

CSE 100: HASH TABLES

Where we've been and where we are going...

Our goal so far: We want to store and retrieve data (keys) fast

Tree structures

- **BSTs**: simple, fast in the average case, but can perform poorly with ordered data (you implemented)
- **AVL trees**: Guaranteed fast, but tricky to implement (you will not implement)
- **RSTs (built on treaps)**: Simpler than AVL trees, usually fast, even with ordered data (you will not implement)
- **Red-black trees**: Guaranteed fast, built in to C++ (coming soon... but you will not implement)

Hash tables (today thru Friday):

- Very fast in the average case
- No sorted access to data
- Can be tricky to do well (you are using an implementation built into C++ in PA3)

Note: you are responsible for knowing how to work with all of these structures as we did in class/in the reading.

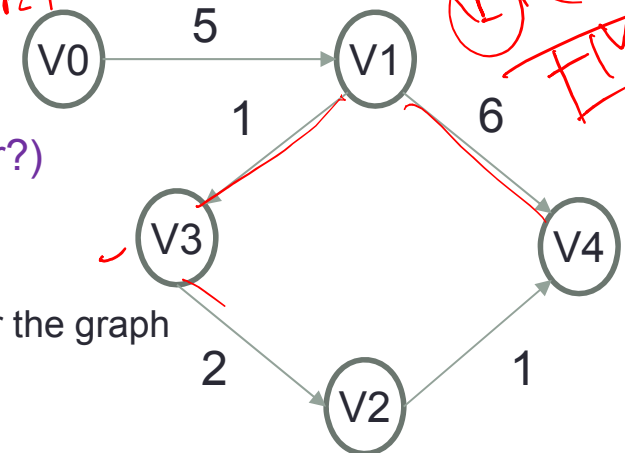
Dijkstra's Algorithm: Run time

- Initialize the graph: Give all vertices a dist of INFINITY, set all "done" flags to false
- Start at s ; give s dist = 0 and set prev field to -1
- Enqueue $(s, 0)$ into a priority queue. This queue contains pairs (v, cost) where cost is the best cost path found so far from s to v . It will be ordered by cost, with smallest cost at the head.
- While the priority queue is not empty:
 - Dequeue the pair (v, c) from the head of the queue.
 - If v 's "done" is true, continue
 - Else set v 's "done" to true. We have found the shortest path to v . (It's prev and dist field are already correct).
 - For each of v 's adjacent nodes, w :
 - Calculate the best path cost, c , to w via v by adding the edge cost for (v, w) to v 's "dist".
 - If c is less than w 's "dist", replace w 's "dist" c and enqueue (w, c) .

What is the running time of this algorithm in terms of $|V|$ and $|E|$? (More than one might be correct—which is tighter?)

- A. $O(|V|^2)$
- B. $O(|E| + |V|)$
- C. $O(|E| \log(|V|))$
- D. $O(|E| \log(|E|) + |V|)$
- E. $O(|E| * |V|)$

Assume adjacency list representation for the graph



Prim's MST Algorithm: Run Time

1. Create an empty graph T . Initialize the vertex vector for the graph. Set all "done" fields to false. Pick an arbitrary start vertex s . Set its "done" field to true. Iterate through the adjacency list of s , and put those edges in the priority queue.
2. While the priority queue is not empty:
 - Remove from the priority queue the edge (v, w, cost) with the smallest cost.
 - Is the "done" field of the vertex w marked true?
 - If Yes: this edge connects two vertices already connected in the spanning tree, and we cannot use it. Go to 2.
 - Else accept the edge:
 - Mark the "done" field of vertex w true, and add the edge (v, w) to the spanning tree T .
 - Iterate through w 's adjacency list, putting each edge in the priority queue.

What is the running time of this algorithm in terms of $|V|$ and $|E|$? (More than one might be correct—which is tighter?)

- A. $O(|V|^2)$
- B. $O(|E| + |V|)$
- C. $O(|E| \log(|V|))$
- D. $O(|E| \log(|E|) + |V|)$
- E. $O(|E| * |V|)$

Handwritten analysis:

$$\log V^2 = \frac{2 \log(V)}{\log(V)} \neq O(|V|)$$

$$O(\log(E)) = O(\log(V))$$

$$E \log(E) + |V|$$

$$V^2$$

Fast Lookup: Hash tables

- So far, almost all of our data structures have had performance $O(\log N)$ for insert and find.
- Operations supported by hash tables:
 - Find (key based look up) $\leftarrow O(1)$
 - Insert $O(1)$
 - Delete $O(1)$
- Average run times:
- No worst case guarantees

Hash table Motivation: Two sum problem

- Consider the 2-sum problem: Given an unsorted array (A) of N integers between 0 and 1000,000, find all pairs of elements that sum to a given number T

