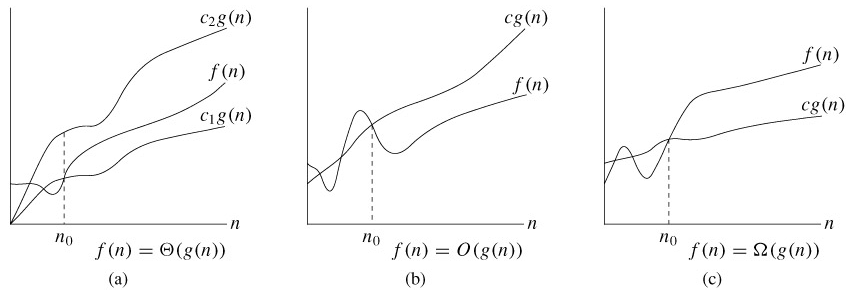
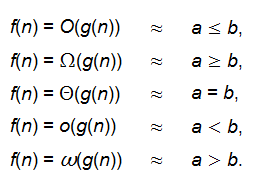
# myNote

## Asymptotic notation





### Question1

Exercises 3.1-3

|  |
| --- |
|  |

Explain why the statement, "The running time of algorithm *A* is at least *O*(*n*2)," is meaningless.

|  |
| --- |
|  |

|  |
| --- |
|  |

Let the running time be *T (n)*. *T (n)* ≥ *O(n*2*)* means that *T (n)* ≥ *f (n)* for some

function *f (n)* in the set *O(n*2*)*. This statement holds for any running time *T (n)*,

since the function *g(n)* = 0 for all *n* is in *O(n*2*)*, and running times are always

nonnegative. Thus, the statement tells us nothing about the running time.

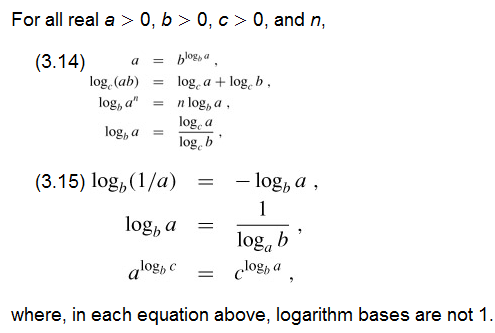
## Standard notations and common functions

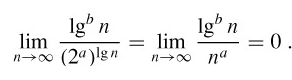
### Floors and ceilings

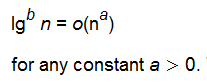
The floor function *f*(*x*) = ⌊*x*⌋ is monotonically increasing, as is the ceiling function *f*(*x*) = ⌈*x*⌉.



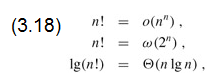
### Logarithms





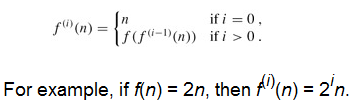


### Factorials



### Functional iteration

We use the notation *f*(*i*)(*n*) to denote the function *f*(*n*) iteratively applied *i* times to an initial value of *n*. Formally, let *f*(*n*) be a function over the reals. For nonnegative integers *i*, we recursively define



### The iterated logarithm function

lg\* *n* = min {*i* = 0: lg(*i*) *n* ≤ 1}.

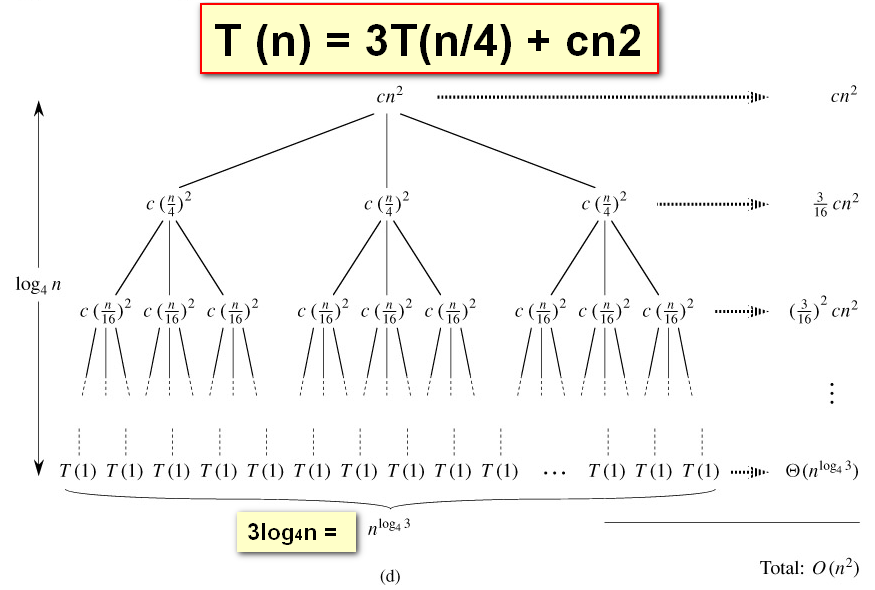
The iterated logarithm is a *very* slowly growing function:

|  |  |  |
| --- | --- | --- |
| lg\* 2 | = | 1, |
| lg\* 4 | = | 2, |
| lg\* 16 | = | 3, |
| lg\* 65536 | = | 4, |
| lg\*(265536) | = | 5. |

we rarely encounter an input size n such that lg\* n > 5.

## The recursion-tree method

**T(n) = aT(n/b) + f (n) ,**



f(n) = cn2:计算出每行计算数目，再除去最后一行外，对所有行数的计算量求和，得O（n2）,其实就是

*n*log*ba*：最后一行的最大数目，叶子增长速率\*高度=3log4n=nlog43< O（n2）

nlog43+O（n2）= O（n2）

## The master method

**T(n) = aT(n/b) + f (n) ,**

where a ≥ 1, b > 1, and f is asymptotically positive.

Compare *f* (*n*) with *n*log*ba*:

1. *f* (*n*) = *O*(*n*log*ba* – ε) for some constant ε > 0.

• *f* (*n*) < *n*log*ba ：f* (*n*) grows polynomially slower than *n*log*ba* (by an *n*ε factor).

***Solution:*** *T*(*n*) = Θ(*n*log*ba* ) .

EX: ***T (n) = 4T (n/2) + n***

*n*log*ba*= *n*log2*4* ***> n****=f(n)**Solution:T (n) =Θ(n2).*

2. *f* (*n*) = Θ(*n*log*ba* lg*kn*) for some constant *k* ≥ 0.

• *f* (*n*) ≈*n*log*ba ：f* (*n*) and *n*log*ba* grow at similar rates.

***Solution:*** *T*(*n*) = Θ(*n*log*ba* lg*k*+1*n*) . 默认k=0 *T*(*n*) = Θ(*n*log*ba* lg*n*)

EX: ***T (n) = 4T (n/2) + n2***

*n*log*ba*= *n*log2*4* ***= n2****=f(n)**Solution:T (n) =*Θ(*n2* lg*n*)*.*

3. *f* (*n*) = Ω(*n*log*ba* + ε) for some constant ε > 0.

• *f* (*n*) > *n*log*ba : f* (*n*) grows polynomially faster than *n*log*ba* (by an *n*εfactor),

***and*** *f* (*n*) satisfies the ***regularity condition*** that

*a f* (*n/b*) ≤ *c f* (*n*) for some constant *c* < 1. ***即：f (n)=n k ,*** ***a/b k < 1***

***Solution:*** *T*(*n*) = Θ( *f* (*n*)) .

EX: ***T (n) = 4T (n/2) + n3***

*n*log*ba*= *n*log2*4* ***<n3****=f(n)* ***& a/b k =4/23 < 1*** *Solution:T (n) =Θ(****n3****).*

***Notice:***

It is important to realize that the three cases do not cover all the possibilities for *f* (*n*). There is a gap between cases 1 and 2 when *f* (*n*) is smaller than *n*log*ba* but not polynomially smaller. Similarly, there is a gap between cases 2 and 3 when *f* (*n*) is larger than *n*log*ba* but not polynomially larger. If the function *f* (*n*) falls into one of these gaps, or if the regularity condition in case 3 fails to hold, the master method cannot be used to solve the recurrence

### Typical forms:

***T (n) = T (n − 1) + n****Solution:T (n) =Θ(n2).*

***T (n) = T (√n) + 1****Solution:T (n) =Θ(lg lg n).*

***T (n) = T (n − 1) + 1/n****Solution:T (n) =Θ(lg n).*

***T (n) = T (n − 1) + lg n*** *Solution:T (n) =Θ(nlg n).*

***T (n) = T (n/2) + T (n/4) + T (n/8) + n*** *Solution:T (n) =Θ(n).*

## The divide-and-conquer design paradigm

***1. Divide the problem (instance)***

***into subproblems.***

***2. Conquer the subproblems by***

***solving them recursively.***

***3. Combine subproblem solutions.***

***Powering a number***

***Problem: Compute a n, where n ∈ N.***

***Naive algorithm: Θ(n).***

***Divide-and-conquer algorithm:***

***a n =an/2 ⋅ an/2  if n is even;****//偶数*

***a n =a(n–1)/2 ⋅ a(n–1)/2 ⋅ a if n is odd.*** *//奇数*

***T(n) = T(n/2) + Θ(1) ⇒ T(n) =*** ***Θ(lg n) .***

## Quicksort *Θ ( nlgn )*

### Properties

**• Worst-case running time:** Θ( *n*2 )

**• Expected running time:** Θ( *n*lg*n* )

**• Constants hidden in**Θ( *n*lg*n* )**are small.**

**• Sorts in place.**

**Quicksort is typically over twice as fast as merge sort.**

Quicksort can benefit substantially from ***code tuning***.

Quicksort behaves well even with caching and virtual memory.

### method

**Quicksort an *n*-element array:**

***1. Divide:*** Partition the array into two subarrays

around a ***pivot*** *x* such that elements in lower

subarray ≤ *x* ≤ elements in upper subarray.



