5.4.2 Balls and bins

- Consider tossing identical balls randomly into b bins, numbered $1,2,\ldots,b$
- Tosses are independent, and on each toss the ball is equally likely to end up in any bin
- The probability that a tossed ball lands in any given bin is $1/b$
- The ball-tossing process is a sequence of Bernoulli trials with a probability $1/b$ of success ≡ the ball falls in the given bin

- ***How many balls fall in a given bin?***
  - The number of balls that fall in a given bin follows the binomial distribution $b(k;n,1/b)$
  - If we toss n balls, the expected number of balls that fall in the given bin is n/b
- ***How many balls must we toss, on the average, until a given bin contains a ball?***
  - The number of tosses until the given bin receives a ball follows the geometric distribution with probability $1/b$ and
  - the expected number of tosses until success is $1/(1/b) = b$

- ***How many balls must we toss until every bin contains at least one ball?***
  - Call a toss in which a ball falls into an empty bin a "hit"
  - We want to know the expected number n of tosses required to get b hits
  - We can partition the n tosses into stages
  - The ith stage consists of the tosses after the $(i-1)$st hit until the ith hit
  - The first stage consists of the first toss, since we are guaranteed to have a hit when all bins are empty

  - During the ith stage, $i-1$ bins contain balls and $b-i+1$ bins are empty
  - For each toss in the ith stage, the probability of obtaining a hit is $(b-i+1)/b$
  - n_i is the number of tosses in the ith stage
  - The number of tosses required to get b hits is $n = \sum_{i=1}^{b} n_i$
  - Each n_i has a geometric distribution with probability of success $(b-i+1)/b$

$$\mathrm{E}[n_i] = \frac{b}{b-i+1}$$

$$\mathrm{E}[n] = \mathrm{E}\left[\sum_{i=1}^{b} n_i\right] = \sum_{i=1}^{b} E[n_i]$$
$$= \sum_{i=1}^{b} \frac{b}{b-i+1}$$
$$= b\sum_{i=1}^{b} \frac{1}{i}$$
$$= b(\ln b + O(1))$$

- By harmonic series

- It therefore takes approximately $b \ln b$ tosses before we can expect that every bin has a ball
- This problem is also known as the ***coupon collector's problem***, which says that a person trying to collect each of b different coupons expects to acquire approximately $b \ln b$ randomly obtained coupons in order to succeed

TAMPERE UNIVERSITY OF TECHNOLOGY

II (Sorting and) Order Statistics

~~Heapsort~~
~~Quicksort~~
~~Sorting in Linear Time~~
Medians and Order Statistics

9 Medians and Order Statistics

- The ith order statistic of a set of n elements is the ith smallest element
  - E.g., the minimum of a set of elements is the first order statistic ($i = 1$), and the maximum is the nth order statistic ($i = n$)
- A **median** is the "halfway point" of the set
- When n is odd, the median is unique, occurring at $i = (n + 1)/2$

- When n is even, there are two medians, occurring at $i = n/2$ and $i = n/2 + 1$
- Thus, regardless of the parity of n, medians occur at
  - $\lfloor i = (n + 1)/2 \rfloor$ (the **lower** median) and
  - $\lceil i = (n + 1)/2 \rceil$ (the **upper** median)
- For simplicity, we use "the median" to refer to the lower median

- The problem of selecting the ith order statistic from a set of n distinct numbers
- We assume that the set contains distinct numbers
  - virtually everything extends to the situation in which a set contains repeated values
- We formally specify the problem as follows:
  - **Input:** A set A of n (distinct) numbers and an integer i, with $1 \leq i \leq n$
  - **Output:** The element $x \in A$ that is larger than exactly $i - 1$ other elements of A

- We can solve the problem in $O(n \lg n)$ time by heapsort or merge sort and then simply index the ith element in the output array
- There are faster algorithms
- First, we examine the problem of selecting the minimum and maximum of a set of elements
- Then we analyze a practical randomized algorithm that achieves an $O(n)$ expected running time, assuming distinct elements

9.1 Minimum and maximum

- How many comparisons are necessary to determine the minimum of a set of n elements?
- We can easily obtain an upper bound of $n - 1$ comparisons
  - examine each element of the set in turn and keep track of the smallest element seen so far.
- In the following procedure, we assume that the set resides in array A, where $A.length = n$

