Algorithm Design and Analysis

David N. JANSEN, Bohua ZHAN
名 姓

算法设计与分析

詹博华，杨大卫

This week's content

- Today Wednesday:
  - Chapter 9: Selection
  - Chapter 11: Hashing (start)
    - 11.1–11.2
  - Exercises
- Tomorrow Thursday:
  - Exercise solutions
  - Chapter 11: Hashing (end)
    - 11.3–11.5

这周的内容

- 今天周三：
  - 第9章：
  - 第11章：
    - 11.1–11.2
  - 练习
- 明天周四：
  - 练习题解答
  - 第11章：
    - 11.3–11.5

Exercises

练习

7.3-1

- Why do we analyze the expected running time of a randomized algorithm and not its worst-case running time?

为什么我们分析随机化算法的期望运行时间，而不是其最坏运行时间呢？

7.3-1

- We use randomization to get an algorithm that has a good running time with high probability.

- If we need an algorithm with a good running time in the worst case, we should not use a randomized algorithm.

Problem 7-4

(Stack depth for Quicksort)
The QUICKSORT algorithm of Section 7.1 contains two recursive calls to itself. After QUICKSORT calls PARTITION, it recursively sorts the left subarray and then it recursively sorts the right subarray. The second recursive call in QUICKSORT is not really necessary; we can avoid it by using an iterative control structure.
This technique, called tail recursion, is provided automatically by good compilers.

（快速排序的栈深度） 7.1 节中的 QUICKSORT 算法包含了两个对其自身的递归调用。在调用 PARTITION 后，QUICKSORT 分别递归调用了左边的子数组和右边的子数组。QUICKSORT 中的第二个递归调用并不是必须的。我们可以用一个循环控制结构来代替它。这一技术称为**尾递归**，好的编译器都提供这一功能。考虑下面这个版本的快速排序，它模拟了尾递归情况：

Problem 7-4

```
TAIL-RECURSIVE-QUICKSORT(A, p, r)
1   while p < r
2       // Partition and sort left subarray.
3       q = PARTITION(A, p, r)
4       TAIL-RECURSIVE-QUICKSORT(A, p, q − 1)
5       p = q + 1
```

Problem 7-4

a. Argue that TAIL-RECURSIVE-QUICKSORT(A, 1, A.length) correctly sorts array A.

a. 证明：TAIL-RECURSIVE-QUICKSORT(A，1，$A.length$)能正确地对数组 A 进行排序。

Compilers usually execute recursive procedures by using a stack that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. Upon calling a procedure, its information is pushed onto the stack; when it terminates, its information is popped. Since we assume that array parameters are represented by pointers, the information for each procedure call on the stack requires O(1) stack space. The stack depth is the maximum amount of stack space used at any time during a computation.

编译器通常使用**栈**来存储递归执行过程中的相关信息，包括每一次递归调用的参数等。最新调用的信息存在栈的顶部，而第一次调用的信息存在栈的底部。当一个过程被调用时，其相关信息被**压入**栈中；当它结束时，其信息则被**弹出**。因为我们假设数组参数是用指针来指示的，所以每次过程调用只需要 $O(1)$ 的栈空间。**栈深度**是在一次计算中会用到的栈空间的最大值。

Problem 7-4 a.

- We should prove by induction over *r*–*p*: TAIL-RECURSIVE-QUICKSORT(*A*, *p*, *r*) sorts the array *A*[*p*] ... *A*[*r*] correctly.

- Induction basis: p = r, so r–p = 0. Then TAIL-RECURSIVE-QUICKSORT(*A*, *p*, *r*) terminates without doing anything. This is correct because a 1-element array is always sorted.

Problem 7-4 a.

- Induction step:
  Let $p_0 := p$ at the initial call of TAIL-RECURSIVE-QUICKSORT.

- We use the following loop invariant for the while loop:
  $A[p_0]$ ... $A[p-1]$ is sorted. And if $p_0 < p$, $A[p-1]$ is $\leq$ all elements in $A[p]$ ... $A[r]$.

- Initialisation: trivial because $p \leq p_0$.

Problem 7-4 a.

- Maintenance of loop invariant:
  We use the induction hypothesis to prove that after line 4, $A[p] \ldots A[q-1]$ is also sorted.
  And we use the specification of PARTITION to prove that
  $A[q] \geq$ every element of $A[p] \ldots A[q-1]$,
  $A[q] \leq$ every element of $A[q+1] \ldots A[r]$.
  Both together imply that at the end of the loop iteration, $A[p_0] \ldots A[q]$ is sorted, and
  $A[q] \leq$ every element of $A[q+1] \ldots A[r]$.
  This is the loop invariant for the next iteration!

Problem 7-4 a.

- Termination: At the end of the loop, we get from the loop invariant that A[p_0] ... A[r–1] is sorted. And if $r > p_0$, $A[r–1] \leq A[r]$.
  This implies that A[p_0] ... A[r] is sorted.

Problem 7-4

b. Describe a scenario in which TAIL-RECURSIVE-QUICKSORT's stack depth is Θ(n) on an n-element input array.

c. Modify the code for TAIL-RECURSIVE-QUICKSORT so that the worst-case stack depth is O(log n). Maintain the O(n log n) expected running time of the algorithm.

b. 请描述一种场景，使得针对一个包含 n 个元素数组的 TAIL-RECURSIVE-QUICKSORT 的栈深度是 $\Theta(n)$。

c. 修改 TAIL-RECURSIVE-QUICKSORT 的代码，使其最坏情况下栈深度是 $\Theta(\lg n)$，并且能够保持 $O(n \lg n)$ 的期望时间复杂度。

Problem 7-4 b.

- PARTITION always uses $A[r]$ as pivot.
  If this is the largest element, then the recursive call in line 4 works on $A[p]$ ... $A[r–1]$.

- This leads to a recurrence formula for the stack space used:
  $S(n) = S(n–1) + \Theta(1)$
  $S(1) = 0$

- The recurrence has solution $S(n) = \Theta(n)$.

Problem 7-4 c.

```
TAIL-RECURSIVE-QUICKSORT(array A, p, r)
// Specification: sorts the array A[p] ... A[r]
while p < r
        q = PARTITION(A, p, r)
        if q–p < r–q
                TAIL-RECURSIVE-QUICKSORT(A, p, q–1)
                p = q+1
        else
                TAIL-RECURSIVE-QUICKSORT(A, q+1, r)
                r = q–1
```

Loop invariant:
$A[p_0]$ ... $A[p–1]$ is sorted.
If $p_0 < p$, then $A[p–1] \leq$ every element in $A[p]$ ... $A[r]$.

