ADT Dictionary

Dictionary: a **collection of items**, each of which **contains a key and a value**

↳ called a "**key-value pair**" (KVP)

↳ **Keys can be compared** and are (typically) unique.

Operations:

- **Search(k)**, aka, lookup(k)
- **insert(k, v)**
- **delete(k)**, aka, remove(k)

optionals: successor, merge, is-empty, size, etc

Common Assumptions:

1) Dictionary has **n KVPs**
2) Each KVP uses **constant space**
3) Keys can be compared in **constant time**
4) (Usually:) dictionary is **non-empty** before and after operation

	search	insert	delete
unsorted list/array	$O(n)$	$O(1)$	$O(1)$
sorted list/array	$O(\log n)$	$O(n)$	$O(n)$
binary search tree	$O(\text{height})$	$O(\text{height})$	$O(\text{height})$

Review: binary search

↳ note: only applies to a sorted array!

```
binary-search(A, n, k):
1) l=0, r=n-1
2) while (l < r) {
3)     m = ⌊(l+r)/2⌋
4)     if (A[m]==k) return "found at A[m]"
5)     else if (A[m] < k) then l=m+1
6)     else r=m-1
7) }
8) return "not found :c, but would be between A[l-1] and A[l]"
```

Review: Binary Search Trees (BSTs)

Structure:

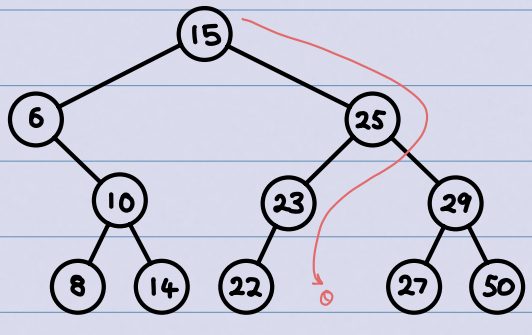
- all nodes have two (possibly empty) subtrees
- every node stores a KVP
- empty subtrees usually not shown

Ordering:

- every key k in T.left is less than the root key
- every key k in T.right is greater than the root key

BST::Search(k) → start at root, compare k to current node's key. Stop if found or subtree is empty, else recurse at subtree.

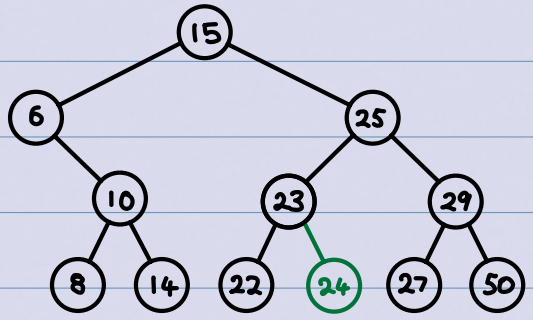
Eg: BST::Search(24):



∴, 24 not found in the BST!

BST:: insert(k, v) → Search for k, then insert (k, v) as a new node.

Eg: BST:: insert(24, v):



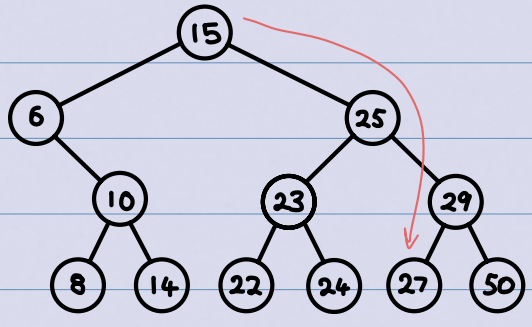
∴, 24 inserted!

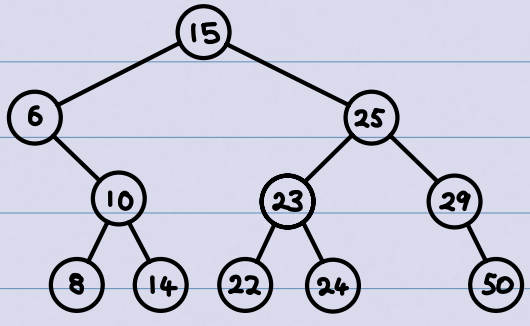
BST:: Delete(k) → first search for the node x that contains the key.

- If x is a leaf, delete it.
- If x has one empty subtree, move the child up
- Else, swap key at x with key at successor node and then delete that node

- Successor: next-smallest among all keys in the dict.

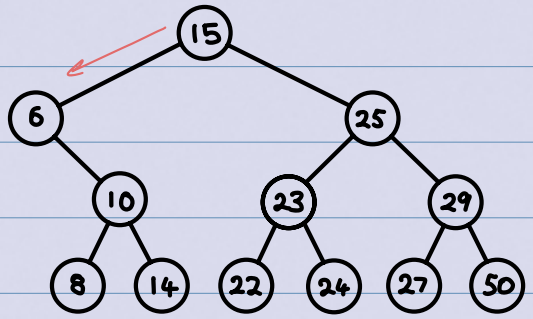
Eg: BST:: Delete(27): (leaf)

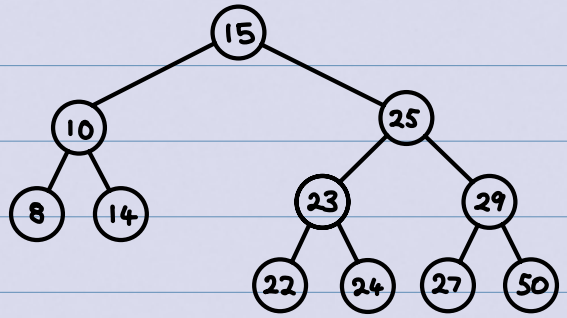




∴ Deleted using case 1

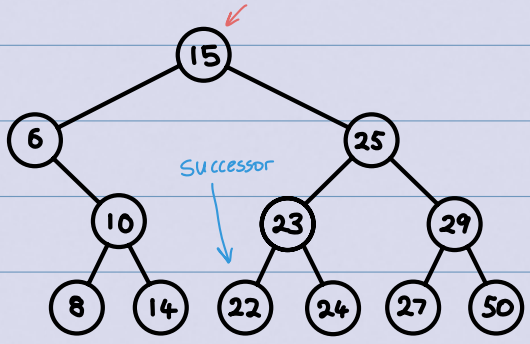
Eg: BST:: Delete(6):

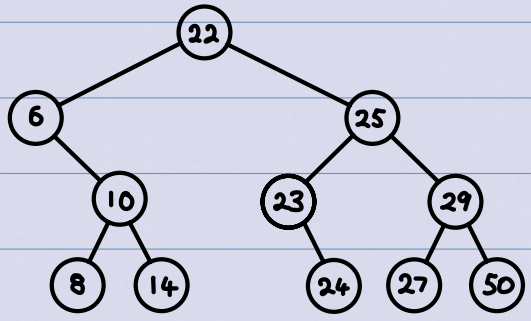




∴ Deleted using case 2

Eg: BST::Delete(15)





∴, deleted using case 3

→ BST::Search, BST::insert, and BST::delete all $O(h)$, where h = maximum number for which level h contains nodes.

↳ single-node tree has height 0, empty tree has height of -1.

- if n items are inserted one-at-a-time, how big is h?

↳ worst-case: $n-1 = O(n)$

↳ best-case: $O(\log n)$

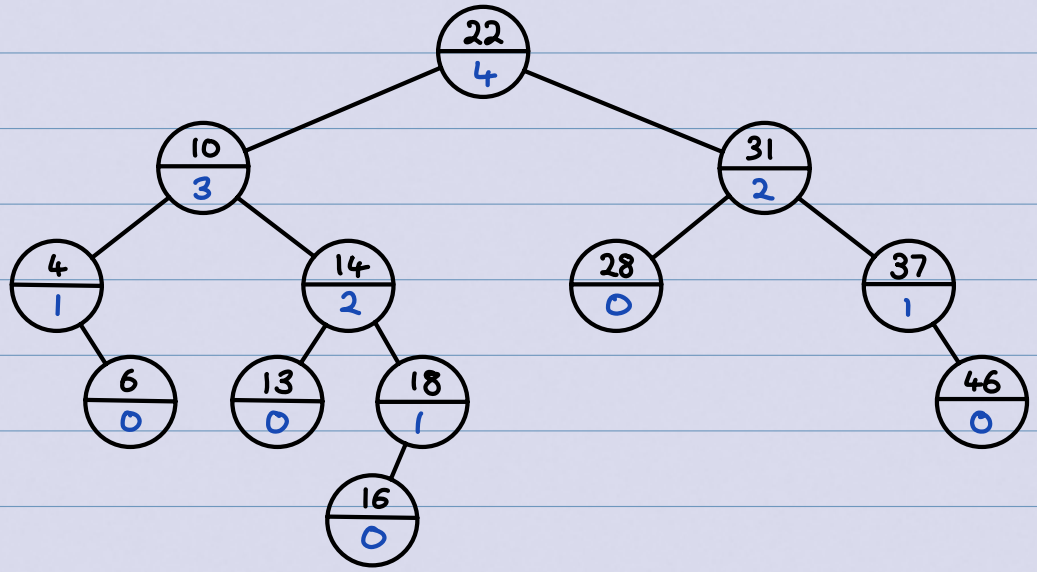
AVL Trees

an AVL tree is a BST with an additional height-balance property at every node:

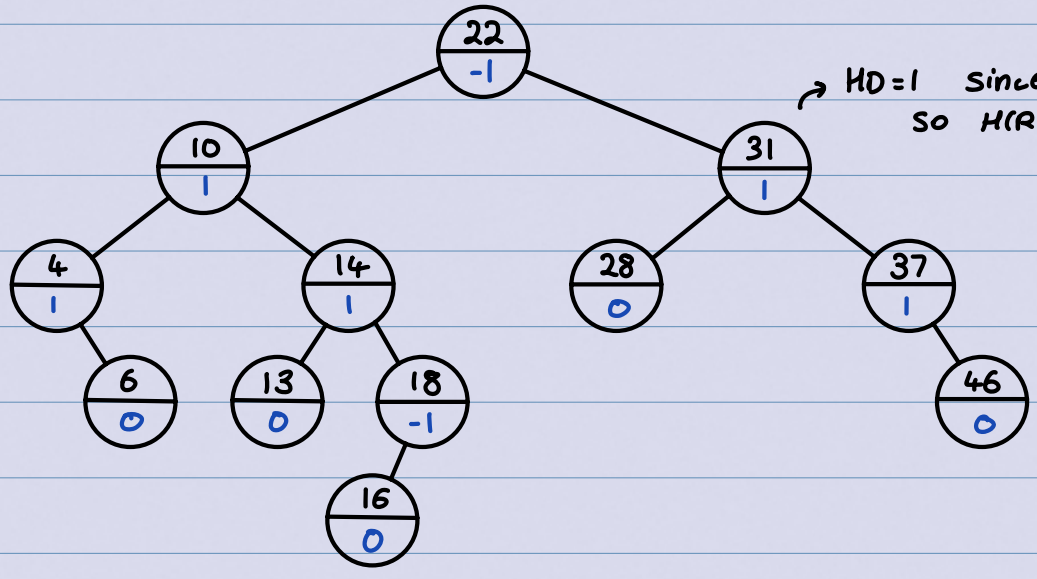
The heights of the left and right subtree differ by at most 1.

