CSE - 5311 Advanced Algorithms

Instructor : Nomaan Mufti

Submitted by

Raja Rajeshwari Anugula & Srujana Tiruveedhi

Contents

- Hash Tables
- Union Find

Hash Tables

- Hash table is an effective data structure for implementing dictionaries.
- Although searching for an element in hash table in the worst case is $\Theta(n)$ time, under reasonable assumptions the expected time to search for an element is $O(1)$.
- With hashing this element is stored in slot $h(k)$ i.e we use a hash function *h* to compute the slot from the key k.
- Two keys may hash to the same slot. This is called collision.

A Generalization

A hash table is a generalization of an ordinary array.

- With an ordinary array, we store the element whose key is k in position k of the array.
- Given a key k, we find the element whose key is k by just looking in the kth position of the array. This is called ***direct addressing***.
- Direct addressing is applicable when we can afford to allocate an array with one position for every possible key.

We use a hash table when we do not want to (or cannot) allocate an array with one position per possible key.

- Use a hash table when the number of keys actually stored is small relative to the number of possible keys.
- A hash table is an array, but it typically uses a size proportional to the number of keys to be stored (rather than the number of possible keys).
- Given a key k, don't just use k as the index into the array.
- Instead, compute a function of k, and use that value to index into the array. We call this function a ***hash function***.

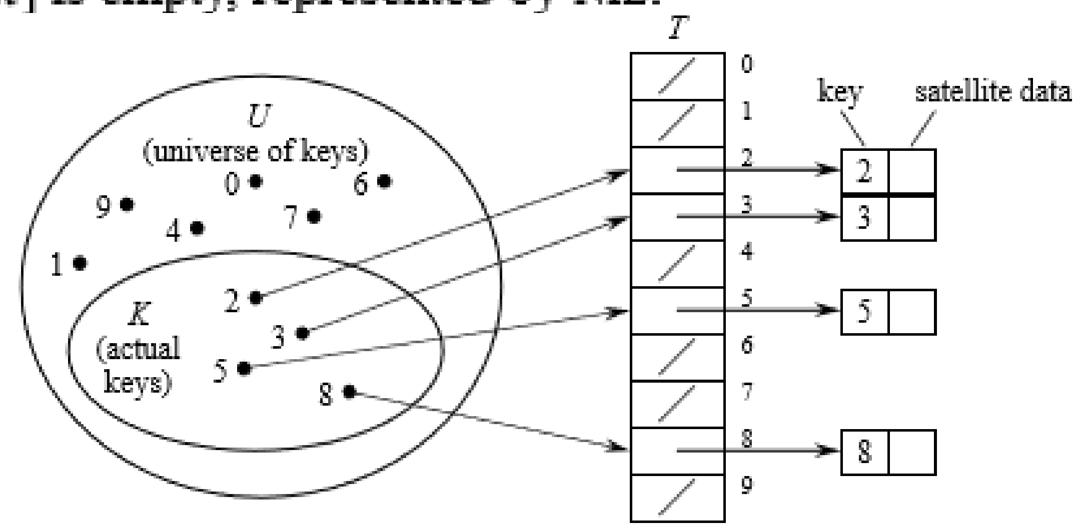
Direct Access Tables

Scenario

- Maintain a dynamic set.
- Each element has a key drawn from a universe $U = \{0, 1, \ldots, m-1\}$ where m isn't too large.
- No two elements have the same key.

Represent by a ***direct-address table***, or array, $T[0 \ldots m-1]$:

- Each ***slot***, or position, corresponds to a key in U.
- If there's an element x with key k, then $T[k]$ contains a pointer to x.
- Otherwise, $T[k]$ is empty, represented by NIL.



Direct Access Tables

Dictionary operations are trivial and take $O(1)$ time each:

DIRECT-ADDRESS-SEARCH(T, k)