$A[r+1]$ ... $A[r_0]$ is sorted.
If $r < r_0$, then $A[r+1] \geq$ every element in $A[p]$ ... $A[r]$.

8.2-2

- Prove that COUNTING-SORT is stable.

试证明 COUNTING-SORT 是稳定的。

Counting sort

COUNTING-SORT(*A*, *B*, *k*)
Let *C*[0 ... k] be a new array
Initialize every element of *C* to 0
for *j* = 1 **to** *A.length*
    C[A[j]] = C[A[j]] + 1
for *i* = 1 **to** *k*
    C[*i*] = *C*[*i*] + *C*[*i*–1]
for *j* = *A.length* **downto** 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] – 1

- input: array A, containing elements in {0, ..., k}
- output: array *B*
- additional array *C* is used to count how many elements are ≤ a value.

8.2-2

- The only loop where elements are copied to array B is the last loop. Note that array A is never changed.

- If two elements are equal, say $A[i_1] = A[i_2]$ with $i_1 < i_2$, then their positions will be determined by $C[A[i_2]]$. The values in array C only diminish over time in the last loop. Therefore, when $j = i_2$, the value of $C[A[j]]$ is larger than the value of $C[A[j]]$ when $j = i_1$.

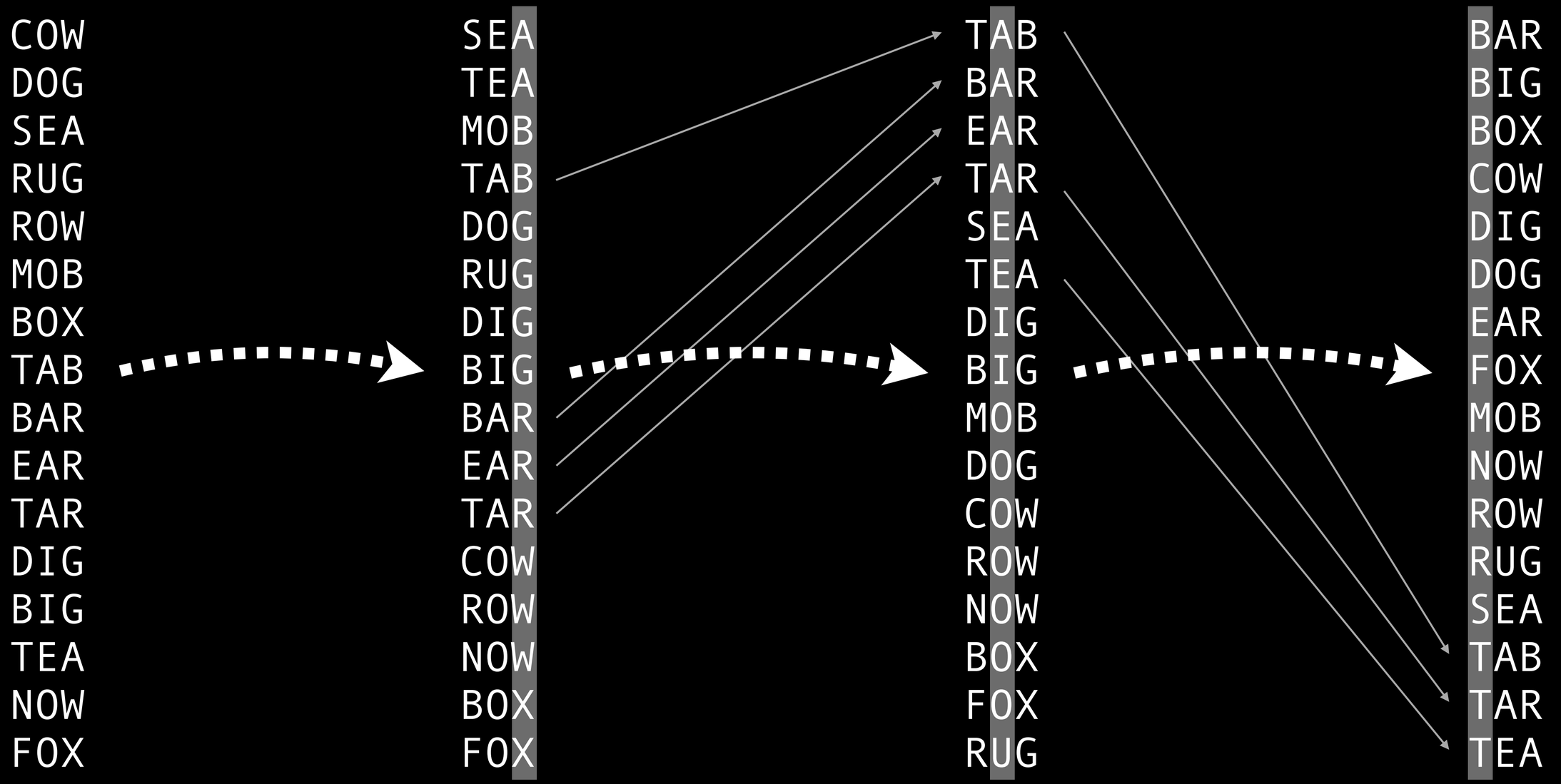
8.3-1

- Using Figure 8.3 as a model, illustrate the operation of radix sort on the following list of English words:

参照图 8-3 的方法，说明 RADIX-SORT 在下列英文单词上的操作过程：

COW, DOG, SEA, RUG, ROW,
MOB, BOX, TAB, BAR, EAR,
TAR, DIG, BIG, TEA, NOW, FOX.

8.3-1


COW
DOG
SEA
RUG
ROW
MOB
BOX
TAB
BAR
EAR
TAR
DIG
BIG
TEA
NOW
FOX
SEA
TEA
MOB
TAB
DOG
RUG
DIG
BIG
BAR
EAR
TAR
COW
ROW
NOW
BOX
FOX
TAB
BAR
EAR
TAR
SEA
TEA
DIG
BIG
MOB
DOG
COW
ROW
NOW
BOX
FOX
RUG
BAR
BIG
BOX
COW
DIG
DOG
EAR
FOX
MOB
NOW
ROW
RUG
SEA
TAB
TAR
TEA

9.1-1

Show that the second smallest of n elements can be found with $n + \lceil \lg n \rceil - 2$ comparisons in the worst case. (*Hint:* Also find the smallest element.)