MINIMUM(A)

1. min $\leftarrow A[1]$
2. **for** $i \leftarrow 2$ **to** $A.length$
3. **if** min $> A[i]$
4. min $\leftarrow A[i]$
5. **return** min

- We can, of course, find the maximum with $n-1$ comparisons as well

- This is the best we can do, since we can obtain a lower bound of $n-1$ comparisons
  - Think of any algorithm that determines the minimum as a tournament among the elements
  - Each comparison is a match in the tournament in which the smaller of the two elements wins
  - Observing that every element except the winner must lose at least one match, we conclude that $n-1$ comparisons are necessary to determine the minimum
- Hence, the algorithm MINIMUM is optimal w.r.t. the number of comparisons performed

Simultaneous minimum and maximum

- Sometimes, we must find both the minimum and the maximum of a set of n elements
- For example, a graphics program may need to scale a set of (x, y) data to fit onto a rectangular display screen or other graphical output device
- To do so, the program must first determine the minimum and maximum value of each coordinate

- $\Theta(n)$ comparisons is asymptotically optimal:
  - Simply find the minimum and maximum independently, using $n - 1$ comparisons for each, for a total of $2n - 2$ comparisons
- In fact, we can find both the minimum and the maximum using at most $3\lfloor n/2 \rfloor$ comparisons by maintaining both the minimum and maximum elements seen thus far
- Rather than processing each element of the input by comparing it against the current minimum and maximum, we process elements in pairs

- Compare pairs of input elements first with each other, and then we compare the smaller with the current min and the larger to the current max, at a cost of 3 comparisons for every 2 elements
- If n is odd, we set both the min and max to the value of the first element, and then we process the rest of the elements in pairs
- If n is even, we perform 1 comparison on the first 2 elements to determine the initial values of the min and max, and then process the rest of the elements in pairs as in the case for odd n

- If n is odd, then we perform

$$3\lfloor n/2 \rfloor$$

comparisons

- If n is even, we perform 1 initial comparison followed by

$$3(n-2)/2$$

comparisons, for a total of $3n/2 - 2$

- Thus, in either case, the total number of comparisons is at most $3\lfloor n/2 \rfloor$

9.2 Selection in expected linear time

- The selection problem appears more difficult than finding a minimum, but the asymptotic running time for both is the same: $\Theta(n)$
- A divide-and-conquer algorithm RANDOMIZED-SELECT is modeled after the quicksort algorithm
- Unlike quicksort, RANDOMIZED-SELECT works on only one side of the partition
- Whereas quicksort has an expected running time of $\Theta(n \lg n)$, the expected running time of RANDOMIZED-SELECT is $\Theta(n)$, assuming that the elements are distinct

- RANDOMIZED-SELECT uses the procedure RANDOMIZED-PARTITION of RANDOMIZED-QUICKSORT

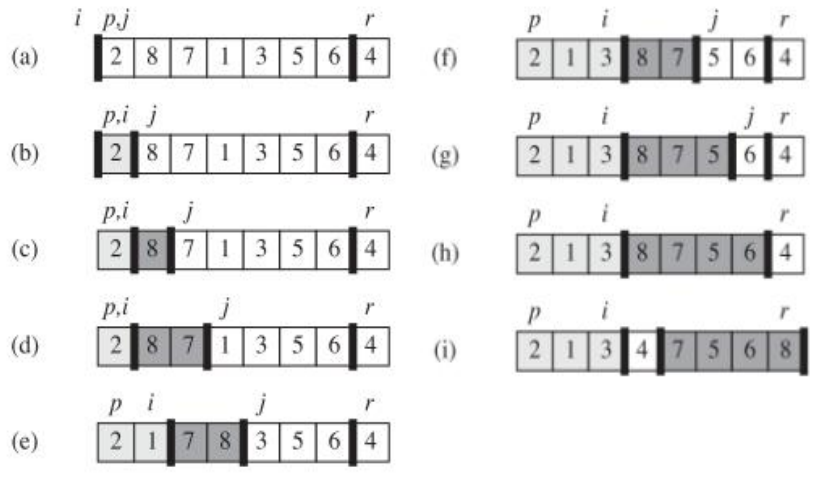
RANDOMIZED-PARTITION(A, p, r)