***2. Conquer:*** Recursively sort the two subarrays.

***3. Combine:*** Trivial.

### Pseudocode for quicksort

QUICKSORT(*A*, *p, r*)

**if** *p* < *r*

**then** *q* = PARTITION(*A*, *p, r*)

QUICKSORT(*A*, *p, q*–1)

QUICKSORT(*A*, *q+*1*, r*)

**Initial call:** QUICKSORT(*A*, 1*, n*)

PARTITION(*A*, *p*, *q*)

K=random(p , q) //**比较值，为优化计算，不出现最差情况，用随机值**

exchange *A*[*p*] ↔ *A*[ *k*]

***x* = *A*[*p*]** *i =* *p*

**for** *j =* *p* + 1 **to** *q {*

**do if** *A*[ *j*] ≤ *x {*

**then** *I =* *i* + 1

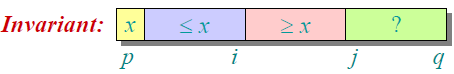
exchange *A*[*i*] ↔ *A*[ *j*]

}

}

exchange *A*[*p*] ↔ *A*[*i*]

**return** *i*



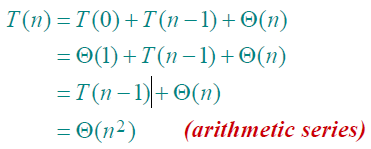
### Worst-case of quicksort Θ( *n*2 )

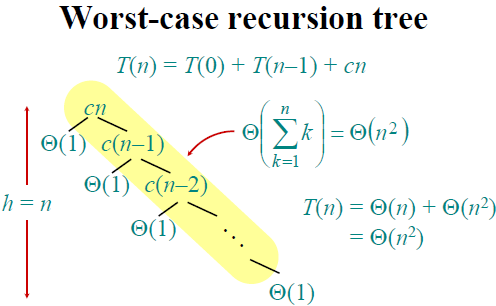
• Input sorted or reverse sorted.

• Partition around min or max element.

• One side of partition always has no elements

***Unlucky:***



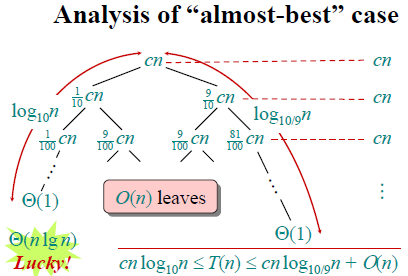


### best-case Θ(*n* lg *n*)

If we’re lucky, PARTITION splits the array evenly:

*T*(*n*) = 2*T*(*n*/2) + Θ(*n*) = Θ(*n* lg *n*) (same as merge sort)





***Lucky:***

*T*(*n*) = *T*(ni/k) + *T*(n(k-i)/k) + Θ(*n*) = Θ(*n* lg *n*)

### Normal-case Θ(*n* lg *n*)

Suppose we alternate lucky, unlucky,

lucky, unlucky, lucky, ….

*L*(*n*) = 2*U*(*n*/2) + Θ(*n*) ***lucky***

*U*(*n*) = *L*(*n* – 1) + Θ(*n*) ***unlucky***

Solving:

*Becase: U*(*n*/2) = *L*(*n*/2 – 1) + Θ(*n*/2)

*So: L*(*n*) = 2(*L*(*n*/2 – 1) + Θ(*n*/2)) + Θ(*n*)

= 2*L*(*n*/2 – 1) + Θ(*n*)

= Θ(*n* lg *n*) ***Lucky!***

### optimal design 1 - Randomized quicksort Θ(*n* lg *n*)

**IDEA**: Partition around a ***random*** element.

• Running time is independent of the input order.

• No assumptions need to be made about the input distribution.

• No specific input elicits the worst-case behavior.

• The worst case is determined only by the output of a random-number generator.

### optimal design2 - Randomized quicksort worst caseΘ(*n* lg *n*)

BEST-CASE-QUICKSORT*(A, p, r )*

**if** *p < r*

**then** *i* ← *(r* − *p* + 1*)/*2 //half of number

[*x* ←SELECT*(A, p, r, i )*](#_SELECT__Θ(n)) *//get the middle value*

*q* ←PARTITION*(x)*

BEST-CASE-QUICKSORT*(A, p, q* − 1*)*

BEST-CASE-QUICKSORT*(A, q* + 1*, r )*

## Heapsort

• *O(n* lg *n)* worst case.like merge sort.

• Sorts in place.like insertion sort.

• Combines the best of both algorithms.

### Pseudocode for HEAPSORT

HEAPSORT*(A, n)*

BUILD-MAX-HEAP*(A, n)* // Builds a max-heap from the array.

**for** *i* ← *n* **downto** 2 // Starting with the root (the maximum element)

//places the maximum element into the correct place

**do** exchange *A*[1] ↔ *A*[*i* ]

MAX-HEAPIFY*(A,* 1*, i* − 1*)* //Builds a max-heap just for the changed item

Builds a max-heap

BUILD-MAX-HEAP*(A, n)* Time: O(n).

**for** *i* ← *n/*2 **downto** 1 // i starts off in the middle

**do** MAX-HEAPIFY*(A, i, n)*

MAX-HEAPIFY*(A, i, n)* Time: O(lg n).

*l* ← LEFT*(i ) ,r* ← RIGHT*(i )*

//swap A[i ] with the larger of the two children to preserve heap property.

//until subtree rooted at i is max-heap or a leaf

**if** *l* ≤ *n* and *A*[*l*] *> A*[*i* ]

**then** *largest* ←*l* **else** *largest* ←*i*

**if** *r* ≤ *n* and *A*[*r* ] *> A*[*largest*]

**then** *largest* ←*r*

**if** *largest* \_= *i*

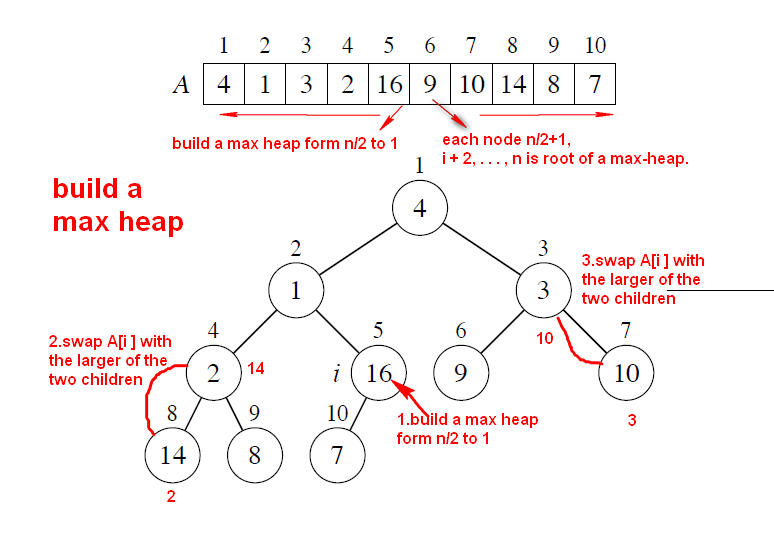
**then** exchange *A*[*i* ] ↔ *A*[*largest*]

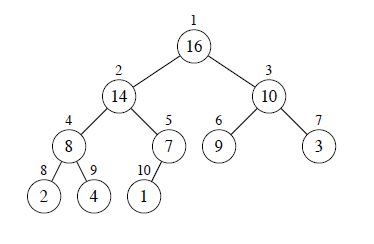
MAX-HEAPIFY*(A, largest, n)*

BUILD-MAX-HEAP*(A, n)* Time: O(n).

**for** *i* ← *n/*2 **downto** 1 // i starts off in the middle

**do** MAX-HEAPIFY*(A, i, n)*





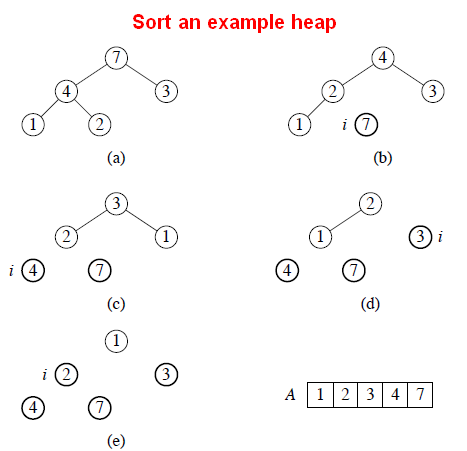
HEAPSORT*(A, n)*

BUILD-MAX-HEAP*(A, n)*

**for** *i* ← *n* **downto** 2

**do** exchange *A*[1] ↔ *A*[*i* ]

MAX-HEAPIFY*(A,* 1*, i* − 1*)*



### Priority queues

Basic operations

* INSERT(*S*, *x*) inserts the element *x* into the set *S*. This operation could be written as *S* ← *S* ∪ {*x*}.
* MAXIMUM(*S*) returns the element of *S* with the largest key.
* EXTRACT-MAX(*S*) removes and returns the element of *S* with the largest key.
* INCREASE-KEY(*S*, *x*, *k*) increases the value of element *x*'s key to the new value *k*, which is assumed to be at least as large as *x*'s current key value

HEAP-EXTRACT-MAX(A) // 取出最大元素，即为根元素

1 if heap-size[A] < 1

2 then error "heap underflow"

3 max ← A[1] //根元素

4 A[1] ← A[heap-size[A]] // 将末尾元素替换到根，再进行最大堆处理

5 heap-size[A] ← heap-size[A] - 1

6 MAX-HEAPIFY(A, 1)

7 return max

HEAP-INCREASE-KEY(*A*, *i*, *key*) // 将i元素优先级升到key

1 **if** *key* < *A*[*i*]

2 **then error** "new key is smaller than current key"

3 *A*[*i*] ← *key*

// 升级后的i元素向上递归，构建最大堆

4 **while** *i* > 1 and *A*[PARENT(*i*)] < *A*[*i*]

5 **do** exchange *A*[*i*] ↔ *A*[PARENT(*i*)]

6 *i* ← PARENT(*i*)

// 插入元素，先放到末尾，在调用升级程序HEAP-INCREASE-KEY(A, i, key)

MAX-HEAP-INSERT(*A*, *key*)

1 *heap-size*[*A*] ← *heap-size*[*A*] + 1

2 *A*[*heap-size*[*A*]] ← -∞

3 HEAP-INCREASE-KEY(*A*, *heap-size*[*A*], *key*)

## How fast can we sort?

### Comparison sorting

The only operation that may be used to gain order information about a sequence

is comparison of pairs of elements.

All sorts seen so far are comparison sorts: insertion sort, selection sort, merge

sort, quicksort, heapsort, treesort

time

•Ω(n)to examine all the input.

•Ω(nlgn)is a lower bound for comparison sorts.

### Sorting in linear time

#### Counting sort

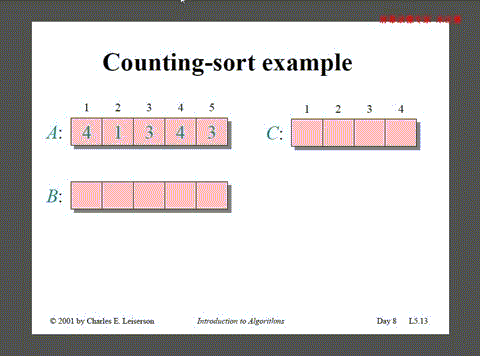
Non-comparison sorts.-**Counting sort** Θ(*n* + *k*)

**Counting sort:** No comparisons between elements. ***Stable，最适合于 k<<n***

***Counting sort*** assumes that each of the *n* input elements is an integer in the range 0 to *k*

• ***Input***: *A*[1 . . *n*], where *A*[ *j*]∈{1, 2, …, *k*} .，下标为1->n，数组值1->k

• ***Output***: *B*[1 . . *n*], sorted.

