

CSE 100
Midterm Review
Ternary Tries

Announcements

- PA4 checkpoint deadline extended to Tues, May 26 at 11:00pm

Dijkstra's Algorithm

The array of vertices, which include dist, prev, and done fields (initialize dist to 'INFINITY' and done to 'false'): *Source: v0*

V0: dist=0 prev=-1 done=~~t~~

adj: (v1, 1), (v2, 6), (v3, 3)

V1: dist=~~1~~ prev=~~v0~~ done=~~t~~

adj: (v2, 4)

V2: dist=~~6~~ ~~3~~ prev=~~v0~~ ~~v3~~ done=~~t~~

adj: ↓

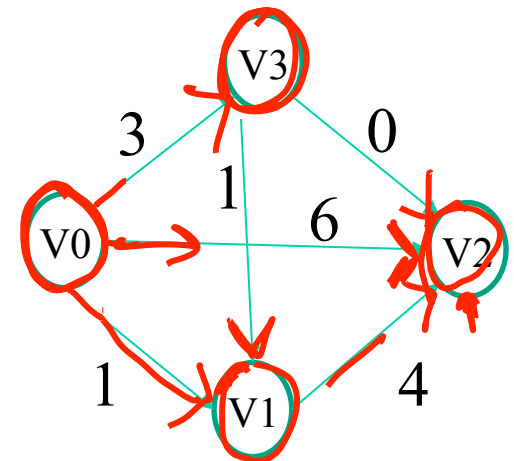
V3: dist=~~3~~ prev=~~v0~~ done=~~t~~

adj: (v2, 0) (v1, 1)

The priority queue (set start vertex dist=0, prev=-1, and insert it with priority 0 to start)

1. (v0, 0)
2. (v1, 1) (v2, 6) (v3, 3)
3. (v2, 5) (v2, 6) (v3, 3)
4. (v2, 3) (v2, 5) (v2, 6)

5. (v2, 5) (v2, 6)



Union by height

Start with 4 items: 0, 1, 2, 3, 4, 5, 6, 7



Ties: **Union**(*i*, *j*) makes the node **Find**(*i*) the parent of the node **Find**(*j*)

Perform these operations:

- ✓ **Union**(2, 3)
- ✓ **Union**(1, 2)
- ✓ **Union**(0, 1)
- ✓ **Union**(4, 5)
- Union**(6, 7)
- Union**(4, 6)

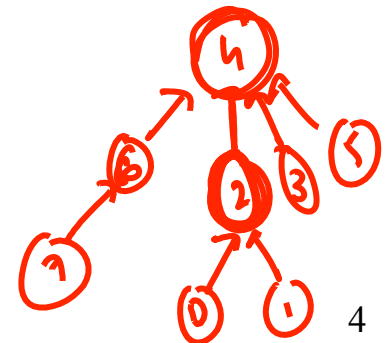
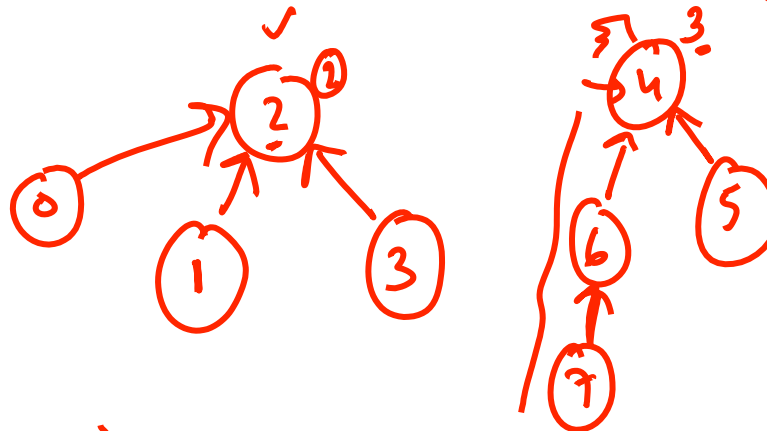
Union runtime $O(1)$

Union(2, 7)
find(3) (path compression)

