MergeSort(A[1..n]) {

if (size of A == 1) return A; /\* base case, nothing to sort \*/

B1 := MergeSort(A[1..n/2]);

B2 := MergeSort(A[n/2+1..n]);

B := Merge(B1, B2);

return B; /\* sorted array \*/

}

/\*

merge two sorted array a[1..n] and b[1..n] into output c[]

\*/

Merge() {

i := 1; j := 1; k := 1;

/\* pointer to arrays a, b, c \*/

while (i <= n and j <= n) {

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

{c[k] := a[i]; i++; k++; }

else {c[k] := b[j]; j++; k++; }

}

Copy all remaining elements

}

T(n)=2T(n/2)+O(n)

So T(n)=O(n log n)