****• ***Auxiliary storage***: *C*[1 . . *k*] . 计数器，下标1->k代表数组数值，c[k]值代表出现的次数

**Pseudocode for Counting sort**

COUNTING-SORT(*A*, *B*, *k*)

**for** *i* ← 1 **to** *k*

**do** *C*[*i*] ← 0 //

**for** *j* ← 1 **to** *n // A*[ *j*]作为C的下标，C储存A[ j]出现的次数

**do** *C*[*A*[ *j*]] ← *C*[*A*[ *j*]] + 1 ⊳ //*C*[*i*] = |{key = *i*}|

**for** *i* ← 2 **to** *k //给元素定位置*

**do** *C*[*i*] ← *C*[*i*] + *C*[*i*–1] ⊳ //*C*[*i*] = |{key ≤ *i*}|

**for** *j* ← *n* **downto** 1 //将A[ *j*]的值放到B中，位置由C决定

**do** *B*[*C*[*A*[ *j*]]] ← A[ *j*]

*C*[*A*[ *j*]] ← *C*[*A*[ *j*]] – 1

Time：Θ(*n* + *k*) If *k* = *O*(*n*), then counting sort takes Θ(*n*) time

Counting sort will be used in radix sort.

#### radix sort

***Key idea:*** Sort *least* signiÞcant digits Þrst.

To sort *d* digits:

RADIX-SORT*(A, d)*

**for** *i* ← 1 **to** *d*

**do** use a stable sort to sort array *A* on digit *i（usually Counting sort）*

How to break each key into digits?

• *n* words.

• *b* bits/word.

• Break into *r* -bit digits( *r* ≈ lg *n* ). Have *d* = *b/r*

• Use counting sort, *k* = 2*r* − 1.

Example: 32-bit words, 8-bit digits. *b* = 32, *r* =lg32= 8, *d* = 32*/*8 = 4, *k* =28 − 1 = 255

32长度字符，分割成每段长度为*r* =lg32= 8位的子段，共分成*d* = 32*/*8 = 4段，每个子段长度为*k* = 2*r* – 1=255，调用计数排序时，原始待排序数组大小为n，而辅助计数数组为k

• Time =**Θ((n + 2r ) b/r)** **= Θ((n + n) b/lgn)= Θ(bn/ lgn)**

#### Bucket sort

bucket sort assumes that the input is generated by a random process that distributes elements uniformly over the interval [0, 1).

**Input:** *A*[1 *. . n*], where 0 ≤ *A*[*i* ] *<* 1 for all *i* .

**Auxiliary array:** *B*[0 *. . n* − 1] of linked lists, each list initially empty.

**Pseudocode for Bucket sort**

BUCKET-SORT*(A, n)*

**for** *i* ← 1 **to** *n*

**do** insert *A*[*i* ] into list *B*[*n* · *A*[*i* ]]

**for** *i* ← 0 **to** *n* − 1

**do** sort list *B*[*i* ] with insertion sort

concatenate lists *B*[0]*, B*[1]*, . . . , B*[*n* − 1] together in order

**return** the concatenated lists

## Medians and Order Statistics

### Selection in expected linear time

#### RANDOMIZED-SELECT

Selection of the *i* th smallest element of the array *A*

**Pseudocode for** **RANDOMIZED-SELECT**

RANDOMIZED-SELECT*(A, p, r, i )*

**if** *p* = *r*

**then return** *A*[*p*]

*q* ←RANDOMIZED-PARTITION*(A, p, r )*

*k* ← *q* − *p* + 1 // q – p中比 A[q]小的值的个数

**if** *i* = *k* pivot value is the answer

**then return** *A*[*q*]

**else if** *i < k*

**then return** RANDOMIZED-SELECT*(A, p, q* − 1*, i )*

**else return** RANDOMIZED-SELECT*(A, q* + 1*, r, i* − *k)* // 往后比较，减前面K个最小值

PARTITION(*A*, *p*, *q*)

***x* = *A*[RANDOM*(p, q )*]**  //**比较值，为优化计算，不出现最差情况，用随机值**

*i =* *p*

**for** *j =* *p* + 1 **to** *q {*

**do if** *A*[ *j*] ≤ *x {*

**then** *I =* *i* + 1

exchange *A*[*i*] ↔ *A*[ *j*]

}

}

exchange *A*[*p*] ↔ *A*[*i*]

**return** *i*

After the call to RANDOMIZED-PARTITION, the array is partitioned into two subarrays

*A*[*p . . q* − 1] and *A*[*q* + 1 *. . r* ], along with a ***pivot*** element *A*[*q*].

• The elements of subarray *A*[*p . . q* − 1] are all ≤ *A*[*q*].

• The elements of subarray *A*[*q* + 1 *. . r* ] are all *> A*[*q*].

• The pivot element is the *k*th element of the subarray *A*[*p . . r* ], where *k* =

*q* − *p* + 1.

***Worst-case running time:*** Θ(*n2*)

***Expected running time:*** Θ(*n*)

#### SELECT Θ(*n*)

**Worst-case linear-time order statistics**

**Pseudocode for SELECT**

SELECT(*i, n*)

1. Divide the *n* elements into groups of 5.(7 is ok,but not 3) Θ(*n*)

Find the median of each 5-element group by rote.

2. Recursively SELECT the median *x* of the *n*/5 *T*(*n*/5)

group medians to be the pivot.

3. Partition around the pivot *x*. Let *k* = rank(*x*). Θ(*n*)

4. **if** *i* = *k* **then return** *x T*(3*n*/4)

**else if** *i* < *k*

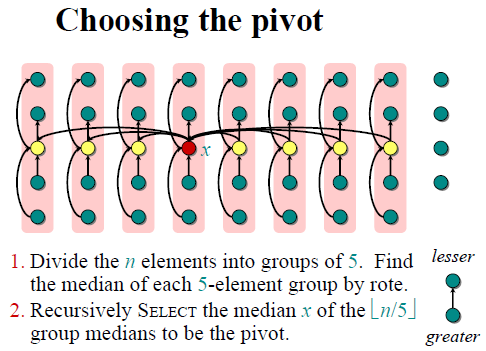
**then** recursively SELECT the *i*th

smallest element in the lower part

**else** recursively SELECT the (*i–k*)th

smallest element in the upper part

*T*(*n*)=Θ(*n*)+*T*(*n*/5)+Θ(*n*)+*T*(3*n*/4)= Θ(*n*)



**Use:** **solves the selection problem for an arbitrary order statistic-A good** procedure to get the median value

**Example:** procedure MEDIAN (*A, p, r*) array *A* and subarray indices *p* and *r* , returns the value of the median element of *A*[*p . . r*] in *O(n)* time in the worst case.

SELECT*(A, p, r, i ) //* *T (n) ≤ T (n/2) + O(n) = O(n).*

**if** *p* = *r*

**then return** *A*[*p*]

*x* ←MEDIAN*(**A, p, r )*

*q* ←PARTITION*(x)*

*k* ← *q* − *p* + 1

**if** *i* = *k*

**then return** *A*[*q*]

**else if** *i < k*

**then return** SELECT*(A, p, q* − 1*, i )*

**else return** SELECT*(A, q* + 1*, r, i* − *k)*

## Summary

Notice that SELECT and RANDOMIZED-SELECT determine information about the

relative order of elements only by comparing elements.

• Sorting requiresΘ(nlgn)time in the comparison model.

• Sorting algorithms that run in linear time need to make assumptions about their

input.

• Linear-time *selection* algorithms do not require any assumptions about their

input.

• Linear-time selection algorithms solve the selection problem without sorting

and therefore are not subject to theΩ(nlgn)lower bound

### Exercises 9.3-8

**Question:**Let *X*[1 .. *n*] and *Y* [1 .. *n*] be two arrays, each containing *n* numbers already in sorted order. Give an *O*(lg *n*)-time algorithm to find the median of all 2*n* elements in arrays *X* and *Y*

**Analyse:**

Let.s start out by supposing that the median (the lower median, since we know we have an even number of elements) is in *X*. Let.s call the median value *m*, and let.s

suppose that it.s in *X*[*k*]. Then *k* elements of *X* are less than or equal to *m* and

*n* −*k* elements of *X* are greater than or equal to *m*. We know that in the two arrays

combined, there must be *n* elements less than or equal to *m* and *n* elements greater

than or equal to *m*, and so there must be *n* − *k* elements of *Y* that are less than or

equal to *m* and *n* − *(n* − *k)* = *k* elements of *Y* that are greater than or equal to *m*.

Thus, we can check that *X*[*k*] is the lower median by checking whether *Y* [*n*−*k*] ≤

*X*[*k*] ≤ *Y* [*n* − *k* + 1]. A boundary case occurs for *k* = *n*. Then *n* − *k* = 0, and

there is no array entry *Y* [0]; we only need to check that *X*[*n*] ≤ *Y* [1].

Now, if the median is in *X* but is not in *X*[*k*], then the above condition will not

hold. If the median is in *X*[*k*\_], where *k*\_*< k*, then *X*[*k*] is above the median, and

*Y* [*n* − *k* + 1] *< X*[*k*]. Conversely, if the median is in *X*[*k*\_\_], where *k*\_\_*> k*, then

*X*[*k*] is below the median, and *X*[*k*] *< Y* [*n* − *k*].

Thus, we can use a binary search to determine whether there is an *X*[*k*] such that

either *k < n* and *Y* [*n*−*k*] ≤ *X*[*k*] ≤ *Y* [*n*−*k*+1] or *k* = *n* and *X*[*k*] ≤ *Y* [*n*−*k*+1];

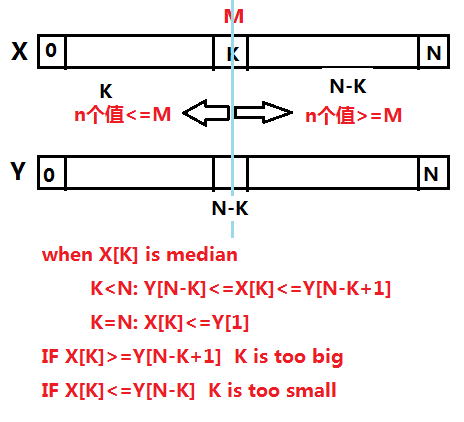
if we find such an *X*[*k*], then it is the median. Otherwise, we know that the median

is in *Y* , and we use a binary search to Þnd a *Y* [*k*] such that either *k < n* and

*X*[*n* − *k*] ≤ *Y* [*k*] ≤ *X*[*n* − *k* + 1] or *k* = *n* and *Y* [*k*] ≤ *X*[*n* − *k* + 1]; such a

*Y* [*k*] is the median. Since each binary search takes *O(*lg *n)* time, we spend a total

of *O(*lg *n)* time.



