Algorithms

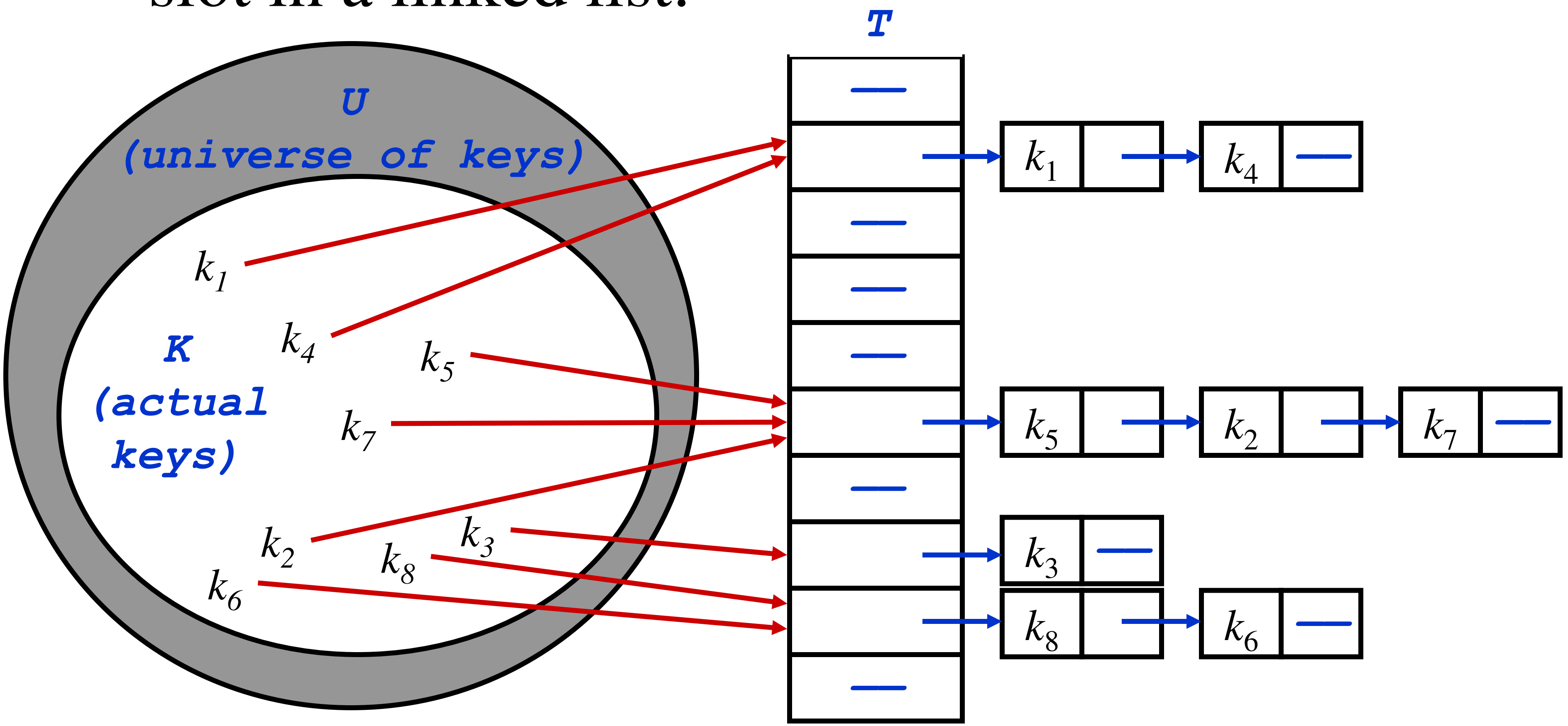
Universal Hashing

Review: Resolving Collisions

- *How can we solve the problem of collisions?*
- *Open addressing*
  - To insert: if slot is full, try another slot, and another, until an open slot is found (*probing*)
  - To search, follow same sequence of probes as would be used when inserting the element
- *Chaining*
  - Keep linked list of elements in slots
  - Upon collision, just add new element to list

Review: Chaining

- Chaining puts elements that hash to the same slot in a linked list:



Review: Analysis Of Hash Tables

- *Simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- *Load factor* $\alpha = n/m$ = average # keys per slot
  - Average cost of unsuccessful search = $O(1+\alpha)$
  - Successful search: $O(1+ \alpha/2) = O(1+ \alpha)$
  - If n is proportional to m, $\alpha = O(1)$
- So the cost of searching = O(1) if we size our table appropriately

Review: Choosing A Hash Function

- Choosing the hash function well is crucial
  - Bad hash function puts all elements in same slot
  - A good hash function:
    - Should distribute keys uniformly into slots
    - Should not depend on patterns in the data
- We discussed three methods:
  - Division method
  - Multiplication method
  - Universal hashing