0	1	2	3	4	5	6	7
-1	-1	-1	-1	-1	-1	-1	-1

2 2 -2 2 -3 4 4 6

final up tree representation



Kruskal's algorithm run time with smart union

no path compression: $|E|$: No. of edges
 $|V|$: No. of vertices

1. Sort edges in increasing order of cost

2. Set of edges in MST, $T = \{\}$

3. For $i = 1$ to $|E|$

$s1 = \text{find}(u); s2 = \text{find}(v);$ $O(\log |V|)$

if ($s1 \neq s2$) { // If $T \cup \{e_i = u, v\}$ has no cycles

Add e_i to T ✓ $O(1)$

union($s1, s2$) $O(1)$

}

A. $O(|E| \log |E|)$
B. $O(|E| \log |V|)$

C. $O(\log |E|)$

D. $O(|E|) \times$

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

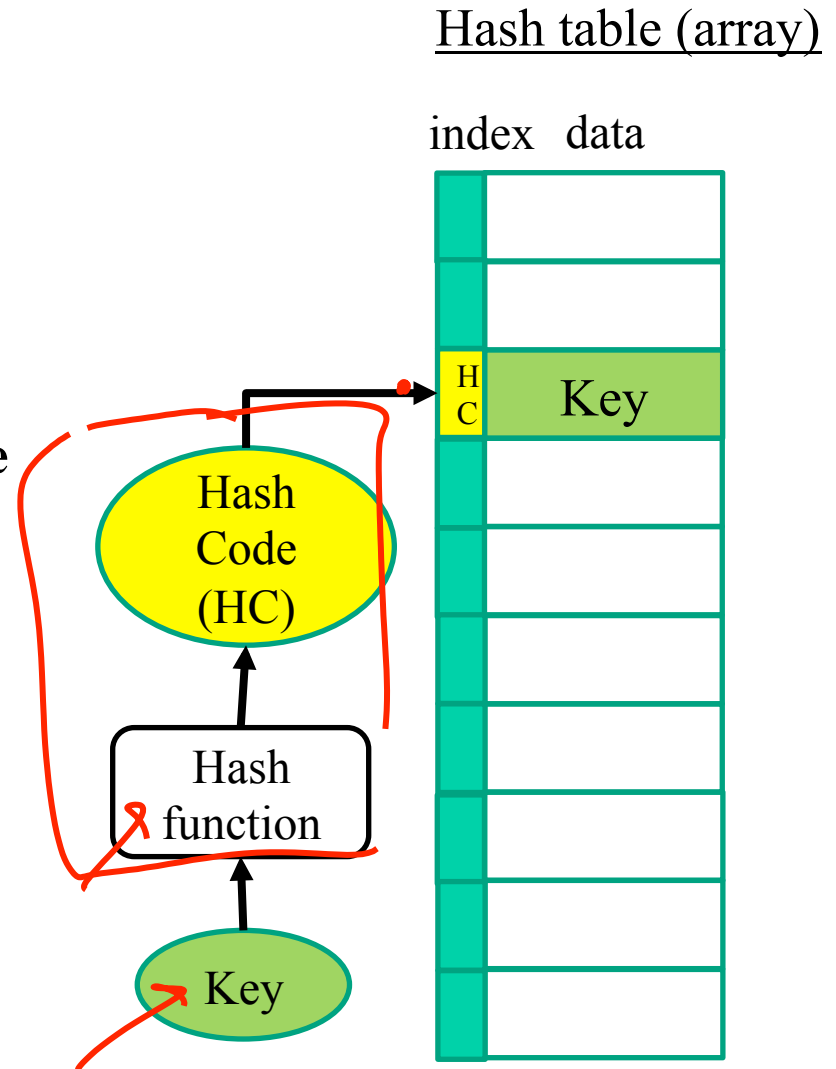
$O(|E|)$

$O(|V|)$

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

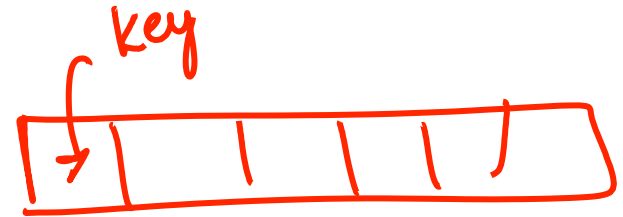
Making hashing work

- 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

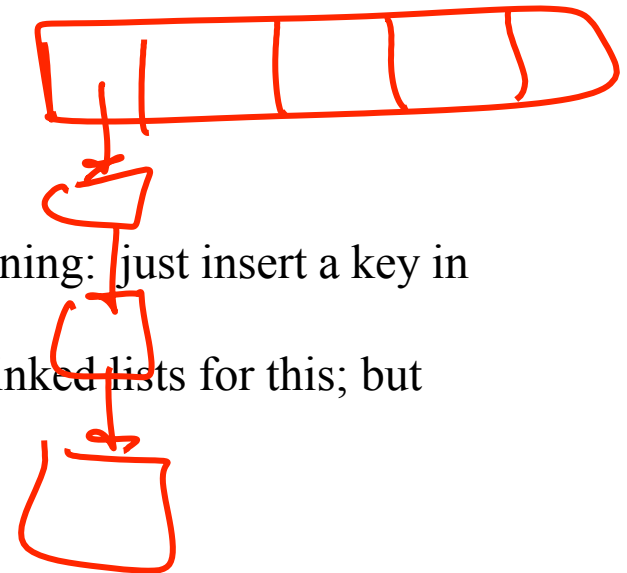


Open addressing vs. separate chaining

- Linear probing and random hashing are appropriate if the keys are kept as entries in the hashtable itself...
 - doing that is called "open addressing"
 - it is also called "closed hashing"



- Another idea: Entries in the hashtable are just pointers to the head of a linked list ("chain"); elements of the linked list contain the keys...