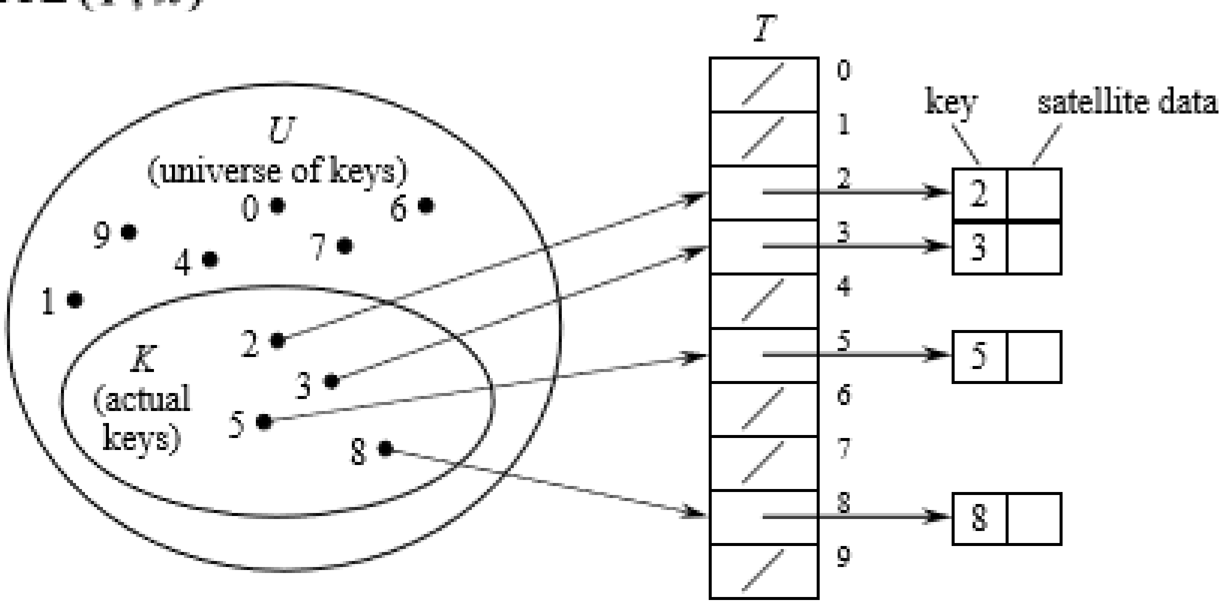
return $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

$T[key[x]] = x$

DIRECT-ADDRESS-DELETE(T, x)

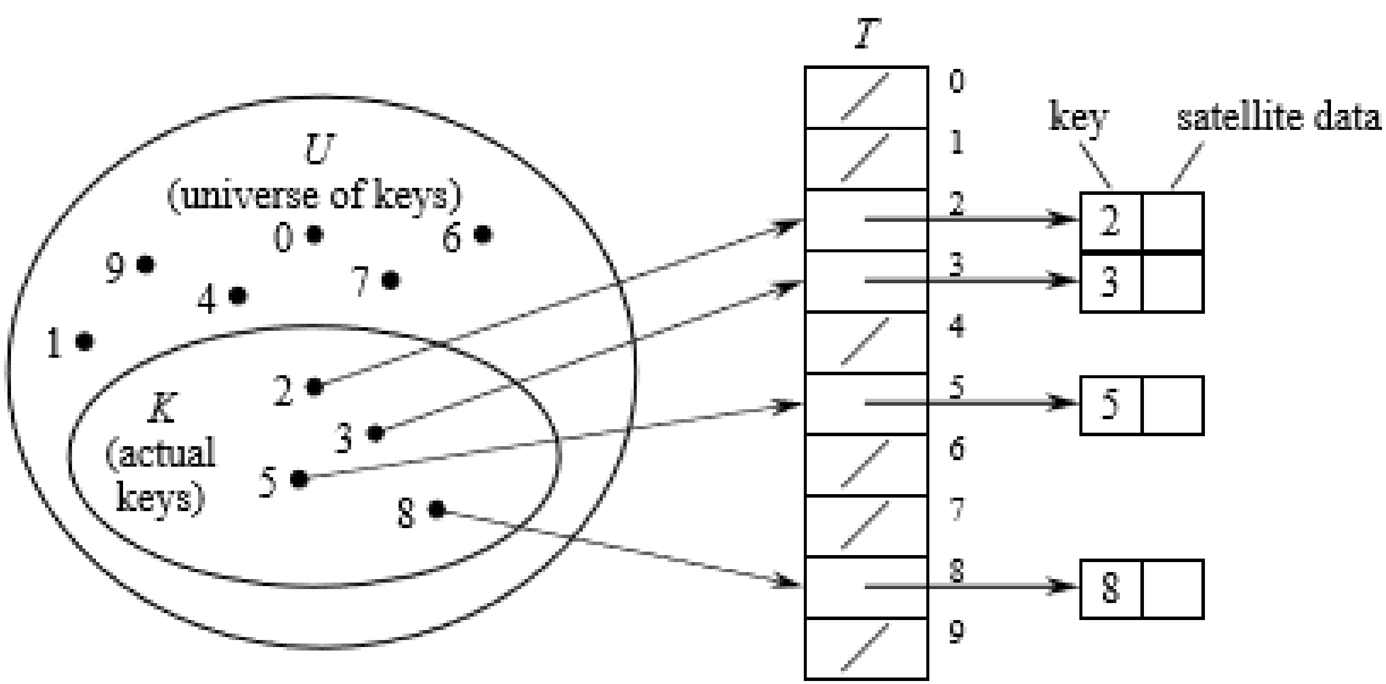
$T[key[x]] = \text{NIL}$



Direct Access Tables

The problem with direct addressing is if the universe U is large, storing a table of size $|U|$ may be impractical or impossible.

Often, the set K of keys actually stored is small, compared to U, so that most of the space allocated for T is wasted.



Hash Tables

- When K is much smaller than U, a hash table requires much less space than a direct-address table.
- Can reduce storage requirements to $\Theta(|K|)$.
- Can still get $O(1)$ search time, but in the *average case*, not the *worst case*.

