Randomized Algorithms

Rangaballav Pradhan

ITER, SOADU

Randomized Algorithms

- Randomization can be incorporated in algorithm design in two ways:
  - Randomization of the input: average case analysis
  - Randomization of the algorithms: algorithm to make random decisions as it processes the input
- Why randomize?
  - Algorithm design technique
  - Can lead to simplest, fastest, or only known algorithm for a particular problem.
- Examples. Graph algorithms, quicksort, hashing, load balancing, closest pair, Monte Carlo integration, cryptography, …and many more.

Finding the Median

- Suppose we are given a set of n numbers $S = \{a_1, a_2, \ldots, a_n\}$. Their *median* is the number that would be in the middle position if we were to sort them.
- The median of $S = \{a_1, a_2, \ldots, a_n\}$ is equal to the k_{th} largest element in S, where $k = (n + 1)/2$ if n is odd, and $k = n/2$ if n is even.
- Assume for the sake of simplicity that all the numbers are distinct.
- It is easy to compute the median in time O(n log n) if we simply sort the numbers first.
- But a simple randomized approach, based on divide-and-conquer, can yield an expected running time of O(n).

Median Finding Using Splitters

- We can map the median-finding problem to the more general problem of *selection*.
- Given a set of n numbers S and a number k between 1 and n, consider the function Select(S, k) that returns the k_{th} largest element in S.
- As special cases, Select includes the problem of finding the median of S via Select(S, $n/2$) or Select(S, $(n + 1)/2$); it also includes the easier problems of finding the minimum (Select(S, 1)) and the maximum (Select(S, n)).
- Our goal is to design an algorithm that implements Select so that it runs in expected time $O(n)$.
- The basic structure of the algorithm implementing Select is as follows.
- We choose an element $a_i \in S$, the "splitter," and form the sets $S^- = \{a_j : a_j < a_i\}$ and $S^+ = \{a_j : a_j > a_i\}$. We can then determine which of S^- or S^+ contains the k_{th} largest element, and iterate only on this one.

Finding the Median

Select(S, k):
Choose a splitter $a_i \in S$
For each element a_j of S
    Put a_j in S^- if $a_j < a_i$
    Put a_j in S^+ if $a_j > a_i$
Endfor
If $|S^-| = k - 1$ then
    The splitter a_i was in fact the desired answer
Else if $|S^-| \geq k$ then
    The kth largest element lies in S^-
    Recursively call *Select*(S^-, k)
Else suppose $|S^-| = l < k - 1$
    The kth largest element lies in S^+
    Recursively call *Select*(S^+, $k - 1 - l$)
Endif

Choosing a Good Splitter

- Assuming we can select a splitter in linear time, the rest of the algorithm takes linear time plus the time for the recursive call.
- It's important that the splitter significantly reduce the size of the set being considered, so that we don't keep making passes through large sets of numbers many times.
- So a good choice of splitter should produce sets S^- and S^+ that are approximately equal in size.
- If we choose the median as the splitter, the running time will be, $T(n) \leq T(n/2) + cn$, Which is linear $O(n)$ but this idea is circular.
- Any "well-centered" element can serve as a good splitter: If we had a way to choose splitters a_i such that there were at least εn elements both larger and smaller than a_i, for any fixed constant $\varepsilon > 0$, then the size of the sets in the recursive call would shrink by a factor of at least $(1-\varepsilon)$ each time. Thus the running time T(n) would be bounded by the recurrence $T(n) \leq T((1-\varepsilon)n) + cn$, which will converge to $T(n) \leq \frac{1}{\varepsilon} cn$.

Random Splitters

Select(S, k):
Choose a splitter $a_i \in S$ **uniformly at random**
For each element a_j of S
    Put a_j in S^- if $a_j < a_i$
    Put a_j in S^+ if $a_j > a_i$
Endfor
If $|S^-| = k - 1$ then
    The splitter a_i was in fact the desired answer
Else if $|S^-| \geq k$ then
    The kth largest element lies in S^-
    Recursively call *Select*(S^-, k)
Else suppose $|S^-| = l < k - 1$
    The kth largest element lies in S^+
    Recursively call *Select*(S^+, $k - 1 - l$)
Endif

Random Splitters

- The idea is, to reduce the size of the set under consideration by a fixed constant fraction every iteration, so that to get a convergent series and hence a linear bound.
- The algorithm is in *phase* j when the size of the set under consideration is at most $n(3/4)^j$ but greater than $n(3/4)^{j+1}$.
- In a given iteration of the algorithm, we say that an element of the set under consideration is *central* if at least a quarter of the elements are smaller than it and at least a quarter of the elements are larger than it.
- If a central element is chosen as a splitter, then at least a quarter of the set will be thrown away, the set will shrink by a factor of (3/4) or better.