**pseudocode:**

TWO-ARRAY-MEDIAN*(X, Y )*

*n* ← *length*[*X*] *\_ n* also equals *length*[*Y* ]

*median* ← FIND-MEDIAN*(X, Y, n,* 1*, n)*

**if** *median* = NOT-FOUND

**then** *median* ← FIND-MEDIAN*(Y, X, n,* 1*, n)*

**return** *median*

FIND-MEDIAN*(A, B, n, low, high)*

**if** *low > high*

**then return** NOT-FOUND

**else** *k* ←*(low*+*high)/*2

**if** *k* = *n* and *A*[*n*] ≤ *B*[1]

**then return** *A*[*n*]

**else if** *k < n* and *B*[*n* − *k*] ≤ *A*[*k*] ≤ *B*[*n* − *k* + 1]

**then return** *A*[*k*]

**else if** *A*[*k*] *> B*[*n* − *k* + 1]

**then return** FIND-MEDIAN*(A, B, n, low, k* − 1*)*

**else return** FIND-MEDIAN*(A, B, n, k* + 1*, high)*

## Hash table

Let *n* be the number of keys in the table, and

let *m* be the number of slots.

Define the ***load factor*** of *T* to be

α = *n*/*m* = average number of keys per slot

### Hash functions

**Keys as natural numbers**

• Hash functions assume that the keys are natural numbers.

• When they.re not, have to interpret them as natural numbers.

• ***Example:*** Interpret a character string as an integer expressed in some radix notation. Suppose the string is CLRS:

• ASCII values: C = 67, L = 76, R = 82, S = 83.

• There are 128 basic ASCII values.

• So interpret CLRS as *(*67 · 1283*)*+ *(*76 · 1282*)*+ *(*82 · 1281*)*+ *(*83 · 1280*)* =141,764,947.

#### functions *1 h*(*k*) = *k* mod *m*

Pick *m* to be a prime(质数) not too close to a power of 2 or 10 and not otherwise used prominently in the computing environment

**Extreme deficiency:**

If *m* = 2*r*, then the hash doesn’t even depend on all the bits of *k*:

• If *k* = 10110001110110102 and *r* = 6, then *h*(*k*) = 0110102 .

If *k* = 10110001110110102 and *r* = 4, then *h*(*k*) =10102 .

#### functions 2 better h(k) = (A·k mod 2w) rsh (w – r)

Assume that all keys are integers, *m* = 2*r*（插槽数） and our computer has *w*-bit words(normally 32bits or 64bits).

Define *h*(*k*) = (*A***·***k* mod 2*w*) rsh (*w* – *r*),

where rsh is the “bit-wise right-shift” operator 位右移

and *A* is an odd integer in the range 2*w*–1 < *A* < 2*w*

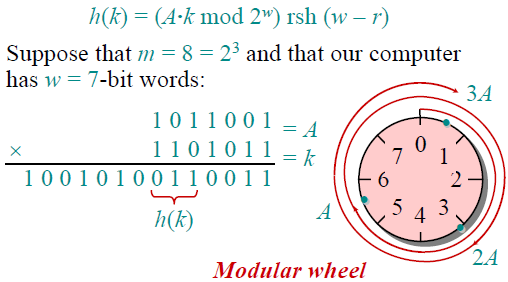
• Don’t pick *A* too close to 2*w*.

• Multiplication modulo 2*w* is fast.

• The rsh operator is fast.

理由：当k逐渐增大时，h(k)不会填充插槽的同一位置，而是类似逐渐遍历，减少碰撞

Example:



### handle collisions

Use two methods: chaining and open addressing.

Chaining is usually better than open addressing

#### Open addressing

**we have at most one element per slot, and thus *n* ≤ *m*, which implies α= *n*/*m*≤ 1.**

**HASH-INSERT(*T*, *k*)**

1 *i* ← 0

2 **repeat** *j* ← *h*(*k*, *i*)

3 **if** *T*[*j*] = NIL

4 **then** *T*[*j*] ← *k*

5 **return** *j*

6 **else** *i* ← *i* + 1

7 **until** *i* = *m*

8 **error** "hash table overflow"

**HASH-SEARCH(*T*, *k*)**

1 *i* ← 0

2 **repeat** *j* ← *h*(*k*, *i*)

3 **if** *T*[*j*] = *k*

4 **then return** *j*

5 *i* ← *i* + 1

6 **until** *T*[*j*] = NIL or *i* = *m*

7 **return** NIL

**HASH-DELETE(*T*, *k*)**

1 *i* ← 0

2 **repeat** *j* ← *h*(*k*, *i*)

3 **if** *T*[*j*] = *k*

4 **then** *T*[*j*] = DELETE(NOT null) **return** *success*

5 *i* ← *i* + 1

6 **until** *T*[*j*] = NIL or *i* = *m*

7 **return** un*success*

**Linear probing 线性探查法**

*h*(*k*, *i*) = (*h*'(*k*) + *i*) mod *m*

for *i* = 0, 1, ..., *m* – 1

initial *T*[*h*'(*k*)];

**Quadratic probing 二次探查法**

**better than linear probing**

*h*(*k*, *i*) = (*h*'(*k*) + c1*i+c2i²*) mod *m*

for *i* = 0, 1, ..., *m* – 1

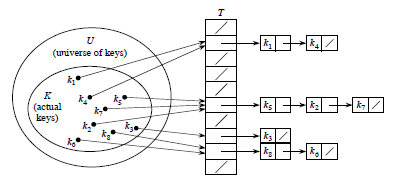
initial *T*[*h*'(*k*)];

**Double hashing**

*h*(*k*, *i*) = (*h*1(*k*) + *ih*2(*k*)) mod *m*,

initial *T*[*h*1(*k*)];

#### chaining



#### 11.3.3 ★ Universal hashing

select the hash function at random from a carefully designed class of functions at the beginning of execution

Let ℋ be a finite collection of hash functions that map a given universe *U* of keys into the range {0, 1, ..., *m* - 1}. with a hash function randomly chosen from ℋ, the chance of a collision between distinct keys *k* and *l* is no more than the chance 1/*m* of a collision if *h*(*k*) and *h*(*l*) were randomly and independently chosen from the set {0, 1, ..., *m* - 1}.

**Designing a universal class of hash functions**

***ha,b* (k)=((ak+b) mod p) mod m**

0<=k<=p-1 key *k* is in the range 0 to *p* – 1

*p* > *m m hash table legth, p* is prime

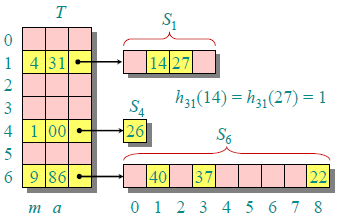
b=Z*p=*{0, 1, ..., *p* - 1}

a=Z*p\*=*{ 1,2, ..., *p* - 1}

**there are *p*(*p* - 1) hash functions in ℋ*p,m*.**

### Perfect hashing

static hash table SEARCH takes Θ(1) time in the *worst case*



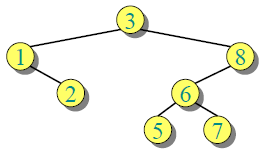
The first level is essentially the same as for hashing with chaining

a small ***secondary hash table*** *Sj* with an associated hash function *hj*. By choosing the hash functions *hj* carefully, we can guarantee that there are no collisions at the secondary level. let the size *mj* of hash table *Sj* be the square of the number *nj* of keys hashing to slot *j*.

by choosing the first level hash function well, the expected total amount of space ,including the secondary hash table ,used is still *O*(*n*).

If we store *n* keys in a hash table of size *m* = *n* using a hash function *h* randomly chosen from a universal class of hash functions and we set the size of each secondary hash table to for *j* = 0, 1, ..., *m* - 1, then the expected amount of storage required for all secondary hash tables in a perfect hashing scheme is less than 2*n*.

## Binary Search Trees （inorder）



### Create an empty BST

**for** *i* = 1 to *n*

**do** TREE-INSERT(*T*, *A*[*i*])

### 中序遍历 *O(h)*

INORDER-TREE-WALK*(x)*

**if** *x* ≠ NIL

**then** INORDER-TREE-WALK*(left*[*x*]*)*

print *key*[*x*]

INORDER-TREE-WALK*(right*[*x*]*)*

### Searching *O(h)*

TREE-SEARCH*(x, k)*

if *x* = NIL or *k* = *key*[*x*]

then return *x*

**if** *k < key*[*x*]

**then return** TREE-SEARCH*(left*[*x*]*, k)*

**else return** TREE-SEARCH*(right*[*x*]*, k)*

Initial call ： TREE-SEARCH*(root*[*T* ]*, k)*.

### Minimum and maximum *O(h)*

**Property：**

• the minimum key of a binary search tree is located at the leftmost node, and

• the maximum key of a binary search tree is located at the rightmost node.

**TREE-MINIMUM*(x)***

**while** *left*[*x*] ≠ NIL

**do** *x* ← *left*[*x*]

**return** *x*

**TREE-MAXIMUM*(x)***

**while** *right*[*x*] ≠ NIL

**do** *x* ← *right*[*x*]

**return** *x*

### Successor and predecessor *O(h)*

Assuming that all keys are distinct

**Successor 前驱**

the successor of a node *x* is the node *y* such that *key*[*y*] is the smallest key *> key*[*x*].

There are two cases:

1. If node *x* has a ***non-empty right subtree***, then *x*.s successor is the minimum in

*x*.s right subtree.( leftmost in right subtree)

2. If node *x* has an ***empty right subtree***, notice that:

• As long as we move to the left up the tree (**move up** through right children),we are visiting smaller keys.

• if the right subtree of node *x* is empty and *x* has a successor *y*, then *y* is the lowest ancestor of *x* whose left child is also an ancestor of *x* (***x* is the maximum in *y*.s left subtree**).

**TREE-SUCCESSOR*(x)***

**if** *right*[*x*] ≠ NIL // non-empty right subtree

**then return** TREE-MINIMUM*(right*[*x*]*)*

*y* ← *p*[*x*] // move up

// x is the maximum in y.s left subtree

**while** *y* ≠ NIL and *x* = *right*[*y*] // when x = left[y] break

**do** *x* ← *y*

*y* ← *p*[*y*]

**return** *y*

### Insertion *O(h)*

**TREE-INSERT*(T, z)***

*y* ← NIL

*x* ← *root*[*T* ]

**while** *x* ≠ NIL // 找到插入尾结点

**do** *y* ← *x* // keep track of parent of x

**if** *key*[*z*] *< key*[*x*]

**then** *x* ← *left*[*x*]

**else** *x* ← *right*[*x*]

*p*[*z*] ← *y* //尾结点

**if** *y* = NIL

**then** *root*[*T* ] ← *z* //Tree T was empty

**else if** *key*[*z*] *< key*[*y*] //插入尾结点孩子

**then** *left*[*y*] ← *z*

**else** *right*[*y*] ← *z*

### Deletion *O(h)*

TREE-DELETE is broken into three cases.