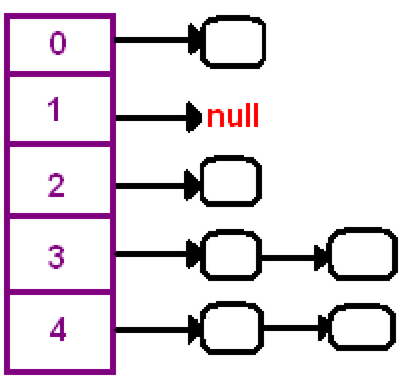
Instead of storing an element with key k in slot k, use a function h and store the element in slot $h(k)$.

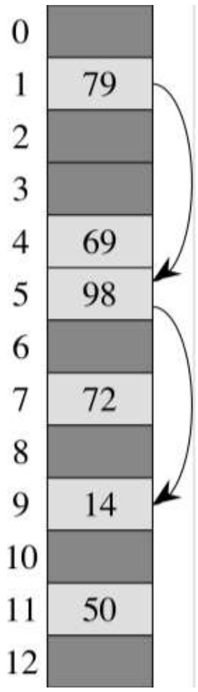
- We call h a ***hash function***.
- $h : U \rightarrow \{0, 1, \ldots, m-1\}$, so that $h(k)$ is a legal slot number in T.
- We say that k ***hashes*** to slot $h(k)$.

Hash Tables, Collisions

When two or more keys hash to the same slot.

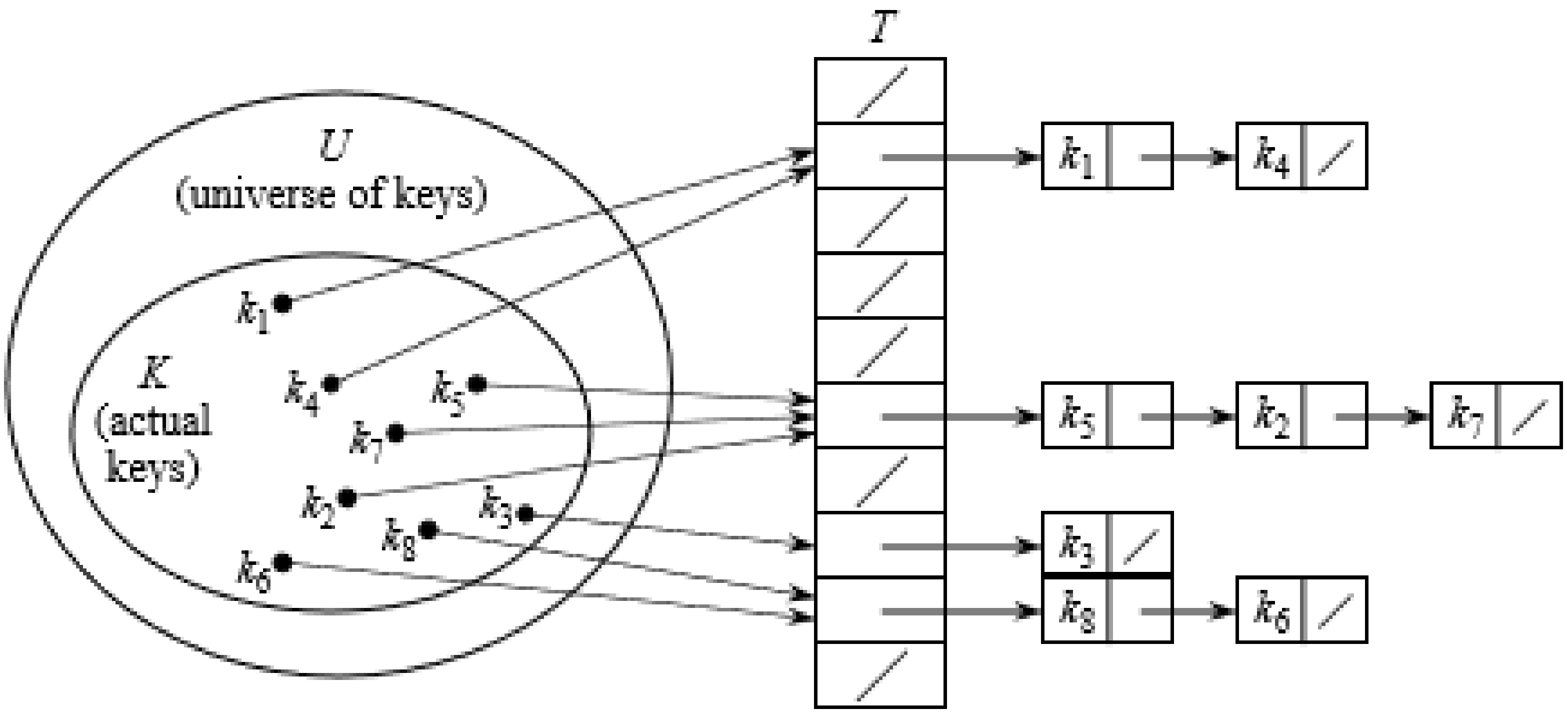
- Can happen when there are more possible keys than slots ($|U| > m$).
- For a given set K of keys with $|K| \leq m$, may or may not happen. Definitely happens if $|K| > m$.
- Therefore, must be prepared to handle collisions in all cases.
- Use two methods: chaining and open addressing.
- Chaining is usually better than open addressing.