Random Splitters

- It can be observed that half of the elements in the set are central, and so the probability that our random choice of splitter produces a central element is (1/2).
- Hence, the expected number of iterations before a central element is found is 2; and so the expected number of iterations spent in phase j, for any j, is at most 2.
- Now, let X be a random variable equal to the number of steps taken by the algorithm. We can write it as, $X = X_0 + X_1 + X_3 + \ldots$, where X_j is the expected number of steps spent by the algorithm in phase j.
- When the algorithm is in phase j, the set has size at most $n(3/4)^j$, and so the number of steps required for one iteration in phase j is at most $cn(3/4)^j$ for some constant c.

Random Splitters

- We know that the expected number of iterations spent in phase j is at most two. So, $E[X_j] \leq 2cn(3/4)^j$.
- Thus we can bound the total expected running time using linearity of expectation,

$$E[X] = \sum_j E[X_j] \leq \sum_j 2cn\left(\frac{3}{4}\right)^j = 2cn\sum_j\left(\frac{3}{4}\right)^j \leq 8cn.$$

- Conclusion. The expected running time of Select(n, k) is O(n).

Median Finding using Random Splitters

Select(S, k):
While no central splitter has been found
    Choose a splitter $a_i \in S$ **uniformly at random**
    For each element a_j of S
        Put a_j in S^- if $a_j < a_i$
        Put a_j in S^+ if $a_j > a_i$
    Endfor
    If $|S^-| \geq |S|/4$ and $|S^+| \geq |S|/4$ then
        a_i is a central splitter
    Endif
Endwhile
If $|S^-| = k - 1$ then
    The splitter a_i was in fact the desired answer
Else if $|S^-| \geq k$ then
    The kth largest element lies in S^-
    Recursively call *Select*(S^-, k)
Else suppose $|S^-| = l < k - 1$
    The kth largest element lies in S^+
    Recursively call *Select*(S^+, $k - 1 - l$)
Endif

Randomized Quicksort

- The idea is similar to the randomized divide-and-conquer technique we used to find the median.
- As before, we choose a splitter for the input set S, and separate S into S^-, the elements below the splitter value and S^+,those above it.
- The difference is that, rather than looking for the median on just one side of the splitter, we sort both sides recursively and glue the two sorted pieces together (with the splitter in between) to produce the overall output.
- Also, we need to explicitly include a base case for the recursive code: we only use recursion on sets of size at least 4.

Randomized Quicksort

QuickSort(S):
If $|S| \leq 3$ then
    Sort S.
    Output the sorted list.
Else
    Choose a splitter $a_i \in S$ **uniformly at random**
    For each element a_j of S
        Put a_j in S^- if $a_j < a_i$
        Put a_j in S^+ if $a_j > a_i$
    Endfor
    Recursively call *QuickSort*(S^-) and *QuickSort*(S^+)
    Output the sorted set S^-, then a_i, then the sorted set S^+
EndIf

Randomized Quicksort

- If we always select the smallest element as a splitter, then the running time T(n) on n-element sets satisfies the same recurrence as before:
- $T(n) \leq T(n-1) + cn$, and so we end up with a time bound of $T(n) = O(n^2)$.
- On the positive side, if the splitters selected happened to be the medians of the sets at each iteration, then we get the recurrence $T(n) \leq 2T(n/2) + cn$, the running time in this lucky case is $O(n \log n)$.
- Here we are concerned with the *expected running time* which can be bounded by $O(n \log n)$.

Randomized Quicksort

- The analysis of Quicksort will closely follow the analysis of median-finding. Just as in the Select procedure that we used for median-finding, the crucial definition is that of a *central splitter*—one that divides the set so that each side contains at least a quarter of the elements.
- The idea is that a random choice is likely to lead to a central splitter, and central splitters work well. In the case of sorting, a central splitter divides the problem into two considerably smaller subproblems.
- We will slightly modify the algorithm so that it only issues its recursive calls when it finds a central splitter. Any off-center splitter will be thrown away and the algorithm repeat until it selects a central splitter.

Randomized Quicksort

Modified_QuickSort(S):
If $|S| \leq 3$ then
    Sort S.
    Output the sorted list.
Else
    While no central splitter has been found
        Choose a splitter $a_i \in S$ **uniformly at random**
        For each element a_j of S
            Put a_j in S^- if $a_j < a_i$
            Put a_j in S^+ if $a_j > a_i$
        Endfor
        If $|S^-| \geq |S|/4$ and $|S^+| \geq |S|/4$ then
            a_i is a central splitter
        Endif
    Endwhile
    Recursively call Modified_*QuickSort*(S^-) and *Modified_QuickSort*(S^+)
    Output the sorted set S^-, then a_i, then the sorted set S^+
EndIf

Randomized Quicksort Analysis

- Consider a subproblem for some set *S*. Each iteration of the While loop selects a possible splitter a_i and spends *O(|S|)* time splitting the set and deciding if a_i is central.
- Earlier we argued that the number of iterations needed until we find a central splitter is at most 2. Hence,
- **Claim.** The expected running time for the algorithm on a set S, excluding the time spent on recursive calls, is O(|S|).
- The algorithm is called recursively on multiple subproblems. We will group these subproblems by size. We'll say that the subproblem is of *type j* if the size of the set under consideration is at most $n(3/4)^j$ but greater than $n(3/4)^{j+1}$.
- So, the expected time spent on a subproblem of type *j*, excluding recursive calls, is $O(n(3/4)^j)$.

