COMP319 Algorithms 1
Lecture 10
Sets and Hashing

Instructor: Gil-Jin Jang

Disjoint set representation and operations

Hashing and collision resolution

Textbook Chapters 5 and 12

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Disjoint set representation

Union-Find operations

Collapsing rule

Textbook chapter 5

SETS

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Set Definition and Operations

- Set definition
  - A collection of data with **UNIQUE** elements
    - No DUPLICATE elements
- Notations
  - Ø: empty set
  - *Z*: integer set
  - *R*: real numbers
  - *N*: natural numbers

- Operations
  - Union
  - Intersection
  - Difference
  - Complement
  - Combinations of the above
- Membership: $\in$
  - Check if a specific element belongs to a set

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Disjoint Set Definition

- Pairwise disjoint set definition

  $S_i \cap S_j = \emptyset \leftrightarrow$ If S_i and S_j, for $i \neq j$, are disjoint sets, then there is no element that is in both S_i and S_j.

- Disjoint set operations
  - Cardinality: $|S|$ = the number of elements of set S
  - Disjoint set union
    - If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j$ = {all elements x such that $x \in S_i$ or $x \in S_j$}.
    - $|S_i \cup S_j| = |S_i| + |S_j| - |S_i \cap S_j| = |S_i| + |S_j|$
  - Find(x)
    - Find the set containing element x
    - UNIQUE set is found, by the definition of disjoint set

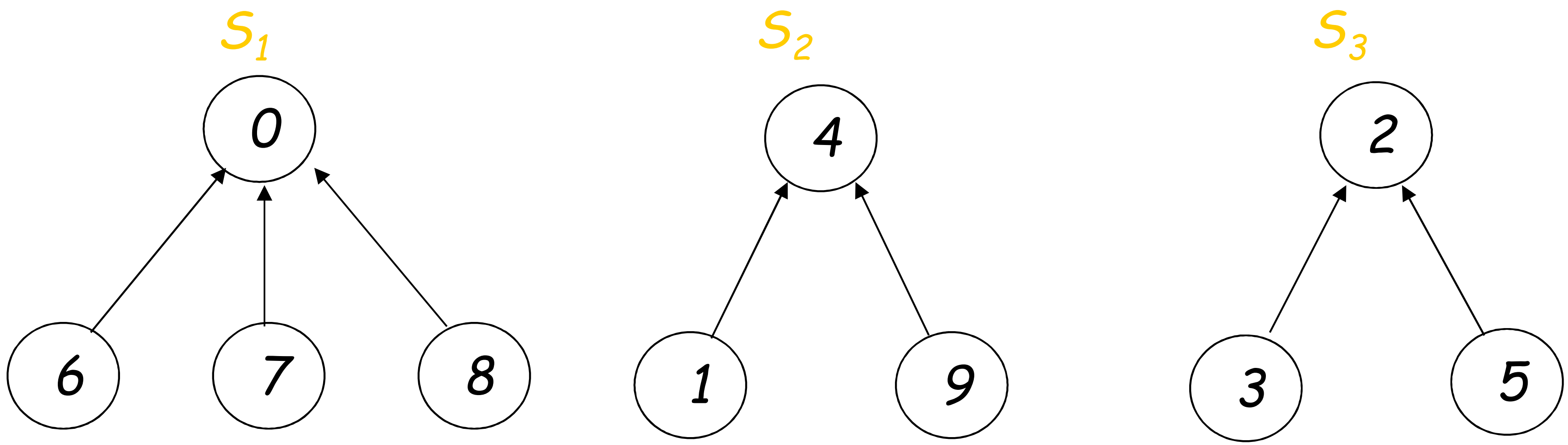
Disjoint Set ADT

- The universe consists of N elements, named 1, …, N .
- The disjoint set ADT is a collection of sets, such that
  - Sets are disjoint: each element is in exactly one set
- Operations
  - NewSet = **union** (Set1, Set2)
    - creates a **NEW** set, composed of all elements which are either in Set1 or Set2
  - Setname = **find** (element_name)
    - returns the name of the UNIQUE set that contains the given element

Tree Representation of Disjoint Sets

*The **has-a** relationship can be represented by TREEs*

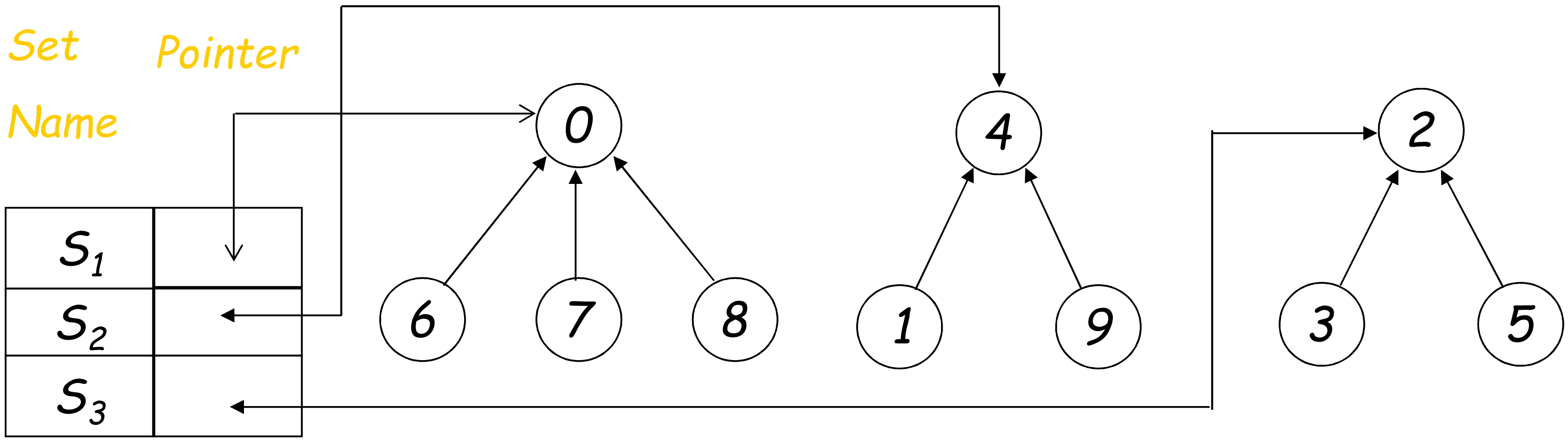
Set 2 = {4 , 1 ,9}



Set I = {0 , 6 ,7 ,8 }

Set 3 = {2 , 3 , 5}

Pointer Representation for S1, S2, S3



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

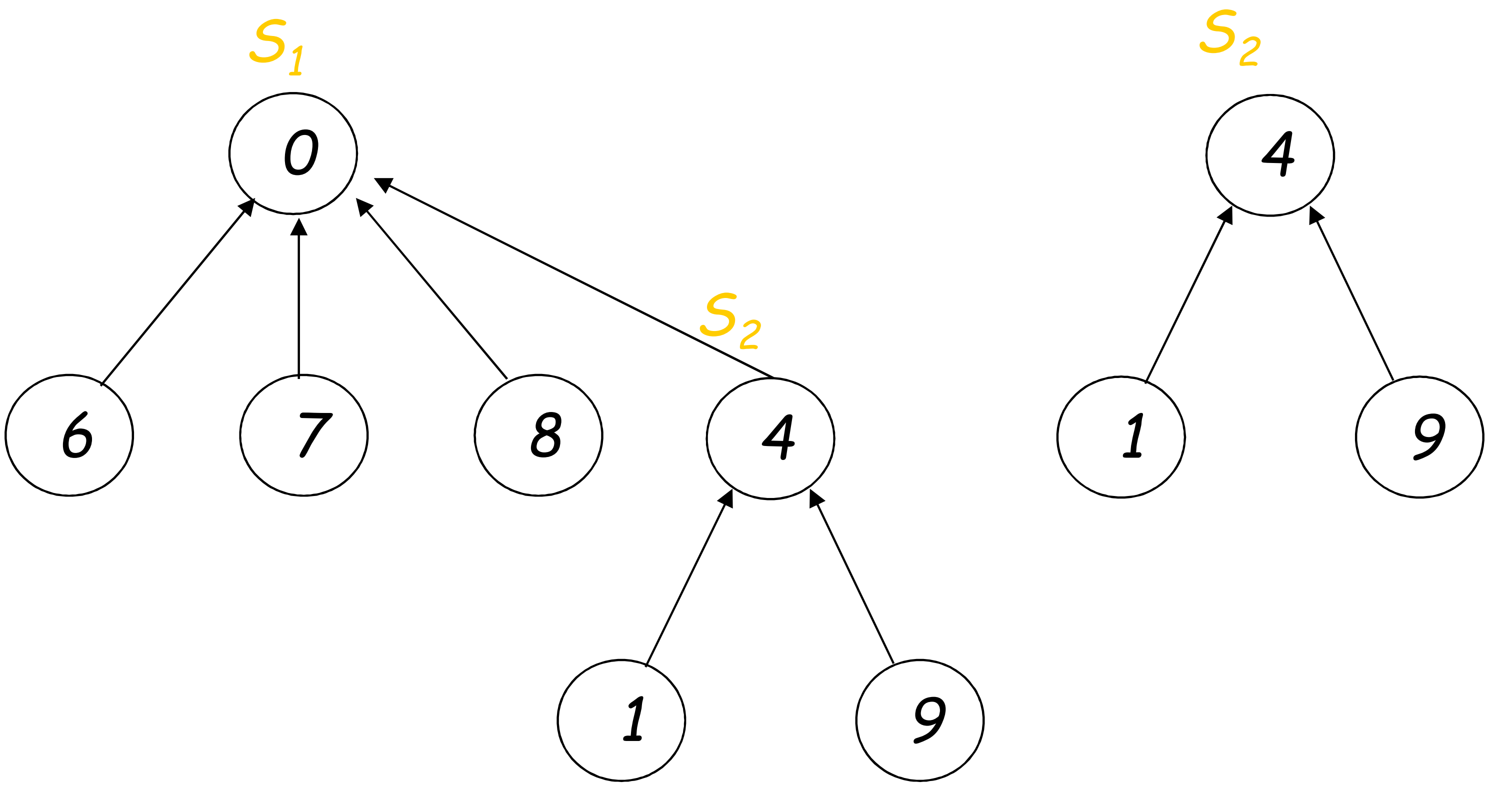
Unions of Sets

- To obtain the union of two sets, just set the parent field of one of the roots to the other root.

- To figure out which set an element is belonged to, just follow its parent link to the root and then follow the pointer in the root to the set name.