证明：在最坏情况下，找到 n 个元素中第二小的元素需要 $n+\lceil \lg n \rceil - 2$ 次比较。（提示：可以同时找最小元素。）

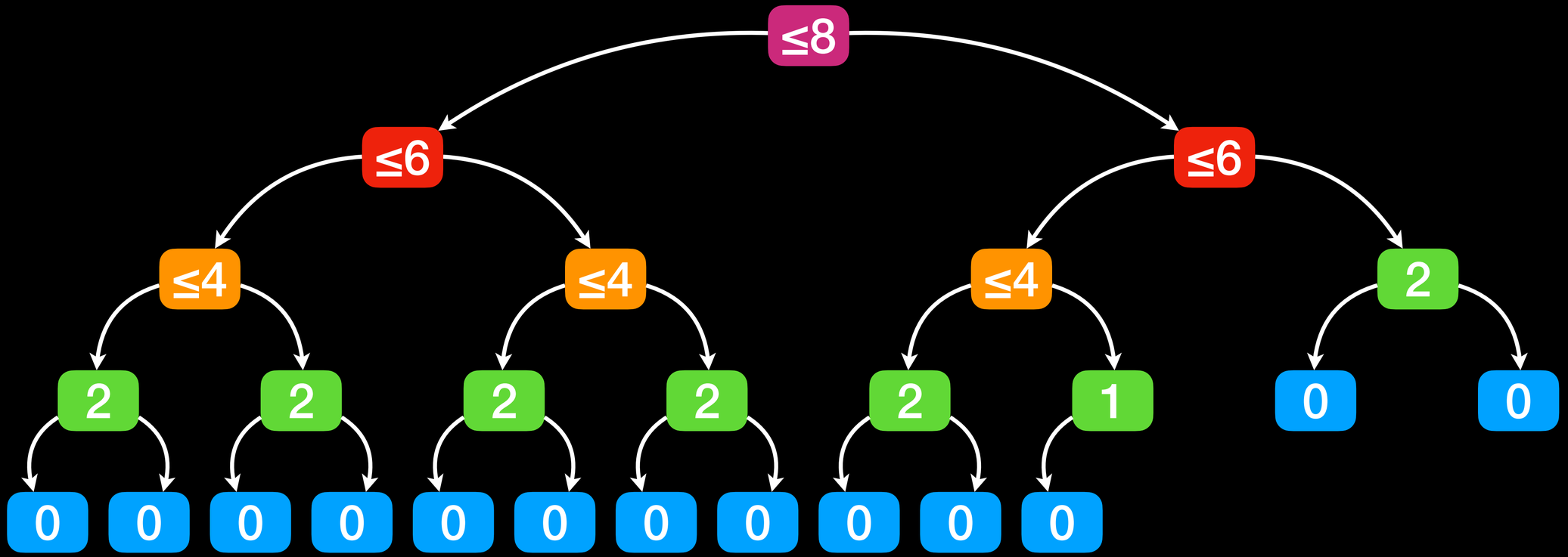
9.1-1

- First idea: Heapsort
  Construct a min-heap and extract the smallest two elements.
- Requires time $O(n)$ to construct the heap, and time $O(\log n)$ to extract the smallest element.
- Problem: constructing the heap may require almost $4n$ comparisons.
  Correct complexity class, but still too large!

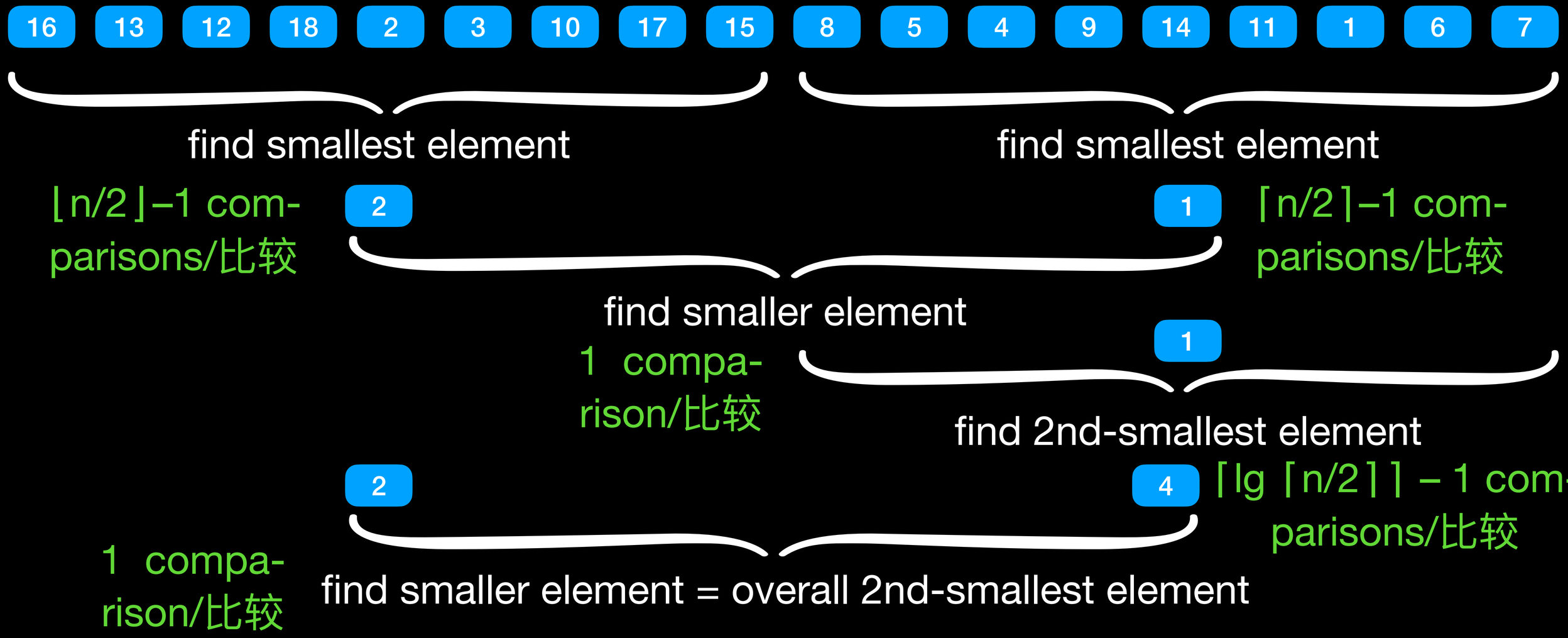
- 第一个想法：堆排序
  构造一个最小堆并提取最小的两个元素。
- 构建堆需要时间 $O(n)$，提取最小元素需要时间 $O(\log n)$。
- 问题：构建堆可能需要将近 $4n$ 个比较。
  正确的复杂性类，但仍然太大！