Randomized Quicksort Analysis

- To bound the overall running time, we need to bound the number of subproblems for each type j.
- Splitting a type j subproblem via a central splitter creates two subproblems of higher type. So the subproblems of a given type j are disjoint. This gives us a bound on the number of subproblems.
- **Claim.** The number of *type j* subproblems created by the algorithm is at most $(4/3)^{j+1}$.
- There are at most $(4/3)^{j+1}$ subproblems of *type j*, and the expected time spent on each is $O(n(3/4)^j)$. Thus, by linearity of expectation, the expected time spent on subproblems of type j is O(n). The number of different types is bounded by $\log_{\frac{4}{3}} n = O(\log n)$, which gives the desired bound.
- The expected running time of Modified Quicksort is O(n log n).

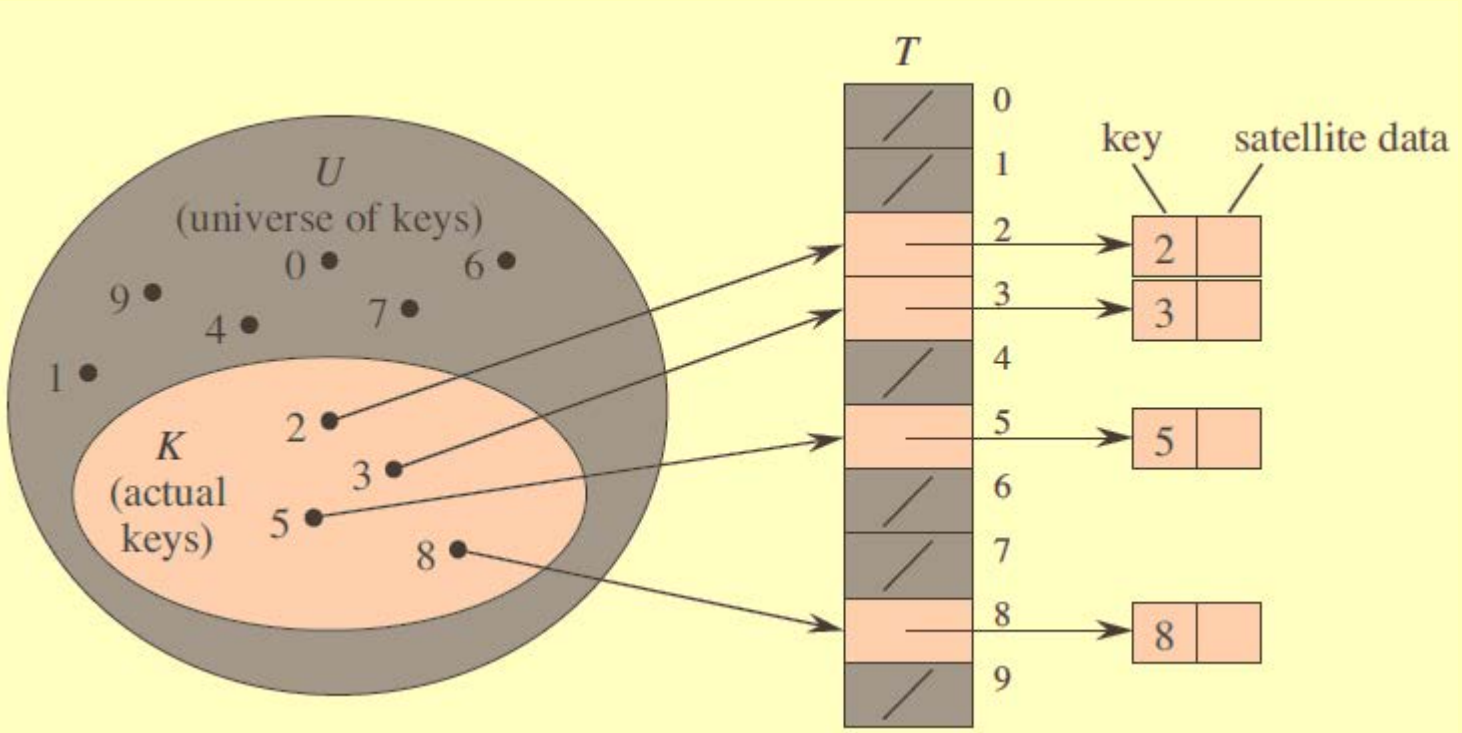
Hashing: A Randomized Implementation of Dictionaries