A

--	--	--	--	--	--	--	--	--	--	--	--

- Method 1: Exhaustive search

```

for (i = 0; i < N; i++) {
  for (j = i + 1; j < N; j++) {
    if ((A[i] + A[j]) == T)
      store (A[i], A[j]);
  }
}

```

$$\begin{array}{r}
 N \times N \\
 \hline
 (N^2) \\
 \hline
 O(N^2)
 \end{array}$$

Q: What is the worst case run time of this method? Assume the store method (inserts the elements into an unsorted linked list)

- A. $O(N)$
- B. $O(N \log N)$
- C. $O(N^2)$

Hash table Motivation: Two sum problem

- Consider the 2-sum problem: Given an unsorted array (A) of N integers between 0 and 1000,000, find all pairs of elements that sum to a given number T

A

--	--	--	--	--	--	--	--	--	--	--	--

- Method 2: Sort first then search

```

sort (A);
for (i = 0; i < N; i++) {
    j = binary_search(T - A[i]);
    store (A[i], A[j]);
}

```

Handwritten notes: $\leftarrow$ (pointing to `sort(A)`), $\leftarrow (108M) \times M$ (pointing to `binary_search`), $\leftarrow$ (pointing to `store`)

Q: What is the worst case run time of this method? Assume the store method (inserts the elements into an unsorted linked list)

- A. $O(N)$
- ☒ B. $O(N \log N)$
- C. $O(N^2)$

De-tour: Finding data fast

- Recall:

The data you will store will always be integers between 0 and 1,000,000. You decide to use an array with 1,000,000 Boolean values to store your data. An entry will be true if the item is in your structure, and false otherwise.

- What is the (Big-O) running time to insert an item into this proposed structure?

A. $O(N)$

B. $O(\log N)$

C. $O(1)$



(i)

$$A[i] = \underline{1}$$

The Big O running time of a find will be:

Can we use this results to improve the performance of the two-sum problem

Hash table Motivation: Two sum problem

- Consider the 2-sum problem: Given an unsorted array (A) of N integers between 0 and 1000,000, find all pairs of elements that sum to a given number T



- Method 3: Use an array based lookup table
- ```

bool Hashtable[ARRAY_SIZE]={false};
for (i = 0; i<N; i++){
 insert(Hashtable, A[i]);
}
for (i = 0; i<N; i++){
 if (find(Hashtable, T-A[i]) == true)
 store (A[i], A[j]);
}

```
- Handwritten notes:*  $O(1)$  is written next to the insert and store operations. A large red bracket spans the two loops.

*Handwritten complexity analysis:*

$$\frac{O(N^2)}{O(N \log N)} = O(N)$$

Q: What is the worst case run time of this method? Assume the store method (inserts the elements into an unsorted linked list)

- ☒ A.  $O(N)$
- ☐ B.  $O(N \log N)$
- ☐ C.  $O(N^2)$

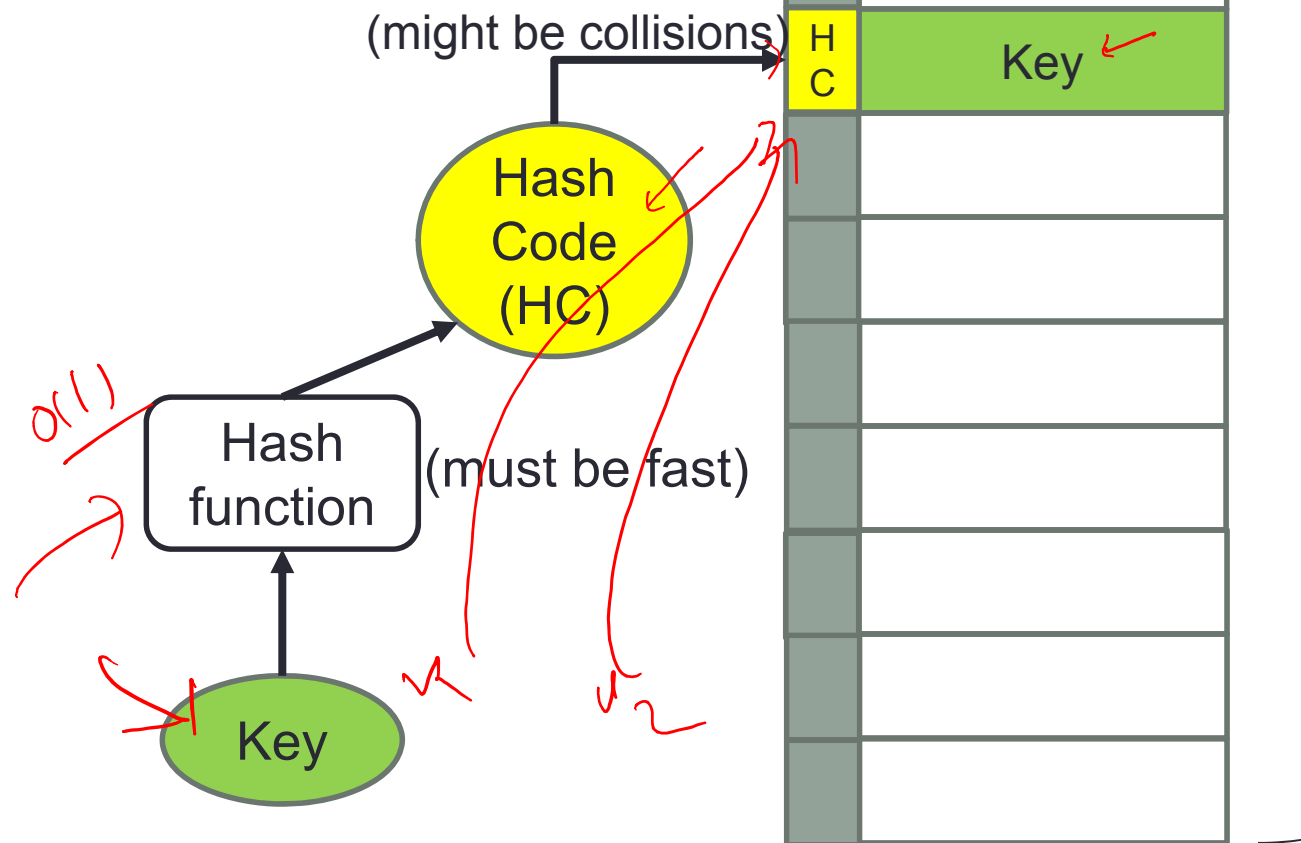
What is the problem with this method?