↳ ie, if node z has left subtree L and right subtree R, then height(R) - height(L) must be in {-1, 0, 1}

AVL Tree Example (w/ height):



AVL Tree Example (w/ height-difference):



Theorem: the height of an AVL tree on n nodes is in $\Theta(\log n)$.

↳ BST::search, BST::insert, BST::delete all cost $\Theta(\log n)$ in the worst case.

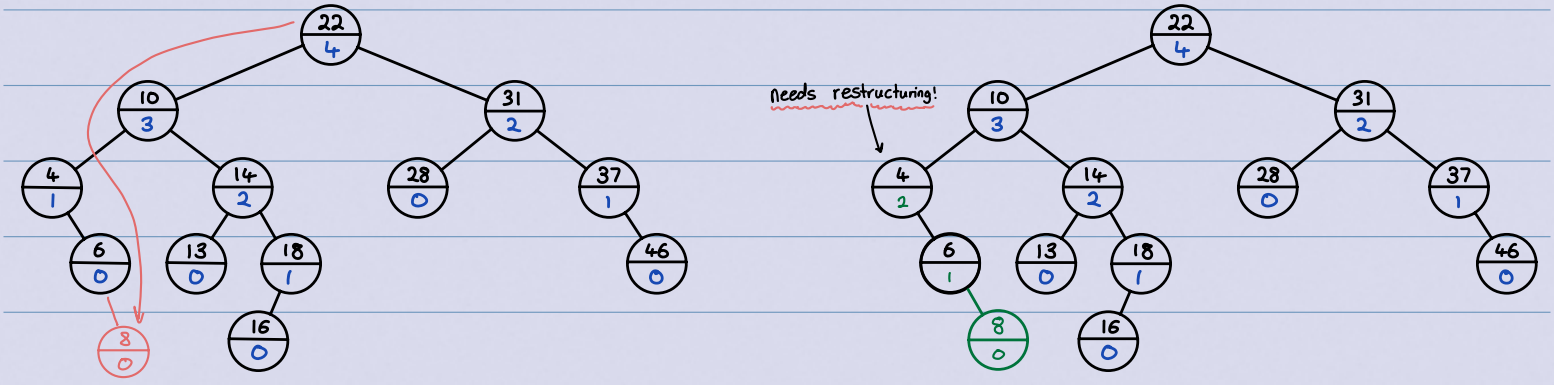
AVL::insert(k,v) → first, insert(k,v) with usual BST

insertion. We assume that this returns the new leaf z where the key was sorted. Then, move up the tree from z, and update the height (easy to do in constant time). If the height difference becomes ±2 at node z, then z is unbalanced, so we must re-structure the tree.

→ note, to set the height, we simply do:

u.height = 1 + max {u.left.height, u.right.height}

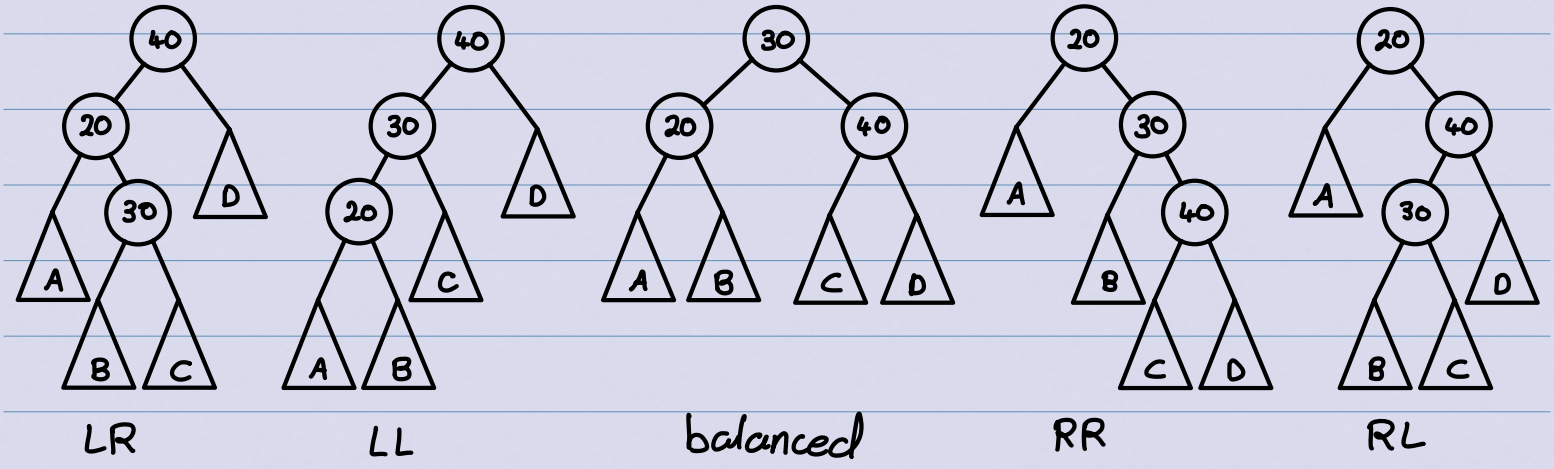
Eg: AVL: insert (8):



∴ After restructuring (later), 8 is inserted correctly!

Restructuring in a BST: Rotations

There are many different BSTs with the same keys. Eg:

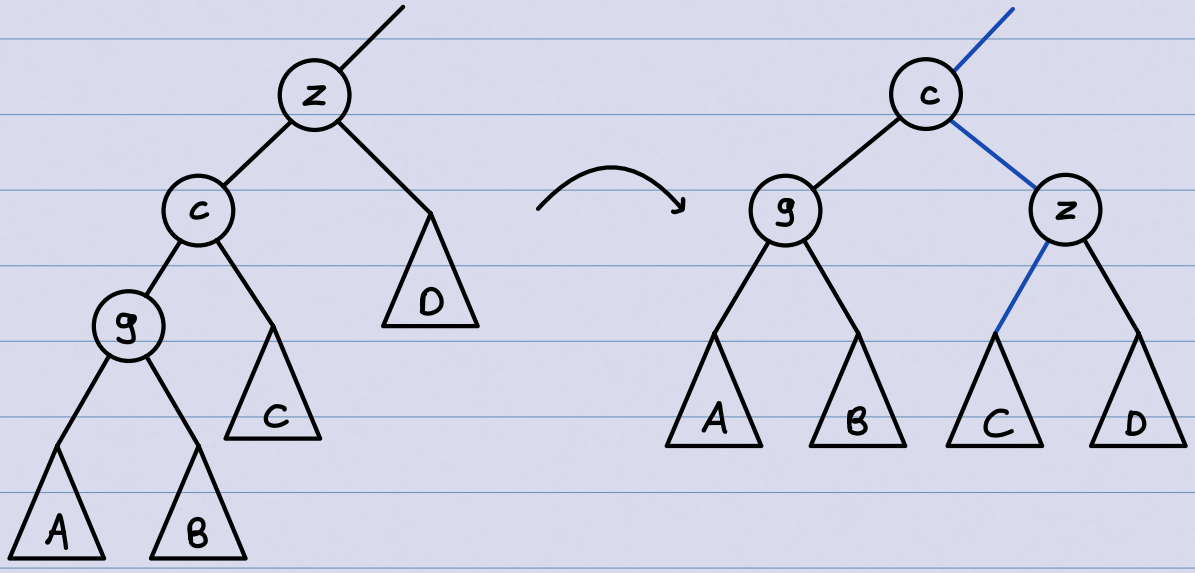


→ goal: change the structure without changing the order, and

restructure such that the subtree becomes balanced.

Right Rotation

this is a right-rotation on node z:



→ only O(1) links are changed. Useful to fix left-left imbalances.

Right-Rotation Pseudocode:

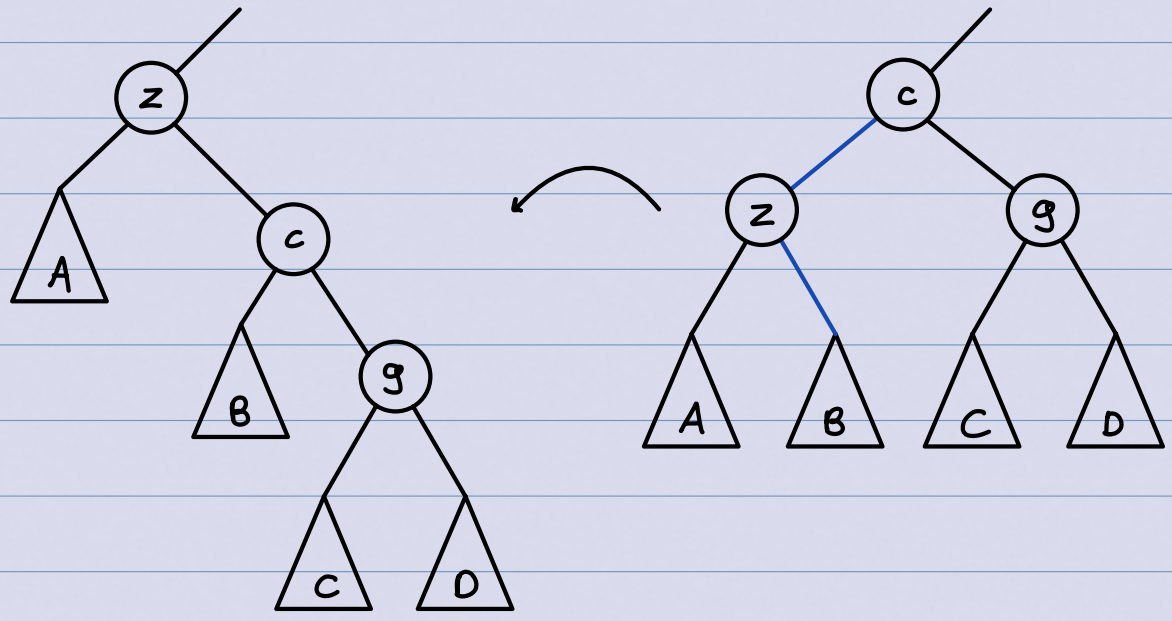
```
rotate-right(z)
1) c = z.left
2) // fix links connecting to above
3) c.parent = (p = z.parent)
4) if (p == null) { root = c }
5) else {
6)     if (p.left == z) { p.left = c }
7)     else { p.right = c }
8) }
9) // actual rotation
10) z.left = c.right,    c.right.parent = z
11) c.right = z,    z.parent = c
12) set-height-from-subtrees(z),    set-height-from-subtrees(c)
```

```
13) return c // returns new root of subtree
```

↳ runs in O(1)!