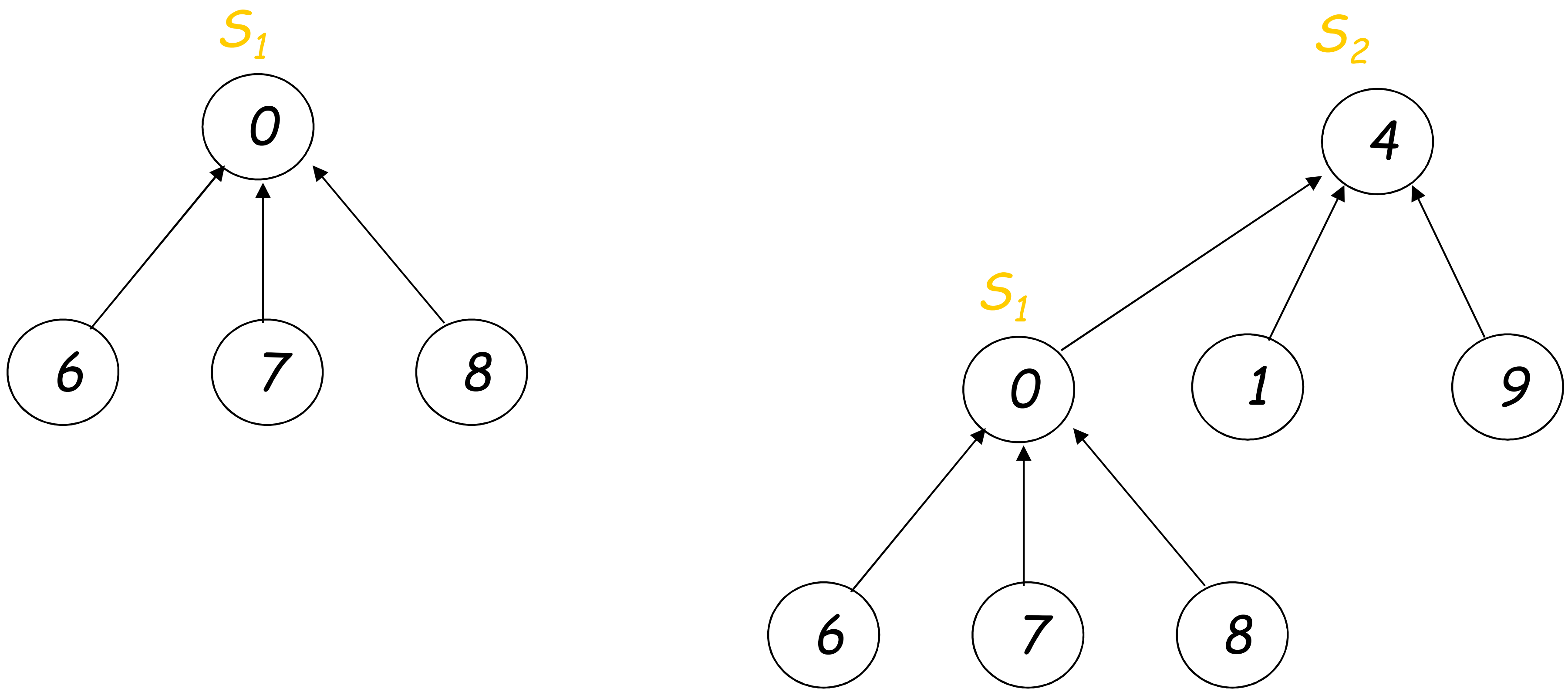
Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Representations of S1 ∪ S2 (1)



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

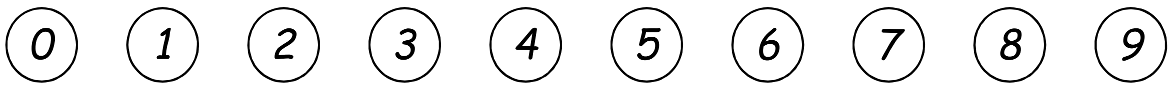
Representations of S1 ∪ S2 (2)



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Array Representation

- We could use an array to represent each set.
  - Assume set elements are numbered 0 through n-1.
- To start, each set contains one element
- Array value: **-1**
  - It is a root of a set with itself only



i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

-1 means no parents ➔ root: a set of one element

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

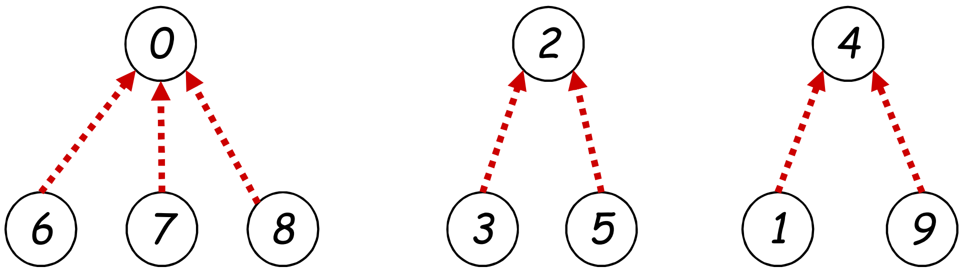
Array Representation

- Array value -1: root
- Any nonnegative integer: index of its parent
  - Parent can be either root or any other element

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4

Root

Try drawing this array into set graph



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Array Representation

```
void Union1( int i , int j )
{
    parent[i] = j ;
}
```

EX: S1 ∪ S2   Union1(0 , 2) ;

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4
	2									

EX: Find1(5) ;

i = 2

```
int Find1( int i )
{
        for(;parent[i] >= 0 ; i = parent[i]) ;
        return i ;
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

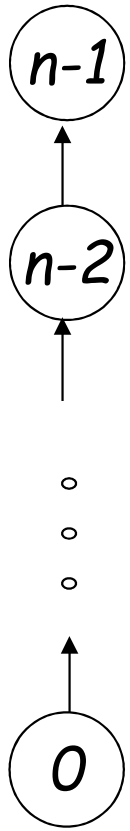
Worst-case total running time of a sequence of f finds and u unions

Collapsing rule for efficient find

DISJOINT SET ANALYSIS AND COLLAPSING RULE

Analysis Union-Find Operations

- For a set of n elements each in a set of its own, then the result of the union function is a degenerate tree (chained).
- The time complexity of the following sequence of $(n-1)$ union-find operation is $O(n^2)$.
- The complexity can be improved by using weighting rule for union.



union(0, 1), find(0)   *Union operation*

union(1, 2), find(0)   *O(n)*

⋮

Find operation

union(n-2, n-1), find(0) $O(n^2)$

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

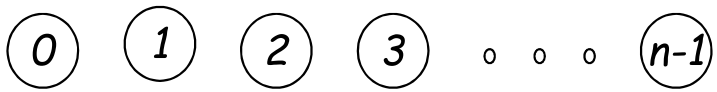
Weighting Rule

- The purpose of weighting rule is to prevent a degenerate tree as a result of successive union operations
- Definition [Weighting rule for union(i, j)]
  - If the number of nodes in the tree with root i is less than the number in the tree with root j, then make j the parent of i ; otherwise make i the parent of j.
  - root(union(S_i,S_j)) := root(S_i) if $|S_i| > |S_j|$; otherwise root(S_j)
- Modification of the value of root in the array
  - Instead of simple -1, assign $-|S|$ to store the number of elements in a set (cardinality)

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

```
void union2 (int i, int j)
{
    int temp = parent[i] + parent[j];
    if ( parent[i]>parent[j]) {
        parent[i]=j;
        parent[j]=temp;
    }
    else {
        parent[j]=i;
        parent[i]=temp;
    }
}
```



EX: unoin2 (0 , 1) , unoin2 (0 , 2) , unoin2 (0 , 3)

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

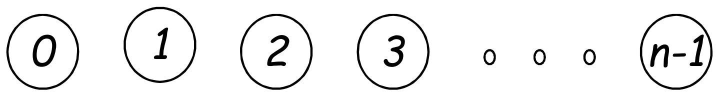
i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

```
void union2 (int i, int j)
{
    int temp = parent[i] + parent[j];
    if ( parent[i]>parent[j]) {
        parent[i]=j;
        parent[j]=temp;
    }
    else {
        parent[j]=i;
        parent[i]=temp;
    }
}
```

unoin2 (0 , 1)

temp = -2

The absolute value is the number of elements in the union set



EX: unoin2 (0 , 1) , unoin2 (0 , 2) , unoin2 (0 , 3)

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

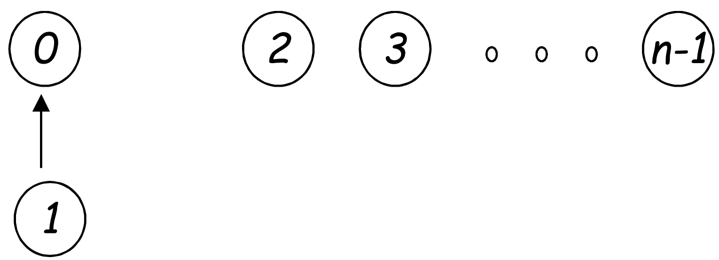
i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-2	0	-1	-1	-1	-1	-1	-1	-1	-1

```
void union2 (int i, int j)
{
   int temp = parent[i] + parent[j];
   if ( parent[i]>parent[j]) {
      parent[i]=j;
      parent[j]=temp;
   }
   else {
      parent[j]=i;
      parent[i]=temp;
   }
}
```

unoin2 (0 , 1)

temp = -2

The absolute value is the number of elements in the union set



EX: unoin2 (0 , 1) , unoin2 (0 , 2) , unoin2 (0 , 3)

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

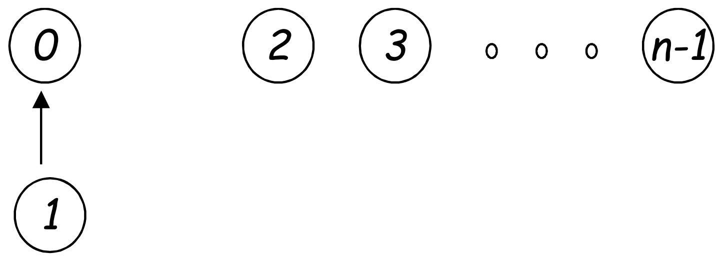
i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-2	0	-1	-1	-1	-1	-1	-1	-1	-1

```
void union2 (int i, int j)
{
    int temp = parent[i] + parent[j];
    if ( parent[i]>parent[j]) {
        parent[i]=j;
        parent[j]=temp;
    }
    else {
        parent[j]=i;
        parent[i]=temp;
    }
}
```

unoin2 (0 , 1)
unoin2 (0 , 2)

temp = -3

The absolute value is the number of elements in the union set



EX: unoin2 (0 , 1) , unoin2 (0 , 2) , unoin2 (0 , 3)

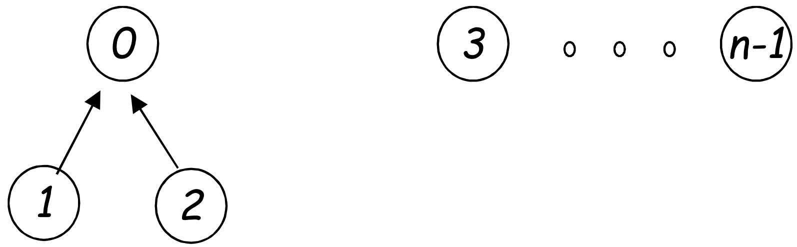
i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-3	0	0	-1	-1	-1	-1	-1	-1	-1

```
void union2 (int i, int j)
{
    int temp = parent[i] + parent[j];
    if ( parent[i]>parent[j]) {
        parent[i]=j;
        parent[j]=temp;
    }
    else {
        parent[j]=i;
        parent[i]=temp;
    }
}
```

unoin2 (0 , 1)
unoin2 (0 , 2)

temp = -3

The absolute value is the number of elements in the union set



EX: unoin2 (0 , 1) , unoin2 (0 , 2) , unoin2 (0 , 3)

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

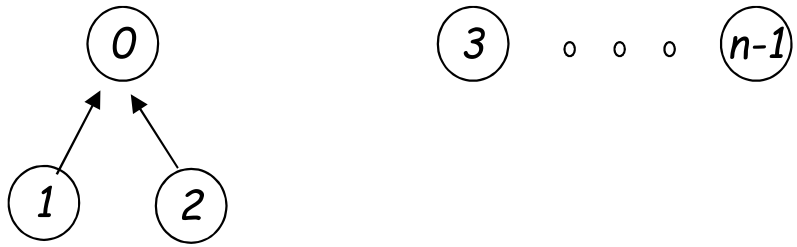
i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-3	0	0	-1	-1	-1	-1	-1	-1	-1