- Randomization can be a powerful technique in the design of data structures. The most fundamental use of randomization in this setting is a technique called *hashing* that can be used to maintain a dynamically changing set of elements.
- Dictionary. Given a universe U of possible elements, maintain a subset $S \subseteq U$ so that inserting, deleting, and searching in S is efficient.$|S| << |U|$

  - *MakeDictionary*(): initialize a dictionary with $S = \varnothing$.
  - *insert*(u): add element $u \in U$ to S.
  - *delete*(u): delete u from S (if u is currently in S).
  - *lookup*(u): is u in S ?

Implementation of Dictionaries

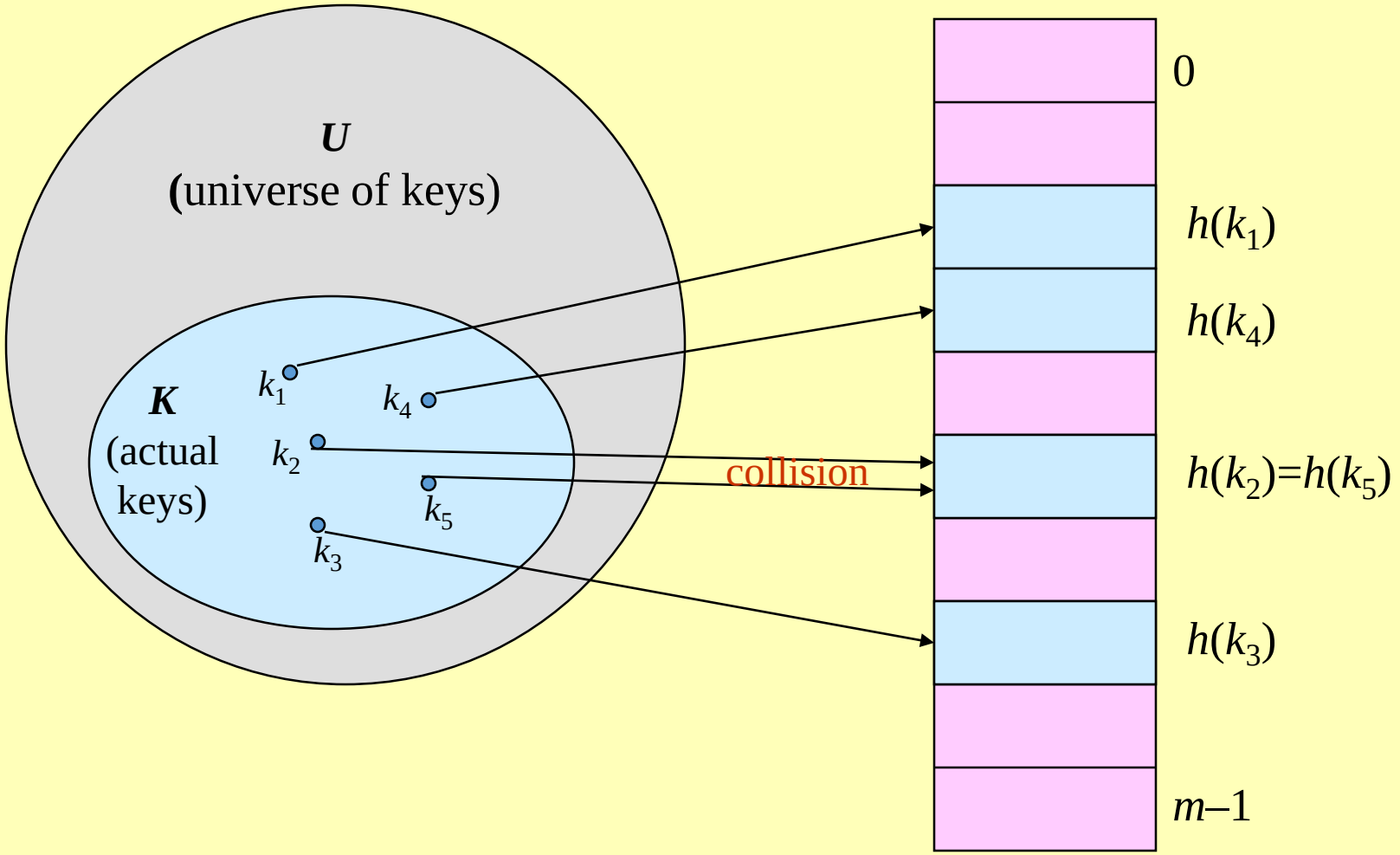
- Array. Insertion, Deletion in O(1), but Search in O(n).
- Linked List. Insertion, Deletion in O(1), but Search in O(n).
- BST. Insertion, Deletion and Search in O(log n).
- Direct Address Table. Insertion, Deletion and Search in O(1).