Left Rotation

this is a left-rotation on node z:



Again, only O(1) links need to be changed. Useful to fix right-right imbalances.

Left-Rotation Pseudocode:

```
rotate-left (z)
1) c = z.right
2) // fix links connecting to above
3) c.parent = (p = z.parent)
4) if (p == null) { root = c }
5) else {
6)     if (p.right == z) { p.right = c }
7)     else { p.left = c }
8) }
9) // actual rotation
```

10) z.right = c.left, c.left.parent = z

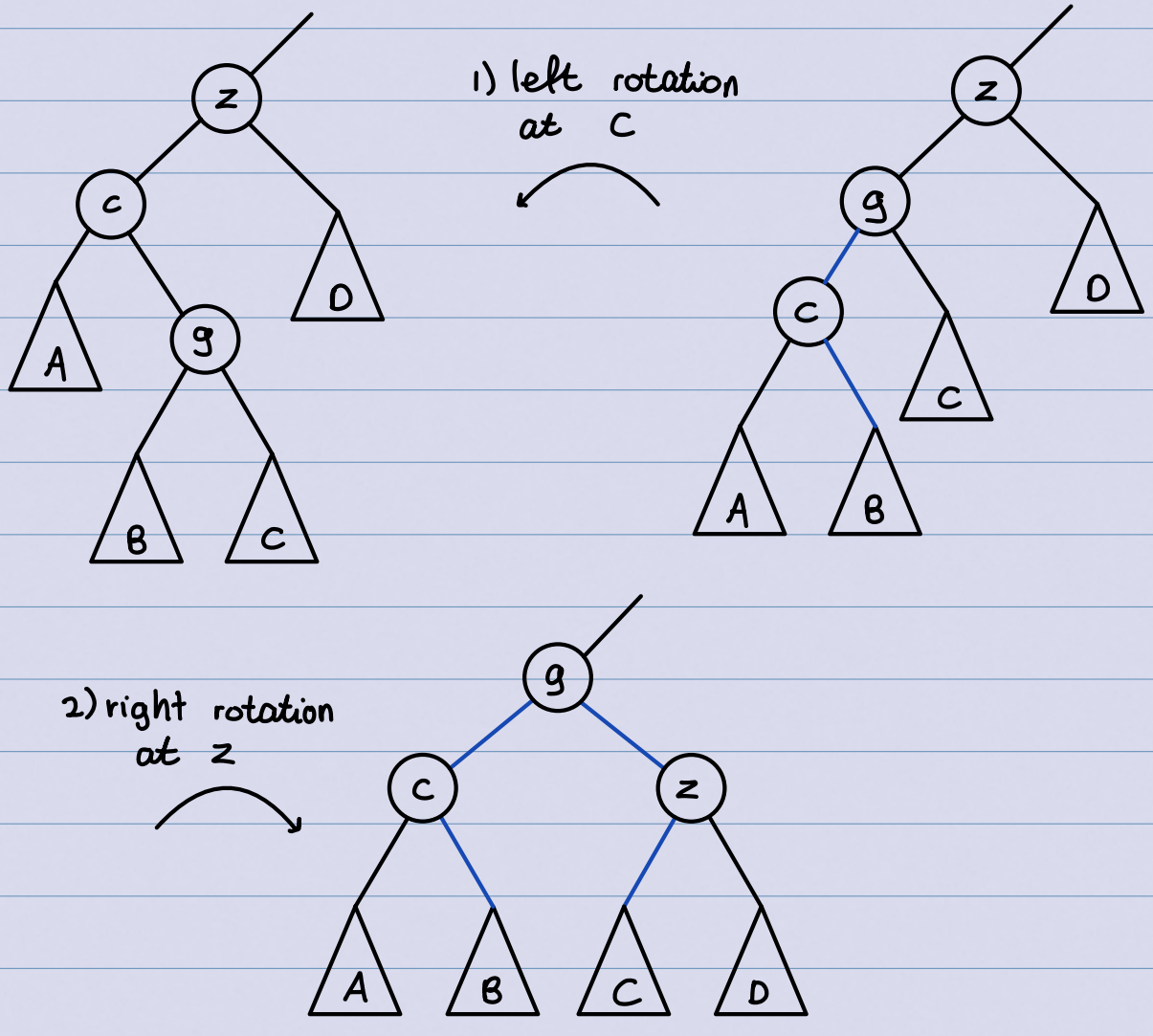
11) c.left = z, z.parent = c

12) set-height-from-subtrees(z), set-height-from-subtrees(c)

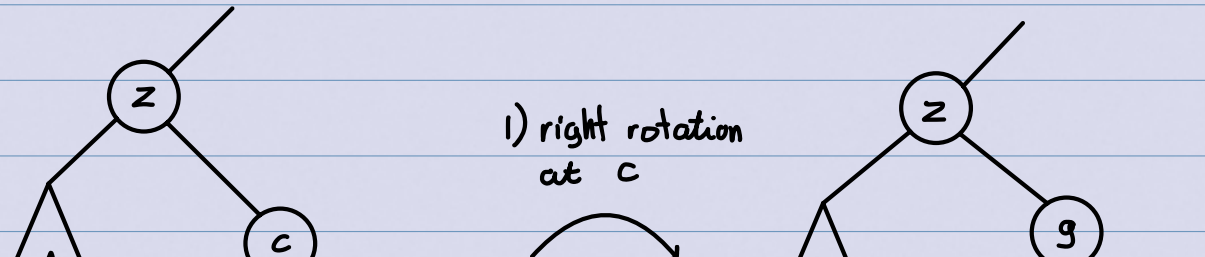
13) return c // returns new root of subtree

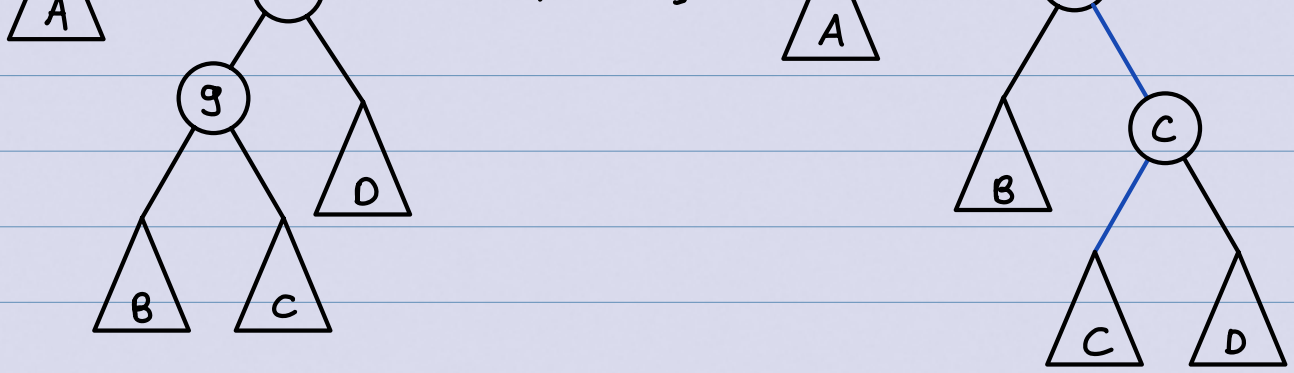
↳ runs in O(1)!

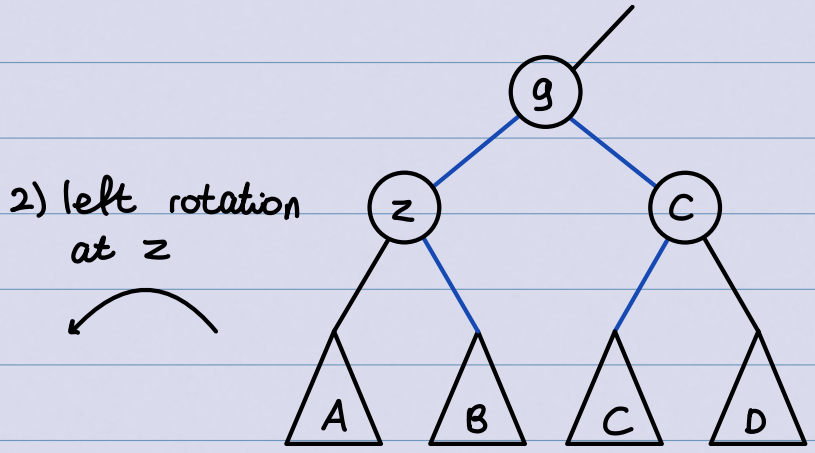
Double Right Rotation



Double Left Rotation







AVL Insertion Revisited

Imbalance at z: do (single or double) rotation

- Choose c as child where subtree has bigger height.

```
AVL::insert(k,v)
1)  z = BST::insert(k, v)      // new leaf with k
2)  while (z is not null) {
3)    if ( |z.left.height - z.right.height| > 1) {
4)       c = taller child of z
5)       g = taller child of c    //grandchild of z
6)       z = restructure(g, c, z)  // see in next code block
7)       break
8)    }
9)    set-height-from-subtrees (z)
10)   z = z.parent
11) }
```

↳ for insertion, one rotation restores all heights of subtrees.