```
void union2 (int i, int j)
{
    int temp = parent[i] + parent[j];
    if ( parent[i]>parent[j]) {
        parent[i]=j;
        parent[j]=temp;
    }
    else {
        parent[j]=i;
        parent[i]=temp;
    }
}
```

unoin2 (0 , 1)
unoin2 (0 , 2)
unoin2 (0 , 3)

temp = -4

The absolute value is the number of elements in the union set



EX: unoin2 (0 , 1) , unoin2 (0 , 2) , unoin2 (0 , 3)

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-4	0	0	0	-1	-1	-1	-1	-1	-1

```
void union2 (int i, int j)
{
    int temp = parent[i] + parent[j];
    if ( parent[i]>parent[j]) {
        parent[i]=j;
        parent[j]=temp;
    }
    else {
        parent[j]=i;
        parent[i]=temp;
    }
}
```

unoin2 (0 , 1)
unoin2 (0 , 2)
unoin2 (0 , 3)

temp = -4

The absolute value is the number of elements in the union set

EX: unoin2 (0 , 1) , unoin2 (0 , 2) , unoin2 (0 , 3)

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

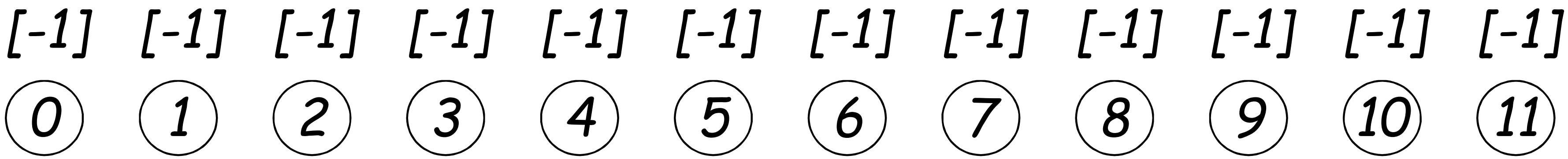
Weighted Union Time Complexity

- Lemma 5.5: Assume that we start with a forest of trees, each having one node.
  - Let T be a tree with m nodes created as a result of a sequence of unions each performed using function **WeightedUnion**.
  - The height of T is no greater than

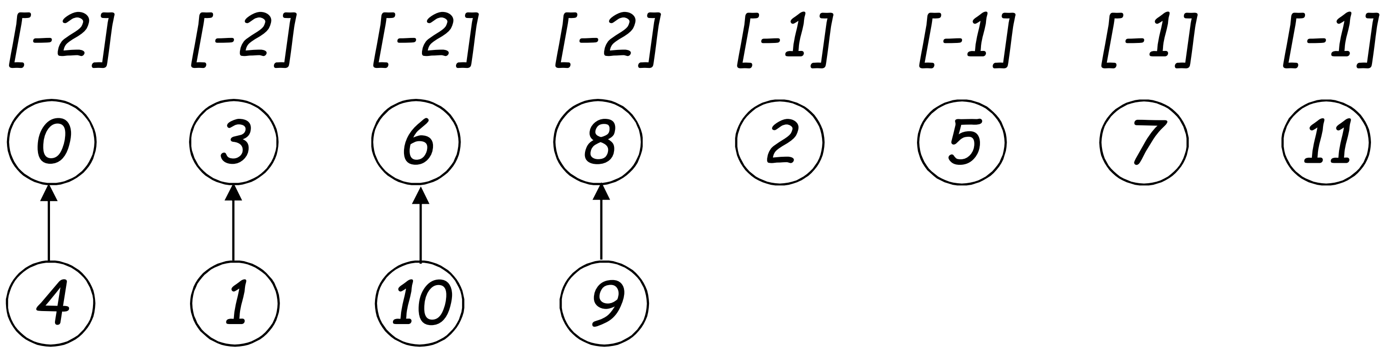
$$floor\ (\log_2 m) + 1 .$$

- For the processing of an intermixed sequence of $u - 1$ unions and f find operations, the time complexity is $O(u + f \log u)$.