# Fast Lookup: Hash functions

- Setup for hashing:
  - Universe of possible keys  $U$  
  - Keep track of evolving set  $S$
  - $|S|$  is approximately known 

# Hashing

- Let's modify our array-based look up table
- Need a hash-function  $h(x)$ : takes in a key, returns an index in the array
- gold standard: random hash function



# Probability of Collisions

- Suppose you have a hash table that can hold 100 elements. It currently stores 9 elements (in 9 different locations in the hash table). What is the probability that your next insert will cause a collision (assuming a totally random hash function)?

- A. 0.09
- B. 0.10
- C. 0.37
- D. 0.90
- E. 1.00

# Probability of Collisions

- Suppose you have a hash table that can hold 100 elements. It currently stores 30 elements (in one of 30 possible different locations in the hash table). What is the probability that your next two inserts will cause at least one collision (assuming a totally random hash function)? (Choose the closest match)

- A. .09
- B. .30
- ☒ C. .52
- D. .74
- E. .90

$$\begin{aligned}
 &1 - P(\text{No collision}) \\
 &1 - \left( \frac{70}{100} \times \frac{69}{100} \right) \\
 &1 - 0.48 \\
 &= 0.52
 \end{aligned}$$

# Probability of Collisions

- If you have a hash table with M slots and N keys to insert in it, then the probability of at least 1 collision is:

$$\begin{aligned}
 P_{N,M}(\text{collision}) &= 1 - P_{N,M}(\text{no collision}) \\
 &= 1 - \prod_{i=1}^N P_{N,M}(\text{ith key no collision})
 \end{aligned}$$

# Hashtable collisions and the "birthday paradox"

- Suppose there are 365 slots in the hash table:  $M=365$
- What is the probability that there will be a collision when inserting  $N$  keys?
  - For  $N=10$ ,  $\text{prob}_{N,M}(\text{collision}) = 12\%$
  - For  $N=20$ ,  $\text{prob}_{N,M}(\text{collision}) = 41\%$
  - For  $N=30$ ,  $\text{prob}_{N,M}(\text{collision}) = 71\%$
  - For  $N=40$ ,  $\text{prob}_{N,M}(\text{collision}) = 89\%$
  - For  $N=50$ ,  $\text{prob}_{N,M}(\text{collision}) = 97\%$
  - For  $N=60$ ,  $\text{prob}_{N,M}(\text{collision}) = 99+\%$
- So, among 60 randomly selected people, it is almost certain that at least one pair of them have the same birthday
- On average one pair of people will share a birthday in a group of about  $\sqrt{2 \times 365} \approx 27$  people
- In general: collisions are likely to happen, unless the hash table is quite sparsely filled
- So, if you want to use hashing, can't use perfect hashing because you don't know the keys in advance, and don't want to waste huge amounts of storage space, you have to have a strategy for dealing with collisions