Fixing an slightly-unbalanced AVL tree:

restructure(g, c, z) → node g is child of c which is child of z

1) case

// right rotation
u = rotate-right(z)

// double-right rotation
rotate-left(c)
u = rotate-right(z)

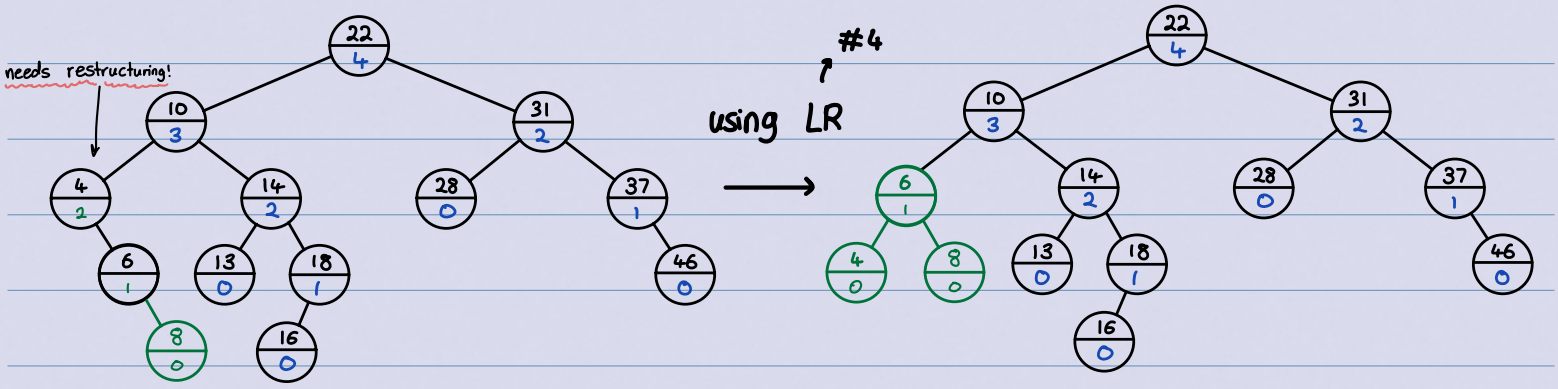
// double-left rotation
rotate-right(c)
u = rotate-left(z)

// left rotation
u = rotate-left(z)

2) return u

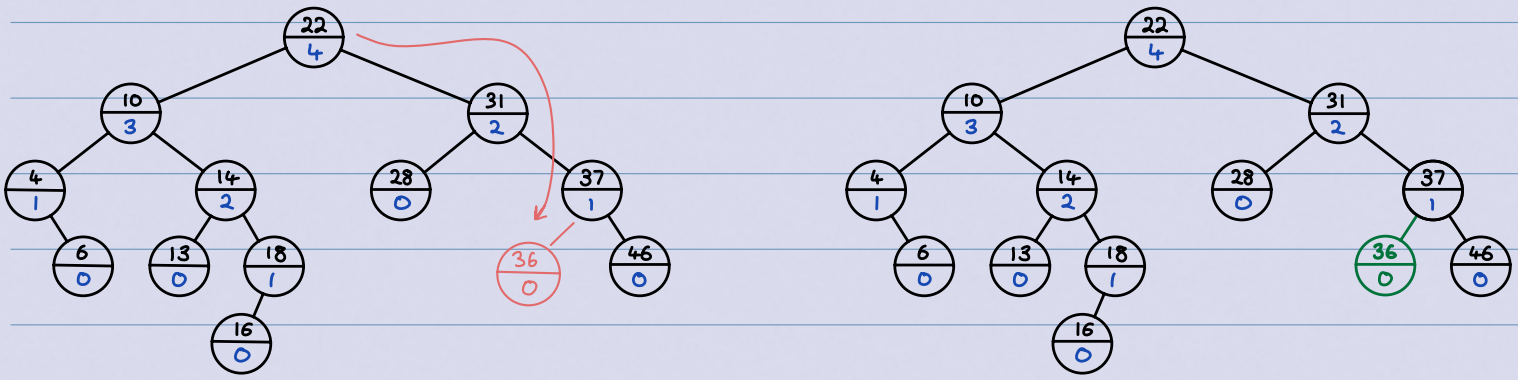
Eg: revisiting AVL::insert(8)

we had previously ended with an unbalanced AVL after inserting 8. Now, we can balance it:

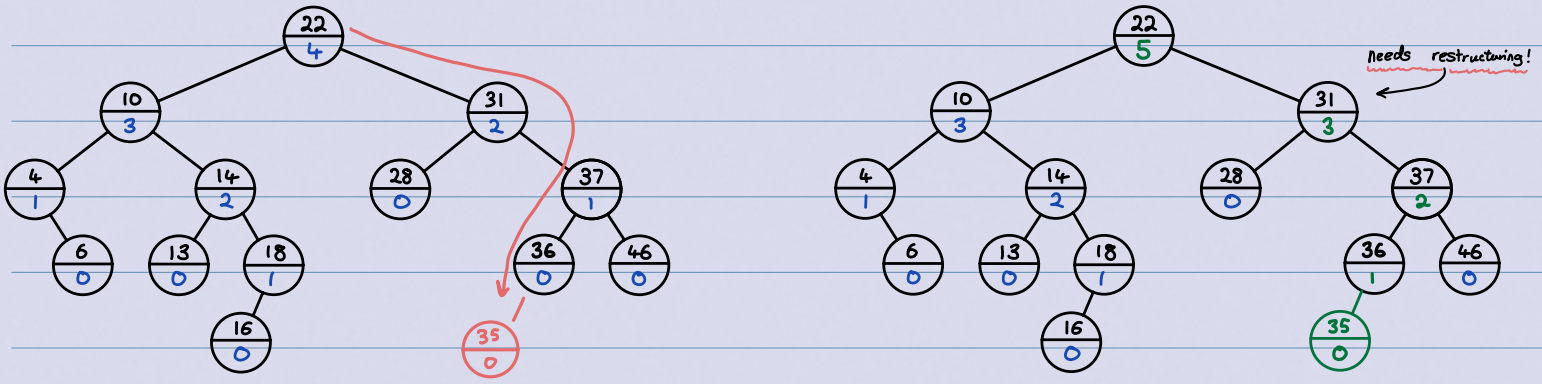


∴, we have correctly restructured the subtree using a left rotation.

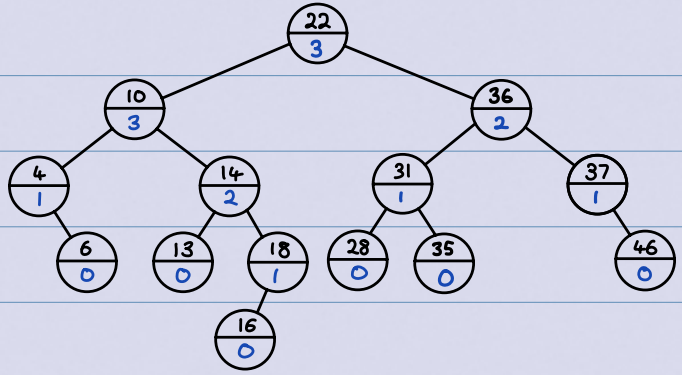
Eg: AVL::insert(35), AVL::insert(36):



→ 35 was inserted and the tree is still correctly structured!



→ 36 was added, but we must restructure, as node 31 is unbalanced! We will restructure node 31 because it's the first unbalanced node as we go up the tree. See that it follows the right-left (31→37, 37→36) pattern, so we must use a double-left rotation:



∴, we have inserted 35 and 36, and restructured accordingly!

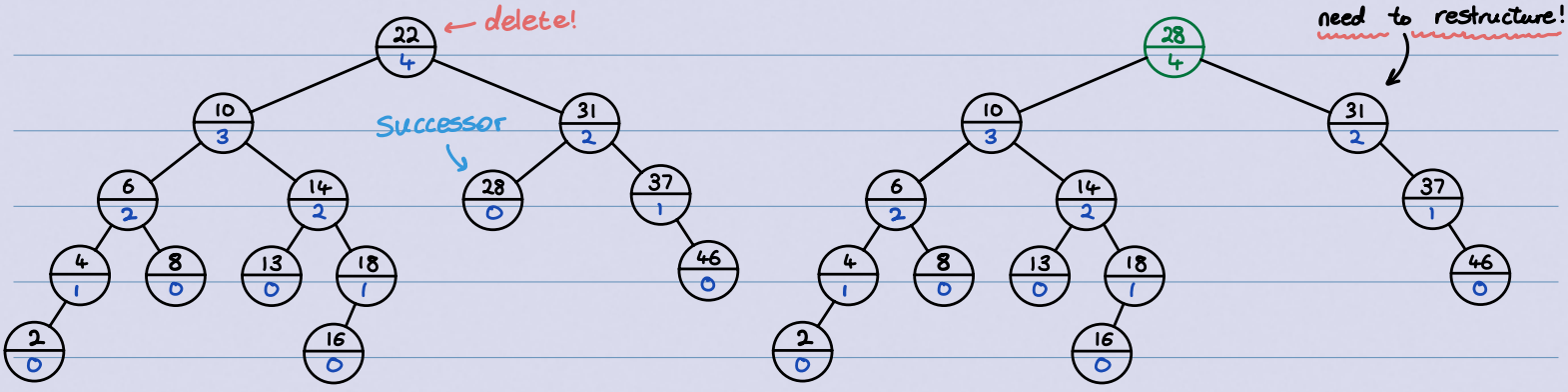
Deletion in AVL Trees

AVL::delete(k) → first, remove the key k with BST::delete. Then, find node where structural change happened (not necessarily near the node that had k!). Go back up to the root, update heights, and rotate if needed.