Example 5.3

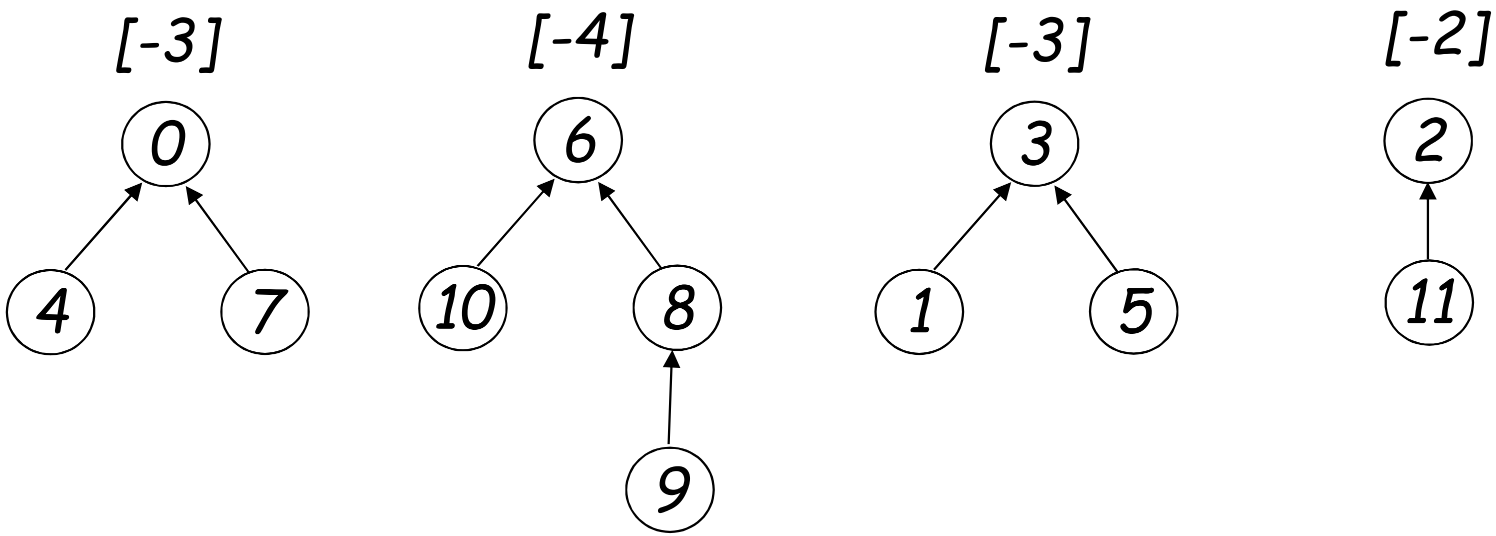


(a) Initial trees



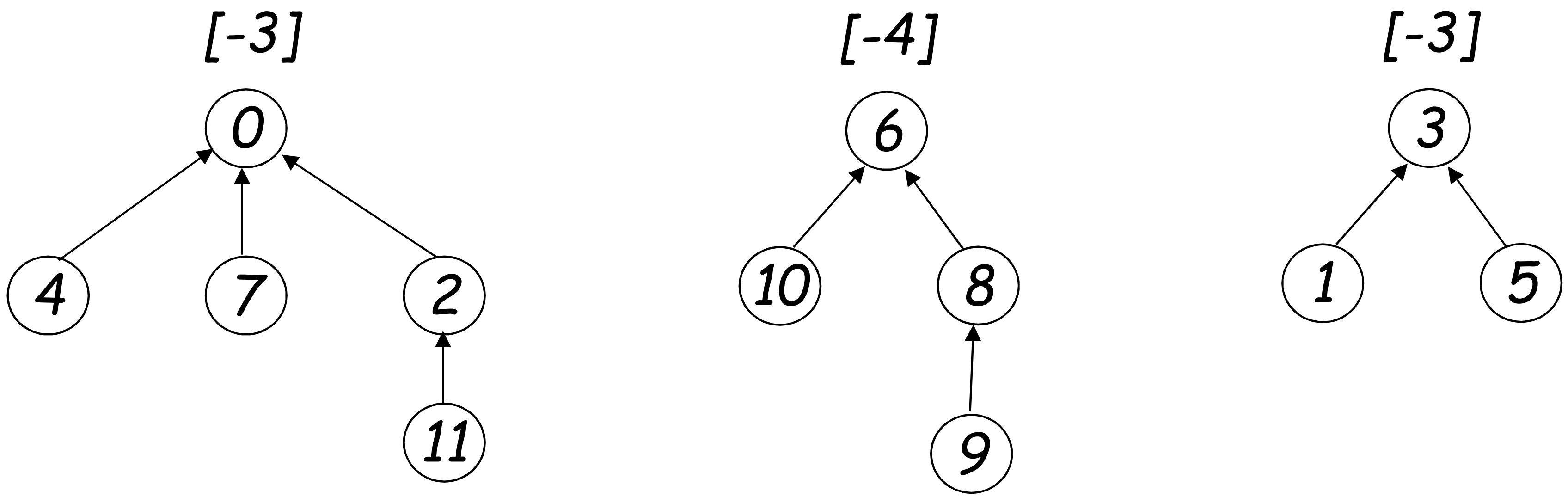
(b) Height-2 trees following $0 \cup 4$, $3 \cup 1$, $6 \cup 10$, and $8 \cup 9$

Example 5.3 (Cont.)



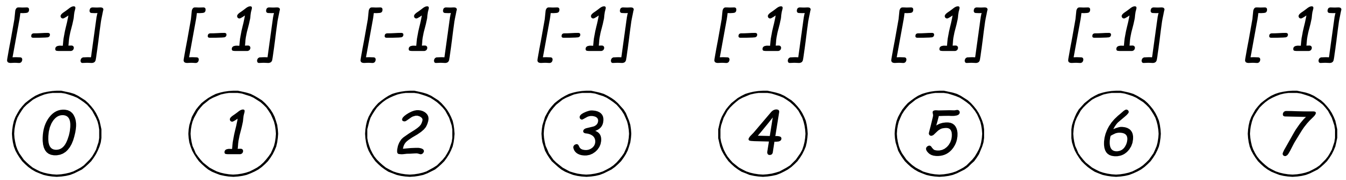
(c) Tree following $7 \cup 4$, $6 \cup 8$, $3 \cup 5$, and $2 \cup 11$

Example 5.3 (Cont.)

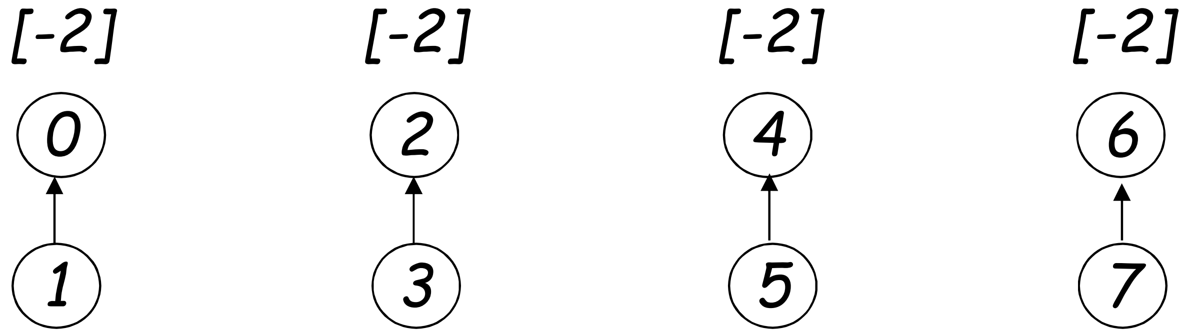


(d) Tree following 11 ∪ 0

Trees Achieving Worst-Case Bound

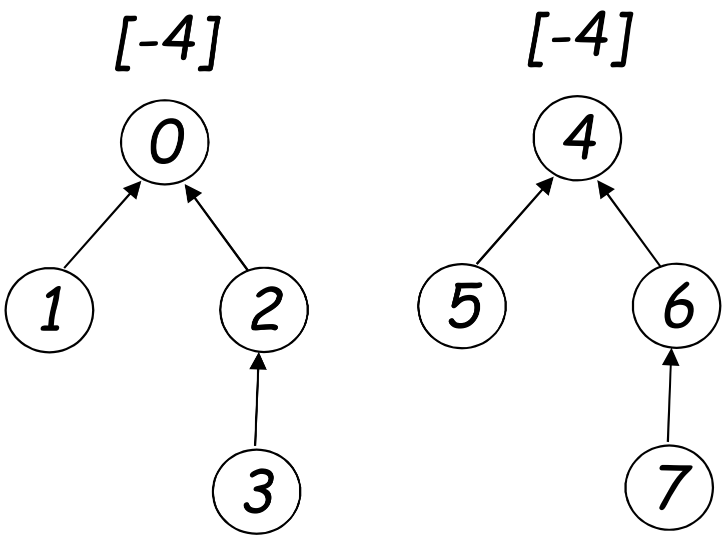


(a) Initial height trees

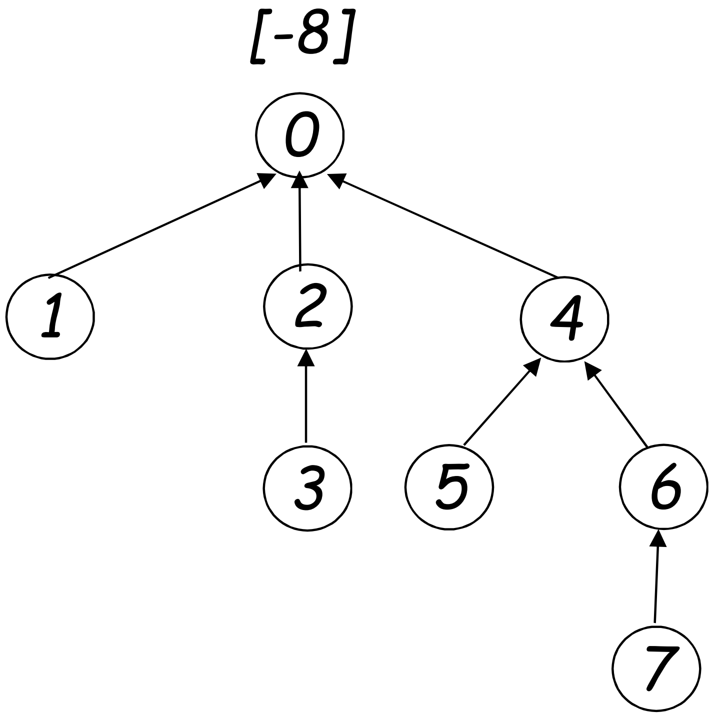


(b) Height-2 trees following union (0, 1), (2, 3), (4, 5), and (6, 7)

Worst-Case Bound (Cont.)



(c) Height-3 trees following union (0, 2), (4, 6)



(d) Height-4 trees following union (0, 4)

Collapsing Rule

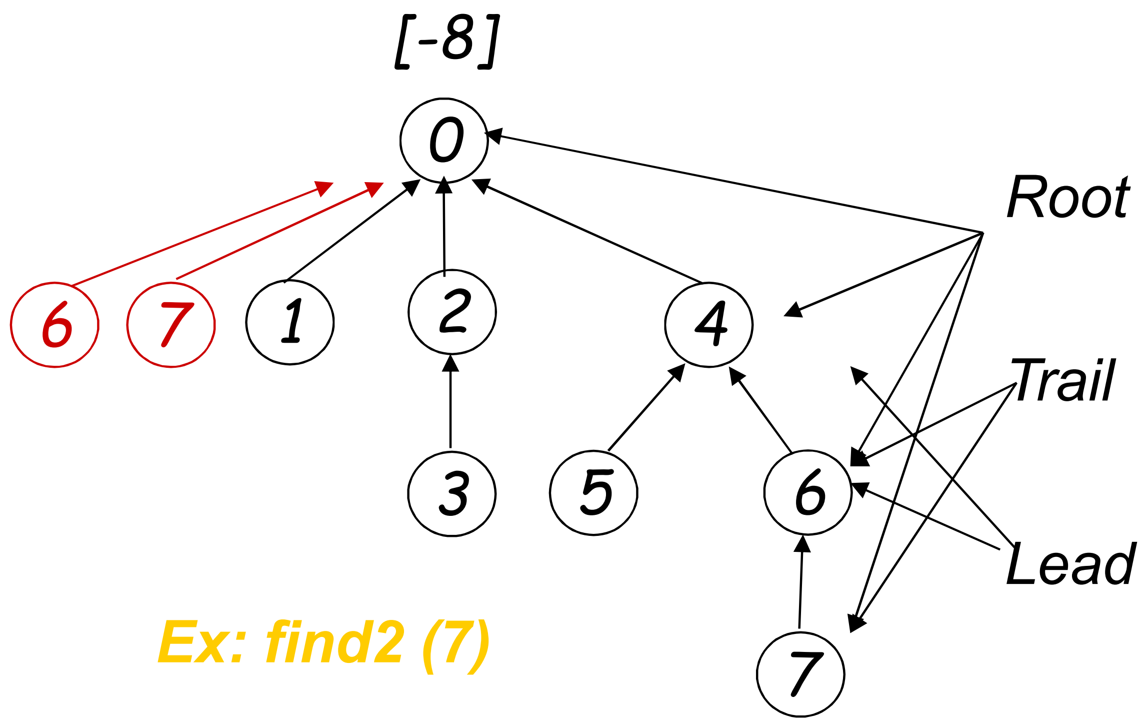
- Definition [Collapsing rule]
  - While processing operation find, if *j* is a node on the path from *i* to its root,
    and parent[*i*]≠ root(*i*), then set parent[*j*] to root(*i*).

- The first run of find operation will collapse the tree. Therefore, all following find operation of the same element only goes up one link to find the root.

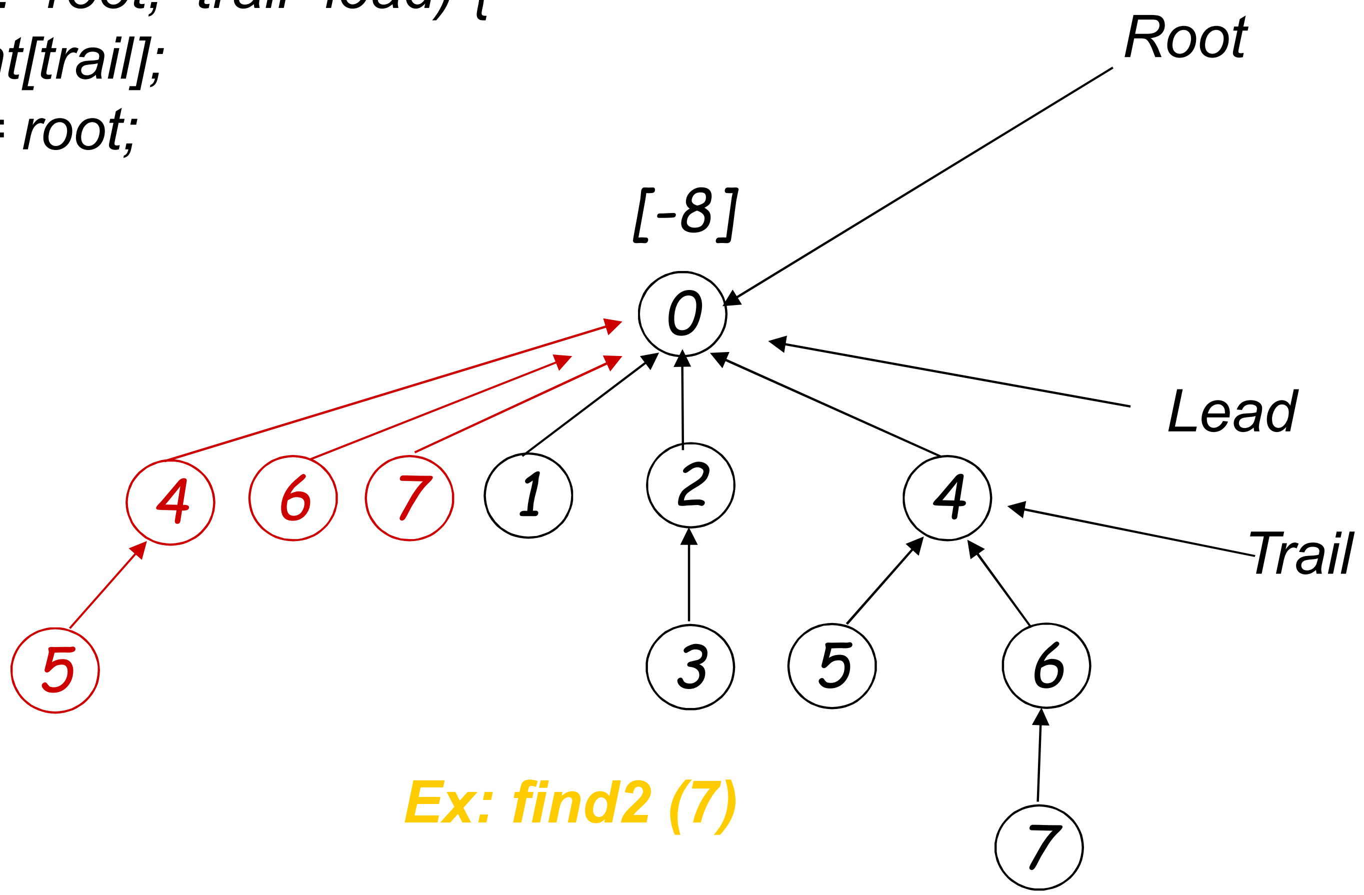
- All the parents of node *i* become its siblings – flattened.

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