$$N=23$$

$$M=2 - \left(\frac{1}{2}\right)$$

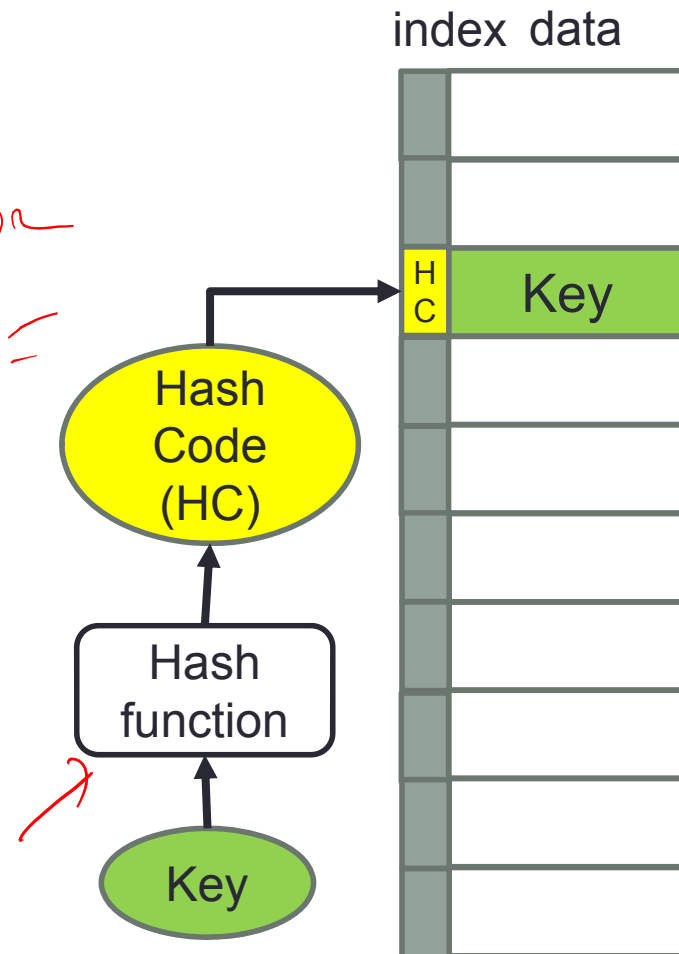
$$(25) -$$

$$\sqrt{2 \times 365} = 27$$

# Making hashing work

## Hash table (array)

- Important issues in implementing hashing are:
  - Deciding on the **hash function**
  - Deciding on the **size of the hash table**
  - Deciding on the **collision resolution strategy**
- With a good hashtable design,  $O(1)$  average-case insert and find operation time costs can be achieved, with  $O(1)$  space cost per key stored
- This makes hashtables a very useful and very commonly used data structure



## Hash functions: desiderata

Important considerations in designing a hash function for use with a hash table

- It is fast to compute (must be  $O(1)$ )
- It distributes keys evenly
- It is consistent with the equality testing function (i.e. two keys that are equal will have the same hash value)

Designing a good hash function is not easy!

We'll look at a few, but there's much more to explore.

# Simple (and effective?) hash function for integers: $H(K) = K \bmod M$

- When is the function  $H(K) = K \bmod M$  where  $K$  is the key and  $M$  is the table size a good hash function for integer keys?

- A. Always
- B. When  $M$  is prime
- C. When values of  $K$  are evenly distributed
- D. In the case of either B or C
- E. Never



## Simple (and effective?) hash function for integers: $H(K) = K \bmod M$

- When is the function  $H(K) = K \bmod M$  where  $K$  is the key and  $M$  is the table size a good hash function for integer keys?

When  $M$  is prime OR When values of  $K$  are evenly distributed

Think of an example of a non-prime  $M$  and a distribution of keys that would cause poor behavior from this function.

## Simple (and effective?) hash function for integers: $H(K) = K \bmod M$

- When is the function  $H(K) = K \bmod M$  where  $K$  is the key and  $M$  is the table size a good hash function for integer keys?

When  $M$  is prime OR When values of  $K$  are evenly distributed

Because you can never count on evenly distributed keys, always use prime-size table with this hash function

## Hash table size

- By "size" of the hash table we mean how many slots or buckets it has
- Choice of hash table size depends in part on choice of hash function, and collision resolution strategy
- But a good general “rule of thumb” is:
  - The hash table should be an array with length about 1.3 times the maximum number of keys that will actually be in the table, and
  - Size of hash table array should be a prime number (especially with the simple hash function we looked at)
- So, let  $M$  = the next prime larger than 1.3 times the number of keys you will want to store in the table, and create the table as an array of length  $M$
- (If you underestimate the number of keys, you may have to create a larger table and rehash the entries when it gets too full; if you overestimate the number of keys, you will be wasting some space)

# Next time

- Hash functions
- Resolving collisions in hash tables