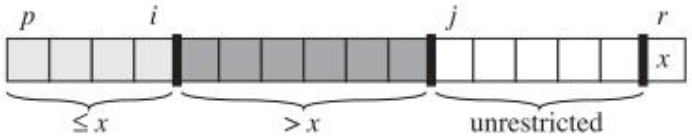
1. $i \leftarrow$ RANDOM(p, r)
2. exchange $A[r]$ with $A[i]$
3. **return** PARTITION(A, p, r)

Partitioning the array

PARTITION(A, p, r)

1. $x \leftarrow A[r]$
2. $i \leftarrow p - 1$
3. **for** $j \leftarrow p$ **to** $r - 1$
4. **if** $A[j] \leq x$
5. $i \leftarrow i + 1$
6. exchange $A[i]$ with $A[j]$
7. exchange $A[i + 1]$ with $A[r]$
8. **return** $i + 1$





- PARTITION always selects an element $x = A[r]$ as a ***pivot*** element around which to partition the subarray $A[p..r]$
- As the procedure runs, it partitions the array into four (possibly empty) regions
- At the start of each iteration of the **for** loop in lines 3–6, the regions satisfy properties, shown above

- At the beginning of each iteration of the loop of lines 3–6, for any array index k,
  1. If $p \leq k \leq i$, then $A[k] \leq x$
  2. If $i + 1 \leq k \leq j - 1$, then $A[k] > x$
  3. If $k = r$, then $A[k] = x$
- Indices between j and $r - 1$ are not covered by any case, and the values in these entries have no particular relationship to the pivot x
- The running time of PARTITION on the subarray $A[p..r]$ is $\Theta(n)$, $n = p - r + 1$

- Return the ith smallest element of $A[p..r]$

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

1. **if** $p = r$
2. **return** $A[p]$
3. $q \leftarrow$ RANDOMIZED-PARTITION(A, p, r)
4. $k \leftarrow q - p + 1$
5. **if** $i = k$ // the pivot value is the answer
6. **return** $A[q]$
7. **elseif** $i < k$
8. **return** RANDOMIZED-SELECT$(A, p, q - 1, i)$
9. **else return** RANDOMIZED-SELECT$(A, q + 1, r, i - k)$

- (1) checks for the base case of the recursion
- Otherwise, RANDOMIZED-PARTITION partitions $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ s.t. each element in the former is $\leq A[q] <$ each element of the latter
- (4) computes the # k of elements in $A[p..q]$
- (5) checks if $A[q]$ is ith smallest element
- Otherwise, determine in which of the two subarrays the ith smallest element lies
- If $i < k$, then the desired element lies on the low side of the partition, and (8) recursively selects it

- If $i > k$, then the desired element lies on the high side of the partition
- Since we already know k values that are smaller than the ith smallest element of $A[p..r]$ the desired element is the $(i - k)$th smallest element of $A[q + 1..r]$, which (9) finds recursively

- Worst-case running time for RANDOMIZED-SELECT is $\Theta(n^2)$, even to find the minimum
- The algorithm has a linear expected running time, though
- Because it is randomized, no particular input elicits the worst-case behavior
- Let the running time of RANDOMIZED-SELECT on an input array $A[p..r]$ of n elements be a random variable $T(n)$
- We obtain an upper bound on $\mathrm{E}[T(n)]$ as follows

- The procedure RANDOMIZED-PARTITION is equally likely to return any element as the pivot
- Therefore, for each k such that $1 \leq k \leq n$, the subarray $A[p\,..\,q]$ has k elements (all less than or equal to the pivot) with probability $1/n$
- For $k = 1,2,\ldots,n$, define indicator random variables X_k where

$X_k = \mathrm{I}\{\text{the subarray } A[p..q] \text{ has exactly } k \text{ elements}\}$

- Assuming that the elements are distinct, we have $\mathrm{E}[X_k] = 1/n$
- When we choose $A[q]$ as the pivot element, we do not know, *a priori*,
  1) if we will terminate immediately with the correct answer,
  2) recurse on the subarray $A[p\,..\,q-1]$, or
  3) recurse on the subarray $A[q+1\,..\,r]$
- This decision depends on where the ith smallest element falls relative to $A[q]$