Hashing

- Challenge. As opposed to the direct address tables, the Universe U can be extremely large so defining an array of size $|U|$ is infeasible.
- Applications. File systems, databases, Google, compilers, checksums, P2P networks, associative arrays, cryptography, web caching, etc.
- Suppose we want to be able to store a set S of size up to m.
- Hash function. $h : U \rightarrow \{ 0, 1, \ldots, m-1 \}$.
- Hashing. Create an array H of length m. When processing element u, access array element $H[h(u)]$.
- The function h that maps elements of U to array positions is called a hash function, and the array H a hash table.
- To add an element u to the set S, we simply place u in position $h(u)$ of the array H.

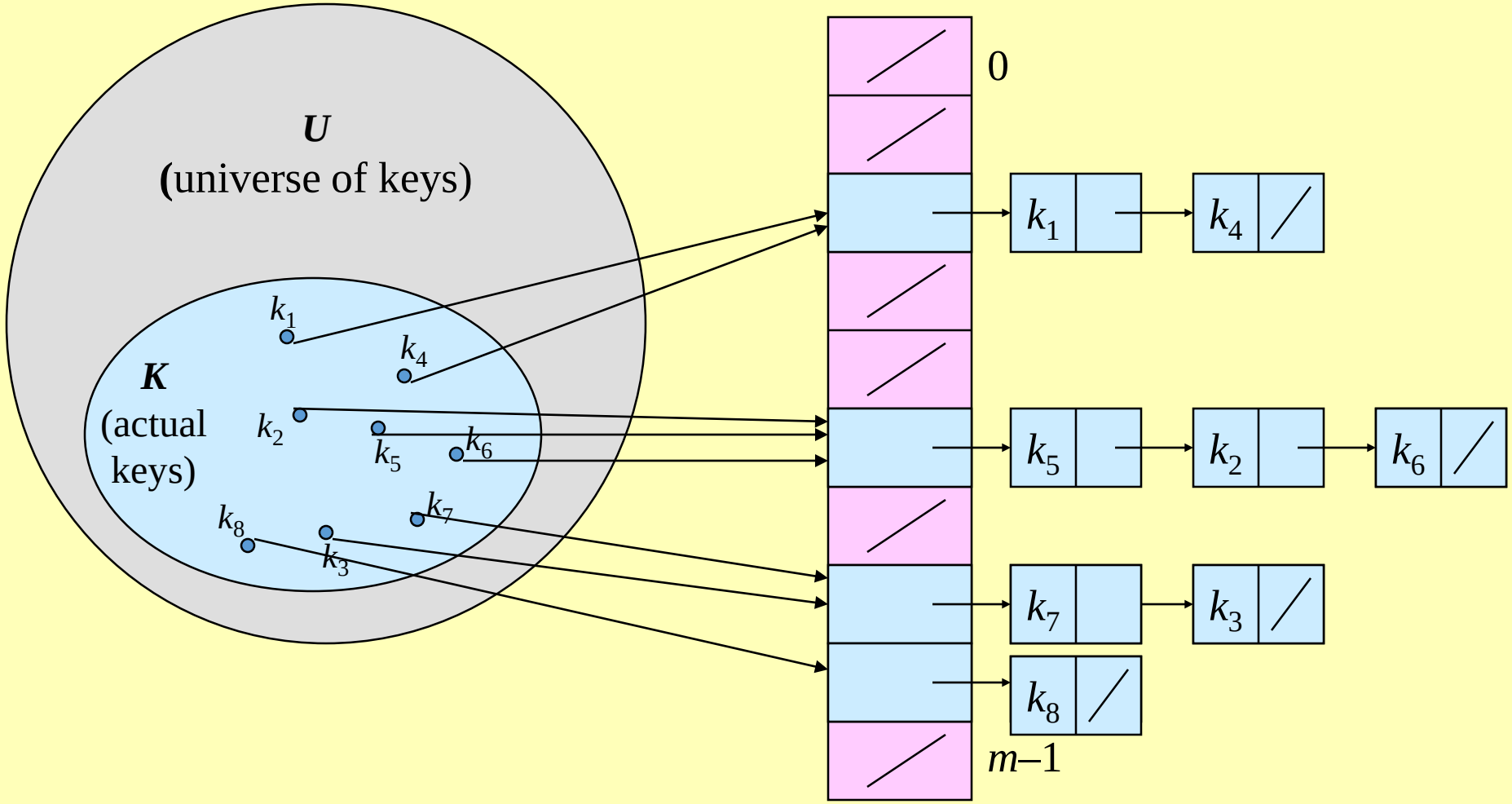
Hashing


U
(universe of keys)
K
(actual
keys)
k1
k4
k2
k5
k3
collision
0
h(k1)
h(k4)
h(k2)=h(k5)
h(k3)
m–1