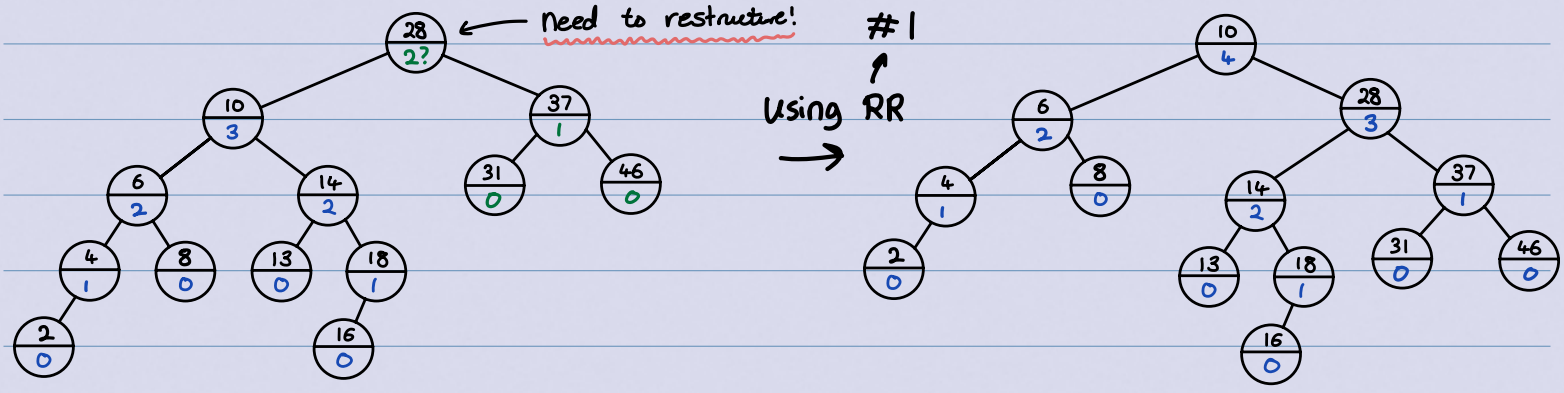
AVL::delete(k)

```
1) z = BST::delete(k)
2) // Assume z is the parent of the BST node that was
      removed
3) while (z is not null) {
4)     if (|z.right.height − z.left.height| > 1) {
5)         c = taller child of z
6)         g = taller child of c   // break ties → avoid double rotation
7)         z = restructure(g, c, z)
8)     }
9)     // always continue up the path
10)    set-height-from-subtrees(z)
11)    z = z.parent
12)}
```

Example: AVL::delete(24):



→ we've deleted the 22 node, but we must now restructure from the 31 node using a left-rotation:



∴, we have deleteded node 22 and restructured accordingly!

- Important: ties must be broken to avoid double rotation while deleting!

AVL Trees - Summary

- Search → just like BSTs, costs $O(height)$
- Insert → BST::insert, the check & update along path to new leaf
  - total cost $O(height)$
  - restructure will be called at most once!
- Delete → BST::delete, then check & update along path to deleted node

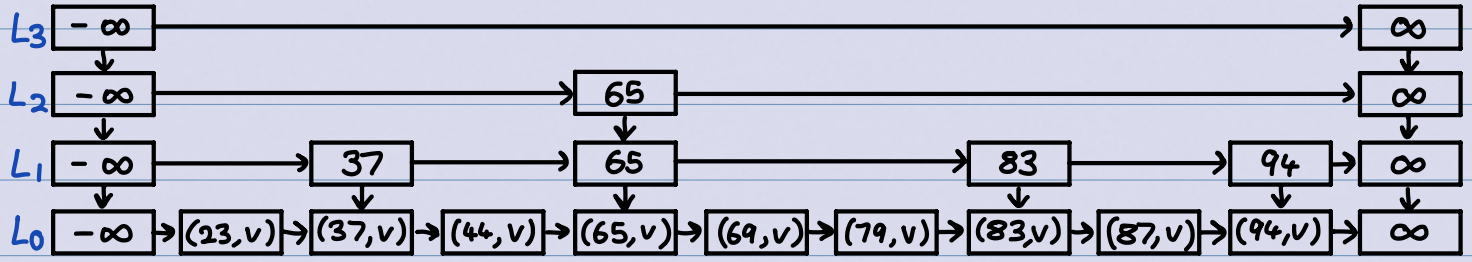
- total cost $\mathcal{O}$(height)
- restructure may be called $\mathcal{O}$(height) times!

- Worst-case for all operations is $\mathcal{O}(\text{height}) = \mathcal{O}(\log n)$

Skip Lists

A heirarchy of ordered linked lists (levels) $L_0, L_1, \dots, L_H$:

- each list L_i contains the special keys $-\infty$ and ∞ (sentinels)
- list L_0 contains the KVPs of S in a non-decreasing order (the other lists store only keys and references)
- each list is a subsequence of the previous one, ie, $L_0 \supseteq L_1 \supseteq \dots \supseteq L_H$
- list L_H contains only the sentinels, all other lists contain at least one non-sentinel.



more definitions:

- node = entry in one list, vs KVP = one non-sentinel entry in L_0
- there are (usually) more nodes than KVPs (above example has 14 non-sentinal nodes and 9 KVPs)
- root = topmost left sentinel is the only field of the skip list.
- each node p has references p.after and p.below.
- each key k belongs to a "tower" of nodes.
  - ↳ height of tower k: maximal index i such that $k \in L_i$
  - ↳ height of skip list: maximal index h such that L_H exists

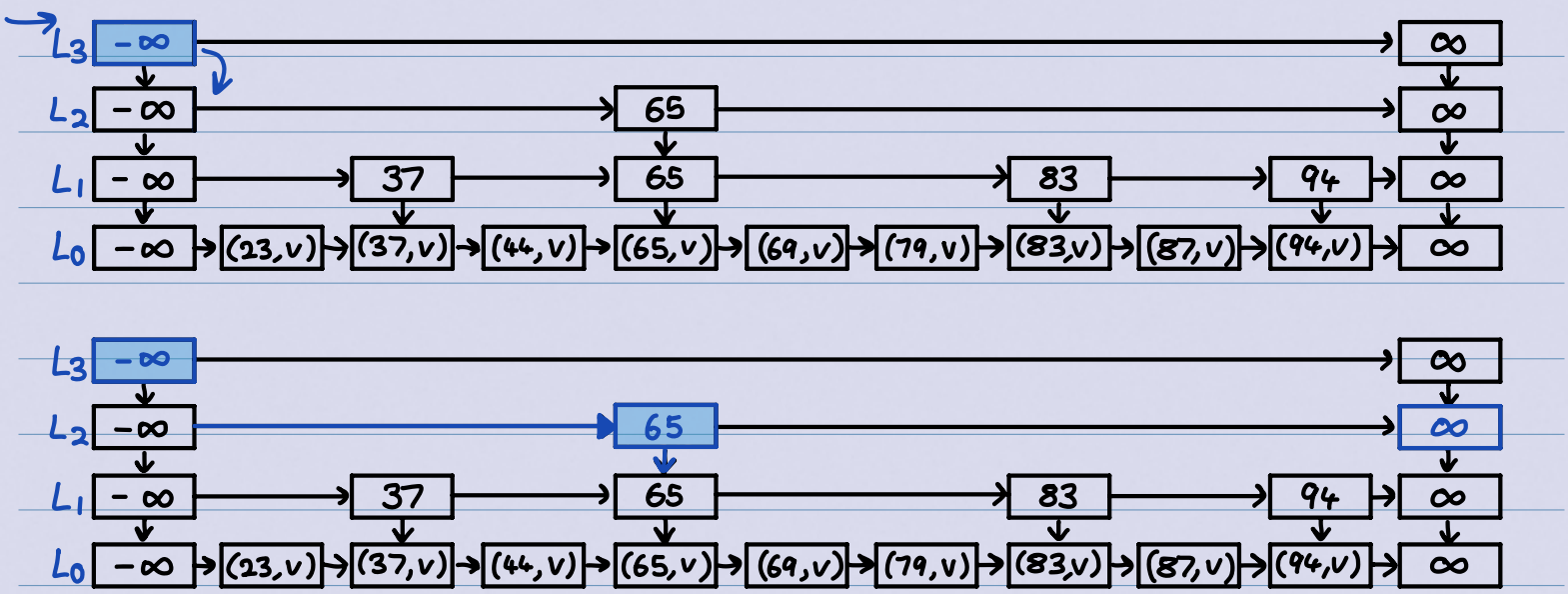
Skip Lists: Search

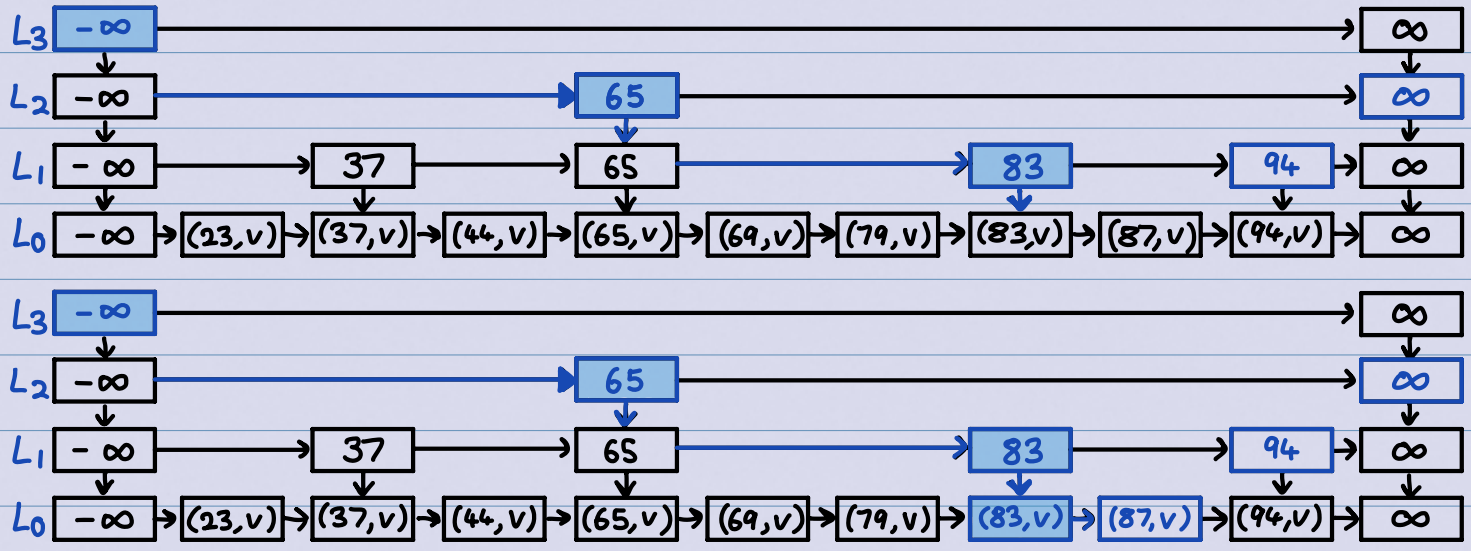
for each list, find predecessor (node before where k would be).

```
get-predecessors(k)
1) p = root
2) P = stack of nodes, initially containing p
3) while (p.below ≠ null) {
4)     p = p.below
5)     while (p.after.key < k) { p = p.after }
6)     P.push(p)
7) }
8) return P
```

```
SkipList::search(k)
1) P = get-predecessors(k)
2) p0 = P.top()   // predecessor of k in L0
3) if (p0.after.key == k) { return KVP at p0.after }
4) else { return "not found, but would be after p0" }
```