Review: The Division Method

- $h(k) = k \bmod m$
  - In words: hash k into a table with m slots using the slot given by the remainder of k divided by m
- Elements with adjacent keys hashed to different slots: good
- If keys bear relation to m: bad
- Upshot: pick table size m = prime number not too close to a power of 2 (or 10)

Review: The Multiplication Method

- For a constant A, $0 < A < 1$:
- $h(k) = \lfloor m \underbrace{(kA - \lfloor kA \rfloor)}_{\text{Fractional part of } kA} \rfloor$
- Upshot:
  - Choose $m = 2^P$
  - Choose A not too close to 0 or 1
  - Knuth: Good choice for $A = (\sqrt{5} - 1)/2$

Review: Universal Hashing

- When attempting to foil an malicious adversary, randomize the algorithm
- *Universal hashing*: pick a hash function randomly when the algorithm begins (*not* upon every insert!)
  - Guarantees good performance on average, no matter what keys adversary chooses
  - Need a family of hash functions to choose from

Universal Hashing

- Let ς be a (finite) collection of hash functions
  - …that map a given universe U of keys…
  - …into the range $\{0, 1, \ldots, m - 1\}$.
- ς is said to be *universal* if:
  - for each pair of distinct keys $x, y \in U$, the number of hash functions $h \in \varsigma$ for which $h(x) = h(y)$ is $|\varsigma|/m$
  - In other words:
    - With a random hash function from ς, the chance of a collision between x and y is exactly $1/m$ $(x \neq y)$

Universal Hashing

- Theorem 12.3:
  - Choose h from a universal family of hash functions
  - Hash n keys into a table of m slots, $n \le m$
  - Then the expected number of collisions involving a particular key x is less than 1
  - Proof:
    - For each pair of keys y, z, let c_{yx} = 1 if y and z collide, 0 otherwise
    - $E[c_{yz}] = 1/m$ (by definition)
    - Let C_x be total number of collisions involving key x
    - $E[C_x] = \sum_{\substack{y \in T \\ y \neq x}} E[c_{xy}] = \frac{n-1}{m}$
    - Since $n \le m$, we have $E[C_x] < 1$

A Universal Hash Function

- Choose table size m to be prime
- Decompose key x into r+1 bytes, so that $x = \{x_0, x_1, \ldots, x_r\}$
  - Only requirement is that max value of byte < m
  - Let $a = \{a_0, a_1, \ldots, a_r\}$ denote a sequence of r+1 elements chosen randomly from $\{0, 1, \ldots, m - 1\}$
  - Define corresponding hash function $h_a \in \varsigma$:

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m$$

  - With this definition, ς has m^{r+1} members

A Universal Hash Function

- ς is a universal collection of hash functions (Theorem 12.4)
- How to use:
  - Pick r based on m and the range of keys in U
  - Pick a hash function by (randomly) picking the a's
  - Use that hash function on all keys

Augmenting Data Structures

- This course is supposed to be about design and analysis of algorithms
- So far, we've only looked at one design technique (*What is it?*)

Augmenting Data Structures

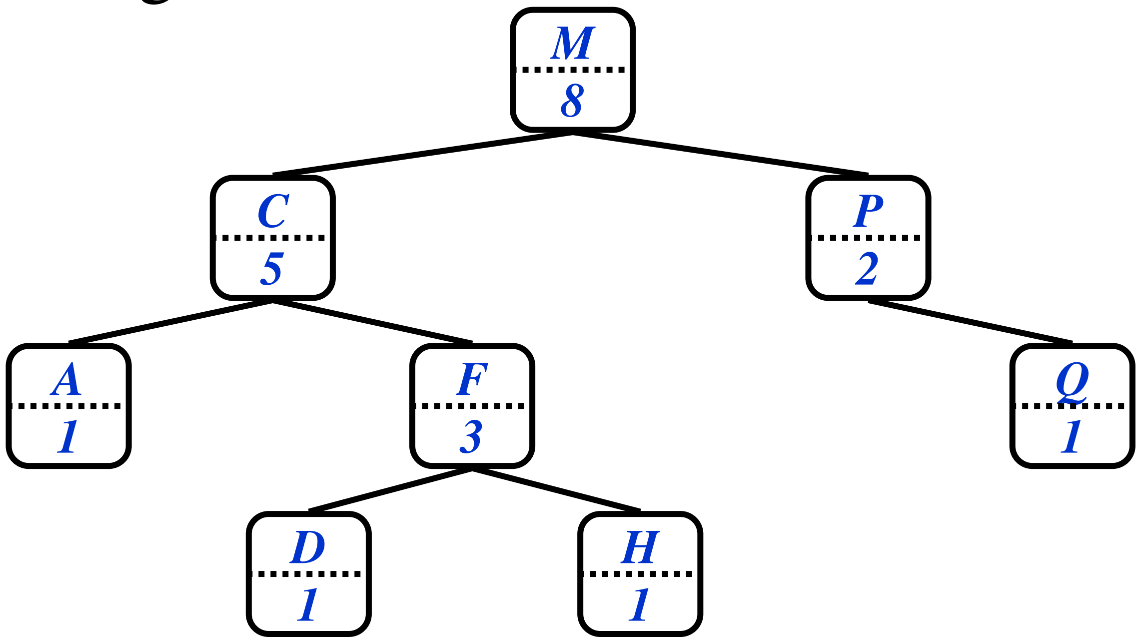
- This course is supposed to be about design and analysis of algorithms
- So far, we've only looked at one design technique: *divide and conquer*
- Next up: augmenting data structures
  - Or, "One good thief is worth ten good scholars"