9.1-1

comparisons required to place the element correctly
when constructing the heap:



9.1-1


16 13 12 18 2 3 10 17 15 8 5 4 9 14 11 1 6 7
find smallest element
find smallest element
⌊n/2⌋–1 comparisons/比较
2
1
⌈n/2⌉–1 comparisons/比较
find smaller element
1 comparison/比较
1
find 2nd-smallest element
2
4
⌈lg ⌈n/2⌉⌉ – 1 comparisons/比较
1 comparison/比较
find smaller element = overall 2nd-smallest element

9.1-1

- I want to prove the following using strong induction over $n \geq 2$:
  The smallest element of an array with n elements can be found with $n–1$ comparisons, and if the smallest element has been found, the 2nd-smallest element can be found with $\lceil \lg n \rceil - 1$ additional comparisons.

- 我想用 $n \geq 2$ 上的强归纳法
  证明以下几点：
  具有n个元素的数组中的最小元素
  可以通过$n–1$的比较找到，
  如果找到了最小元素，则可以通过
  $\lceil \lg n \rceil - 1$ 额外的比较找到第二小元素。

9.1-1:
Induction basis

归纳法基础

- $n = 2$.
  The smallest element of a 2-element array can be found with 1 comparison, and to find the 2nd-smallest element we need 0 additional comparisons.
  This is obvious.

- n = 3.
  The smallest element of a 3-element array can be found with 2 comparisons, and to find the 2nd-smallest element we need 1 additional comparisons.
  This is easy.

9.1-1
Induction step

- Assume for all $2 \le m \le n$ that the smallest element of an array with m elements can be found with $m-1$ comparisons, and the 2nd-smallest element can be found with $\lceil \lg m \rceil - 1$ additional comparisons.
- We have to prove that the smallest element of an array with $n+1$ elements can be found with n comparisons, and the 2nd-smallest element can be found with $\lceil \lg (n+1) \rceil - 1$ additional comparisons.

归纳步骤

- 归纳假设：。。。
- 需要证明：。。。

9.1-1
Induction step

归纳步骤

- To be proven: The smallest element of an array with $n+1$ elements can be found with n comparisons.

- We split the array into two halves of size $\lfloor (n+1)/2 \rfloor$ and $\lceil (n+1)/2 \rceil$, and find their smallest elements with $\lfloor (n+1)/2 \rfloor - 1 + \lceil (n+1)/2 \rceil - 1 = n-1$ comparisons.
  Then we need one comparison to find the smaller of these two.
  Total n comparisons.

9.1-1
Induction step

归纳步骤

- To be proven: The 2nd-smallest element of an array with $n+1$ elements can be found with $\lceil \lg (n+1) \rceil - 1$ additional comparisons.

- In the half that contained the smallest element, we find the 2nd-smallest element. That requires at most $\lceil \lg \lceil (n+1)/2 \rceil \rceil - 1$ additional comparisons.
  Note that $\lceil \lg \lceil (n+1)/2 \rceil \rceil - 1 = \lceil \lg ((n+1)/2) \rceil - 1 = \lceil \lg (n+1) - 2 \rceil$.

- Then we need 1 comparison to compare this element with the smallest element of the other half. The smaller of the two is the 2nd-smallest element.

- Total we need $\lceil \lg (n+1) - 2 \rceil + 1 = \lceil \lg (n+1) \rceil - 1$ comparisons.

9.3-1

In the algorithm SELECT, the input elements are divided into groups of 5. Will the algorithm work in linear time if they are divided into groups of 7? Argue that SELECT does not run in linear time if groups of 3 are used.

在算法 SELECT 中，输入元素被分为每组 5 个元素。如果它们被分为每组 7 个元素，该算法仍然会是线性时间吗？证明：如果分成每组 3 个元素，SELECT 的运行时间不是线性的。

Worst-case $O(n)$

SELECT(array A, p, r, i)
// Specification: permute A such that $A[p+i-1]$ becomes the ith order statistic in $A[p]$ ... $A[r]$
$n = r - p + 1$
if $n = 1$ **then** **return** $A[p]$
for $j = 0$ **to** $\lfloor n/5 \rfloor - 1$
    INSERTION-SORT(A, $p+5j$, $p+5j+4$); $C[j] = A[p+5j+2]$
if n mod $5 > 0$ **then** INSERTION-SORT(A,$p+5\lfloor n/5 \rfloor$,r); $C[\lfloor n/5 \rfloor] = A[\lfloor (p+5\lfloor n/5 \rfloor + r)/2 \rfloor]$
} $O(n)$
SELECT(C, 0, $\lceil n/5 \rceil - 1$, $\lfloor n/10 \rfloor + 1$) $T(\lceil n/5 \rceil)$
q = PARTITION(A, p, r, but let pivot = $C[\lfloor n/10 \rfloor]$) $O(n)$
$k = (q - p)/step + 1$ // pivot is the kth-smallest element in $A[p]$ ... $A[r]$
if $i < k$ **then** SELECT(A, p, $q-1$, i)
else if $i > k$ **then** SELECT(A, $q+1$, r, $i-k$)
} $\leq T(0.7n)$