Collision Resolution by Chaining

Put all elements that hash to the same slot into a linked list.

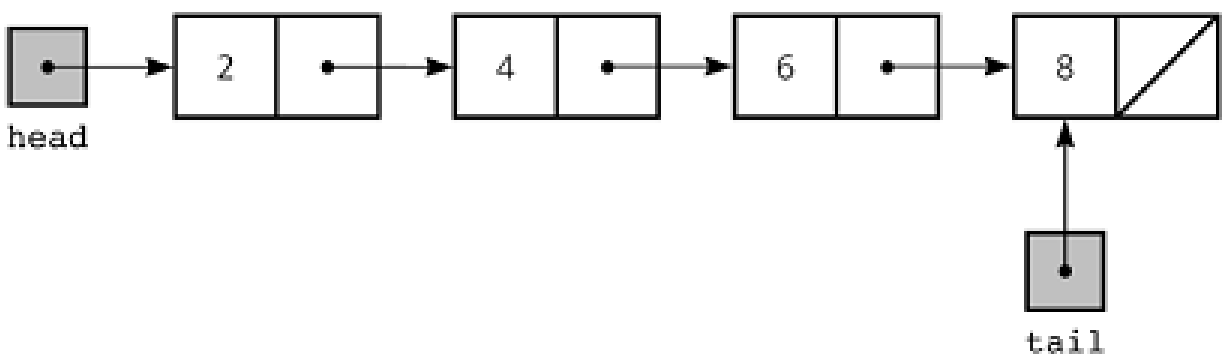


- Slot j contains a pointer to the head of the list of all stored elements that hash to j *[or to the sentinel if using a circular, doubly linked list with a sentinel]* .
- If there are no such elements, slot j contains NIL.

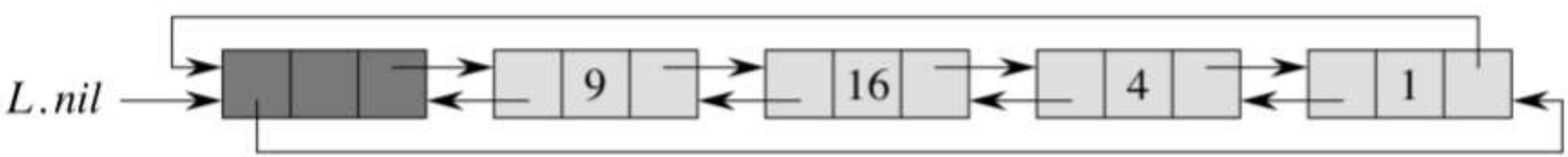
Linked Lists (Review)

A linked list is a linear data structure where each element is a separate object. Linked list elements are not stored at contiguous location; the elements are linked using pointers.

Each node of a list is made up of two items - the data and a reference to the next node. The last node has a reference to null. The entry point into a linked list is called the head of the list. It should be noted that head is not a separate node, but the reference to the first node. If the list is empty then the head is a null reference



Doubly Linked List.



Dictionary Operations on Hash Tables

- ***Insertion:***

  CHAINED-HASH-INSERT(T, x)

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

  - Worst-case running time is $O(1)$.
  - Assumes that the element being inserted isn't already in the list.
  - It would take an additional search to check if it was already inserted.

- ***Search:***

  CHAINED-HASH-SEARCH(T, k)

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

  Running time is proportional to the length of the list of elements in slot $h(k)$.

Dictionary Operations on Hash Tables

- ***Deletion:***

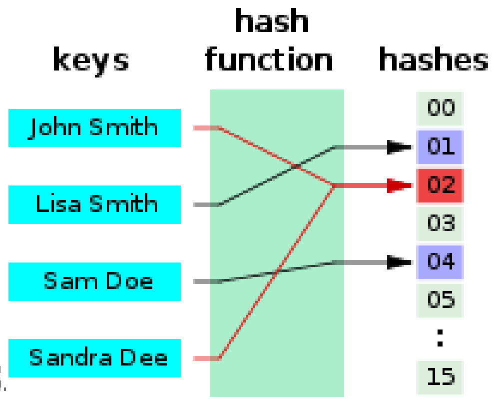
  CHAINED-HASH-DELETE(T, x)

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

  - Given pointer x to the element to delete, so no search is needed to find this element.
  - Worst-case running time is $O(1)$ time if the lists are doubly linked.
  - If the lists are singly linked, then deletion takes as long as searching, because we must find x's predecessor in its list in order to correctly update *next* pointers.