Example: SkipList::search(87):





↳ where □ = key compared w/ k

■ = added to P

⟶ = path taken by p

∴ 83 was found in only 7 comparisons!

final stack:
(83, v)
83
65
-∞

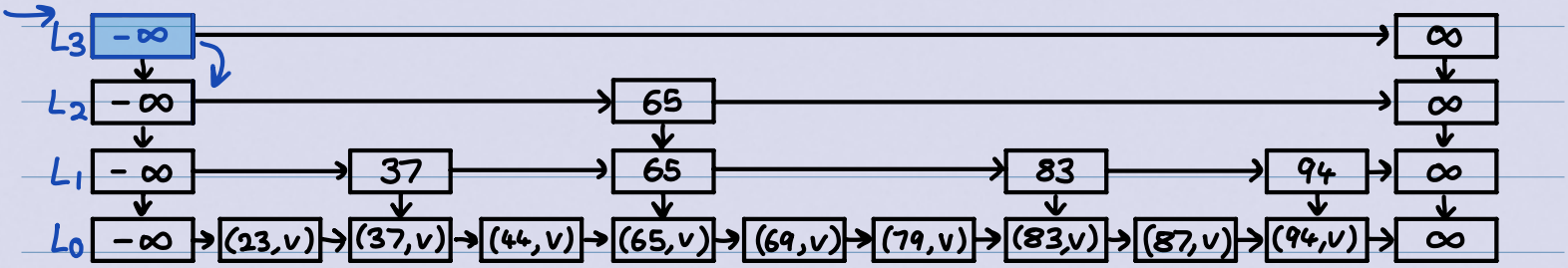
Skip List: Deletion

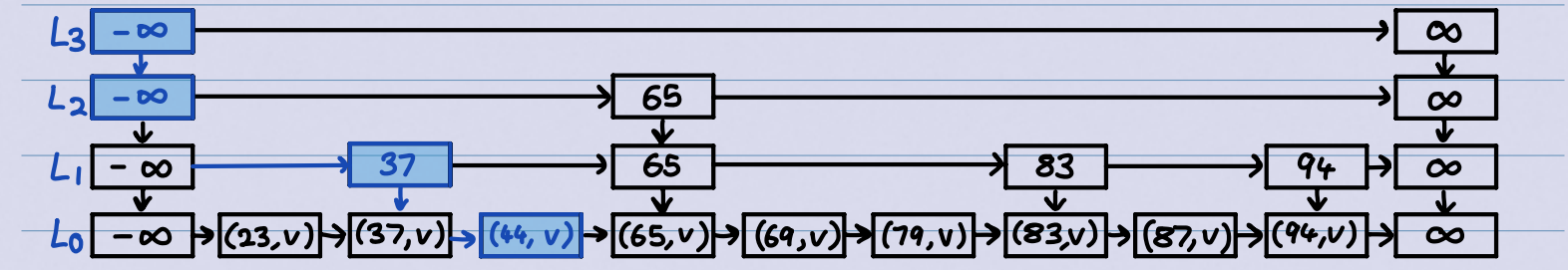
it's easy to remove a key since we can find all predecessors. Then eliminate lists if there are multiple ones will only sentinels.

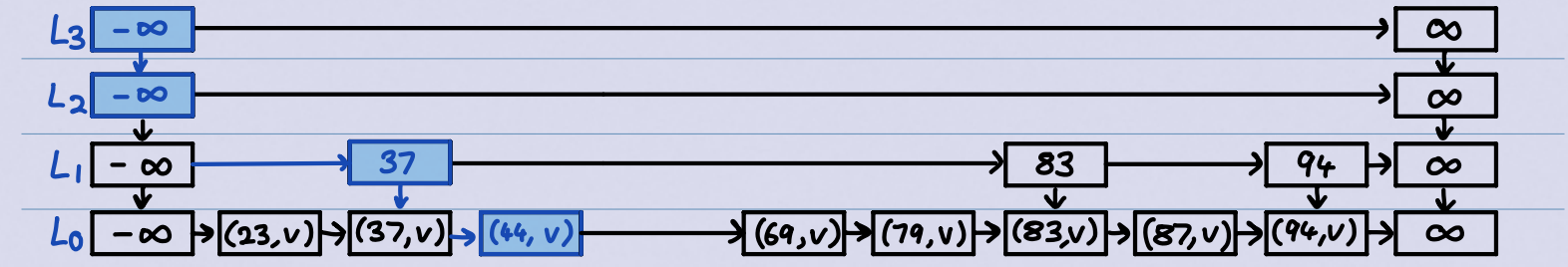
```
skipList:: deletion (k)
1) P = get-predecessors (k)
2) while (P is not empty) {
3)     p = P.pop()  // predecessor of k in some list
4)     if (p.after.key = k) { p.after = p.after.after }
5)     else { break }  // no more copies of k
6) }
7) p = left sentinel of the root list
8) while (p.below.after is the ∞-sentinel) {
       // top 2 lists have only sentinels, remove one
```

```
9)    p.below = p.below.below
10)   p.after.below = p.after.below.below
11) }
```

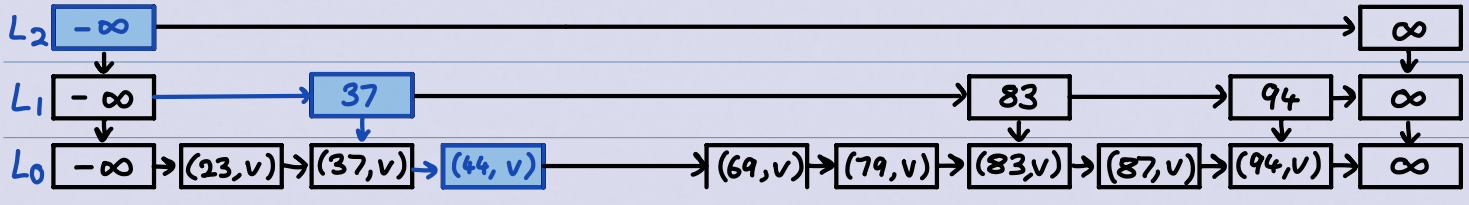
Example: SkipList::delete(65):







but, now we have two lists which are just the sentinels!
so, using the second while loop, we delete one of them:



∴, we have succesfully deleted the node with key = 65!

Skip Lists: Insertion

- there's no choice as to where to put the tower of k

- the only choice is how tall should we make the tower of k
  - ↳ we choose randomly! Toss a coin until you get tails
  - ↳ let i be the number of times the coin came up heads
  - ↳ we want key k to be in lists $L_0, \dots, L_i$, so $i \rightarrow$ height of tower of k
  - ∴, Pr(tower of key k has height i) = $(\frac{1}{2})^i$
- before we can insert, we must check that these lines exist
  - ↳ add sentinel-only lists, if needed, until height h satisfies $h > i$.
- then do the actual insertion
  - ↳ use get-predecessors(k) to get P
  - ↳ the top i items are the predecessors $p_0, \dots, p_i$ where k should be in each list $L_0, L_1, \dots, L_i$
  - ↳ insert (k, v) after p_0 in L_0, and k after p_j in L_j for $1 \leq j \leq i$.

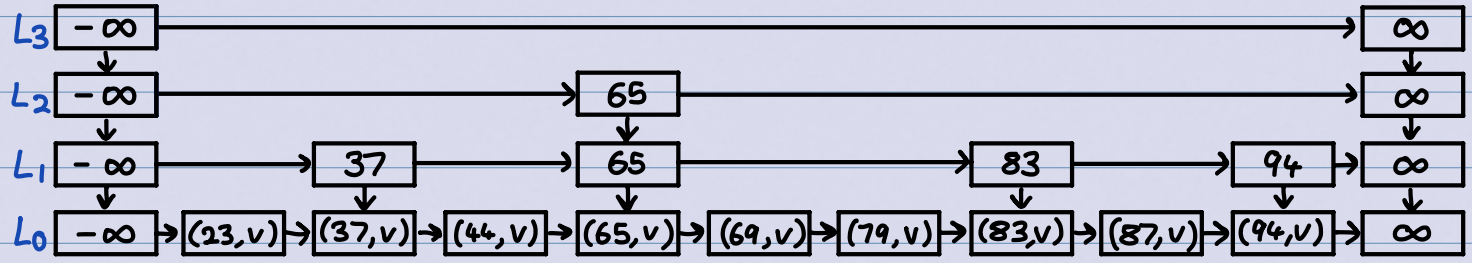
```
SkipList::insert(k, v)
1) for (i=0; random(2)=1; i++) {}   // random tower height
2) for (h=0; p=root.below; p≠null; p=p.below; h++) {}
3) while (i ≥ h) {
4)   create new sentinel-only list; link it in below topmost level
5)   h++
6) }
7) P = get-predecessors(k)
8) p = P.pop()   // insert (k, v) in L0
9) z_below = new node with (k, v)
10) z_below.after = p.after,   p.after = z_below
11) while (i > 0) {   // insert k in L1, ..., Li
12)   p = P.pop()
13)   z = new node with k
```