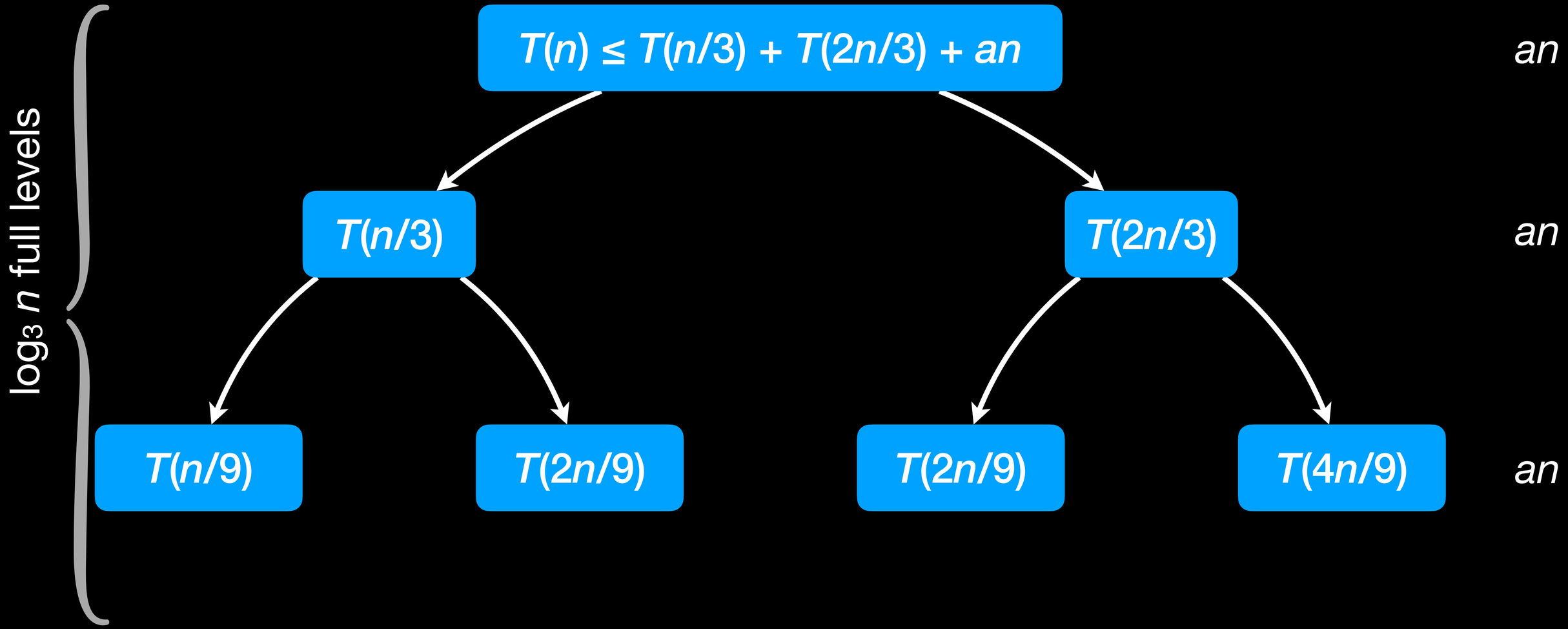
9.3-1: Groups of 3

- In finding the median-of-medians, there are $\lceil n/3 \rceil$ groups, so the first recursive call to SELECT uses time in $T(\lceil n/3 \rceil)$.
- The found median-of-medians ensures that:
  - in one group, there is 1 larger and 1 smaller elements.
  - In $\lceil \lceil n/3 \rceil /2 \rceil$ groups, there are 2 larger elements.
  - In $\lfloor \lceil n/3 \rceil /2 \rfloor$ groups, there are 2 smaller elements.
  - But in the small group, there might be up to 1 larger elements less and up to 1 smaller elements less.

9.3-1: Groups of 3

- In finding the median-of-medians, there are $\lceil n/3 \rceil$ groups, so the first recursive call to SELECT uses time in $T(\lceil n/3 \rceil)$.
- The found median-of-medians ensures that:
  - Overall, there are at least $2(\lceil \lceil n/3 \rceil /2 \rceil$ larger elements, and at least $2(\lfloor \lceil n/3 \rceil /2 \rfloor$ smaller elements.
- The worst-case second recursive call to SELECT uses time in something like $\leq T(2n/3)$.
- When drawing the recursion tree, we would get a time bound in $O(n \log n)$.

9.3-1: Groups of 3



9.3-1

- Groups of 7:
  In finding the median-of-medians, there are only $\lceil n/7 \rceil$ groups, so the first recursive call to SELECT only uses time in $T(\lceil n/7 \rceil)$.

- The found median-of-medians ensures that:
  - in one group, there are 3 larger and 3 smaller elements.
  - In $\lceil \lceil n/7 \rceil /2 \rceil$ groups, there are 4 larger elements.
  - In $\lfloor \lceil n/7 \rceil /2 \rfloor$ groups, there are 4 smaller elements.
  - But in the small group, there might be up to 3 larger elements less and up to 3 smaller elements less.

9.3-1

- Groups of 7:
  In finding the median-of-medians, there are only $\lceil n/7 \rceil$ groups, so the first recursive call to SELECT only uses time in $T(\lceil n/7 \rceil)$.

- The found median-of-medians ensures that:
  - There are at least $4(\lceil \lceil n/7 \rceil /2 \rceil$ larger elements and at least $4(\lceil \lceil n/7 \rceil /2 \rceil$ smaller elements.

- The worst-case second recursive call to SELECT uses time in something like $\leq T(5n/7)$.

- It appears that the recursion would be a little bit less deep (but perhaps INSERTION-SORT takes a little bit longer).

Hashing 散列表 / 哈希表

Dictionary

- (dynamic) dictionary. Operations:
  - insert
  - search
  - delete
- Example: identifiers (variable names) in a program:
  declare = insert,
  access = search,
  end of scope = delete.

- （动态的）字典，需要的操作：
  - insert
  - search
  - delete
- 例子：程序中的标识符（变量名）：
  声明 = insert,
  访问 = search,
  作用域的末尾 = delete.

How to implement a dictionary?

- Hash access:
  h(identifier) = array index
  for a suitable function *h*: {identifiers} –> {array indices}

  - advantage: fast (if *h* is simple)
    uses moderate memory

  - disadvantage:
    if |{identifiers}| > |{array indices}|,
    h cannot be injective.
    ➥ need to resolve conflicts

如何实现字典?

- 哈希方法：
  h(标识符) = 数组索引
  用于合适的函数 *h*: {标识符} –> {数组索引}

  - 优点：快速（如果 *h* 很简单）
    使用适中的内存

  - 缺点：
    如果 |{标识符}| > |{数组索引}|,
    无法 *h* 单射。
    ➥ 需要解决冲突

How to handle collisions?