- Assuming $T(n)$ to be monotonically increasing, we can upper-bound the time needed for a recursive call by that on largest possible input
- To obtain an upper bound, we assume that the ith element is always on the side of the partition with the greater number of elements
- For a given call of RANDOMIZED-SELECT, the indicator random variable X_k has value 1 for exactly one value of k, and it is 0 for all other k
- When $X_k = 1$, the two subarrays on which we might recurse have sizes $k - 1$ and $n - k$

- We have the recurrence

$$T(n) \leq \sum_{i=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n))$$

$$= \sum_{i=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)$$

- Taking expected values, we have

$$\mathrm{E}[T(n)] \leq \mathrm{E}\left[\sum_{i=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)\right]$$

$$= \sum_{i=1}^{n} \mathrm{E}[X_k \cdot T(\max(k-1, n-k))] + O(n)$$

$$= \sum_{i=1}^{n} \mathrm{E}[X_k] \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n)$$

$$= \sum_{i=1}^{n} \frac{1}{n} \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n)$$

- The first Eq. on this slide follows by independence of random variables X_k and $\max(k-1, n-k)$
- Consider the expression

$$\max(k-1, n-k) = \begin{cases} k-1 & \text{if } k > \lceil n/2 \rceil \\ n-k & \text{if } k \leq \lceil n/2 \rceil \end{cases}$$

- If n is even, each term from $T(\lceil n/2 \rceil)$ up to $T(n-1)$ appears exactly twice in the summation, and if n is odd, all these terms appear twice and $T(\lfloor n/2 \rfloor)$ appears once
- Thus, we have

$$\mathrm{E}[T(n)] \leq \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} \mathrm{E}[T(k)] + O(n)$$

- We can show that $\mathrm{E}[T(n)] = O(n)$ by substitution
- In summary, we can find any order statistic, and in particular the median, in expected linear time, assuming that the elements are distinct

TAMPERE UNIVERSITY OF TECHNOLOGY

III Data Structures

~~Elementary Data Structures~~

Hash Tables

~~Binary Search Trees~~

~~Red-Black Trees~~

Augmenting Data Structures

Dynamic sets

- Sets are fundamental to computer science
- Algorithms may require several different types of operations to be performed on sets
- For example, many algorithms need only the ability to
  - insert elements into, delete elements from, and test membership in a set
- We call a dynamic set that supports these operations a ***dictionary***

Operations on dynamic sets

- Operations on a dynamic set can be grouped into ***queries*** and ***modifying operations***

SEARCH(S, k)
- Given a set S and a key value k, return a pointer x to an element in S such that $x.key = k$, or NIL if no such element belongs to S

INSERT(S, x)
- Augment the set S with the element pointed to by x. We assume that any attributes in element x have already been initialized

DELETE(S, x)
- Given a pointer x to an element in the set S, remove x from S. Note that this operation takes a pointer to an element x, not a key value

MINIMUM(S)
- A query on a totally ordered set S that returns a pointer to the element of S with the smallest key

MAXIMUM(S)
- Return a pointer to the element of S with the largest key

SUCCESSOR(S, x)
- Given an element x whose key is from a totally ordered set S, return a pointer to the next larger element in S, or NIL if x is the max element

PREDECESSOR(S, x)
- Given an element x whose key is from a totally ordered set S, return a pointer to the next smaller element in S, or NIL if x is the min element

- We usually measure the time taken to execute a set operation in terms of the size of the set
- E.g., red-black trees can support any of the operations on a set of size n in time $O(\lg n)$

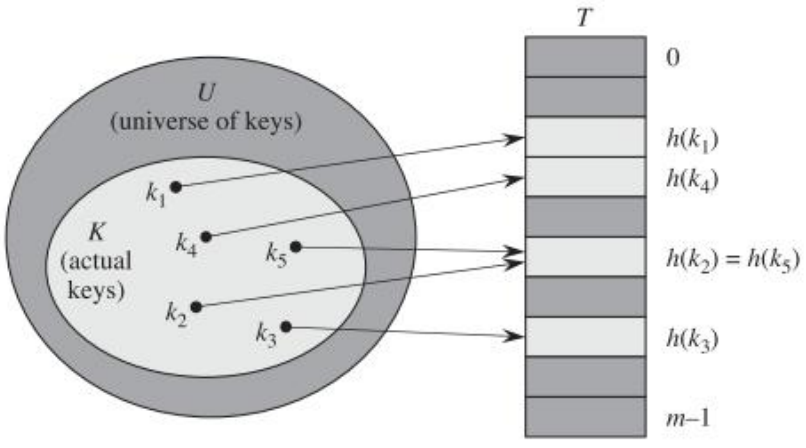
11 Hash Tables

- Often one only needs the dictionary operations INSERT, SEARCH, and DELETE
- Hash table effectively implements dictionaries
- In the worst case, searching for an element in a hash table takes $\Theta(n)$ time
- In practice, hashing performs extremely well
- Under reasonable assumptions, the average time to search for an element is $O(1)$