14) z.after = p.after, p.after = z, z.below = z_{below}, z_{below} = z

15) i = i+1

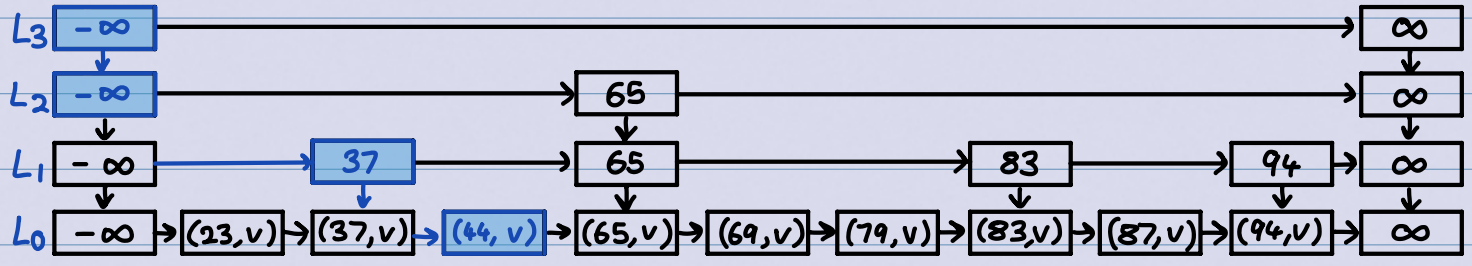
16) }

Example (1): SkipList::insert(52, v). Coin tosses: H, T → i = 1

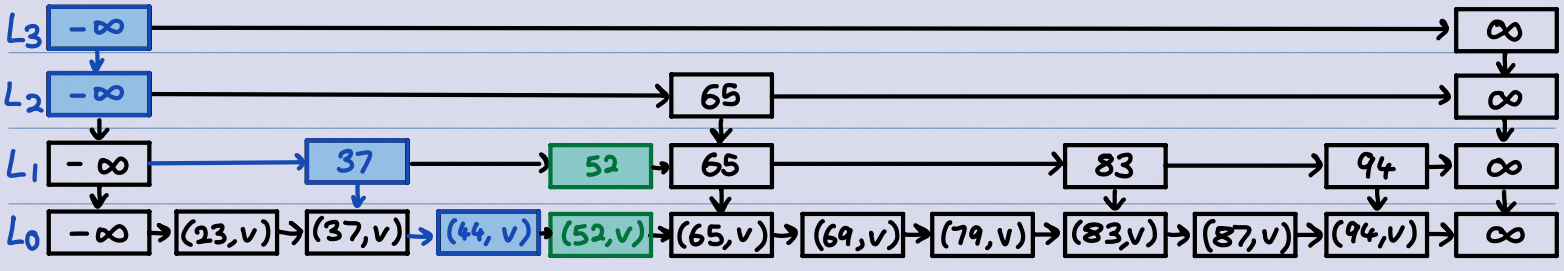


Since h = 3 > i = 1, we don't need to insert any sentinel-only lists!

get-predecessors(52):

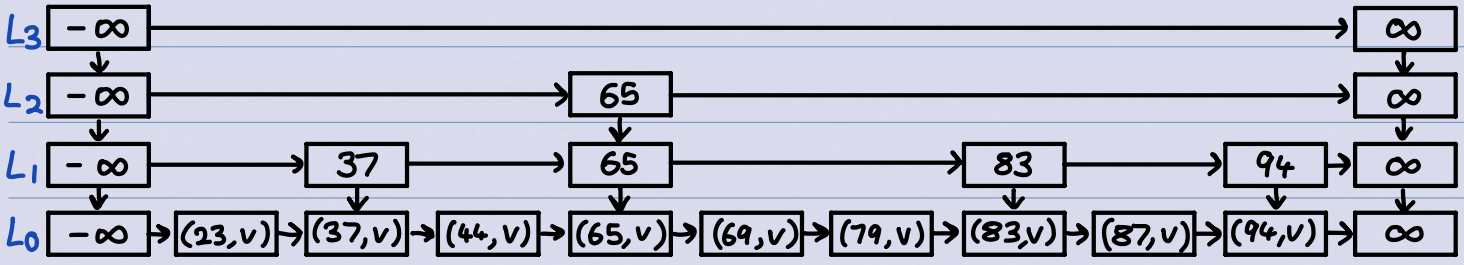


insert 52 in lists $L_0, \ldots, L_i$ (ie, L_0 and L_1):

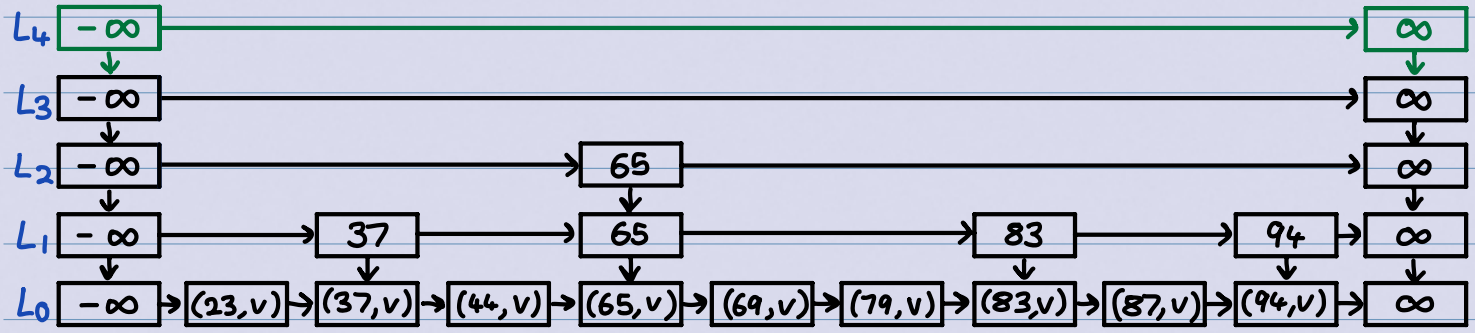


∴, we have succesfully inserted node with key = 52!

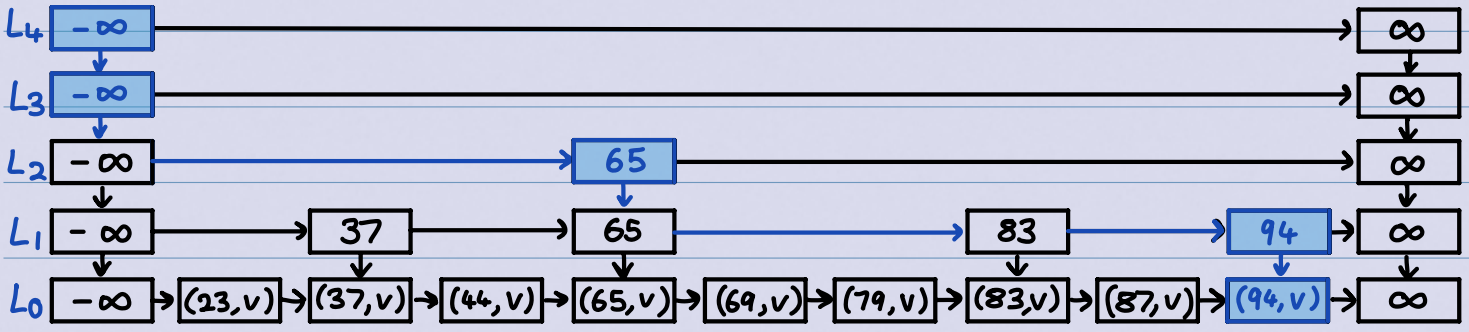
Example (2): SkipList::insert(100, v). Coin tosses: H, H, H, T → i = 3.



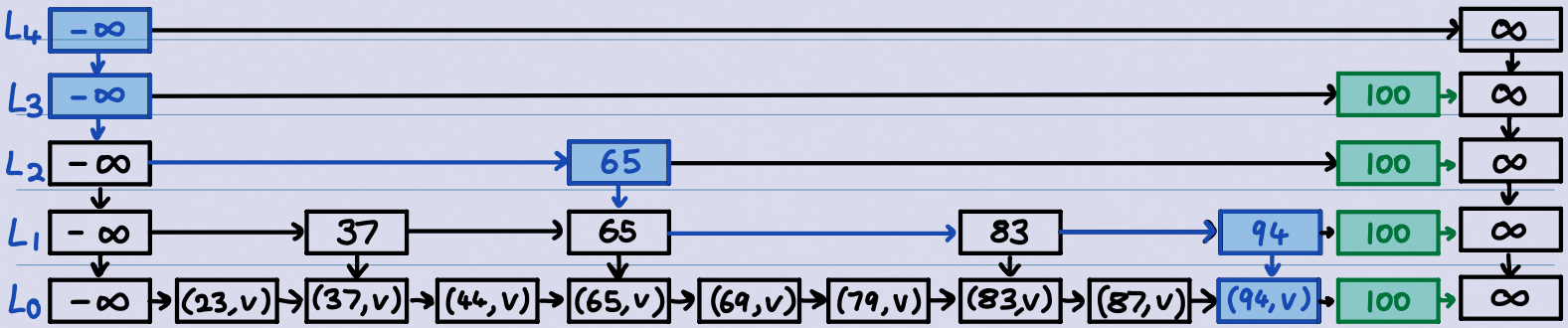
Since $h=3$ is not greater than $i=3$, we must add another level:



now, we call get-predecessors(100):



then, we insert 100 in lists $L_0, \ldots, L_i$



∴, we have succesfully inserted 100 into the skip list!

Skip Lists: Analysis

- expected space: $O(\#\text{non-sentinels} + \text{height})$
  - ↳ expected number of non-sentinels: $O(n)$
  - ↳ expected height: $O(\log n)$
- ↳∴, expected space is $O(n)$

- run-time of operations is dominated by get-predecessors:

↳ how often do we drop down (execute p=p.below)? height.

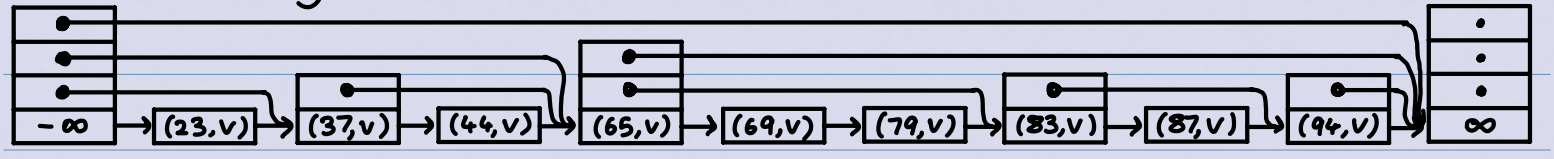
↳ how often do we step forward (execute p=p.after)?

- Expect O(1) forward-steps per list

∴, so search, insert, delete have O(log n) expected run-time.

Skip Lists: Summary

- O(n) expected space, all operations take O(log n) expected time
- Lists make it easy to implement. We can easily add more operations (eg, successor, merge, etc)
- No better than randomized BSTs
- But, we can make improvements on the space
  ↳ can save links (hence space) by implementing towers as array.



Biased Search Requests

So far, we've been assuming all keys to be equal. But, in reality, some keys are accessed more frequently than others. (access: insertion / sucessful search)

- 80/20 rule: 80% of outcomes result from 20% of causes.
- Rule of Temporal Locality: a recently accessed item is likely to be accessed soon again

↳ Intuition says that we should put frequently accessed items near the font (where we first search in the data structure).

Optimal Static Ordering

- Let's say we know the access distribution, and we want the best order of a list.

- access probability of key $k = \frac{\#\text{ accesses of } k}{\text{total } \#\text{ accesses}}$.