Hash Functions

- Ideally, the hash function satisfies the assumption of simple uniform hashing.
- In practice, it's not possible to satisfy this assumption, since we don't know in advance the probability distribution that keys are drawn from, and the keys may not be drawn independently.
- Often use heuristics, based on the domain of the keys, to create a hash function that performs well.



Keys as natural numbers

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

Hash Functions: Division

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

Example: $m = 20$ and $k = 91 \Rightarrow h(k) = 11$.

Advantage: Fast, since requires just one division operation.

Disadvantage: Have to avoid certain values of m:

- Powers of 2 are bad. If $m = 2^p$ for integer p, then $h(k)$ is just the least significant p bits of k.
- If k is a character string interpreted in radix 2^p (as in `CLRS` example), then $m = 2^p - 1$ is bad: permuting characters in a string does not change its hash value (Exercise 11.3-3).

Good choice for m: A prime not too close to an exact power of 2.

Hash Functions: Multiplication

1. Choose constant A in the range $0 < A < 1$.
2. Multiply key k by A.
3. Extract the fractional part of kA.
4. Multiply the fractional part by m.
5. Take the floor of the result.

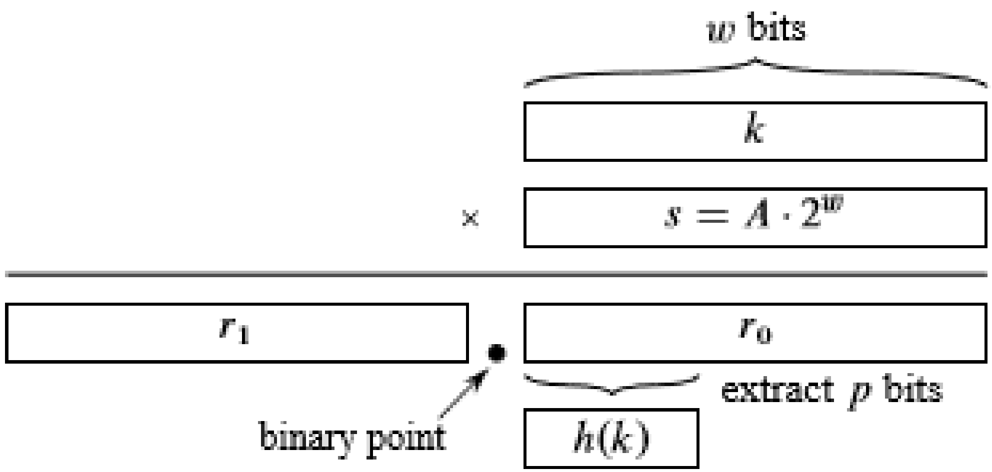
Put another way, $h(k) = \lfloor m\,(k\,A \bmod 1) \rfloor$, where $k\,A \bmod 1 = kA - \lfloor kA \rfloor =$ fractional part of kA.

Disadvantage: Slower than division method.

Advantage: Value of m is not critical.

Implementation of Multiplication Method

- Choose $m = 2^p$ for some integer p.
- Let the word size of the machine be w bits.
- Assume that k fits into a single word. (k takes w bits.)
- Let s be an integer in the range $0 < s < 2^w$. (s takes w bits.)
- Restrict A to be of the form $s/2^w$.



- Multiply k by s.
- Since we're multiplying two w-bit words, the result is $2w$ bits, $r_1 2^w + r_0$, where r_1 is the high-order word of the product and r_0 is the low-order word.
- r_1 holds the integer part of kA ($\lfloor kA \rfloor$) and r_0 holds the fractional part of kA ($kA \bmod 1 = kA - \lfloor kA \rfloor$). Think of the "binary point" (analog of decimal point, but for binary representation) as being between r_1 and r_0. Since we don't care about the integer part of kA, we can forget about r_1 and just use r_0.

Implementation of Multiplication Method

Example: $m = 8$ (implies $p = 3$), $w = 5$, $k = 21$. Must have $0 < s < 2^5$; choose $s = 13 \Rightarrow A = 13/32$.

- Using just the formula to compute $h(k)$: $kA = 21 \cdot 13/32 = 273/32 = 8\frac{17}{32}$ $\Rightarrow k\,A \bmod 1 = 17/32 \Rightarrow m\,(k\,A \bmod 1) = 8 \cdot 17/32 = 17/4 = 4\frac{1}{4} \Rightarrow$ $\lfloor m\,(k\,A \bmod 1)\rfloor = 4$, so that $h(k) = 4$.
- Using the implementation: $ks = 21 \cdot 13 = 273 = 8 \cdot 2^5 + 17 \Rightarrow r_1 = 8$, $r_0 = 17$. Written in $w = 5$ bits, $r_0 = 10001$. Take the $p = 3$ most significant bits of r_0, get 100 in binary, or 4 in decimal, so that $h(k) = 4$.