```
int find2(int i)
{
   int root, trail, lead;
   for (root=i; parent[root]>=0; root=parent[root]);
   for (trail=i; trail!=root;  trail=lead) {
      lead = parent[trail];
      parent[trail]= root;
   }
   return root:
}
```



```
int find2(int i)
{
    int root, trail, lead;
    for (root=i; parent[root]>=0; root=parent[root]);
    for (trail=i; trail!=root;  trail=lead) {
        lead = parent[trail];
        parent[trail]= root;
    }
    return root:
}
```



WeightedUnion and CollapsingFind

- Analysis of WeightedUnion and CollapsingFind
  - The use of collapsing rule roughly double the time for an individual find. However, it reduces the worst-case time over a sequence of finds.

```
int find2(int i)
{
    int root, trail, lead;
    for (root=i; parent[root]>=0; root=parent[root]);
    for (trail=i; trail!=root;  trail=lead) {
        lead = parent[trail];
        parent[trail]= root;
    }
    return root:
}
```

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Textbook chapter 12

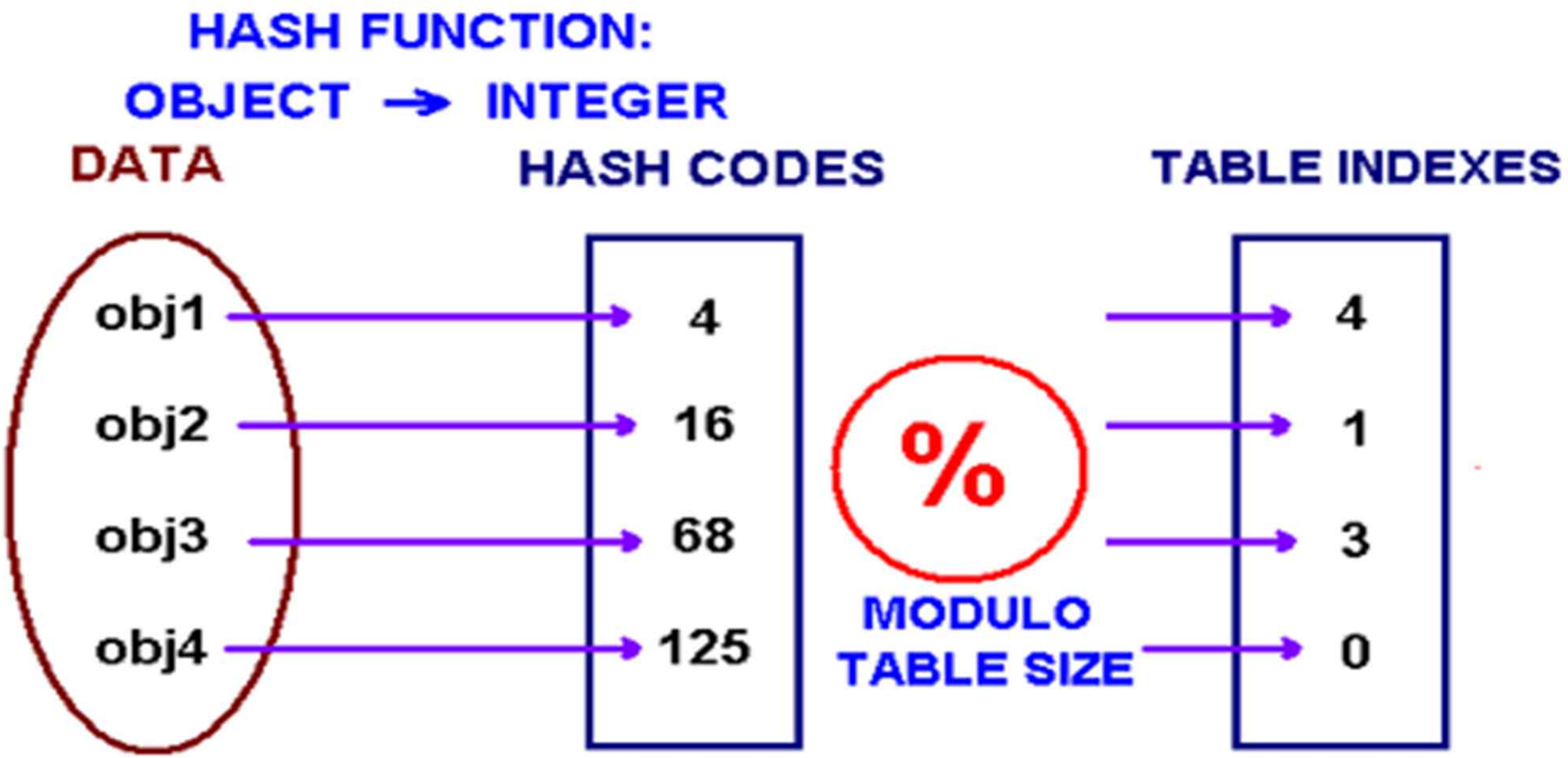
HASHING

Searching in General

- Consider the problem of searching an array for a given value
  - If the array is ***not sorted***, the search requires O(*n*) time
    - If the value isn't there, we need to search all *n* elements
    - If the value is there, we search *n*/2 elements on average
  - If the array is ***sorted***, we can do in O(log *n*) time
    - A binary search requires O(log *n*) time
    - About equally fast whether the element is found or not
  - It doesn't seem like we could do much better
    - How about an **O(1)**, that is, constant time search?
    - We can do it *if the array is organized* in a particular way

Direct Mapping

- Suppose we were to come up with a MAGIC FUNCTION that, given a value to search for, would tell us exactly where in the array to look
  - If it is in that location, it is in the array
  - If it is NOT in that location, it is NOT in the array
- If we look at the function's inputs and outputs, their relationships WON'T MAKE SENSE
  - This function would have no other purpose but searching
- This function is called a HASH FUNCTION because it MAKES HASH of its inputs



HASH TABLE

OBJ4	OBJ2		OBJ3	OBJ1
0	1	2	3	4

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Hash Function Definition

- A hash function is a function that:
  - When applied to an Object, returns a number
  - When applied to *equal* Objects, returns the *same* number for each
  - When applied to ***unequal*** Objects, is ***very unlikely*** to return the same number for each
- Hash function is often called hash map
- Hash functions turn out to be very important for searching, that is, looking things up fast
  - Required for associative memory

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

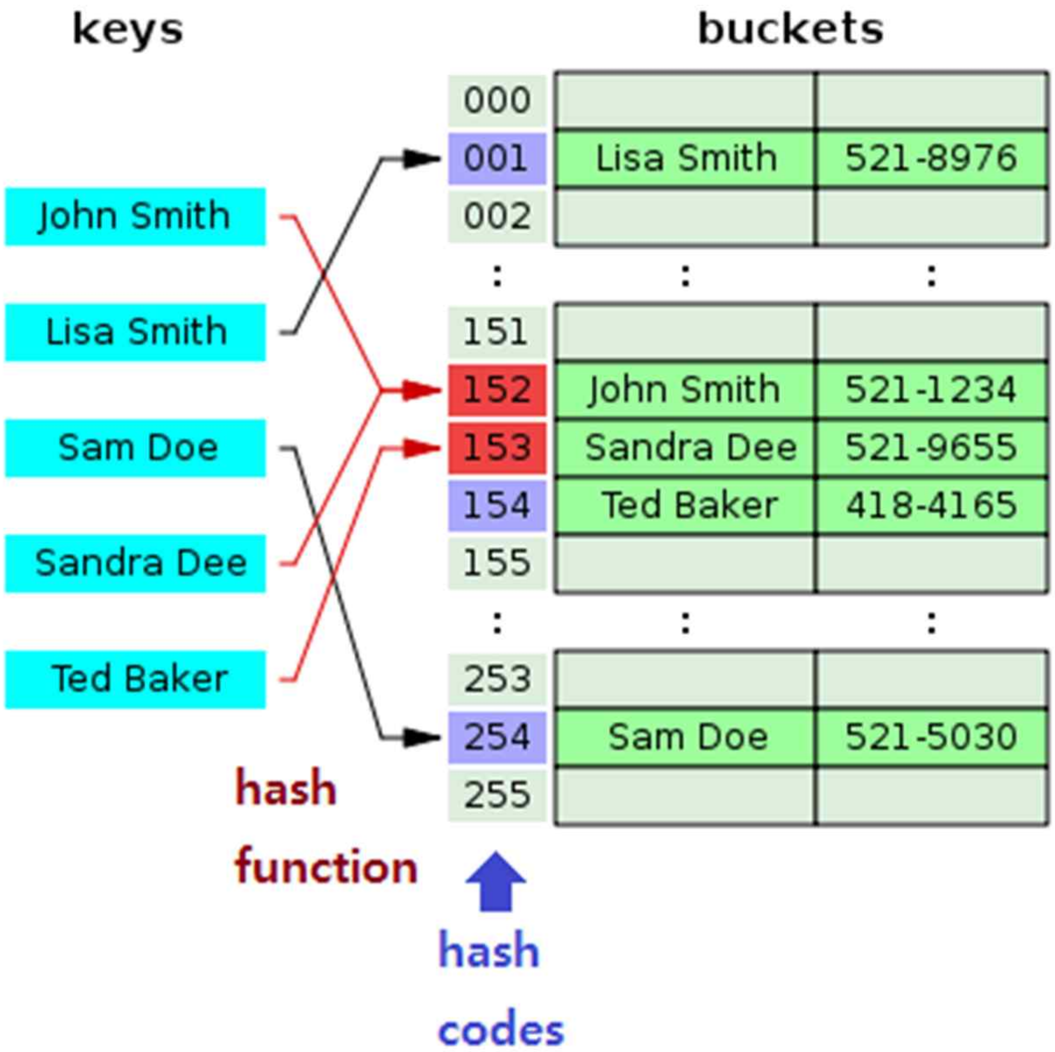
Hash Tables

- Motivation: symbol tables
  - A compiler uses a *symbol table* to relate symbols to associated data
    - Symbols: variable names, procedure names, etc.
    - Associated data: memory location, call graph, etc.
  - For a symbol table (also called a *dictionary*), we care about search, insertion, and deletion
- Hash table:
  - Given a table T and a record x, with key (= symbol) and satellite data, support:
    - Insert (T, x)　　Delete (T, x)　　Search(T, x)
  - We want these to be fast, but don't care about sorting the records
  - Supports all the above in O(1) expected time!

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Hash Table Constituents

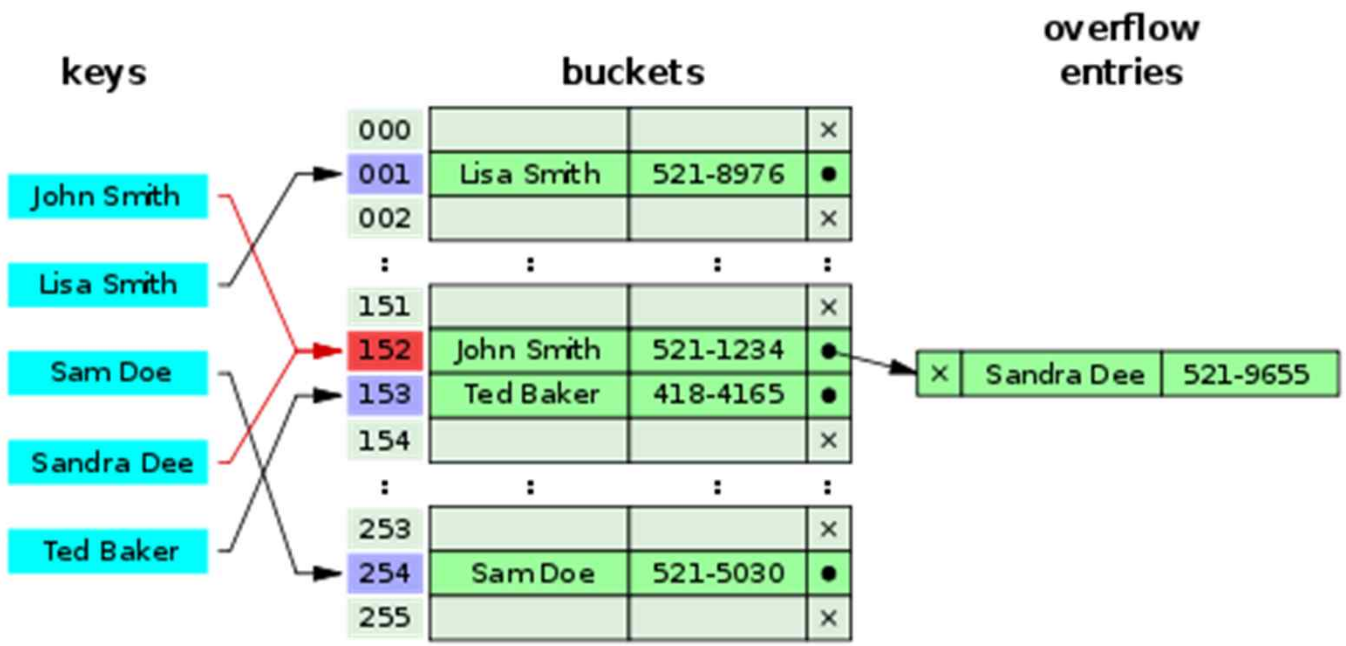
- Keys
  - Unique values to identify objects
- Hash function
  - Mapping from keys to hash codes
- Hash codes (values)
  - Output of the hash function
- Buckets
  - Actual data storage



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Issues in Hashing

- Keys assignment
  - How can we convert floats to natural numbers for hashing purposes?
  - How can we convert ASCII strings to natural numbers for hashing purposes?
  - How to ensure ***uniqueness***?
- Collision in hash codes
  - What if the output of hash functions are the same for distinct key values?
  - May results in bucket entry ***overflow***



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

HASH FUNCTION DESIGN

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Finding the hash function

- Clearly choosing the hash function well is crucial
  - *What will a worst-case hash function do?*
  - *What will be the time to search in this case?*
- *What are desirable features of the hash function?*
  - Should distribute keys uniformly into slots
    - Collision may be unavoidable, but try being as less as possible
  - Should not depend on patterns in the data
    - Fast only for specific data patterns ➔ not desirable
- *How can we come up with MAGIC hash function?*
  - In a few specific cases, where all the possible values are known in advance, it has been possible to compute a perfect hash function

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Simple Example: Direct Addressing

- Suppose:
  - The range of keys is $0..m-1$
  - Keys are all distinct
- The idea:
  - Set up an array $T[0..m-1]$ in such a way that
    - $T[i] = x$ if $x \in T$ and $key[x] = i$
    - $T[i]$ = NULL otherwise
  - This is called a *direct-address table*
    - Operations take O(1) time!
    - *So what's the problem?*

"NULL" in this context does not mean "NULL pointer" only, but a special value to indicate that the array location is empty. For example, -1 can be used in this case because the valid key value ranges from 0 to (m-1)

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

The Problem With Direct Addressing

- Direct addressing works well when the range of *m* keys is relatively small
- But what if the keys are 32-bit integers?
  - Problem 1: direct-address table will have $2^{32} = 4.295*10^9$ entries > 4 billion
  - Problem 2: even if memory is not an issue, the time to initialize the elements to NULL may be time-consuming
- Solution: map keys to smaller range

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Example (ideal) Hash Function

- Suppose our hash function gave us the following values:

```
hashCode("apple") = 5
hashCode("watermelon") = 3
hashCode("grapes") = 8
hashCode("cantaloupe") = 7
hashCode("kiwi") = 0
hashCode("strawberry") = 9
hashCode("mango") = 6
hashCode("banana") = 2
```

- Note that there is no same hash values so no collision in the buckets

0	*kiwi*
1	
2	*banana*
3	*watermelon*
4	
5	*apple*
6	*mango*
7	*cantaloupe*
8	*grapes*
9	*strawberry*

Sets and Tables

- Hashing for set:
  - Sometimes we just want a set of things—objects are either in it, or they are not in it
- Hashing for map:
  - Sometimes we want a map—a way of looking up one thing based on the value of another
  - We use a *key* to find a place in the map
  - The associated *value* is the information we are trying to look up
- Hashing works the same for both sets and maps
  - Most of our examples will be sets

	key	value
. . .		
141		
142	robin	robin info
143	sparrow	sparrow info
144	hawk	hawk info
145	seagull	seagull info
146		
147	bluejay	bluejay info
148	owl	owl info

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
- Three methods:
  - Division method
  - Multiplication method
  - Universal hashing

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Hash Functions: 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
- *What happens to elements with adjacent values of k?*
  - Elements with adjacent keys hashed to different slots: good
- *What happens if m is a power of 2 (say 2^P)?*
  - If keys bear relation to m: bad
- *What if m is a power of 10?*
- Upshot: pick table size m = prime number not too close to a power of 2 (or 10)

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

The Multiplication Method

- For a constant A, $0 < A < 1$:
- $h(k) = \lfloor m\ \underbrace{(kA - \lfloor kA \rfloor)}_{\textit{Fractional part of } kA} \rfloor$

- Choose $m = 2^P$
- Choose A not too close to 0 or 1
- Knuth: Good choice for $A = (\sqrt{5} - 1)/2$

Hash Functions: Universal Hashing

- *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

- 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 $|\varsigma|/m$
  - With a random hash function from ς, the chance of a collision between x and y is exactly $1/m$ $(x \neq y)$

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Universal Hashing: Theorem 12.3

- Choose *h* from a universal family of hash functions
- Hash *n* keys into a table of *m* slots, $n \leq m$
- 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 $n \leq m$, we have $E[C_x] < 1$

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

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

Imperfect Hash Function

- Suppose our hash function gave us the following values:

```
hash("apple") = 5
hash("watermelon") = 3
hash("grapes") = 8
hash("cantaloupe") = 7
hash("kiwi") = 0
hash("strawberry") = 9
hash("mango") = 6
hash("banana") = 2
```

- Now the new item's bucket is already filled:

```
hash("honeydew") = 6
```

- *Now what?*

0	*kiwi*
1	
2	*banana*
3	*watermelon*
4	
5	*apple*
6	*mango*
7	*cantaloupe*
8	*grapes*
9	*strawberry*

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Collisions

- A perfect hash function would tell us exactly *where to look*.
- Imperfect hash function gives
  - Same hash values for different key values – called a ***collision***.
  - Collisions are normally treated as "first come, first served"—the first value that hashes to the location gets it
- What is the next best thing?
  - The best we can do is a function that tells us *where to start looking*!
  - Find something to do with the second and subsequent values that hash to this same location

RESOLVING COLLISIONS

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Handling collisions

- What can we do when two different values attempt to occupy the same place in an array?
  - **Solution #1 (open addressing):** Search from there for an empty location
    - Can stop searching when we find the value *or* an empty location
    - Search must be end-around
  - **Solution #2 (rehashing):** Use a 2^{nd} hash function
    - ...and a third, and a fourth, and a fifth, ...
  - **Solution #3 (chaining):** Use the array location as the header of a linked list of values that hash to this location
- All these solutions work, provided:
  - We use the same technique to *add* things to the array as we use to *search* for things in the array

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Open Addressing

- Basic idea (details in Section 12.4):
  - To insert: if slot is full, try another slot, ..., until an open slot is found (probing)
  - To search, follow same sequence of probes as would be used when inserting the element
    - If reach element with correct key, return it
    - If reach a NULL pointer, element is not in table
- Good for fixed sets (adding but no deletion)
  - Example: spell checking
- Table needn't be much bigger than *n*

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Open Addressing: Insertion, I

- Suppose you want to add seagull to this hash table
- Also suppose:
  - hashCode(seagull) = 143
  - table[143] is not empty
  - table[143] != seagull
  - table[144] is not empty
  - table[144] != seagull
  - table[145] is empty
- Therefore, put seagull at location 145

Open Addressing: Searching, I

- Suppose you want to look up seagull in this hash table
- Also suppose:
  - hashCode(seagull) = 143
  - table[143] is not empty
  - table[143] != seagull
  - table[144] is not empty
  - table[144] != seagull
  - table[145] is not empty
  - table[145] == seagull !
- We found seagull at location 145

. . .	
141	
142	*robin*
143	*sparrow*
144	*hawk*
145	*seagull*
146	
147	*bluejay*
148	*owl*
. . .	

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Open Addressing: Searching, II

- Suppose you want to look up cow in this hash table
- Also suppose:
  - hashCode(cow) = 144
  - table[144] is not empty
  - table[144] != cow
  - table[145] is not empty
  - table[145] != cow
  - table[146] is empty
- If cow were in the table, we should have found it by now
- Therefore, it isn't here

...	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
...	

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Open Addressing: Insertion, II

- Suppose you want to add hawk to this hash table
- Also suppose
  - hashCode(hawk) = 143
  - table[143] is not empty
  - table[143] != hawk
  - table[144] is not empty
  - table[144] == hawk
- hawk is already in the table, so do nothing