11.2 Hash tables

- A set K of keys is stored in a dictionary, it is usually much smaller than the universe U of all possible keys
- A hash table requires storage $\Theta(|K|)$ while search for an element in it only takes $O(1)$ time
- The catch is that this bound is for the *average-case* time

- An element with key k is stored in slot $h(k)$; that is, we use a hash function h to compute the slot from the key k
- $h: U \to \{0,1, \ldots, m-1\}$ maps the universe U of keys into the slots of hash table $T[0..m-1]$
- The size m of the hash table is typically $\ll |U|$
- We say that an element with key k ***hashes*** to slot $h(k)$ or that $h(k)$ is the ***hash value*** of k
- If $k_1 \neq k_2$ hash to same slot we have a ***collision***
- Effective techniques resolve the conflict



- The ideal solution avoids collisions altogether
- We might try to achieve this goal by choosing a suitable hash function h
- One idea is to make h appear to be random, thus minimizing the number of collisions
- Of course, h must be deterministic so that a key k always produces the same output $h(k)$
- Because $|U| > m$, there must be at least two keys that have the same hash value;
  - avoiding collisions altogether is therefore impossible

Collision resolution by chaining

- In chaining, we place all the elements that hash to the same slot into the same linked list
- Slot j contains a pointer to the head of the list of all stored elements that hash to j
- If no such elements exist, slot j contains NIL
- The dictionary operations on a hash table T are easy to implement when collisions are resolved by chaining

CHAINED-HASH-INSERT(T, x)

1. insert x at the head of list $T[h(x.key)]$

CHAINED-HASH-SEARCH(T, k)

1. search for element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(T, x)

1. delete x from the list $T[h(x.key)]$

- Worst-case running time of insertion is $O(1)$
- It is fast in part because it assumes that the element x being inserted is not already present in the table
- For searching, the worst-case running time is proportional to the length of the list;
  - we analyze this operation more closely soon
- We can delete an element (given a pointer) in $O(1)$ time if the lists are doubly linked
- In singly linked lists, to delete x, we would first have to find x in the list $T[h(x.key)]$

Analysis of hashing with chaining

- A hash table T of m slots stores n elements, define the ***load factor*** $\alpha = n/m$, i.e., the average number of elements in a chain
- Our analysis will be in terms of α, which can be < 1, $= 1$, or > 1

- In the worst-case all n keys hash to one slot
- The worst-case time for searching is thus $\Theta(n)$ plus the time to compute the hash function

- Average-case performance of hashing depends on how well h distributes the set of keys to be stored among the m slots, on the average
- ***Simple uniform hashing*** (SUH):
  - Assume that an element is equally likely to hash into any of the m slots, independently of where any other element has hashed to
- For $j = 0,1, \dots, m-1$, let us denote the length of the list $T[j]$ by n_j, so $n = n_0 + \cdots + n_{m-1}$

  and the expected value of n_j is

  $$\mathrm{E}[n_j] = \alpha = n/m$$

Theorem 11.1 *In a hash table which resolves collisions by chaining, an **unsuccessful** search takes average-case time $\Theta(1 + \alpha)$, under SUH assumption.*

Proof Under SUH, k not already in the table is equally likely to hash to any of the m slots.

The time to search unsuccessfully for k is the expected time to go through list $T[h(k)]$, which has expected length $\mathrm{E}[n_{h(k)}] = \alpha$.

Thus, the expected number of elements examined is α, and the total time required (incl. computing $h(k)$) is $\Theta(1 + \alpha)$. ■

Theorem 11.2 *In a hash table which resolves collisions by chaining, a **successful** search takes average-case time $\Theta(1 + \alpha)$, under SUH.*

Proof We assume that the element being searched for is equally likely to be any of the n elements stored in the table.

The number of elements examined during the search for x is one more than the number of elements that appear before x in x's list.