How to choose A:

- The multiplication method works with any legal value of A.
- But it works better with some values than with others, depending on the keys being hashed.
- Knuth suggests using $A \approx (\sqrt{5} - 1)/2$.

Donald Ervin Knuth is a famous computer scientist, mathematician, and professor emeritus at Stanford University. He is the author of the multi-volume work The Art of Computer Programming

Data Structures for Disjoint Sets

- ✓ Also known as “union find”
- ✓ Applications involve grouping elements into a collection of disjoint sets
- ✓ Maintains a collection of disjoint dynamic sets
  $\mathcal{S} = \{S_1, \ldots, S_k\}$
- ✓ Each set is identified by a representative, which is a member of the set

Union Find: Operations

- MAKE-SET(x): make a new set $S_i = \{x\}$, and add S_i to $\mathcal{S}$.
- UNION(x, y): if $x \in S_x$, $y \in S_y$, then $\mathcal{S} = \mathcal{S} - S_x - S_y \cup \{S_x \cup S_y\}$.
  - Representative of new set is any member of $S_x \cup S_y$, often the representative of one of S_x and S_y.
  - Destroys S_x and S_y (since sets must be disjoint).
- FIND-SET(x): return representative of set containing x.

MAKE-SET OPERATION

- Makes a singleton set; or, creates a new set with a single member (i.e., its representative)
- Every set should have a representative which should be any element of the set

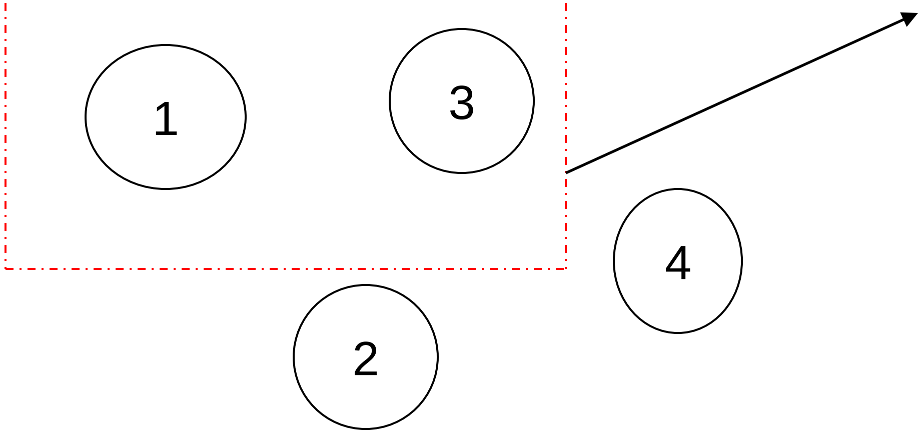
Make-Set(1)

Make-Set(2)

*
*
*
*
*

Make-Set(n)

UNION OPERATION



UNION: Merges two sets and create a new set

Initially each number is a set by itself.

From n singleton sets gradually merge to form a set.

After n-1 union operations we get a single set of n numbers.

FIND OPERATION

- Every set has a name/representative
- Thus Find(x) returns representative of the set
- The time taken for a find operation is $O(n)$ whereas for Union operation it is $O(1)$.

Analysis

- n = # of elements = # of MAKE-SET operations,
- m = total # of operations.
- Since MAKE-SET counts toward total # of operations, $m \geq n$.
- Can have at most $n - 1$ UNION operations, since after $n - 1$ UNIONs, only 1 set remains.
- Assume that the first n operations are MAKE-SET (helpful for analysis, usually not really necessary).

Application: Dynamic Connected Components

For a graph $G = (V, E)$, vertices u, v are in same connected component if and only if there's a path between them.

- Connected components partition vertices into equivalence classes.

CONNECTED-COMPONENTS(G)

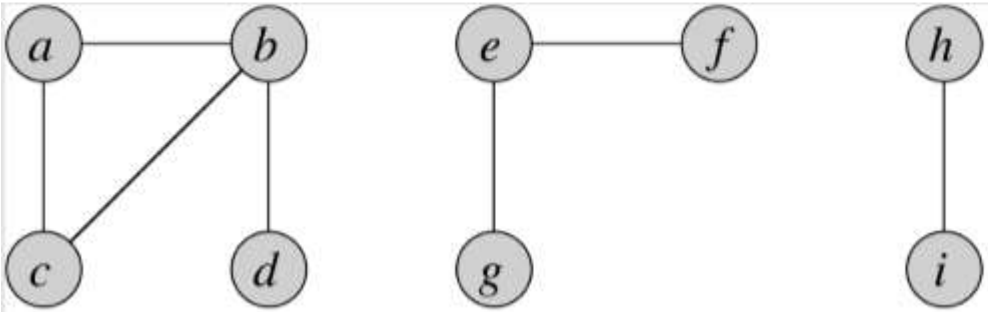
```
for each vertex v ∈ G.V
    MAKE-SET(v)
for each edge (u, v) ∈ G.E
    if FIND-SET(u) ≠ FIND-SET(v)
        UNION(u, v)
```