- Chaining: every array entry stores a linked list of identifiers
- Reduce probability of collisions
  ➥ select *h* wisely
- Open addressing: if an array entry is occupied, calculate alternative index
- Perfect hashing: For a fixed set of identifiers, select *h* so that there are no collisions at all

如何解决冲突?

- 链接法：每个散列表条目都存储一个标识符的链接列表
- 降低碰撞概率
  ➥ 明智地选择*h*
- 开放寻址法：如果数组项被占用，则计算备用索引
- 完全散列：对于一组固定的标识符，选择*h*，这样就不会发生冲突

Chaining

- A hash table entry contains not one element, but (a pointer to) a list of elements

- INSERT(T, x) $O(1)$
  Insert x at the head of list $T[h(x.key)]$
- SEARCH(T, k) $O(1 + \alpha)$, α = load factor
  Linearly search for key k in list $T[h(k)]$
- DELETE(T, x) $O(1)$
  Delete x from its list (namely $T[h(x.key)]$)

链接法

- 哈希表条目不包含一个元素，
  而是包含（指向）元素列表的指针

- INSERT(T, x)
  在列表 $T[h(x.key)]$ 的开头插入 x
- SEARCH(T, k)
  线性搜索列表 $T[h(k)]$ 中的关键字 k
- DELETE(T, x)
  从列表（即 $T[h(x.key)]$ ）中删除 x

Hash Functions

- good hash function:
  - ensures/approximates simple uniform hashing
    (possible if input distribution is known)
  - often found heuristically
- simple choices:
  - division method
  - multiplication method
- universal hashing

哈希函数

- 良好的哈希函数：
  - 确保/近似简单均匀散列
    （如果已知输入分布，则可能）
  - 通常是启发式地找到的
- 简单的选择：
  - 除法
  - 乘法
- 全域哈希

Hash Functions: Division Method

- $h(k) = k \bmod m$
  (m = number of entries in array)
- unsuitable if $m = 2^p$ for some p
  (then $h(k)$ is just the least p bits of k)
- good choice: m is prime but not close to a power of 2
- Example: $m = 701$. Hash the values 100, 821, 1534, 2010.

哈希函数：除法散列表

- $h(k) = k \bmod m$
  （m = 数组中的项数）
- 如果 $m = 2^p$ 对于某些 p 不合适
  （那么 $h(k)$ 只是 k 的最小 p 位）
- 好的选择：m 是素数，但不接近2的幂
- 例子：$m = 701$。请哈希值 100，821，1534，2010。

Hash Functions: Multiplication Method

- Assume given a constant $0 < A < 1$
- $h(k) = \lfloor m \times \text{fractional part of } kA \rfloor$
- $m = 2^p$ is possible
  (and makes the multiplication easy)
- Often A is a multiple of $1/2^w$,
  where w = word size of the computer.
  Then integer calculations are possible:
  $kA \times 2^w$ contains the fractional part of kA
  as lower w bits.

哈希函数：乘法散列表

- 假设给定一个常数 $0 < A < 1$
- $h(k) = \lfloor m \times (kA \text{ 的分数部分}) \rfloor$
- $m = 2^p$ 是可能的
  （并使乘法变得容易）
- 通常 A 是 $1/2^w$ 的倍数，
  其中 w = 计算机的单词大小。
  则可以进行整数计算：
  $kA \times 2^w$ 包含 kA 的小数部分
  作为较低的 w 位。

Hash Functions: Multiplication Method

哈希函数：乘法散列表

- Example: Let $A \approx (\sqrt{5} - 1)/2$.
  $w = 32$.
  $A \times 2^w = 2654435769$.
  $m = 2^{14}$.

- For $k = 123456$, we have
  $(kA \times 2^w) \bmod 2^w =$
  $(123456 \times 2654435769) \bmod 2^w =$
  17612864.
  Then $\lfloor 17612864 / 2^{32-14} \rfloor$ = hash value.

Problem of Hash Functions

- If a malicious adversary knows the constants, it can attack the system.
- Protection: universal hashing = choose a hash function at random
- randomization ensures no input is (always) bad!

哈希函数的问题

- 如果恶意对手知道这些常量，就可以攻击系统。
- 保护：全域哈希 = 随机选择一个散列函数
- 随机化确保没有输入是（总是）坏的！

Universal Hashing

- Define a suitable set of hash functions $\mathcal{H}$.
  $\mathcal{H}$ = finite set of some functions
  U –> {0, 1, ... m–1}.
- Desired property: $\mathcal{H}$ is universal if for all $k, l \in U$ ($k \neq l$), we have that $\mathcal{H}$ contains $\leq |\mathcal{H}|/m$ hash functions h such that $h(k) = h(l)$.
- ensures that chance of collision $\leq 1/m$ if hash function is chosen at random.

全域哈希

- 定义一组合适的散列函数$\mathcal{H}$。
  $\mathcal{H}$ = 某些函数的有限集
  U –> {0, 1, ... m–1}。
- 所需属性： $\mathcal{H}$ 是全域的，如果对于所有 $k, l \in U$ ($k \neq l$)，我们有 $\mathcal{H}$ 包含 $\leq |\mathcal{H}|/m$ 散列函数 h 使得 $h(k) = h(l)$。
- 如果随机选择散列函数，则确保冲突的概率 $\leq 1/m$。

Universal Hashing 全域哈希

Theorem 11.3:
Suppose that a hash function h is chosen randomly from a univesal collection of hash functions and has been used to hash n keys into a table T of size m, using chaining.
If key k is not in T, then the expected length of the list $T[h(k)]$ is at most $\alpha = n/m$.
If key k is in T, then the expected length of the list $T[h(k)]$ is at most $1 + \alpha$.

Universal Hashing 全域哈希

Theorem 11.3: **Proof**

For $k \neq l$, let $X_{kl} = 1$ if $h(k) = h(l)$

$= 0$ otherwise.

Note that the probability that $h(k) = h(l)$ is $\leq 1/m$, so $E[X_{kl}] \leq 1/m$.