Elements before x in the list were all inserted after x was inserted.

- Let us take the average (over the n table elements x) of the expected number of elements added to x's list after x was added to the list + 1
- Let x_i, $i = 1,2,\dots,n$, denote the ith element inserted into the table and let $k_i = x_i.key$
- For keys k_i and k_j, define the indicator random variable $X_{ij} = \mathrm{I}\{h(k_i) = h(k_j)\}$
- Under SUH, $\Pr\{h(k_i) = h(k_j)\} = 1/m$, and by Lemma 5.1, $\mathrm{E}[X_{ij}] = 1/m$

- Thus, the expected number of elements examined in successful search is

$$\mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]$$
$$=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathrm{E}[X_{ij}]\right)$$
$$=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)$$

$$= 1 + \frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

$$= 1 + \frac{1}{nm}\left(\sum_{i=1}^{n} n - \sum_{i=1}^{n} i\right)$$

$$= 1 + \frac{1}{nm}\left(n^2 - \frac{n(n+1)}{2}\right)$$

$$= 1 + \frac{n-1}{2m} = 1 + \frac{\alpha}{2} + \frac{\alpha}{2n}$$

Thus, the total time required for a successful search is $\Theta(2 + \alpha/2 + \alpha/2n) = \Theta(1 + \alpha)$■

- If the number of hash-table slots is at least proportional to the number of elements in the table, we have $n = O(m)$ and, consequently,

$$\alpha = n/m = O(m)/m = O(1)$$

- Thus, searching takes constant time on average
- Insertion takes $O(1)$ worst-case time
- Deletion takes $O(1)$ worst-case time when the lists are doubly linked
- Hence, we can support all dictionary operations in $O(1)$ time on average

11.3 Hash functions

- A good hash function satisfies (approximately) the assumption of SUH:
  - each key is equally likely to hash to any of the m slots, independently of the other keys
- We typically have no way to check this condition
- We rarely know the probability distribution from which the keys are drawn
- The keys might not be drawn independently

- If we, e.g., know that the keys are random real numbers k independently and uniformly distributed in the range $0 \leq k < 1$
- Then the hash function $h(k) = \lfloor km \rfloor$ satisfies the condition of SUH
- In practice, we can often employ heuristic techniques to create a hash function that performs well
- Qualitative information about the distribution of keys may be useful in this design process

- In a compiler's symbol table the keys are strings representing identifiers in a program
- Closely related symbols, such as **`pt`** and **`pts`**, often occur in the same program
- A good hash function minimizes the chance that such variants hash to the same slot
- A good approach derives the hash value in a way that we expect to be independent of any patterns that might exist in the data
- For example, the "division method" computes the hash value as the remainder when the key is divided by a specified prime number

Interpreting keys as natural numbers

- Hash functions assume that the universe of keys is natural numbers $\mathbb{N} = \{0,1,2,\dots\}$
- We can interpret a character string as an integer expressed in suitable radix notation
- We interpret the identifier **`pt`** as the pair of decimal integers $(112,116)$, since $p = 112$ and $t = 116$ in ASCII
- Expressed as a radix-128 integer, **`pt`** becomes $(112 \cdot 128) + 116 = 14{,}452$

11.3.1 The division method

- In the division method, we map a key k into one of m slots by taking the remainder of k divided by m
- That is, the hash function is
$$h(k) = k \bmod m$$
- E.g., if the hash table has size $m = 12$ and the key is $k = 100$, then $h(k) = 4$
- Since it requires only a single division operation, hashing by division is quite fast

- When using the division method, we usually avoid certain values of m
- E.g., m should not be a power of 2, since if $m = 2^p$, then $h(k)$ is just the p lowest-order bits of k
- We are better off designing the hash function to depend on *all* the bits of the key
- A prime not too close to an exact power of 2 is often a good choice for m

- Suppose we wish to allocate a hash table, with collisions resolved by chaining, to hold roughly $n = 2{,}000$ character strings, where a character has 8 bits
- We don't mind examining an average of 3 elements in an unsuccessful search, and so we allocate a hash table of size $m = 701$
- We choose $m = 701$ because it is a prime near $2000/3$ but not near any power of 2 ($2^9 = 512$ and $2^{10} = 1024$)
- Treating each key k as an integer, our hash function would be $h(k) = k \bmod 701$