 - this is called "separate chaining"
 - it is also called "open hashing"



- Collision resolution becomes easy with separate chaining: just insert a key in its linked list if it is not already there
 - (It is possible to use fancier data structures than linked lists for this; but linked lists work very well in the average case)

Linear probing: inserting a key

- When inserting a key K in a table of size M , with hash function $H(K)$
 - Set $\text{indx} = H(K)$
 - If table location indx already contains the key, no need to insert it. Done!
 - Else if table location indx is empty, insert key there. Done!
 - Else collision. Set $\text{indx} = (\text{indx} + 1) \bmod M$.
 - If $\text{indx} == H(K)$, table is full! (Throw an exception, or enlarge table.) Else go to 2.

$M = \underline{7}$, $H(K) = K \bmod M$ 701 mod 7 ✓
insert these keys 701 (1), 145 (5), 217 (0), 19 (5), 13 (6), 749 (0)

in this table, using linear probing:

					19 x	+	
217	701	13	~~749~~		145	19	
index:	0	1	2	3	4	5	6

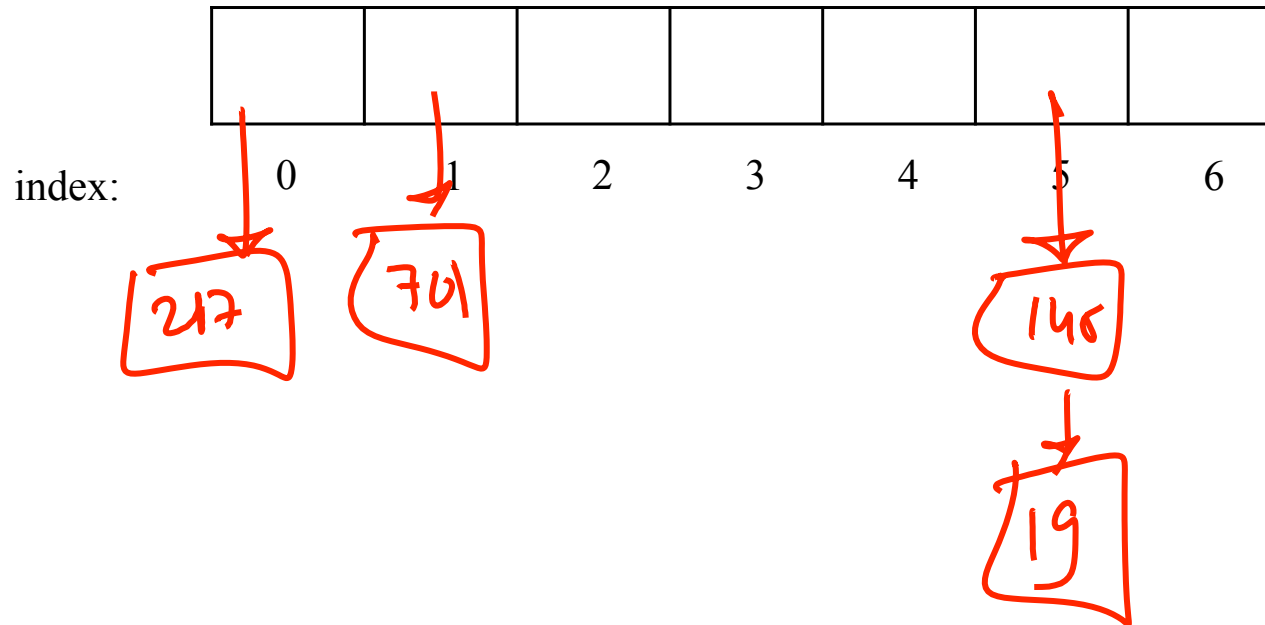
Resolving Collisions: Separate Chaining

- using the hash function $H(K) = K \bmod M$, insert these integer keys:

701 (1), 145 (5), 217 (0), 19 (5), 13 (6), 749 (0)



in this table:

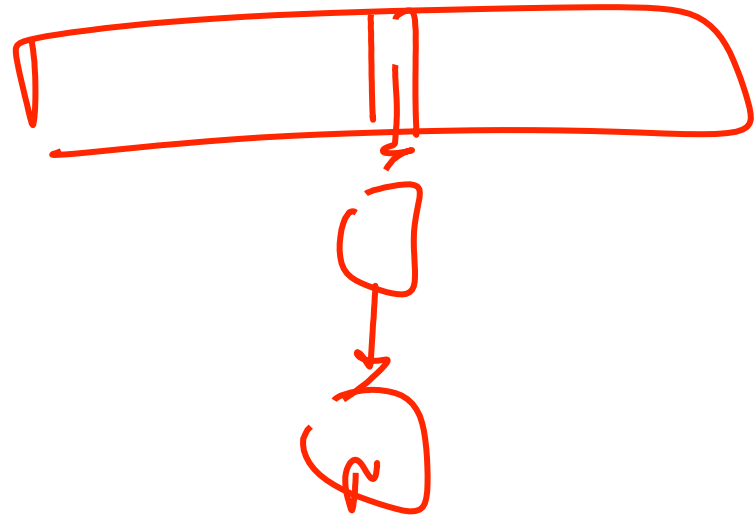


Analysis of open-addressing hashing

↓
linear probing

- What is the *worst-case* time to find a single element in a hash table with N elements in it?

- A. $O(1)$
- B. $O(\log(N))$
- ☒ C. $O(N)$
- D. $O(N\log(N))$
- E. $O(N^2)$



Multi-way tries: Efficient finding of keys by their sequence

Build the trie to store the following numbers:

8

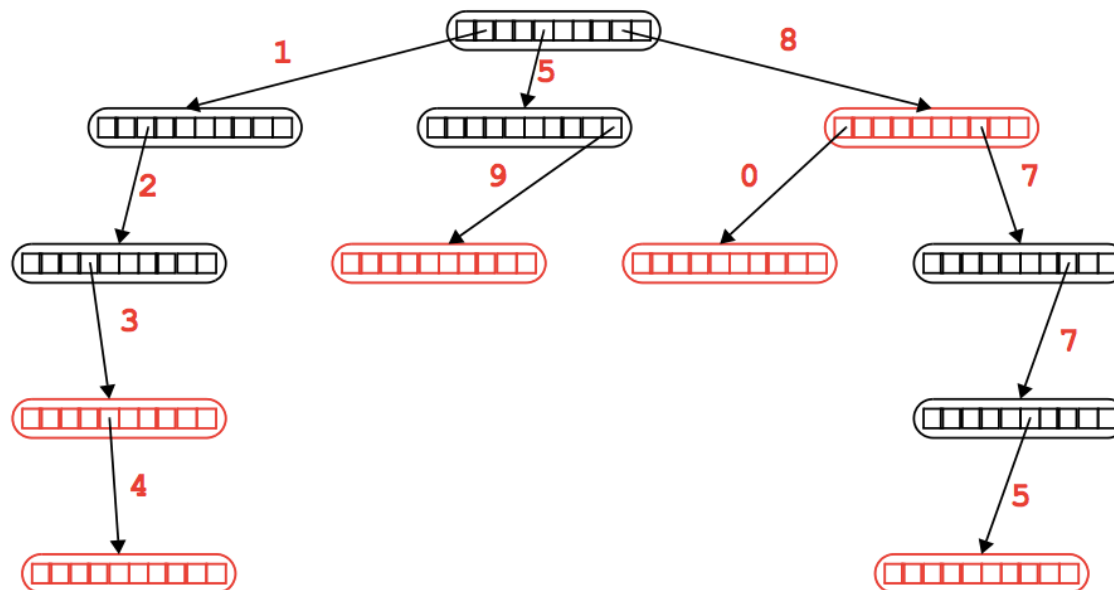
1234

59

123

8775

80

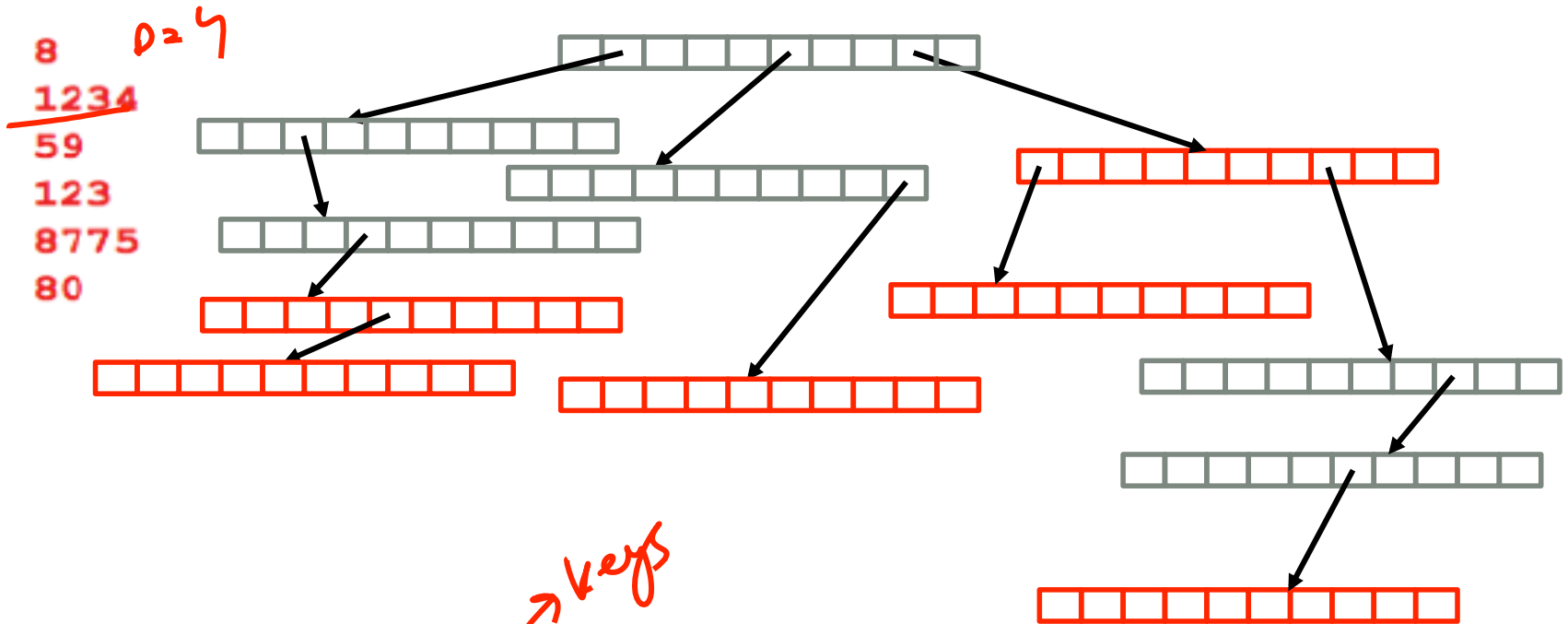


Is there a way to find whether *all* keys contained in a sequence of digits are present in the trie?

- A. Yes
- B. No

Properties of tries

Build the trie to store the following numbers:

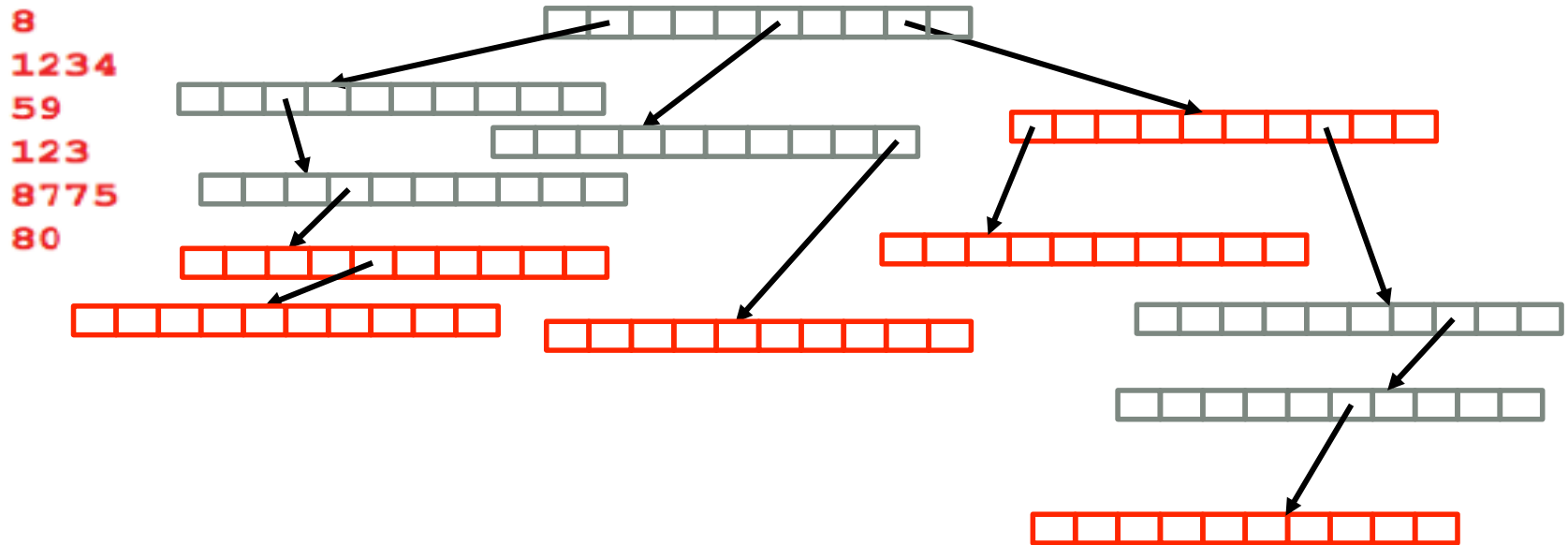


If you stored the same **N**-digit keys in a Binary Search Tree, what would be the worst case height of the tree?

- A. N B. $\lg(10^D)$ C. $\lg(N)$ D. $\lg(D)$ E. Other

Properties of tries

Build the trie to store the following numbers:



Consider storing the full 10^D keys. We know that on average the height of a BST will be $\lg(10^D)$. Which is smaller: D or $\lg(10^D)$?