Then let Y_k = the number of keys $\neq k$ that hash into $T[h(k)]$.

$$Y_k = \sum_{l \in T \setminus \{k\}} X_{kl}, \text{ so } E[Y_k] = E[\sum_{l \in T \setminus \{k\}} X_{kl}]$$

$$= \sum_{l \in T \setminus \{k\}} E[X_{kl}] \leq \sum_{l \in T \setminus \{k\}} 1/m.$$

Universal Hashing 全域哈希

Theorem 11.3: **Proof**

- If $k \notin T$, then Y_k = number of elements in the list $T[h(k)]$.
  Then $E[Y_k] \leq n \cdot 1/m = \alpha$.
- If $k \in T$, then $Y_k + 1$ = number of elements in the list $T[h(k)]$.
  Then E[number of elements in $T[h(k)]$] ≤ $E[Y_k + 1] \leq (n-1) \cdot 1/m + 1 \leq \alpha + 1$.

Universal Hashing 全域哈希

Corollary 11.4

Using universal hashing and collision resolution by chaining in an initially empty table with m slots, it takes expected time $\Theta(s)$ to handle any sequence of s INSERT, SEARCH, and DELETE operations containing $n = O(m)$ INSERT operations.

Proof:
$\alpha = n/m = O(m)/m = O(1)$
So SEARCH takes time $O(1)$.

Universal class of hash functions

全域的函数组

- $h_{ab}(k) = ((ak + b) \bmod p) \bmod m$
  for a prime number $p \geq |U|$ and
  constants $0 < a < p$ and $0 \leq b < p$.

- $\mathcal{H}_{pm} = \{ h_{ab} \mid 0 < a < p \text{ and } 0 \leq b < p \}$.

- contains $(p–1)p$ functions.

- Theorem 11.5: $\mathcal{H}_{pm}$ is universal.

Universal class of hash functions

全域的函数组

Proof of Theorem 11.5:
Let k, l be two distinct keys (k, $l < p$).
Let $r = (ak + b) \bmod p$ and
$s = (al + b) \bmod p$.
Then $r \neq s$. (Otherwise $r - s \equiv a(k-l) \pmod p$,
but $a \not\equiv 0 \not\equiv k-l \pmod p$.)

Moreover, this, regarded as a function
$(a,b) \mapsto (r,s)$, is bijective.
Therefore, if we pick (a, b) at random,
then (r, s) is random as well.

Universal class of hash functions

全域的函数组

Proof of Theorem 11.5:
So, the probability that $h(k) = h(l)$ is the probability that $r \equiv s \pmod{m}$, for any random pair of distinct values $< p$.

If r is fixed, the number of values $s \neq r$ that lead to $r \equiv s \pmod{m}$ is at most:
$\lceil p/m \rceil - 1 \leq ((p + m - 1) / m) - 1 = (p-1) / m$.

There are $p-1$ choices for s, so the probability that $r \equiv s \pmod{m}$ is
$\leq (p-1) / m / (p-1) = 1/m$.
➥ probability that $h(k) = h(l)$ collide $\leq 1/m$.

Open addressing

- alternative to chaining for conflict resolution
- Idea: use a hash function $h(k)$ that can return a sequence $h_0, h_1, h_2, ..., h_{m-1}$ containing every index 0, ..., $m-1$ once.

  If $T[h_0]$ is already occupied,
  try $T[h_1]$, then try $T[h_2]$, etc.
- Noted as $h(k, i) = h_i$.

开放寻址法

- 解决冲突的链接替代方案
- 注意：使用哈希函数 $h(k)$，它可以返回序列 $h_0, h_1, h_2, ..., h_{m-1}$ 包含每个索引 0, ..., $m-1$ 一次。

  如果 $T[h_0]$ 已经被占用，
  尝试 $T[h_1]$，然后尝试 $T[h_2]$，等等。
- 记为 $h(k, i) = h_i$。

Open addressing 开放寻址法

```
INSERT(T, x)
// Specification: insert element x in hash table T
// and return its position
i = 0
repeat
        j = h(x.key, i)
        if T[j] is empty
                T[j] = x
                return j
        i = i + 1
until i == m
error “The hash table is full”
```

Open addressing 开放寻址法

```
SEARCH(T, k)
// Specification: search for key k in hash table T
// and return its position
i = 0
repeat
        j = h(k, i)
        if T[j] is empty
                return “key not found”
        if T[j].key = k
                return j
        i = i + 1
until i == m
return “key not found”
```

Open addressing

- Deleting an element: difficult because it may interrupt the sequence $h(k, 0)$, $h(k, 1)$, $h(k, 2)$, ... for some other key k.
- need to mark deleted elements not as "empty" but differently, so that SEARCH continues.

开放寻址法

- 删除一个元素：很困难，
因为它可能会中断序列 $h(k, 0)$, $h(k, 1)$, $h(k, 2)$, ... 对于某些其他密钥 k。
- 需要将已删除的元素标记为“空”，
而不是不同的，以便 SEARCH 继续搜索。

Open addressing: Linear

- Use a simple auxiliary hash function $h'(k)$.
- $h(k, i) = (h'(k) + i) \bmod m$
- advantage: easy to calculate
  disadvantage: clustering.
  If $h'(k) \approx h'(l) \approx h'(m) \approx \ldots$,
  long chains of occupied slots emerge.

开放寻址法：线性探查

- 使用一个简单的辅助散列函数$h'(k)$。
- $h(k, i) = (h'(k) + i) \bmod m$
- 优点：易于计算
  缺点：群集。
  如果$h'(k) \approx h'(l) \approx h'(m) \approx \ldots$，
  占用插槽的长链浮现。

Open addressing: Quadratic

- Use a simple auxiliary hash function $h'(k)$.
- $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
- advantage: less clustering
  (but still exists if $h'(k) = h'(l)$)
  disadvantage: need to find suitable c_1, c_2. (Book does not explain how.)
  Many values only visit few slots.