The procedure CONNECTED-COMPONENTS initially places each vertex v in its own set
For each edge (u, v), it unites the sets containing u and v

SAME-COMPONENT(u, v)

```
if FIND-SET(u) == FIND-SET(v)
    return TRUE
else return FALSE
```

Dynamic Connected Components: Example



(a)

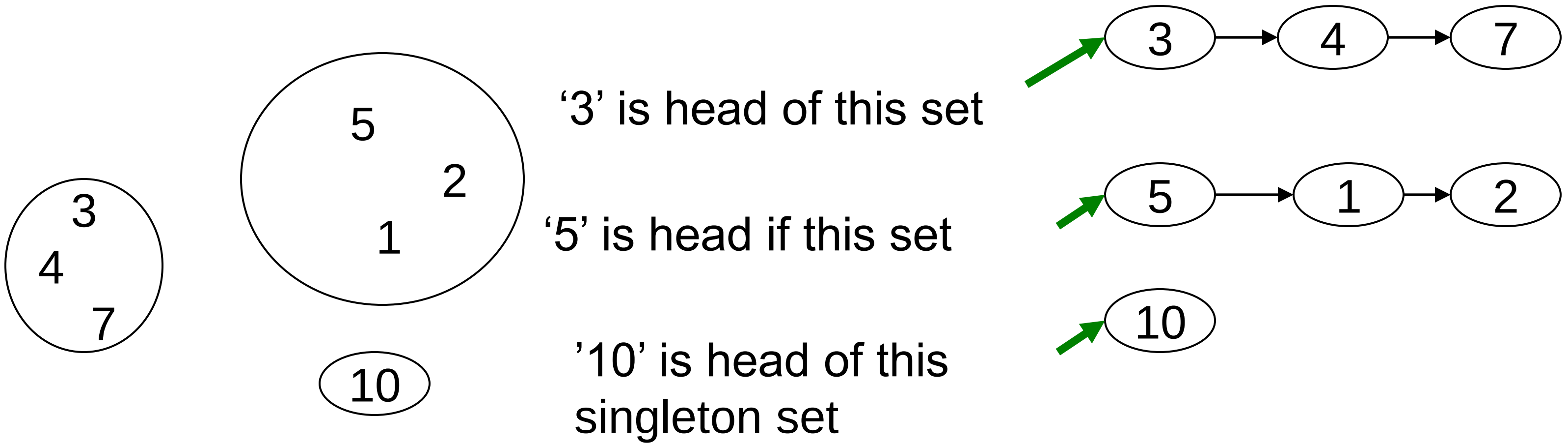
The procedure CONNECTED-COMPONENTS initially places each vertex v in its own set

For each edge (u, v), it unites the sets containing u and v

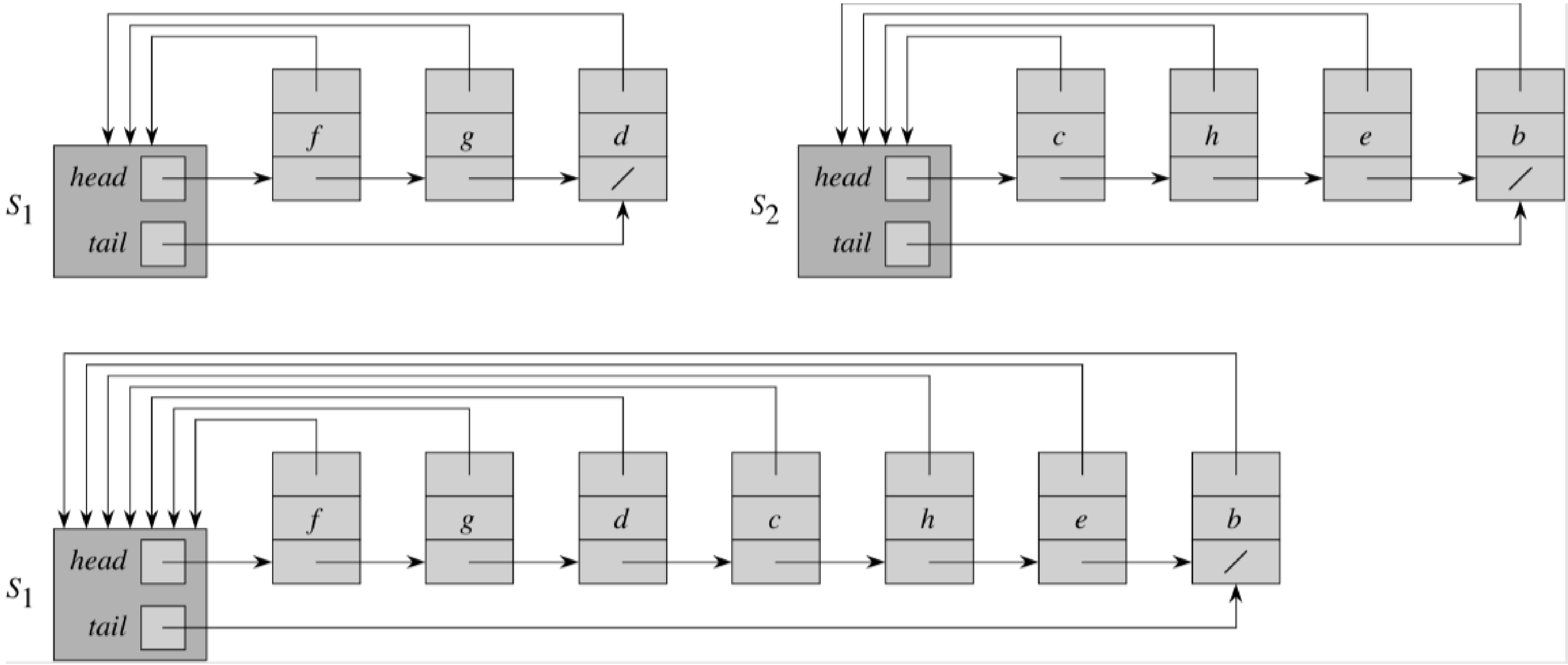
Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$		$\{h,i\}$		$\{j\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