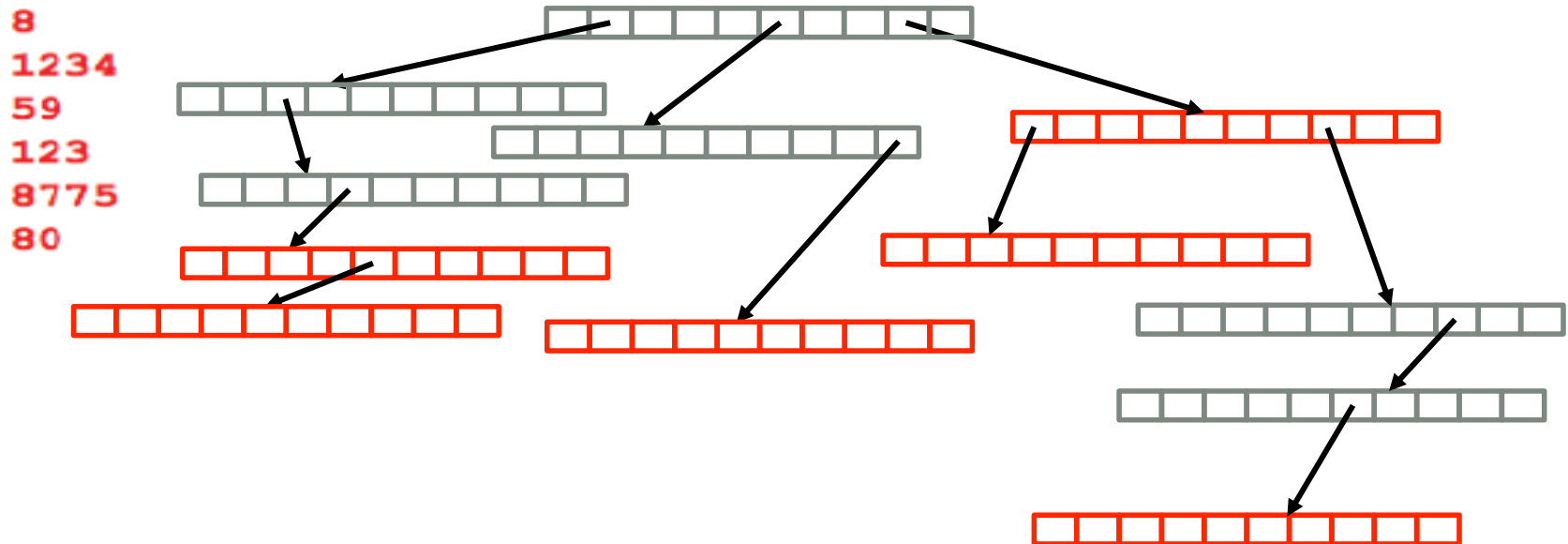
A. D

B. $\lg(10^D)$

C. They are the same

Properties of tries

Build the trie to store the following numbers:



So what is the main drawback of tries?

- A. They are difficult to implement
- B. They (usually) waste a lot of space
- C. They are slow
- D. There is no drawback of tries

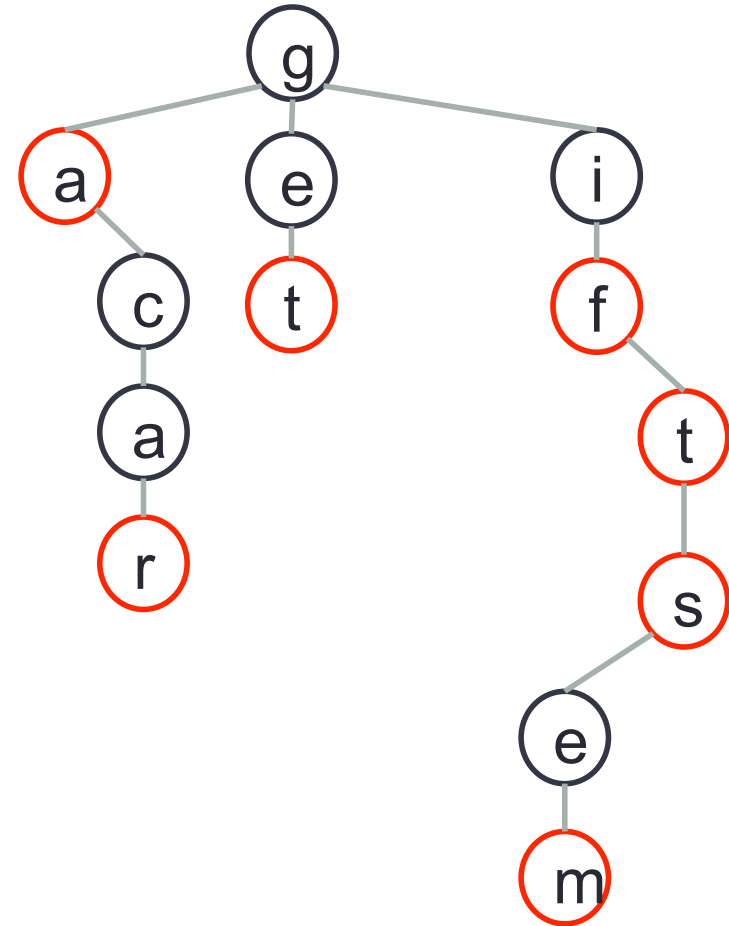
Ternary search trees to the rescue!

- Tries combine binary search trees with tries.
- Each node contains the following:
 - A key **digit** for search comparison
 - Three pointers:
 - **left** and **right**: for when the digit being considered is less than and greater than (respectively) the digit stored in the node (the BST part)
 - **middle**: for when the digit being considered is equal to the digit stored in the node (the trie part)
 - An **end** bit to indicate we've completed a key stored in the tree.

List all the words (strings) you can find in this TST

Are the following in the tree?
(A=yes, B=no)

- get
- if
- gif
- its
- gacar
- tsem



Draw the ternary tree for the following (in this order)

i
just
met
this
is
crazy
call
me
maybe

Does the structure of the tree depend on the order in which keys were inserted? A. Yes B. No

Algorithms for insert and find (in TSTs and MWTs)

- In your reading and/or in Paul Kube's slides