开放寻址法：二次探查

- 使用一个简单的辅助散列函数$h'(k)$。
- $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
- 优点：较少聚群集
  （但如果 $h'(k) = h'(l)$ 仍然存在）
  缺点：需要找到合适的 c_1，c_2。
  （这本书没有解释怎么做。）
  许多选择只访问少数插槽。

Open addressing: double hashing

- use two auxiliary hash functions:
  $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$
- To ensure that $h(k, i)$ visits all slots, construct h_2 carefully, e.g.:
  - $m = 2^{m'}$, and make $h_2(k)$ odd
  - m is prime, and $h_2(k) < m$
- advantage: no clustering; can visit all slots; more hash sequences than other schemes

开放寻址法：双重散列

- 使用两个辅助散列函数：
  $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$
- 为了确保 $h(k, i)$ 访问所有时隙，
  仔细构造 h_2，例如：
  - $m = 2^{m'}$，使 $h_2(k)$ 为奇
  - m 是素数，并且 $h_2(k) < m$
- 优点：无群集；可以访问所有索引；
  哈希序列比其他方案多

Open addressing: running time of search

- How long is the expected running time of SEARCH?
- Assume uniform hashing = every probe sequence is equally likely. (not really true for linear/quadratic probing; double hashing is a good approximation.)
- Theorem 11.6: The number of probes in an unsuccessful search is at most $1/(1-\alpha)$.
- Theorem 11.8: The number of probes in a successful search is at most $1/\alpha \cdot \ln(1/(1-\alpha))$.

开放寻址法：SEARCH的运行时间

- SEARCH的期望运行时间是多长?
- 假设统一散列 = 每个探测序列的可能性相等。（对于线性/二次探测来说，情况并非如此；双重散列近似很好地。）
- 定理11.6：不成功搜索中的探针数量最多为$1/(1-\alpha)$。
- 定理11.8：成功搜索中的探针数量最多为$1/\alpha \cdot \ln(1/(1-\alpha))$。

Perfect Hashing

- Assume the set of keys never changes
  - e.g. data on a CD-ROM
  - e.g. reserved words in a programming language
- We want to achieve $O(1)$ running time for SEARCH and $O(n)$ memory (n = number of entries stored).

完全散列

- 假设密钥集从未更改
  - 例如CD-ROM上的数据
  - 例如编程语言中的保留字
- 我们希望实现SEARCH的 $O(1)$ 运行时间和 $O(n)$ 内存（n = 存储的条目数）。

Perfect Hashing

- Idea:
  - hash functions from suitable $\mathcal{H}_{pm}$.
  - hash table with chaining
    but instead of storing a list in a slot,
    create a secondary hash table.
- Choose secondary hash function
  carefully so there are no collisions at all!
  ➥ SEARCH only calls 2 hash functions,
  no linear search through list

完全散列

- 注意：
  - 来自合适的 $\mathcal{H}_{pm}$ 的散列函数。
  - 带链接的哈希表
    但是，与其将列表存储在槽中，
    不如创建一个辅助哈希表。
- 仔细选择二次散列函数，
  这样就不会有任何冲突！
  ➥ SEARCH只调用2个散列函数，
  无线性搜索列表

Avoid Collisions 避免冲突

- Theorem 11.9:
  Suppose that we store n keys in a hash table of size $m = n^2$ using a hash function h randomly chosen from a universal class of hash functions. Then the probability is less than ½ that there are any collisions.

- If the hash table is large enough, a few random trials allow to find suitable hash function. (But $m = n^2$ may be too large!)

Avoid Collisions 避免冲突

Proof of Theorem 11.9:
There are $n(n-1)/2$ pairs of keys that may collide.
Each pair collides with probability $\leq 1/m$ if h is chosen from a universal family of hash functions.
We calculate the expected number of collisions:

$E[X] = n(n-1)/2 \cdot 1/m = (n^2-n) / 2n^2 \leq ½$.

But if the probability of at least one collision would be $> ½$, then $E[X] > ½$.
So we must have that $Pr\{X > 0\} \leq ½$.

Avoid excessive memory 避免过度使用内存

- To save memory, use Theorem 11.9 only at the secondary level.
- Total memory use should be $O(n)$. (n = number of elements stored)
- On the primary level, we want a hash table of size $m = n$.
  How much space will we need for the secondary tables?

Collisions with $O(n)$ memory

- Theorem 11.10:
  Suppose that 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.
  If n_j = number of keys in slot $T[j]$, then

$$E\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n.$$

Collisions with $O(n)$ memory

Proof of Theorem 11.10:

$$\mathrm{E}\Big[\sum_{j=0}^{m-1} n_j^2\Big] = \mathrm{E}\Big[\sum_{j=0}^{m-1} n_j + 2n_j(n_j-1)/2\Big]$$

$$= \mathrm{E}\Big[\sum_{j=0}^{m-1} n_j\Big] + 2\mathrm{E}\Big[\sum_{j=0}^{m-1} n_j(n_j-1)/2\Big]$$

$$= \quad n \quad + 2\sum_{j=0}^{m-1} \begin{pmatrix}\text{expected col-}\\ \text{lisions in slot } j\end{pmatrix}$$

$$\leq \quad n \quad + 2\, n(n-1)/2 \cdot 1/m$$

$$= n + n - 1 < 2n.$$

Collisions with $O(n)$ memory

Corollary 11.11
Suppose that we store n keys in a hash table of size $m = n$ using a hash function from a universal class of hash function, and we set the size of each secondary hash table to $m_j = n_j^2$.
Then the expected number of table entries required for all secondary hash tables in perfect hashing is $< 2n$.

Collisions with $O(n)$ memory

Corollary 11.12
Suppose that we store n keys in a hash table of size $m = n$ using a hash function from a universal class of hash function, and we set the size of each secondary hash table to $m_j = n_j^2$.
Then the probability is $< ½$ that the total number of table entries required for all secondary hash tables in perfect hashing is $\geq 4n$.

Collisions with $O(n)$ memory

Consequence of Corollary 11.12:

- Pick a random hash function from $\mathcal{H}_{pn}$ as primary hash function repeatedly, until the found one uses $< 4n$ secondary table entries.
- This process will terminate quickly.

推论11.12的结果：

- 从$\mathcal{H}_{pn}$中反复选择一个随机哈希函数作为主哈希函数，直到找到的函数使用$<4n$个辅助表条目。
- 此过程将很快终止。

Summary

- Hashing: can choose tradeoff between time + memory use
- Collisions can be handled by chaining or open addressing
- Universal hashing allows to hash with expected good result independent of input

- 哈希：可以在时间和内存使用之间进行权衡
- 冲突可以通过链接或开放寻址来处理
- 通用散列允许散列取得预期的好成绩与输入无关

Open office

- I want to offer a time to ask questions every week.

开放时间

- 每周有时间可以问我问题。