**Case 1:** *z* has no children.

• Delete *z* by making the parent of *z* point to NIL, instead of to *z*.

**Case 2:** *z* has one child.

• Delete *z* by making the parent of *z* point to *z*.s child, instead of to *z*.

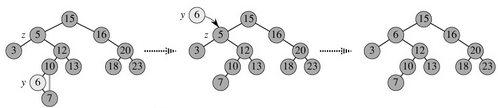
**Case 3:** *z* has two children.

• *z*.s successor *y* has either no children or one child. (*y* is the minimum

node.with no left child.in *z*.s right subtree.)

• Delete *y* from the tree (via Case 1 or 2).

• Replace *z*.s key and satellite data with *y*.s.



**TREE-DELETE*(T, z)*** *O(h)*

△Determine which node y to splice out: either z or z.s successor.

Y结点需要被删除

**if** *left*[*z*] = NIL or *right*[*z*] = NIL

**then** *y* ← *z*

**else** *y* ← TREE-SUCCESSOR*(z)* //two children

//x is set to a non-NIL child of y, or to NIL if y has no children.储存y的孩子节点

**if** *left*[*y*] ≠ NIL

**then** *x* ← *left*[*y*]

**else** *x* ← *right*[*y*]

//y的孩子节点与y的父节点连接

**if** *x* ≠ NIL

**then** *p*[*x*] ← *p*[*y*]

**if** *p*[*y*] = NIL //empty tree

**then** *root*[*T* ] ← *x*

**else if** *y* = *left*[*p*[*y*]]

**then** *left*[*p*[*y*]] ← *x*

**else** *right*[*p*[*y*]] ← *x*

If it was z.s successor that was spliced out, copy its data into z.

**if** *y* ≠ *z*

**then** *key*[*z*] ← *key*[*y*]

copy y.s satellite data into z

**return** *y*

## Red-black trees

• ***Balanced***: height is *O(*lg *n)*, where *n* is the number of nodes.

• Operations will take *O(*lg *n)* time in the worst case.

### Red-black properties

1. Every node is either red or black.

2. The root is black.

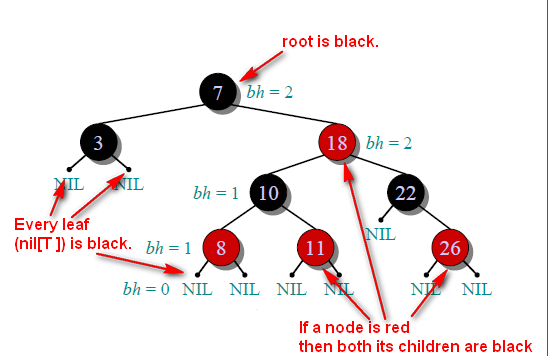
3. Every leaf (*nil*[*T* ]) is black.

4. If a node is red, then both its children are black. (Hence no two reds in a row

on a simple path from the root to a leaf.)

5. For each node, all paths from the node to descendant leaves contain the same

number of black nodes.



***Black-height*** of a node *x*: bh*(x)* is the number of black nodes (including *nil*[*T* ])

on the path from *x* to leaf, not counting *x*. By property 5, black-height is well

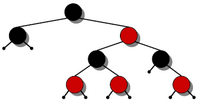
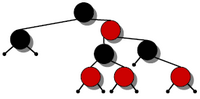
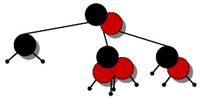
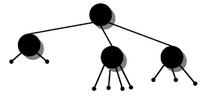
deÞned.

**Claim**

Any node with height *h* has black-height ≥ *h/*2.

The subtree rooted at any node *x* contains ≥ 2bh*(x)* − 1 internal nodes

black-height ≤ 2 lg*(n* + 1*)*.

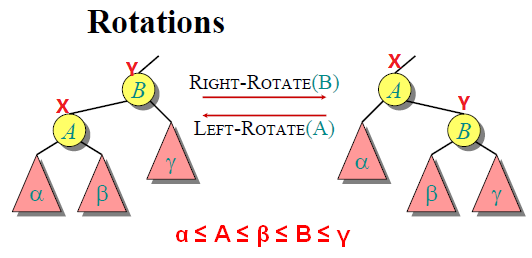
   

### Modifying operations

• the operation itself,

• color changes,

• restructuring the links of the tree via ***“rotations”***.



#### LEFT-ROTATE*(T, x)*

*y* ← *right*[*x*] //Set y. assumes right[x] = nil[T ], and root.s parent is *nil*[*T* ].

*right*[*x*] ← *left*[*y*] // Turn y.s left subtree into x.s right subtree.1

**if** *left*[*y*] ≠ *nil*[*T* ]

**then** *p*[*left*[*y*]] ← *x* //2

*p*[*y*] ← *p*[*x*] //Link x.s parent to y.3

**if** *p*[*x*] = *nil*[*T* ]

**then** *root*[*T* ] ← *y*

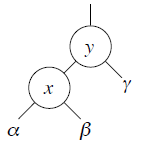
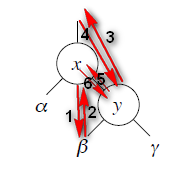
**else if** *x* = *left*[*p*[*x*]] //4

**then** *left*[*p*[*x*]] ← *y* //4

**else** *right*[*p*[*x*]] ← *y* //4

*left*[*y*] ← *x* //5

*p*[*x*] ← *y* //6



#### Insertion *O( lgn )*

**// 先按二叉树方式插入结点，再作调整**

**RB-INSERT*(T, z)***

*y* ← *nil*[*T* ]

*x* ← *root*[*T* ]

**while** *x* ≠ *nil*[*T* ]

**do** *y* ← *x*

**if** *key*[*z*] *< key*[*x*]

**then** *x* ← *left*[*x*]

**else** *x* ← *right*[*x*]

*p*[*z*] ← *y*

**if** *y* = *nil*[*T* ]

**then** *root*[*T* ] ← *z*

**else if** *key*[*z*] *< key*[*y*]

**then** *left*[*y*] ← *z*

**else** *right*[*y*] ← *z*

*left*[*z*] ← *nil*[*T* ]

*right*[*z*] ← *nil*[*T* ]

*color*[*z*] ← RED **//默认插入红色结点**

RB-INSERT-FIXUP*(T, z)* **//调整**

**• RB-INSERT ends by coloring the new node z red.**

**• Then it calls RB-INSERT-FIXUP because we could have violated a red-black property.**

At the start of each iteration of the loop,

1. Node *x* is red.
2. If *p*[*x*] is the root, then *p*[*x*] is black.
3. If there is a violation of the red-black properties, there is at most one violation, and it is of either property 2 or property 4. If there is a violation of property 2, it occurs because *x* is the root and is red. If there is a violation of property 4, it occurs because both *x* and *p*[*x*] are red.

In all three cases, *x*'s grandparent *p*[*p*[*x*]] is black, since its parent *p*[*x*] is red, and property 4 (If a node is red, then both its children are black )is violated only between *x* and *p*[*x*].

it never performs more than two rotations, since the **while** loop terminates if case 2 or case 3 is executed

**RB-INSERT-FIXUP*(T, x)***

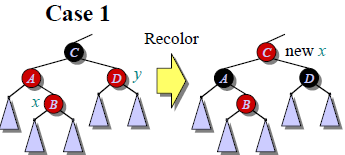
// p[x] is red,it cannot be the root. p[p[x]] exists .the loop terminates when p[x] is black

**while** *color*[*p*[*x*]] = RED

**do if** *p*[*x*] = *left*[*p*[*p*[*x*]]]

**then** *y* ← *right*[*p*[*p*[*x*]]] //x's uncle

//Case 1: z's uncle y is red

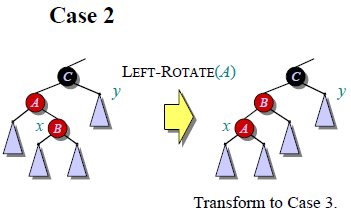
**if** *color*[*y*] = RED

**then** *color*[*p*[*x*]] ← BLACK //Case 1

*color*[*y*] ← BLACK // Case 1

*color*[*p*[*p*[*x*]]] ← RED //Case 1

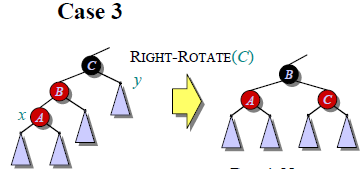
*x* ← *p*[*p*[*x*]] // Case 1 The pointer x moves up two levels in the tree

**else if** *x* = *right*[*p*[*x*]]

**then** *x* ← *p*[*x*] //Case 2

LEFT-ROTATE*(T, x)* // Case 2

*color*[*p*[*x*]] ← BLACK // Case 3

*color*[*p*[*p*[*x*]]] ← RED // Case 3

RIGHT-ROTATE*(T, p*[*p*[*x*]]*)* // Case 3

**else** (same as **then** clause with .right. and .left. exchanged)

*color*[*root*[*T* ]] ← BLACK // The root is black

x's uncle y is red: Case 1 color changes

x's uncle y is black--x is a right child: Case 2 rotation to case 3

x is a left child: Case 3 color changes and a rotation

#### Deletion *O( lgn )*

**RB-DELETE(*T*, *z*)**

1 **if** *left*[*z*] = *nil*[*T*] or *right*[*z*] = *nil*[*T*]

2 **then** *y* ← *z*

3 **else** *y* ← TREE-SUCCESSOR(*z*)

4 **if** *left*[*y*] ≠ *nil*[*T*]

5 **then** *x* ← *left*[*y*]

6 **else** *x* ← *right*[*y*]

7 *p*[*x*] ← *p*[*y*]

8 **if** *p*[*y*] = *nil*[*T*]

9 **then** *root*[*T*] ← *x*

10 **else if** *y* = *left*[*p*[*y*]]

11 **then** *left*[*p*[*y*]] ← *x*

12 **else** *right*[*p*[*y*]] ← *x*

13 **if** *y* ≠ *z*

14 **then** *key*[*z*] ← *key*[*y*]

15 copy *y*'s satellite data into *z*

16 **if** *color*[*y*] = BLACK

17 **then** RB-DELETE-FIXUP(*T*, *x*)

18 **return** *y*

**TREE-DELETE*(T, z)***

**if** *left*[*z*] = NIL or *right*[*z*] = NIL

**then** *y* ← *z*

**else** *y* ← TREE-SUCCESSOR*(z)*

**if** *left*[*y*] ≠ NIL

**then** *x* ← *left*[*y*]

**else** *x* ← *right*[*y*]

**if** *x* ≠ NIL

**then** *p*[*x*] ← *p*[*y*]

**if** *p*[*y*] = NIL

**then** *root*[*T* ] ← *x*

**else if** *y* = *left*[*p*[*y*]]

**then** *left*[*p*[*y*]] ← *x*

**else** *right*[*p*[*y*]] ← *x*

**if** *y* ≠ *z*

**then** *key*[*z*] ← *key*[*y*]

copy y.s satellite data into z

**return** *y*

**differences :**

First:all references to NIL in TREE-DELETE are replaced by references to the sentinel *nil*[*T* ] in RB-DELETE.