Hashing

- Suppose, for all distinct *u* and *v* in *S*, h(u) ≠ h(v). In such a case, the look up(Search operation) will take a constant time: when we check array position H[h(u)], it would either be empty or would contain just u.
- But in actual practice, there can be distinct elements u, v ∈ S for which h(u) = h(v). We say that the two elements collide, since they are mapped to the same place in *H*.
- A collision is said to occur, when h(u) = h(v) but u ≠ v.
- There are a number of ways to deal with collisions.
- In ***chaining***, we will assume that each position H[*i*] of the hash table stores a linked list of all elements *u* ∈ *S* with h(u) = *i*. The operation Lookup(u) would now work as follows
  - Compute the hash function h(u).
  - Scan the linked list at position H[h(u)] to see if *u* is present in this list.

Collision Resolution by Chaining



Hashing with Chaining

Dictionary Operations:

- Chained-Hash-Insert (T, x)
  - Insert x at the head of list $T[h(key[x])]$.
  - Worst-case complexity – $O(1)$.
- Chained-Hash-Delete (T, x)
  - Delete x from the list $T[h(key[x])]$.
  - Worst-case complexity – proportional to length of list with singly-linked lists. $O(1)$ with doubly-linked lists.
- Chained-Hash-Search (T, k)
  - Search an element with key k in list $T[h(k)]$.
  - Worst-case complexity – proportional to length of list.

Analysis on Chained-Hash-Search

- Load factor $\alpha = n/m$ = average keys per slot.
  - m – number of slots.
  - n – number of elements stored in the hash table.
- Worst-case complexity: $\Theta(n)$ + time to compute $h(k)$.
- Average depends on how h distributes keys among m slots.
- **Assumptions:**
  - *Simple uniform hashing*.
    - Any key is equally likely to hash into any of the m slots, independent of where any other key hashes to.
  - $O(1)$ time to compute $h(k)$.
- Time to search for an element with key k is $\Theta(|T[h(k)]|)$.
- Expected length of a linked list = load factor = $\alpha = n/m$.
- Theorem. In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time $\Theta(1+\alpha)$ under the assumption of simple uniform hashing.

Open Addressing

- An alternative to chaining for handling collisions.
- **Idea:**
  - Store all keys in the hash table itself. (Size(H) $\geq |S|$)
  - Each slot contains either a key or NIL.
  - To ***search*** for key k:
    - Examine slot $h(k)$. Examining a slot is known as a **probe**.
    - If slot $h(k)$ contains key k, the search is successful. If the slot contains NIL, the search is unsuccessful.
    - There's a third possibility: **slot $h(k)$ contains a key that is not k**.
      - Compute the index of some other slot, based on k and which probe we are on.
      - Keep probing until we either find key k or we find a slot holding NIL.
- **Advantages:** Avoids pointers; so can use a larger table.

Probe Sequence

- Sequence of slots examined during a key search constitutes a ***probe sequence***.
- Probe sequence must be a permutation of the slot numbers.
  - We examine every slot in the table, if we have to.
  - We don't examine any slot more than once.
- The hash function is extended to:
  - $h : U \times \underbrace{\{0, 1, \ldots, m-1\}}_{\text{probe number}} \rightarrow \underbrace{\{0, 1, \ldots, m-1\}}_{\text{slot number}}$
- $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ should be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$.

Operation Insert

- Act as though we were searching, and insert at the first NIL slot found.
- Pseudo-code for **Insert**:

```
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"
```

Pseudo-code for **Search**:

```
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
```

Deletion

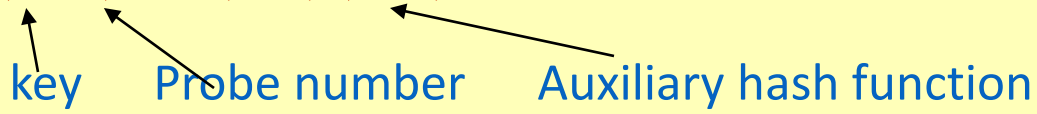
- Cannot just turn the slot containing the key we want to delete to contain NIL.
- Use a special value DELETED instead of NIL when marking a slot as empty during deletion.
  - ***Search*** should treat DELETED as though the slot holds a key that does not match the one being searched for.
  - ***Insert*** should treat DELETED as though the slot were empty, so that it can be reused.
- **Disadvantage:** Search time is no longer dependent on α.
  - Hence, chaining is more common when keys have to be deleted.