Dynamic Order Statistics

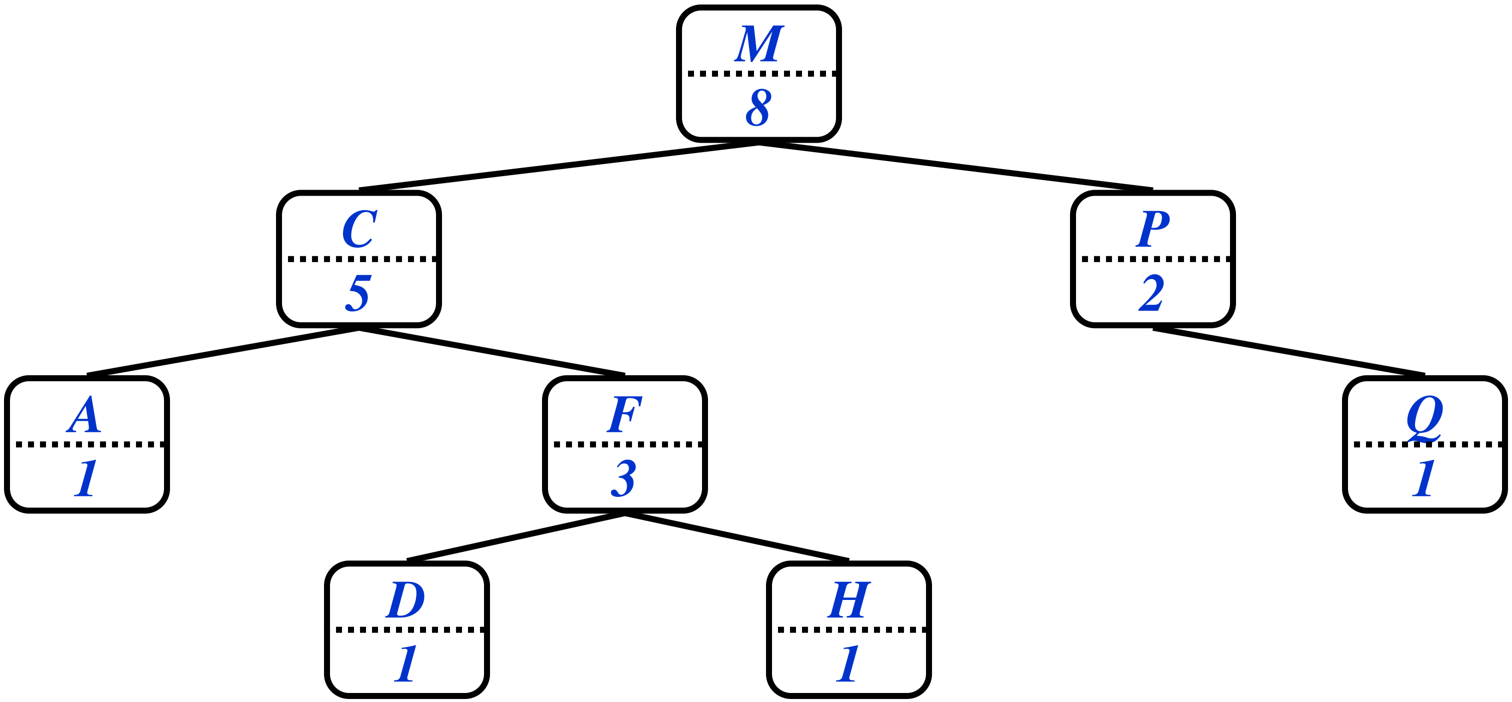
- We've seen algorithms for finding the *i*th element of an unordered set in O(*n*) time
- Next, a structure to support finding the *i*th element of a dynamic set in O(lg *n*) time
  - *What operations do dynamic sets usually support?*
  - *What structure works well for these?*
  - *How could we use this structure for order statistics?*
  - *How might we augment it to support efficient extraction of order statistics?*

Order Statistic Trees

- OS Trees augment red-black trees:
  - Associate a *size* field with each node in the tree
  - **`x->size`** records the size of subtree rooted at **`x`**, including **`x`** itself:



Selection On OS Trees



How can we use this property to select the i*th element of the set?*

OS-Select

```
OS-Select(x, i)
{
    r = x->left->size + 1;
    if (i == r)
        return x;
    else if (i < r)
        return OS-Select(x->left, i);
    else
        return OS-Select(x->right, i-r);
}
```