Second: the test for whether *x* is NIL in line 7 of TREE-DELETE is removed, and the assignment *p*[*x*] ← *p*[*y*] is performed unconditionally in line 7 of RB-DELETE. Thus, if *x* is the sentinel *nil*[*T*], its parent pointer points to the parent of the spliced-out node *y*.

Third, a call to RB-DELETE-FIXUP is made in lines 16-17 if *y* is black. If *y* is red, the red-black properties still hold when *y* is spliced out

**Idea:** If *y* is black, any path containing *y* now has 1 fewer black node.

—— Correct by **giving *x* an .extra black**. Add 1 to count of black nodes on paths containing *x，x* is either ***doubly black*** (if *color*[*x*] = BLACK) or ***red & black*** (if *color*[*x*] =RED).

*Process——*Move the extra black up the tree until

• *x* points to a red & black node ⇒turn it into a black node,

• *x* points to the root ⇒just remove the extra black, or

• we can do certain rotations and recolorings and Þnish.

**RB-DELETE-FIXUP(*T*, *x*)**

1 **while** *x* ≠ *root*[*T*] and *color*[*x*] = BLACK

2 **do if** *x* = *left*[*p*[*x*]]

3 **then** *w* ← *right*[*p*[*x*]]

4 **if** *color*[*w*] = RED //*color*[*w*] = RED 🡪 *p*[*w*]=black

5 **then** *color*[*w*] ← BLACK // Case 1

6 *color*[*p*[*x*]] ← RED // Case 1

7 LEFT-ROTATE(*T*, *p*[*x*]) // Case 1

8 *w* ← *right*[*p*[*x*]] // Case 1

9 **if** *color*[*left*[*w*]] = BLACK and *color*[*right*[*w*]] = BLACK

10 **then** *color*[*w*] ← RED // Case 2

11 *x* ← *p*[*x*] // Case 2

12 **else if** *color*[*right*[*w*]] = BLACK

13 **then** *color*[*left*[*w*]] ← BLACK// Case 3

14 *color*[*w*] ← RED // Case 3

15 RIGHT-ROTATE(*T*, *w*) // Case 3

16 *w* ← *right*[*p*[*x*]] // Case 3

17 *color*[*w*] ← *color*[*p*[*x*]] // Case 4

18 *color*[*p*[*x*]] ← BLACK // Case 4

19 *color*[*right*[*w*]] ← BLACK // Case 4

20 LEFT-ROTATE(*T*, *p*[*x*]) // Case 4

21 *x* ← *root*[*T*] // Case 4

22 **else** (same as **then** clause with "right" and "left" exchanged)

23 *color*[*x*] ← BLACK

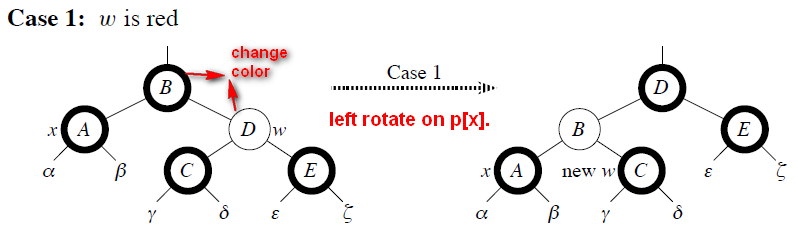
Within the **while** loop:

• *x* always points to a nonroot doubly black node.

• *w* is *x*.s sibling.

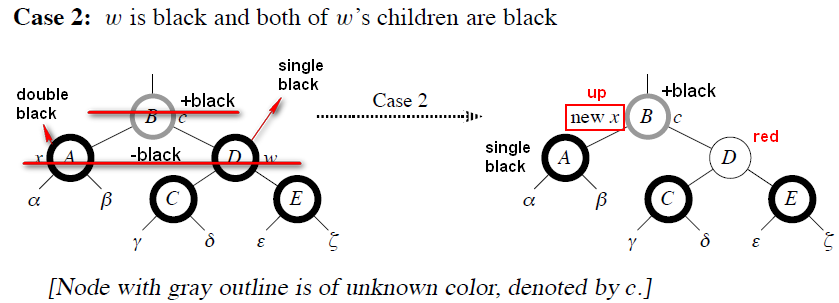
• *w* cannot be *nil*[*T* ], since that would violate property 5 at *p*[*x*].

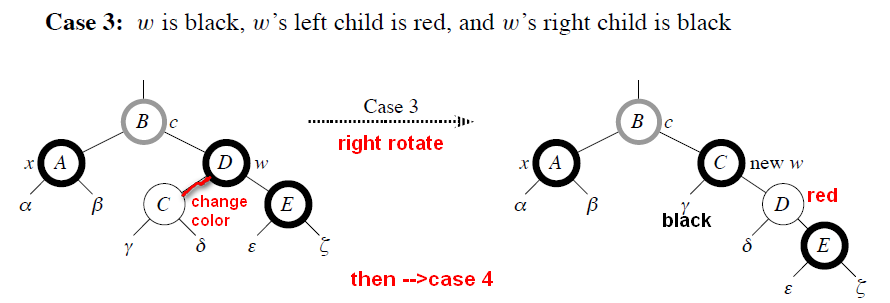
**Core：取一个doubly black node x的black结点，往上层结点分配，直到上层结点为red，改为black后，退出**



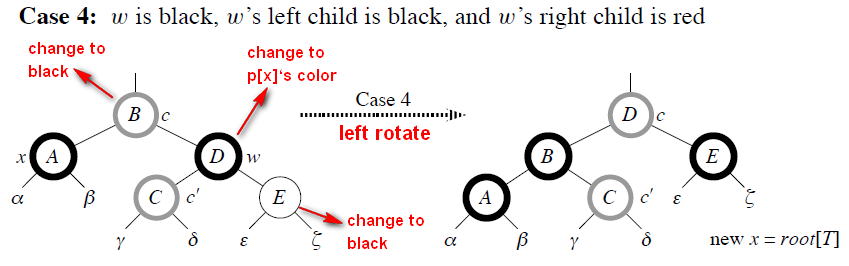
Case1：先保证兄弟结点w为black，then go immediately to case 2, 3, or 4

注意，以下B结点颜色不确定





**Case3：转到左black右red的情况**



## Augmenting Data Structures

### Dynamic order statistics

We want to support the usual dynamic-set operations from R-B trees, plus:

• OS-SELECT*(x, i )*: return pointer to node containing the *i* th smallest key of the

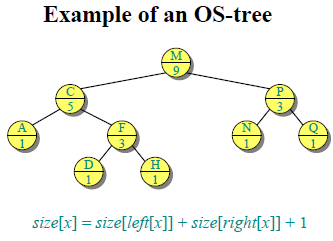
subtree rooted at *x*.

• OS-RANK*(T, x)*: return the rank of *x* in the linear order determined by an

inorder walk of *T* .

DeÞne for sentinel *size*[*nil*[*T* ]] = 0.

Then *size*[*x*] = *size*[*left*[*x*]] + *size*[*right*[*x*]] + 1.



**OS-SELECT*(x, i )*** *O(h)*

*r* ← *size*[*left*[*x*]]+1

**if** *i* = *r*

**then return** *x*

**elseif** *i < r*

**then return** OS-SELECT*(left*[*x*]*, i )*

**else return** OS-SELECT*(right*[*x*]*, i* − *r )*

Initial call: OS-SELECT*(root*[*T* ]*, i )*

**OS-RANK*(T, x)*** *O(h)*

*r* ← *size*[*left*[*x*]] + 1

*y* ← *x*

**while** *y*≠ *root*[*T* ]

**do if** *y* = *right*[*p*[*y*]]

**then** *r* ←*r* + *size*[*left*[*p*[*y*]]] + 1

*y* ← *p*[*y*]

**return** *r*

### Methodology for augmenting a data structure

1. Choose an underlying data structure.

2. Determine additional information to maintain.

3. Verify that we can maintain additional information for existing data structure

operations.

4. Develop new operations.

Don.t need to do these steps in strict order! Usually do a little of each, in parallel.

How did we do them for OS trees?

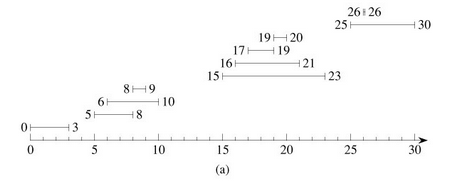
1. R-B tree.

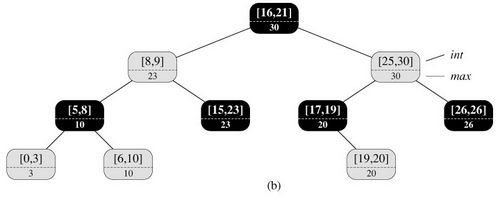
2. *size*[*x*].

3. Showed how to maintain *size* during insert and delete.

4. Developed OS-SELECT and OS-RANK.

### Interval Trees





*max*[*x*] = max(*high*[*int*[*x*]], *max*[*left*[*x*]], *max*[*right*[*x*]]).

**INTERVAL-SEARCH(*T*, *i*)**

1 *x* ← *root*[*T*]

2 **while** *x* ≠ *nil*[*T*] and *i* does not overlap *int*[*x*]

3 **do if** *left*[*x*] ≠ *nil*[*T*] and *max*[*left*[*x*]] ≥ *low*[*i*]

4 **then** *x* ← *left*[*x*]

5 **else** *x* ← *right*[*x*]

6 **return** *x*

## Dynamic Programming

The development of a dynamic-programming algorithm can be broken into a sequence of four steps.