Computing Probe Sequences

- The ideal situation is ***uniform hashing***:
  - Generalization of simple uniform hashing.
  - Each key is equally likely to have any of the $m!$ permutations of $\langle 0, 1, \ldots, m-1 \rangle$ as its probe sequence.
- It is hard to implement true uniform hashing.
  - Approximate with techniques that at least guarantee that the probe sequence is a permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
- Some techniques:
  - Use ***auxiliary hash functions***.
    - Linear Probing.
    - Quadratic Probing.
    - Double Hashing.
  - Can't produce all $m!$ probe sequences.

Linear Probing

- $h(k, i) = (h'(k)+i) \bmod m$.



- The initial probe determines the entire probe sequence.
  - $T[h'(k)], T[h'(k)+1], \ldots, T[m-1], T[0], T[1], \ldots, T[h'(k)-1]$
  - Hence, only m distinct probe sequences are possible.
- Suffers from ***primary clustering***:
  - Long runs of occupied sequences build up.
  - Long runs tend to get longer, since an empty slot preceded by i full slots gets filled next with probability $(i+1)/m$.
  - Hence, average search and insertion times increase.

Quadratic Probing

- $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$ $\quad c_1 \neq c_2$

  key Probe number Auxiliary hash function

- The initial probe position is $T[h'(k)]$, later probe positions are offset by amounts that depend on a quadratic function of the probe number i.

- Must constrain c_1, c_2, and m to ensure that we get a full permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
- Can suffer from ***secondary clustering***:
  - If two keys have the same initial probe position, then their probe sequences are the same.

Double Hashing

- $h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$

  key Probe number Auxiliary hash functions

- Two auxiliary hash functions.
  - h_1 gives the initial probe. h_2 gives the remaining probes.
- Must have $h_2(k)$ relatively prime to m, so that the probe sequence is a full permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
  - Choose m to be a power of 2 and have $h_2(k)$ always return an odd number. Or,
  - Let m be prime, and have $1 < h_2(k) < m$.
- $\Theta(m^2)$ different probe sequences.
  - One for each possible combination of $h_1(k)$ and $h_2(k)$.
  - Close to the ideal uniform hashing.

Analysis of Open-address Hashing

- Analysis is in terms of load factor α.
- **Assumptions:**
  - Assume that the table never completely fills, so $n < m$ and $\alpha < 1$.
  - Assume uniform hashing.
    - In a successful search, each key is equally likely to be searched for.

Theorem:
The expected number of probes in an unsuccessful search in an open-address hash table is at most $1/(1-\alpha)$.

Analysis of Open-address Hashing with Linear Probing

- What is the average number of probes for a successful search and an unsuccessful search for this hash table?
  - Hash Function: h(x) = x mod 11

Successful Search:

- 20: 9 -- 30: 8 -- 2 : 2 -- 13: 2, 3 -- 25: 3,4
- 24: 2,3,4,5 -- 10: 10 -- 9: 9,10, 0

Avg. Probe for SS = (1+1+1+2+2+4+1+3)/8=15/8

Unsuccessful Search:

- We assume that the hash function uniformly distributes the keys.
- 0: 0,1 -- 1: 1 -- 2: 2,3,4,5,6 -- 3: 3,4,5,6
- 4: 4,5,6 -- 5: 5,6 -- 6: 6 -- 7: 7 -- 8: 8,9,10,0,1
- 9: 9,10,0,1 -- 10: 10,0,1

Avg. Probe for US =

(2+1+5+4+3+2+1+1+5+4+3)/11=31/11

0	9
1	
2	2
3	13
4	25
5	24
6	
7	
8	30
9	20
10	10

Good Hash Functions

- Satisfy the assumption of *simple uniform hashing*.
  - Not possible to satisfy the assumption in practice.
- Often use heuristics, based on the domain of the keys, to create a hash function that performs well.
- Regularity in key distribution should not affect uniformity. Hash value should be independent of any patterns that might exist in the data.
  - E.g. Each key is drawn independently from U according to a probability distribution P:

$$\sum_{k:h(k) = j} P(k) = 1/m \quad \text{for } j = 0, 1, \ldots, m-1.$$

  - An example is the division method.

Keys as Natural Numbers

- Hash functions assume that the keys are natural numbers.
- When they are 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, CLRS = $67 \cdot 128^3 + 76 \cdot 128^2 + 82 \cdot 128^1 + 83 \cdot 128^0 = 141{,}764{,}947$.

Division Method

- Map a key k into one of the m slots by taking the remainder of k divided by m. That is,

$$h(k) = k \bmod m$$

- Example: $m = 31$ and $k = 78 \Rightarrow h(k) = 16$.
- **Advantage:** Fast, since requires just one division operation.
- **Disadvantage:** Have to avoid certain values of m.
  - Don't pick certain values, such as $m=2^p$
  - Or hash won't depend on all bits of k.
- **Good choice for m:**
  - Primes, not too close to power of 2 (or 10) are good.