Linked List Representation

- Every set is a singly linked list where the first element (head) is the representative of the set
- Tail is the last element in the list
- Objects may be placed in any order
- Each object in the list has attributes for:
  - The set member
  - Pointer to the set member
  - next



Linked List Representation



With this linked list representation, both MAKE-SET(x) and FIND-SET(x) are easy requiring $O(1)$ time

Above, FIND-SET(g) would return f.

MAKE-SET(x) will create a new linked list whose only object is x

The result of UNION(g, e), is shown in the lower figure. UNION appends the linked list containing e to the linked list containing g. f is the new representative. The set of object for e's list, S_2 is destroyed

UNION with Linked Lists

1. UNION(x, y): append y's list onto end of x's list. Use x's tail pointer to find the end.

   - Need to update the pointer back to the set object for every node on y's list.
   - If appending a large list onto a small list, it can take a while.

Operation	# objects updated
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
UNION(x_5, x_4)	4
$\vdots$	$\vdots$
UNION(x_n, x_{n-1})	$\underline{n-1}$
	$\Theta(n^2)$ total

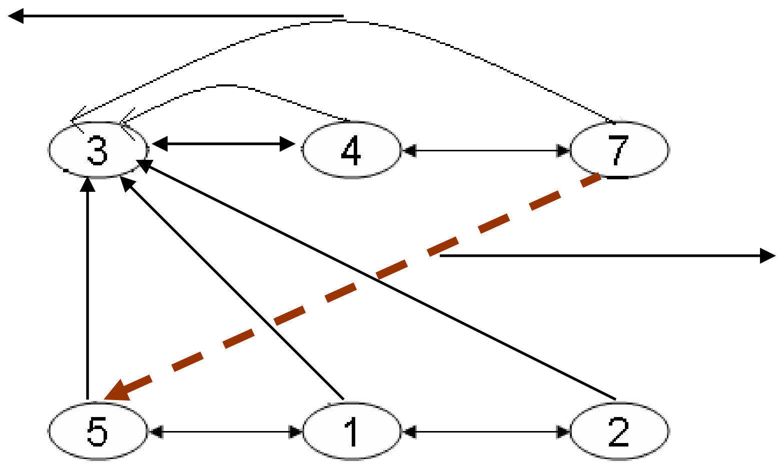
   Amortized time per operation $= \Theta(n)$.

2. ***Weighted-union heuristic:*** Always append the smaller list to the larger list. (Break ties arbitrarily.)

   A single union can still take $\Omega(n)$ time, e.g., if both sets have $n/2$ members.

LINKED LIST REPRESENTATION: UNION

Each element pointing to the head i,e ‘3’ in this example



The 2 sets are being merged by connecting 5 and 7

In this case the CONNECTED-COMPONENTS take $O(m+n^2)$ time.

Theorem 21.1

With weighted union, a sequence of m operations on n elements takes $O(m + n \lg n)$ time.

Sketch of proof Each MAKE-SET and FIND-SET still takes $O(1)$. How many times can each object's representative pointer be updated? It must be in the smaller set each time.

times updated	size of resulting set
1	≥ 2
2	≥ 4
3	≥ 8
$\vdots$	$\vdots$
k	$\geq 2^k$
$\vdots$	$\vdots$
$\lg n$	$\geq n$

Therefore, each merging set is updated in $\leq \lg n$ times

Thus the total time for CONNECTED-COMPONENTS is $O(m+n \lg n)$

HW

Exercises

- 11.2-1 and 11.2-4
- 11.3-1, 11.3-3 and 11.3-4
- 21.1-1, 21.1-2, and 21.1-3
- 21.2-1, 21.2-2, and 21.2-3

Backup Slides