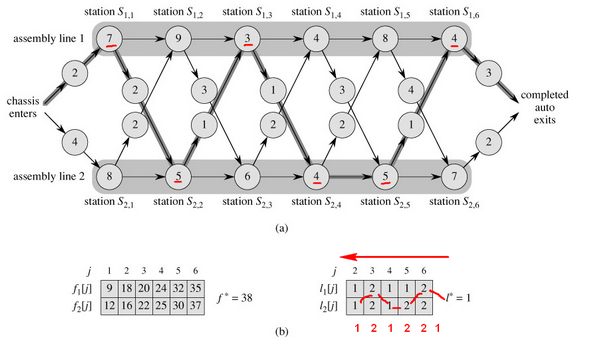
1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

**Dynamic-programming hallmark**

1.Optimal substructure :An optimal solution to a problem (instance) contains optimal solutions to subproblems.

2.Overlapping subproblems: A recursive solution contains a“small” number of distinct subproblems repeated many times.

### Assembly-line scheduling



After going through a station, can either

• stay on same line; no cost, or

• transfer to other line; cost after *Si, j* is *ti, j*. ( *j* = 1*, . . . , n*−1. No *ti,n* , because

the assembly line is done after *Si,n*.)

***Generally:*** An optimal solution to a problem (fastest way through *S*1*, j* ) contains

within it an optimal solution to subproblems (fastest way through *S*1*, j*−1 or *S*2*, j*−1).

***Goal:*** fastest time to get all the way through = *f* ∗.

*f* ∗ = min*( f*1[*n*] + *x*1*, f*2[*n*] + *x*2*)*

*f*1[1] = *e*1 + *a*1*,*1

*f*2[1] = *e*2 + *a*2*,*1

For *j* = 2*, . . . , n*:

*f*1[ *j* ] = min*( f*1[ *j* − 1] + *a*1*, j , f*2[ *j* − 1] + *t*2*, j*−1 + *a*1*, j )*

*f*2[ *j* ] = min*( f*2[ *j* − 1] + *a*2*, j , f*1[ *j* − 1] + *t*1*, j*−1 + *a*2*, j )*

***fi* [ *j* ] ：从起点到 i 工序线的第 j 个 station的最短时长**

• *li* [ *j* ] = line # (1 or 2) whose station *j* − 1 is used in fastest way through *Si, j* .

***li* [ *j* ]用于记录上一个最优位置是在line1 or line2**

• **In other words *Sli* [ *j* ]*, j*−1 precedes *Si, j* .**

• DeÞned for *i* = 1*,* 2 and *j* = 2*, . . . , n*.

• *l*∗ = line # whose station *n* is used.

#### Pseudocode

a:工地时间，t：转换时间，e：进入时间 x：出料时间，n：工地数

**FASTEST-WAY(*a*, *t*, *e*, *x*, *n*)** *Θ(n).*

1 *f*1[1] ← *e*1 + *a*1,1

2 *f*2[1] ←*e*2 + *a*2,1

3 for *j* ← 2 to *n //顺序2— n，都取子问题最优*

4 do if *f*1[*j* - 1] + *a*1,*j* ≤ *f*2[*j* - 1] + *t*2,*j*-1 + *a*1,*j*

5 then *f*1[*j*] ← *f*1[*j* - 1] + *a*1, *j*

6 *l*1[*j*] ← 1

7 else *f*1[*j*] ← *f*2[*j* - 1] + *t*2,*j*-1 + *a*1,*j*

8 *l*1[*j*] ← 2

9 if *f*2[*j* - 1] + *a*2,*j* ≤ *f*1[*j* - 1] + *t*1,*j*-1 + *a*2,*j*

10 then *f*2[*j*] ← *f*2[*j* - 1] + *a*2,*j*

11 *l*2[*j*] ← 2

12 else *f*2[*j*] ∞ *f*1[*j* - 1] + *t*1,*j*-1 + *a*2,*j*

13 *l*2[*j*] ← 1

14 if *f*1[*n*] + *x*1 ≤ *f*2[*n*] + *x*2 *//到出口，标记最优*

15 then *f*\* = *f*1[*n*] + *x*1

16 *l*\* = 1

17 else *f*\* = *f*2[*n*] + *x*2

18 *l*\* = 2

**PRINT-STATIONS(*l*, *n*)**

1 *i* ← *l*\*

2 print "line " *i* ", station " *n*

3 **for** *j* ← *n* **downto** 2

4 **do** *i* ← *li*[*j*]

5 print "line " *i* ", station " *j* - 1

### Matrix-chain multiplication

计算矩阵*A*1 *A*2*A*3相乘，假设维度分别为 10 × 100, 100 × 5, 5 × 50

1. ((*A*1 *A*2) *A*3), we perform 10 · 100 · 5 = 5000 scalar multiplications to compute the 10 × 5 matrix product *A*1 *A*2, plus another 10 · 5 · 50 = 2500 scalar multiplications to multiply this matrix by *A*3, for a total of **7500** scalar multiplications.
2. (*A*1 (*A*2 *A*3)), we perform 100 · 5 · 50 = 25,000 scalar multiplications to compute the 100 × 50 matrix product *A*2 *A*3, plus another 10 · 100 · 50 = 50,000 scalar multiplications to multiply *A*1 by this matrix, for a total of **75,000** scalar multiplications. Thus, computing the product according to the first parenthesization is 10 times faster.

结论：矩阵相乘的顺序不同，需要计算的次数也不同——确定最优计算次序

for 1 ≤ *i* ≤ *j* ≤ *n*. Let *m*[*i*, *j*] be the minimum number of scalar multiplications needed to compute the matrix *Ai‥j*; for the full problem, the cost of a cheapest way to compute *A*1*‥n* would thus be *m*[1, *n*].

Let us assume that the optimal parenthesization splits the product *Ai* *Ai*+1 *Aj* between *Ak* and *Ak*+1, where *i* ≤ *k* < *j*. Then, *m*[*i*, *j*] is equal to the minimum cost for computing the subproducts *Ai‥k* and *Ak*+1*‥j*, plus the cost of multiplying these two matrices together. Recalling that each matrix *Ai* is *pi*-1 × *pi*, we see that computing the matrix product *Ai‥k* *Ak*+1*‥j* takes *pi*-1 *pk* *pj* scalar multiplications. Thus, we obtain



There are only *j* - *i* possible values for *k*, however, namely *k* = *i*, *i* + 1, ..., *j* - 1. Since the optimal parenthesization must use one of these values for *k* : one problem for each choice of *i* and *j* satisfying 1 ≤ *i* ≤ *j* ≤ *n*, or in all. A recursive algorithm may encounter each subproblem many times in different branches of its recursion tree. So we use m[I ,j ] to stort the value we alrealy computed

#### dynamic-programming （bottom-up） algorithm

M[ I j ] 用于记录矩阵i到j最少的计算次数，可为更长的矩阵m[i,b] b>j 服务，s[I,j]记录i到j中的分歧点，最优分歧点有[ I , s[I,j] ] , [ s[I,j]+1 , j ]最后打印最优决策时，可使用递归

**MATRIX-CHAIN-ORDER(*p*) *O*(*n*3)**

1 *n* ← *length*[*p*] – 1 *//****结点个数***

2 **for** *i* ← 1 **to** *n //初始化*

3 **do** *m*[*i*, *i*] ← 0

4 **for** *l* ← 2 **to** *n* *//l is the chain length.从下到上*

5 **do for** *i* ← 1 **to** *n* - *l* + 1 *//起始位—终止位*

6 **do** *j* ← *i* + *l* - 1

7 *m*[*i*, *j*] ← ∞ *//默认最新最长长度为无穷*

8 **for** *k* ← *i* **to** *j* - 1

9 **do** *q* ← *m*[*i*, *k*] + *m*[*k* + 1, *j*] + *pi*-1 *pkpj*

10 **if** *q* < *m*[*i*, *j*] *//依靠子记录，寻找并记录最优子方案*

11 **then** *m*[*i*, *j*] ← *q*

12 *s*[*i*, *j*] ← *k*

13 **return** *m* and *s*

**PRINT-OPTIMAL-PARENS(*s*, *i*, *j*)**

1 **if** *i* = *j*

2 **then** print "*A*"*i*

3 **else** print "("

4 PRINT-OPTIMAL-PARENS(*s*, *i*, *s*[*i*, *j*])

5 PRINT-OPTIMAL-PARENS(*s*, *s*[*i*, *j*] + 1, *j*)

6 print ")"

#### recursive algorithm （inefficient）

用递归解决，重复计算子问题，低效 top-down

The recursive algorithm, must repeatedly resolve each subproblem each time it reappears in the recursion tree.

RECURSIVE-MATRIX-CHAIN(*p*, *i*, *j*) Ω(2*n*)-

1 **if** *i* = *j*

2 **then return** 0

3 *m*[*i*, *j*] ← ∞

4 **for** *k* ← *i* **to** *j* - 1

5 **do** *q* ← RECURSIVE-MATRIX-CHAIN(*p*, *i*, *k*)

+ RECURSIVE-MATRIX-CHAIN(*p*,*k* + 1, *j*)

+ *pi*-1 *pk* *pj*

6 **if** *q* < *m*[*i*, *j*]

7 **then** *m*[*i*, *j*] ← *q*

8 **return** *m*[*i*, *j*]

#### memoized algorithm top-down

**MEMOIZED-MATRIX-CHAIN(*p*)** *O*(*n*3)

1 *n* ← *length*[*p*] - 1

2 **for** *i* ← 1 **to** *n*

3 **do for** *j* ← *i* **to** *n*

4 **do** *m*[*i*, *j*] ← ∞

5 **return** LOOKUP-CHAIN(*p*, 1, *n*)

**LOOKUP-CHAIN(*p*, *i*, *j*)**

1 **if** *m*[*i*, *j*] < ∞

2 **then return** *m*[*i*, *j*]

3 **if** *i* = *j*

4 **then** *m*[*i*, *j*] ← 0

5 **else for** *k* ← *i* **to** *j* - 1

6 **do** *q* ← LOOKUP-CHAIN(*p*, *i*, *k*)

+ LOOKUP-CHAIN(*p*,*k* + 1, *j*) + *pi*-1 *pk* *pj*

7 **if** *q* < *m*[*i*, *j*]

8 **then** *m*[*i*, *j*] ← *q*

9 **return** *m*[*i*, *j*]

### Elements of dynamic programming

optimization dynamic programming problem: optimal substructure and overlapping subproblems

#### Optimal substructure

characterize the structure of an optimal solution: Whenever a problem exhibits optimal substructure, it is a good clue that dynamic programming might apply. It also might mean that a greedy strategy applies.