Open Addressing: Insertion, III

- Suppose:
  - You want to add cardinal to this hash table
  - hashCode(cardinal) = 147
  - The last location is 148
  - 147 and 148 are occupied
- Solution:
  - Treat the table as circular; after 148 comes 0
  - Hence, cardinal goes in location 0 (or 1, or 2, or ...)

. . .	
141	
142	*robin*
143	*sparrow*
144	*hawk*
145	*seagull*
146	
147	*bluejay*
148	*owl*

Efficiency

- Hash tables are actually surprisingly efficient
- Until the table is about 70% full, the number of probes (places looked at in the table) is typically only 2 or 3
- Sophisticated mathematical analysis is required to prove that the expected cost of inserting into a hash table, or looking something up in the hash table, is O(1)
- Even if the table is nearly full (leading to occasional long searches), efficiency is usually still quite high

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

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

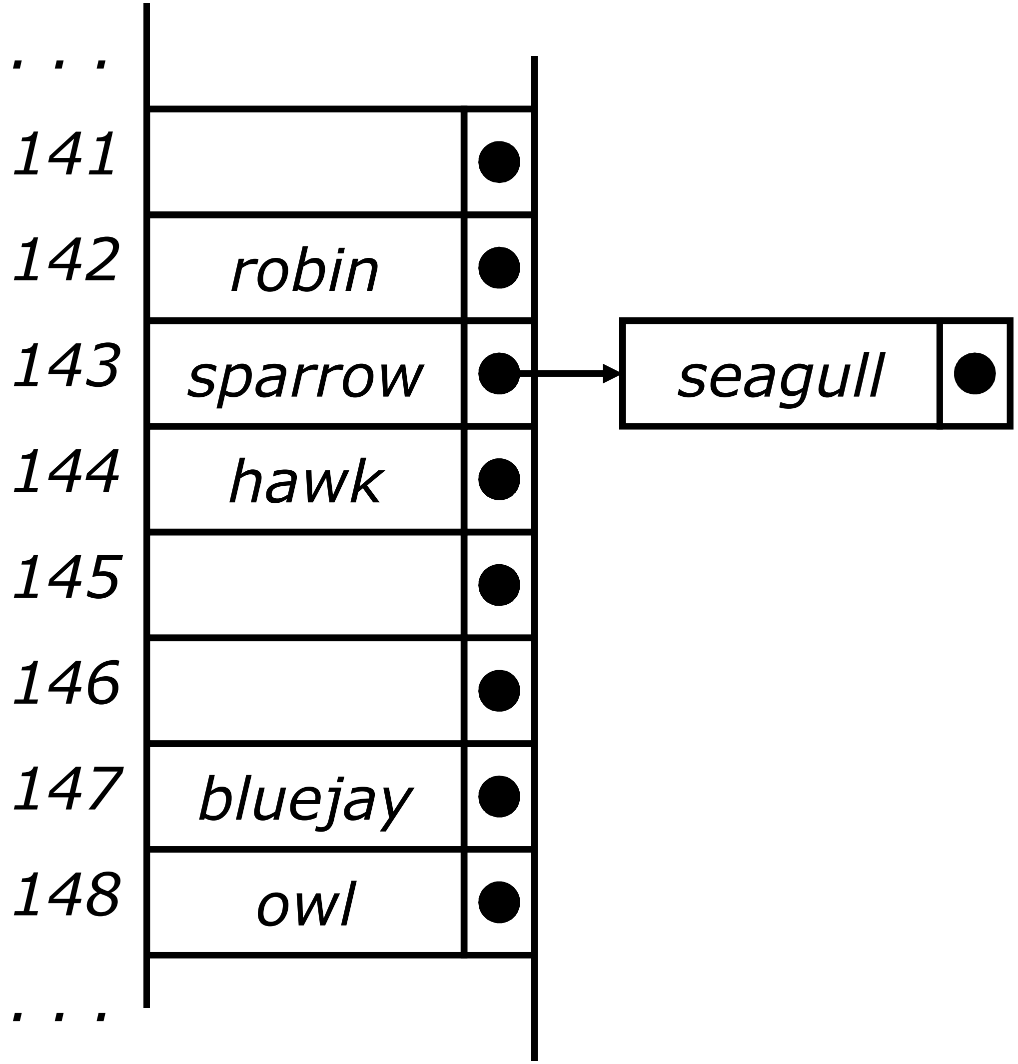
Solution #2: Rehashing

- In the event of a collision, another approach is to rehash: compute another hash function
  - Since we may need to rehash many times, we need an easily computable sequence of functions
- Simple example: in the case of hashing Strings, we might take the previous hash code and add the length of the String to it
  - Probably better if the length of the string was not a component in computing the original hash function
- Possibly better yet: add the length of the String plus the number of probes made so far
  - Problem: are we sure we will look at every location in the array?
- Rehashing is a fairly uncommon approach, and we won't pursue it any further here

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Solution #3: Chaining / Bucket hashing

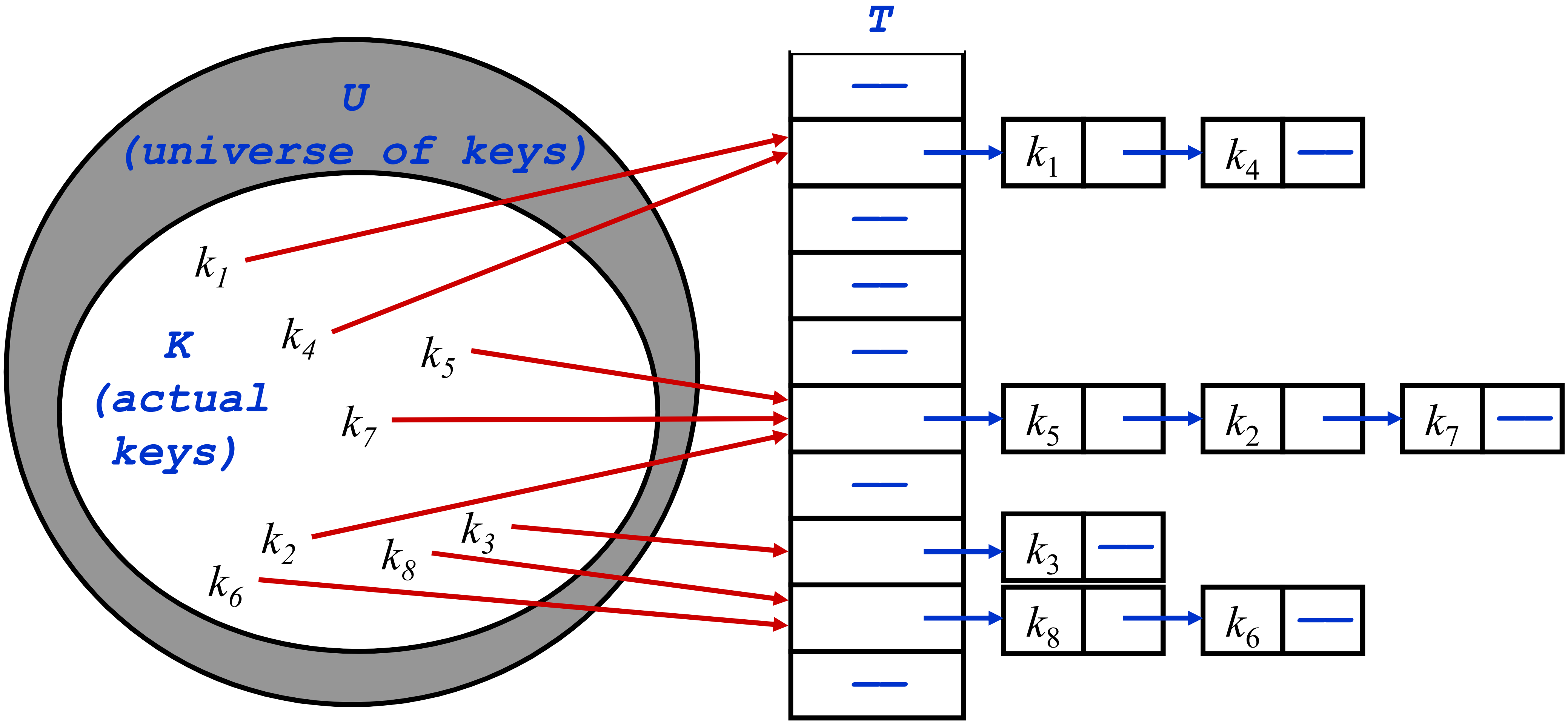
- The previous solutions used open hashing: all entries went into a “flat” (unstructured) array
- Another solution is to make each array location the header of a linked list of values that hash to that location



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Chaining

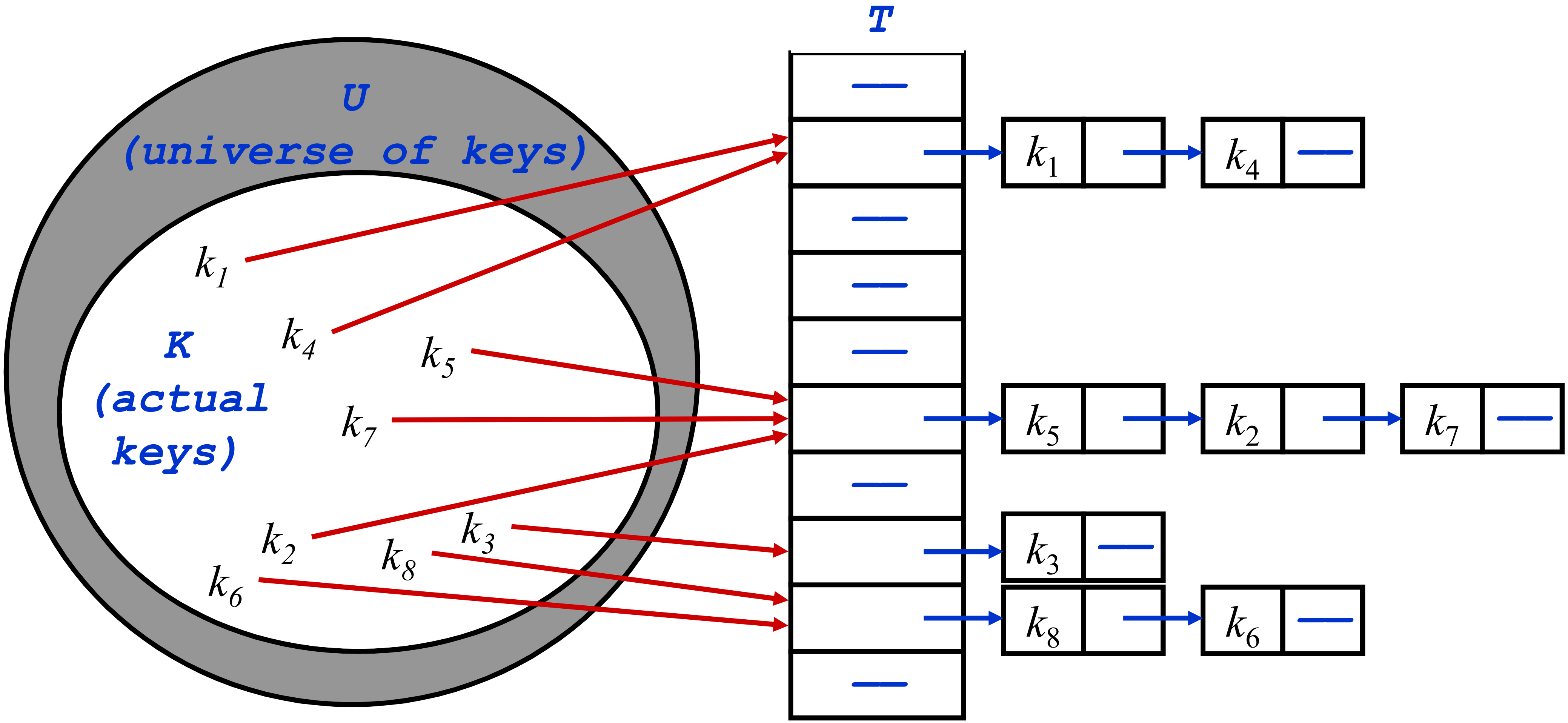
- Chaining puts elements that hash to the same slot in a linked list:



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Chaining

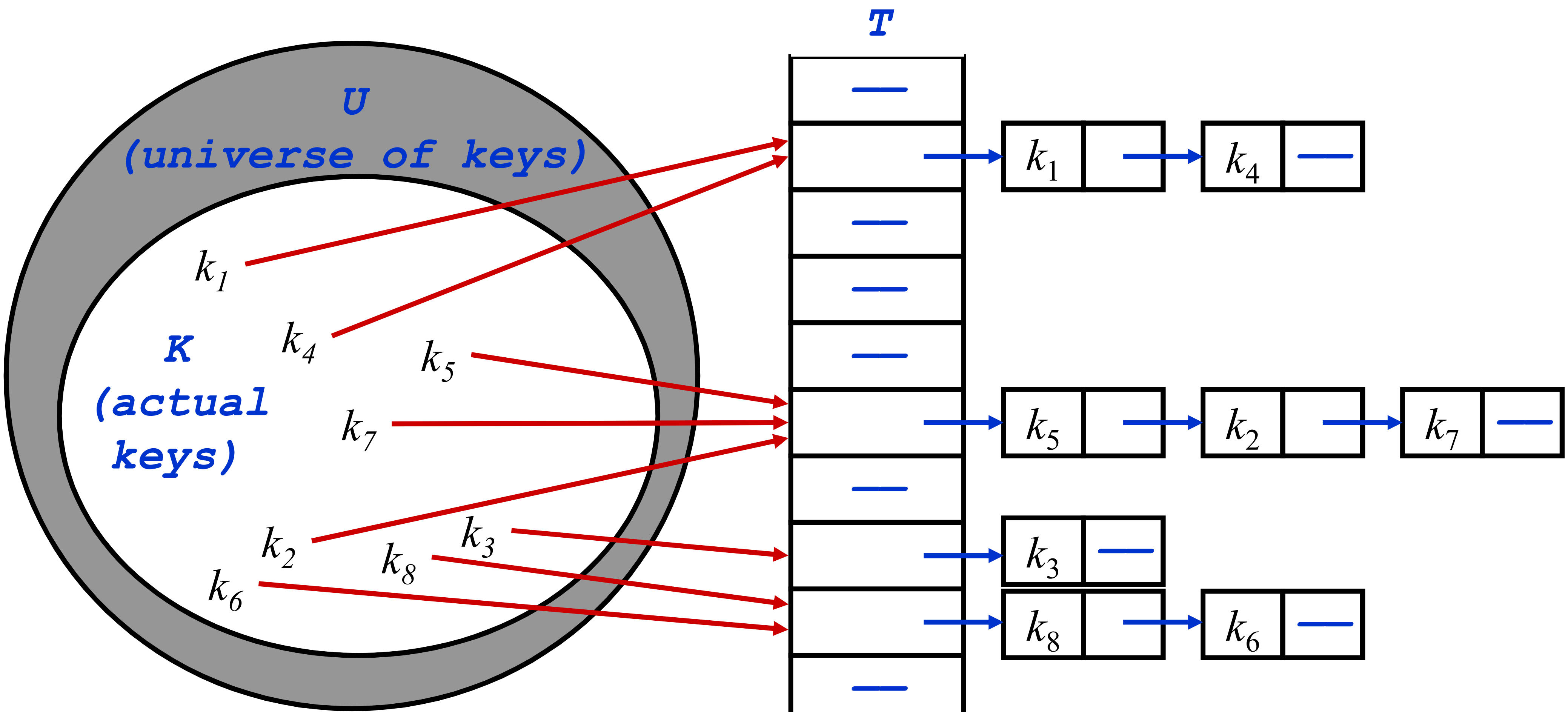
- *How do we insert an element?*



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Chaining

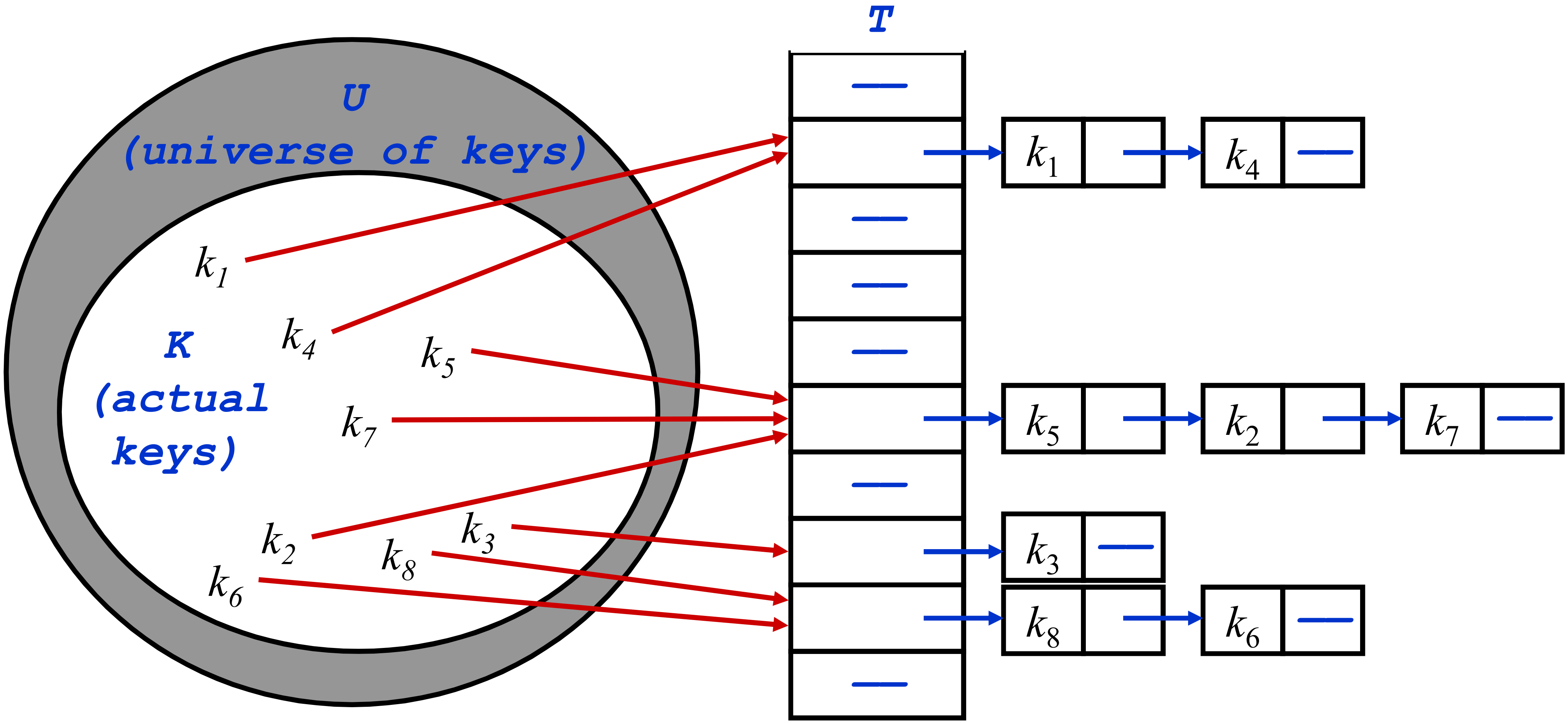
- *How do we delete an element?*
  - *Do we need a doubly-linked list for efficient delete?*



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Chaining

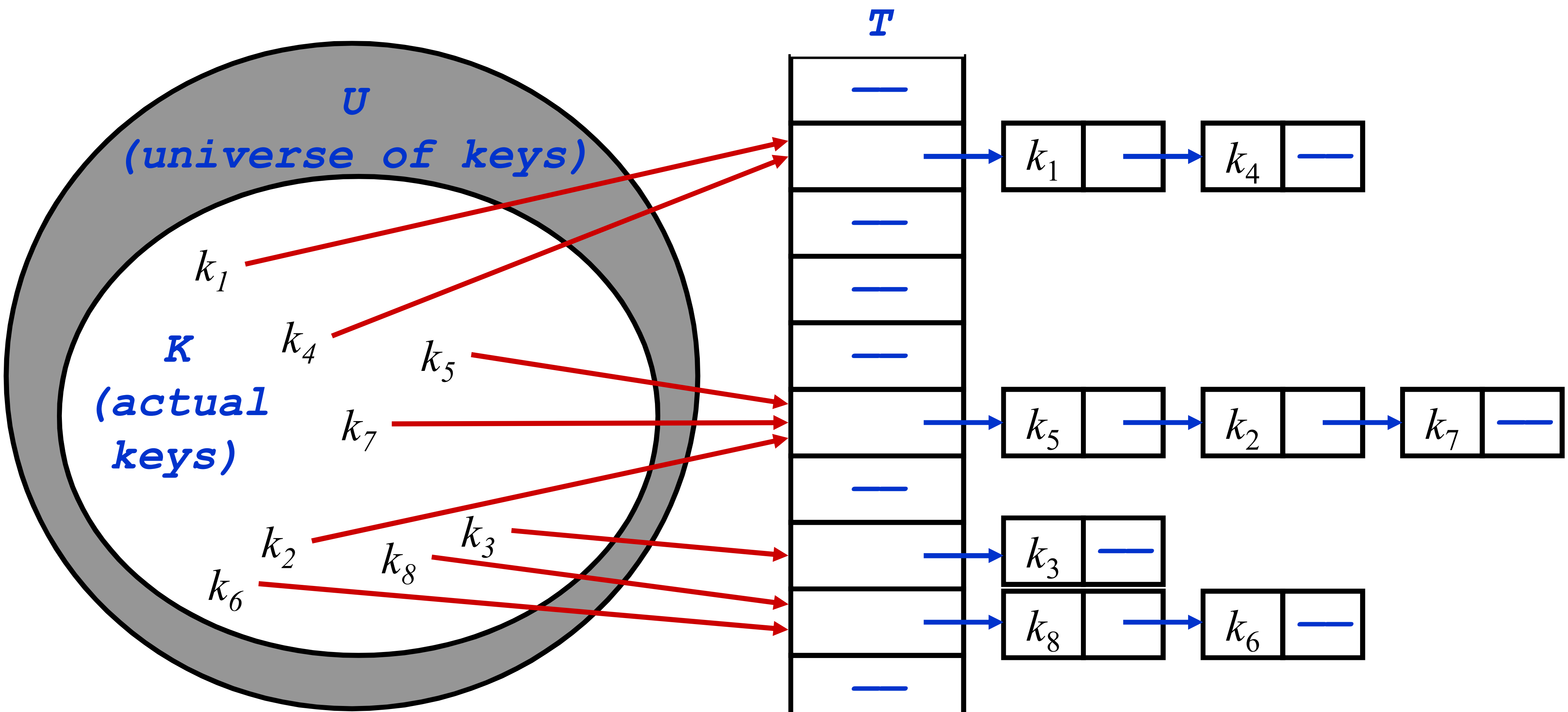
- *How do we search for a element with a given key?*



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Review: Chaining

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



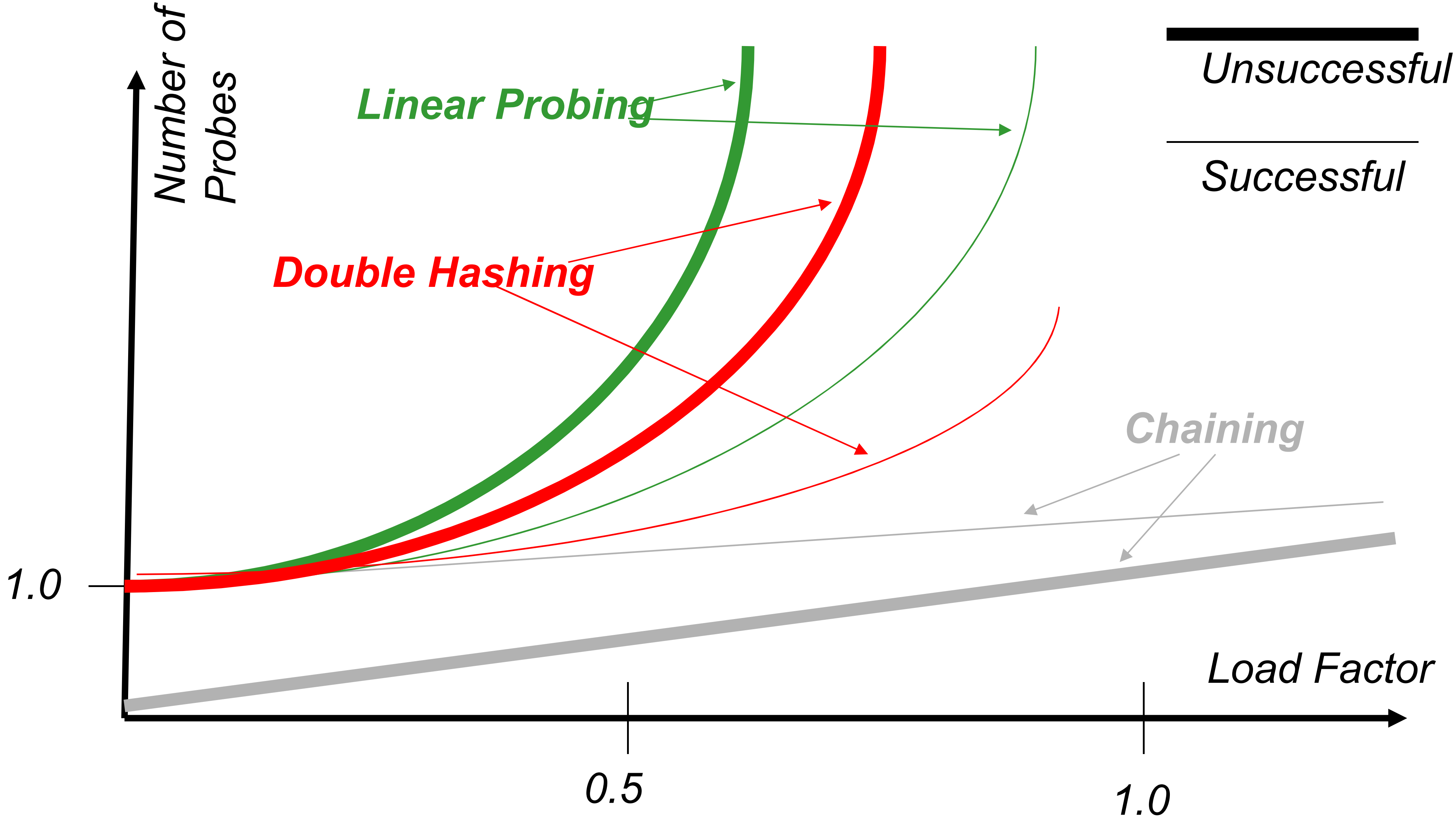
Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given n keys and m slots, the *load factor* $\alpha = n/m$ = average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?*
  - A: O(1+α)
- *What will be the average cost of a successful search?*
  - A: O(1 + α/2) = O(1 + α)
- *If the number of keys n is proportional to the number of slots in the table, what is α?*
  - A: α = O(1)
  - We can make the expected cost of searching constant if we make α constant

Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

Expected Number of Probes vs. Load Factor



Slide credit: J. Lillis, UIC's CS 201 Data Structures and Discrete Mathematics I

END OF LECTURE 10