Multiplication Method

- If $0 < A < 1$, $h(k) = \lfloor m\ (kA \bmod 1) \rfloor = \lfloor m\ (kA - \lfloor kA \rfloor) \rfloor$

  where $kA \bmod 1$ means the fractional part of kA, *i.e.*, $kA - \lfloor kA \rfloor$.
- Disadvantage: Slower than the division method.
- Advantage: Value of m is not critical.
  - Typically chosen as a power of 2, i.e., $m = 2^p$, which makes implementation easy.

- Example: $m = 1000$, $k = 123$, $A \approx 0.6180339887\ldots$

$h(k) = \lfloor 1000(123 \cdot 0.6180339887 \bmod 1) \rfloor$

$= \lfloor 1000 \cdot 0.018169... \rfloor = 18.$

Choosing a Good Hash Function

- There are many simple ways to do this: we could use the first or last few digits of u, or simply take u modulo m.
- While these simple choices may work well in many situations, it is also possible to get large numbers of collisions.
- So, a fixed choice of hash function may run into problems because of the types of elements u encountered in the application.
- The basic idea, is to use randomization in the construction of h.
- For every element $u \in U$, when we go to insert u into S, we select a value $h(u)$ uniformly at random in the set $\{0, 1, \ldots, m-1\}$, independent of all previous choices.
- In this case, the probability that two randomly selected values $h(u)$ and $h(v)$ are equal (and hence cause a collision) is quite small.

Choosing a Good Hash Function

- With this uniform random hashing scheme, the probability that two randomly selected values $h(u)$ and $h(v)$ collide—that is, that $h(u) = h(v)$—is exactly $1/m$.
- But this may lead to problems with other operations such as deletion or lookup as we have no idea where one element u got inserted.

Universal Hashing

- The key idea is to choose a hash function at random, from a carefully selected class of functions.
- Each function h in our class of functions H will map the universe U into the set $\{0, 1, \ldots, m - 1\}$, and we will design it so that it has two properties.
  - For any pair of elements $u, v \in U$, the probability that a randomly chosen $h \in H$ satisfies $h(u) = h(v)$ is at most $1/m$. We say that a class H of functions is universal if it satisfies this first property.
  - Each $h \in H$ can be compactly represented and, for a given $h \in H$ and $u \in U$, we can compute the value $h(u)$ efficiently.

Universal Hashing

A universal family of hash functions is a set of hash functions H mapping a universe U to the set $\{ 0, 1, \ldots, m-1 \}$ such that

- For any pair of elements $u \neq v$: $\Pr_{h \in H}[\, h(u) = h(v) \,] \leq 1/m$
- Can select random h efficiently.

chosen uniformly at random

- Can compute $h(u)$ efficiently.

Ex. $U = \{ a, b, c, d, e, f \}$, $m = 2$.

	a	b	c	d	e	f
$h_1(x)$	0	1	0	1	0	1
$h_2(x)$	0	0	0	1	1	1

$H = \{h_1, h_2\}$

$\Pr_{h \in H}[h(a) = h(b)] = 1/2$
$\Pr_{h \in H}[h(a) = h(c)] = 1$
$\Pr_{h \in H}[h(a) = h(d)] = 0$

. . .

not universal

	a	b	c	d	e	f
$h_1(x)$	0	1	0	1	0	1
$h_2(x)$	0	0	0	1	1	1
$h_3(x)$	0	0	1	0	1	1
$h_4(x)$	1	0	0	1	1	0

$H = \{h_1, h_2, h_3, h_4\}$

$\Pr_{h \in H}[h(a) = h(b)] = 1/2$

$\Pr_{h \in H}[h(a) = h(c)] = 1/2$

$\Pr_{h \in H}[h(a) = h(d)] = 1/2$
$\Pr_{h \in H}[h(a) = h(e)] = 1/2$
$\Pr_{h \in H}[h(a) = h(f)] = 0$

. . .

universal

Universal Hashing

- Choose p prime so that $m \leq p \leq 2m$, where m = | S |.
- Fact: there exists a prime between m and $2m$.
- the universe with vectors of the form x = $(x_1, x_2, \ldots x_r)$ for some integer r, where $0 \leq x_i < p$ for each i.
- Let A be the set of all vectors of the form a = $(a_1, a_2, \ldots a_r)$, where a_i is an integer in the range $[0, p-1]$ for each $i = 1, \ldots, r$.
- For each a ∈ A, we define the linear function

$$h_a(x) = \left(\sum_{i=1}^{r} a_i x_i \right) \bmod p$$

- Universal hash function family. $H = \{ h_a : a \in A \}$.
- Advantages of Universal Hashing:
  - Space used = Θ(m).
  - Expected number of collisions per operation is ≤ 1
  - ⇒ O(1) time per insert, delete, or lookup.