You will find yourself following a common pattern in discovering optimal substructure:

1. You show that a solution to the problem consists of **making a choice,** such as choosing a preceding assembly-line station or choosing an index at which to split the matrix chain. Making this choice leaves one or more subproblems to be solved.
2. You suppose that for a given problem, **you are given the choice that leads to an optimal solution.** You do not concern yourself yet with how to determine this choice. You just assume that it has been given to you.
3. Given this choice, you determine which **subproblems** ensue and how to best characterize the resulting space of subproblems.
4. You show that the solutions to the subproblems used within the optimal solution to the problem must themselves be optimal by using a "cut-and-paste" technique. You do so by supposing that each of the subproblem solutions is not optimal and then deriving a contradiction. In particular, by "cutting out" the nonoptimal subproblem solution and "pasting in" the optimal one, you show that you can get a better solution to the original problem, thus contradicting your supposition that you already had an optimal solution. If there is more than one subproblem, they are typically so similar that the cut-and-paste argument for one can be modified for the others with little effort.

Optimal substructure varies across problem domains in two ways:

1. how many subproblems are used in an optimal solution to the original problem, (assembly-line scheduling:1, *either* through *S*1, *j* -1 *or* *S*2, *j* -1; Matrix-chain multiplication:2, *Ai* *Ai*+1 *Ak* and parenthesizing *Ak*+1 *Ak*+2 *Aj*)
2. how many choices we have in determining which subproblem(s) to use in an optimal solution.

dynamic programming:first finding optimal solutions to subproblems and then making a choice

greedy algorithms： first make a choice-the choice that looks best at the time-and then solve a resulting subproblem.

#### Overlapping subproblems

Dynamic-programming algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed, using constant time per lookup

In contrast, a problem for which a divide-and-conquer approach is suitable usually generates brand-new problems at each step of the recursion

[反例，重新计算子问题而不是储存子问题再查找](#_recursive_algorithm_（inefficient）)

#### Reconstructing an optimal solution

As a practical matter, we **often store which choice we made in each subproblem in a table** so that we do not have to reconstruct this information from the costs that we stored

#### Memoization

There is a variation of dynamic programming that often offers the efficiency of the usual dynamic-programming approach while maintaining a **top-down strategy**. The idea is to ***memoize*** the natural, but inefficient, **recursive** algorithm. As in ordinary dynamic programming, we maintain a table with subproblem solutions, but the control structure for filling in the table is more like the recursive algorithm

After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

[Here is a memoized version of RECURSIVE-MATRIX-CHAIN:](#_memoized_algorithm_)

In general practice, if all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms a top-down memoized algorithm by a constant factor, because there is no overhead for recursion and less overhead for maintaining the table. Moreover, there are some problems for which the regular pattern of table accesses in the dynamic-programming algorithm can be exploited to reduce time or space requirements even further. Alternatively, if some subproblems in the subproblem space need not be solved at all, the memoized solution（top-down） has the advantage of solving only those subproblems that are definitely required.

当所有子问题都至少需要解决一次时，bottom-up dynamic-programming algorithm usually outperforms a top-down memoized algorithm

当只需解决一部分子问题时，the memoized solution（top-down） has the advantage of solving only those subproblems that are definitely required.

### 15.4 Longest common subsequence

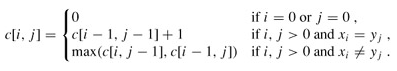
Formally, given a sequence *X* = 〈*x*1, *x*2, ..., *xm*〉, another sequence *Z* = 〈*z*1, *z*2, ..., *zk*〉 is a ***subsequence*** of *X* if there exists a strictly increasing sequence 〈*i*1,*i*2, ..., *ik*〉 of indices of *X* such that for all *j* = 1, 2, ..., *k*, we have *xij* = *zj* . For example, *Z* = 〈*B*, *C*, *D*, *B*〉 is a subsequence of *X* = 〈*A*, *B*, *C*, *B*, *D*, *A*, *B*〉 with corresponding index sequence 〈2, 3, 5, 7〉.

Optimal substructure of an LCS

Let *X* = 〈*x*1, *x*2, ..., *xm*〉 and *Y* = 〈*y*1, *y*2, ..., *yn*〉 be sequences, and let *Z* = 〈*z*1, *z*2, ..., *zk*〉 be any LCS of *X* and *Y*.

1. If *xm* = *yn*, then *zk* = *xm* = *yn* and *Zk*-1 is an LCS of *Xm*-1 and *Yn*-1.
2. If *xm* ≠ *yn*, then *zk* ≠ *xm* implies that *Z* is an LCS of *Xm*-1 and *Y*.
3. If *xm* ≠ *yn*, then *zk* ≠ *yn* implies that *Z* is an LCS of *X* and *Yn*-1.

define *c*[*i*, *j*] to be the length of an LCS of the sequences *Xi* and *Yj* .



#### Computing the length of an LCS

*b*[*i*, *j*] points to the table entry corresponding to the optimal subproblem solution chosen when computing *c*[*i*, *j*].

*c*[*m*, *n*] contains the length of an LCS of *X* and *Y*.

**LCS-LENGTH(*X*, *Y*) *O*(*mn*)**

1 *m* ← *length*[*X*]

2 *n* ← *length*[*Y*]

3 **for** *i* ← 1 **to** *m*

4 **do** *c*[*i*, 0] ← 0

5 **for** *j* ← 0 **to** *n*

6 **do** *c*[0, *j*] ← 0

7 **for** *i* ← 1 **to** *m*

8 **do for** *j* ← 1 **to** *n*

9 **do if** *xi* = *yj*

10 **then** *c*[*i*, *j*] ← *c*[*i* - 1, *j* - 1] + 1

11 *b*[*i*, *j*] ← "↖"

12 **else if** *c*[*i* - 1, *j*] ≥ *c*[*i*, *j* - 1]

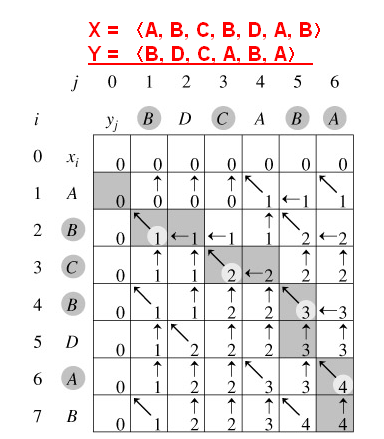
13 **then** *c*[*i*, *j*] ← *c*[*i* - 1, *j*]

14 *b*[*i*, *j*] ← "↑"

15 **else** *c*[*i*, *j*] ← *c*[*i*, *j* - 1]

16 *b*[*i*, *j*] ← ←

17 **return** *c* and *b*



initial invocation ：PRINT-LCS(*b*, *X*, *length*[*X*], *length*[*Y*]).

**PRINT-LCS(*b*, *X*, *i*, *j*)** *O*(*m* + *n*),

1 **if** *i* = 0 or *j* = 0

2 **then return**

3 **if** *b*[*i*, *j*] = "↖"

4 **then** PRINT-LCS(*b*, *X*, *i* - 1, *j* - 1)

5 print *xi*

6 **else if** *b*[*i*, *j*] = "↑"

7 **then** PRINT-LCS(*b*, *X*, *i* - 1, *j*)

8 **else** PRINT-LCS(*b*, *X*, *i*, *j* - 1)

#### Recursive algorithm for LCS & Memoization algorithm

**LCS(x, y, i, j) Θ(mn)**

If c[i, j] = NIL

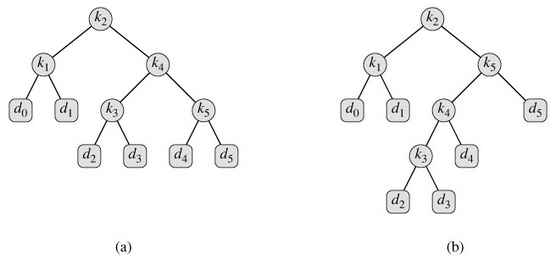
then if x[i] = y[j]

then c[i, j] ←LCS(x, y, i–1, j–1) + 1

else c[i, j] ←max{LCS(x, y, i–1, j), LCS(x, y, i, j–1)}

### 15.5 Optimal binary search trees

*K* = 〈*k1*, *k*2, ..., *kn*〉 of *n* distinct keys in sorted order (so that *k*1 < *k*2 < ··· < *kn*), *n* + 1 "dummy keys" *d*0, *d*1, *d*2, ..., *dn* representing values not in *K*. *d0* < *k*1, *dn* > *kn*, Each key *ki* is an internal node, and each dummy key *di* is a leaf. Every search is either successful (finding some key *ki*) or unsuccessful (finding some dummy key *di*), and so we have



For a given set of probabilities, our goal is to construct a binary search tree whose expected search cost is smallest

***Observations:***

• Optimal BST might not have smallest height.

• Optimal BST might not have highest-probability key at root.

DeÞne *e*[*i, j* ] = expected search cost of optimal BST for *ki , . . . , k j* . Ultimately, we wish to compute *e*[1, *n*].

When a subtree becomes a subtree of a node:

• Depth of every node in subtree goes up by 1.

• Expected search cost increases by



Noting that

i->j的cost由根节点+左*w*(*i*, *r* - 1)+右*w*(*r* + 1, *j*)：

*w*(*i*, *j*) = *w*(*i*, *r* - 1) + *pr* + *w*(*r* + 1, *j*),

*e*[*i*, *j* ] = *pr* + (*e*[*i*, *r* - 1] + *w*(*i*, *r* - 1)) + (*e*[*r* + 1, *j*] + *w*(*r* + 1, *j*)).



**OPTIMAL-BST(*p*, *q*, *n*)** Θ(*n*3)

1 **for** *i* ← 1 **to** *n* + 1

2 **do** *e*[*i*, *i* - 1] ← *qi*-1

3 *w*[*i*, *i* - 1] ← *qi*-1

4 **for** *l* ← 1 **to** *n*

5 **do for** *i* ← 1 **to** *n* - *l* + 1

6 **do** *j* ← *i* + *l* - 1

7 *e*[*i*, *j*] ← ∞

8 *w*[*i*, *j*] ← *w*[*i*, *j* - 1] + *pj* + *qj*

9 **for** *r* ← *i* **to** *j*

10 **do** *t* ← *e*[*i*, *r* - 1] + *e*[*r* + 1, *j*] + *w*[*i*, *j*]

11 **if** *t* < *e*[*i*, *j*]

12 **then** *e*[*i*, *j*] ← *t*

13 *root*[*i*, *j*] ← *r*

14 return *e* and *root*

**Construct an optimal solution**

**CONSTRUCT-OPTIMAL-BST(root)**

r ← root[1, n]

print .k.r .is the root.

CONSTRUCT-OPT-SUBTREE(1, r − 1, r, .left., root)

CONSTRUCT-OPT-SUBTREE(r + 1, n, r, .right., root)

CONSTRUCT-OPT-SUBTREE(i, j, r, dir, root)

if i ≤ j

then t ← root[i, j ]

print .k.t .is. dir .child of k.r

CONSTRUCT-OPT-SUBTREE(i, t − 1, t, .left., root)

CONSTRUCT-OPT-SUBTREE(t + 1, j, t, .right., root)