- we analyse, for any fixed order of keys, the:
  expected access cost $= \sum_{i \geq 1} i \cdot$ (access probability of k at position i).

Example:

key	A	B	C	D	E
# accesses	2	8	1	10	5

the current order has expected access cost:
$2/26 \cdot 1 + 8/26 \cdot 2 + 1/26 \cdot 3 + 10/26 \cdot 4 + 5/26 \cdot 5 = 86/26 \approx 3.31$

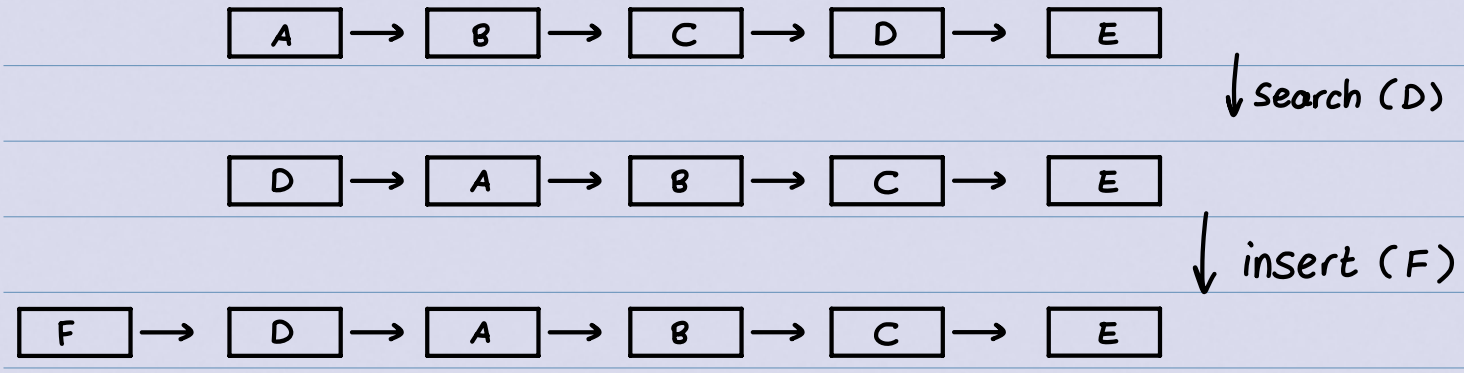
the order [D] → [B] → [E] → [A] → [C] is better!
$10/26 \cdot 1 + 8/26 \cdot 2 + 5/26 \cdot 3 + 2/26 \cdot 4 + 1/26 \cdot 5 = 66/26 \approx 2.54$.

Over all possible static orderings, we minimise the expected access cost by non-increasing access-probability.

Dynamic Ordering: MTF

- We usually don't know the access probabilities ahead of time.
- So, we modify the order dynamically (while we're accessing).

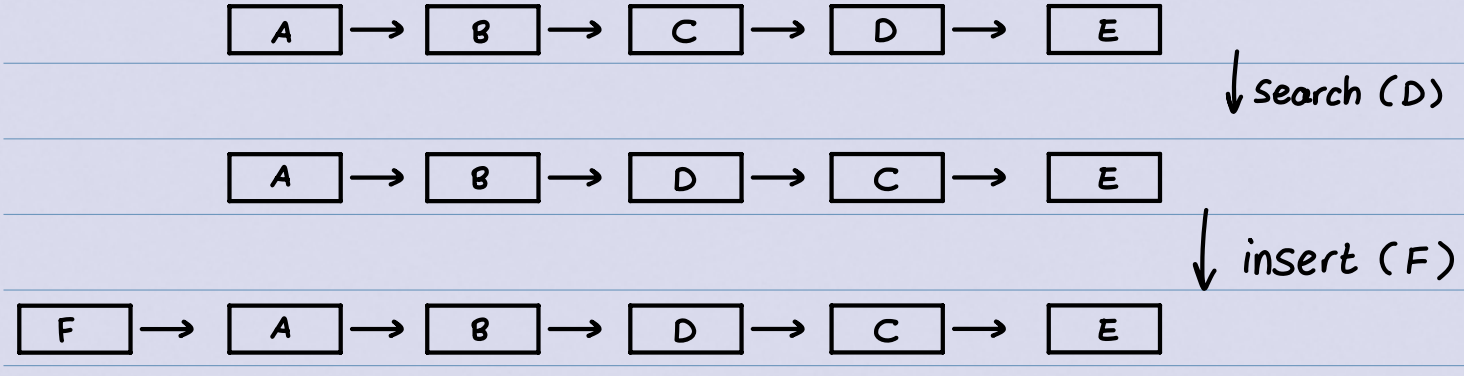
Move-To-Front Heuristic (MTF): upon a successful search, move the accessed item to the front of the list.



We can also do MTF on an array, but then we should insert/search from the back so that we have room to grow.

There are other heuristics we could use:

- Transpose Heuristic: upon a successful search, swap the accessed item with the item immediately preceeding it.



↳ changes are more gradual than MTF.

- Frequency Count Heuristic: keep counters on how often items were accessed, and sort in non-decreasing order.
  ↳ works well in practice, but requires auxiliary space.

- We're unlikely to know the access-probabilities of items, so optimal static order is mostly of theoretical interest.
- For any dynamic reordering heuristic, some sequence will defeat it (have $O(n)$ access-cost for each item).

- MTF and Frequency-Count work well in practice.

Dictionaries for Special Keys (Module 6)

Lower Bound

Can we do better than $O(\log n)$ time for search?

- No: comparison-based searching lower bound is $\Omega(\log n)$.
- Yes: non-comparison-based searching can achieve $o(\log n)$ (under certain conditions!)

Theorem: any **comparison-based algorithm** requires in the **worst-case $\Omega(\log n)$ comparisons to search** among n distinct items.

Interpolation Search

we can match the lower bound asymptotically in a sorted array:

```
binary-search (A, n, k)
1. l = 0,  r = n-1
2. while (l ≤ r) {
3.    m = ⌊(l+r)/2⌋
4.    if (A[m] == k) { return "found at A[m]" }
5.    else if (A[m] < k) { l = m+1 }
6.    else { r = m-1 }
7. }
8. return "not found, but would be between A[l-1] and A[l]"
```

Interpolation search is very similar to binary search, but

we compare at index $l + \left\lceil \frac{k - A[l]}{A[r] - A[l]} \cdot (r - l - 1) \right\rceil$.

- $k - A[l]$ → distance from left key
- $A[r] - A[l]$ → distance between left and right keys
- $(r - l - 1)$ → # unknown keys in range

```
interpolation-search(A, n = A.size(), k)
1. l = 0, r = n-1
2. while (l ≤ r) {
3.   if (k < A[l] or k > A[r]) { return "not found" }
4.   if (k = A[r]) { return "found at A[r]" }
5.   m = l - ⌈(k - A[l]) / (A[r] - A[l]) · (r - l - 1)⌉
6.   if (A[m] == k) { return "found at A[m]" }
7.   else if (A[m] < k) { l = m + 1 }
8.   else { r = m - 1 }
9. }
```

Interpolation Search Example:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	10	20	30	40	50	71	110	112	114	116	118	119	120

interpolation-search(A[0, ..., 13], 14, 71):

- $l = 0$, $r = n - 1 = 13$, $m = l + \left\lceil \frac{71 - 0}{120 - 0} (13 - 0 - 1) \right\rceil = l + 8 = 8$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	10	20	30	40	50	71	110	112	114	116	118	119	120
l								↑					r

Since $k = 71 < 112$, we repeat w/ $r = m - 1 = 7$

- $l = 0$, $r = m - 1 = 7$, $m = l + \left\lceil \frac{71 - 0}{110 - 0} (7 - 0 - 1) \right\rceil = l + 4 = 4$

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	10	20	30	40	50	71	110	112	114	116	118	119	120

l ↑ r

Since $k = 71 > 40$, we repeat w/ $l = m+1 = 5$

- $l = m+1 = 5$, $r = 7$, $m = l + \lceil \frac{71-50}{110-50} (7-5-1) \rceil = l + 1 = 6$

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	10	20	30	40	50	71	110	112	114	116	118	119	120

l ↑ r

∴, we found 71 at A[6] via interpolation search!

- For interpolation search, we have $T^{avg}(n) \in O(\log \log n)$

Tries

Words (=strings) are sequences of characters over alphabet Σ.

- Typical Alphabets: {0,1} (bitstrings), ASCII, {C, G, T, A}.
- Stored in an array: w[i] gets ith character (for i = 0, 1, ...).
- Words have end-sentinel $:

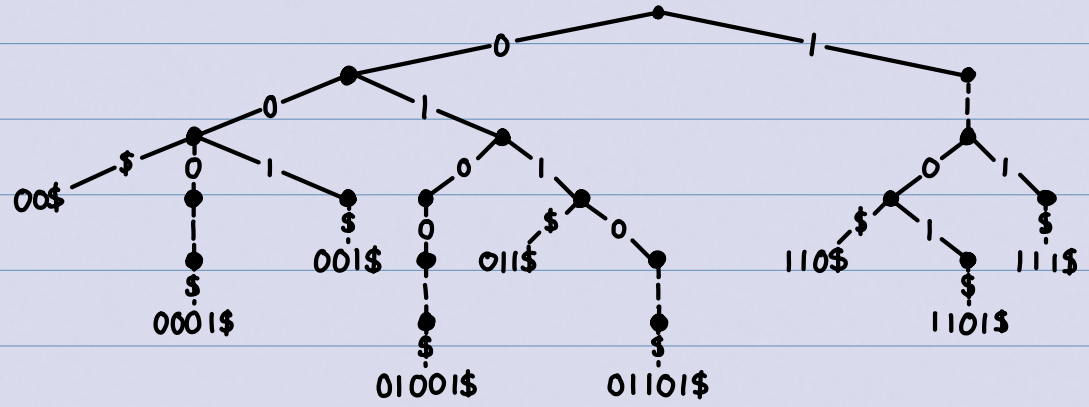
0	1	2	3	4
b	e	a	r	$

- w.size = |w| = # non-sentinel characters (|bear$| = 4)
- Sort words lexicographically: be$ $<_{lex}$ bear$ $<_{lex}$ beer$

Trie (aka, radix tree): A dictionary for bitstrings

- A tree of bitwise comparisons: edge labelled with corresponding bit.
- Similar to radix-sort: use individual bits, not the whole key
- Due to end-sentinels, all key-value pairs are at leaves.
- Note: comes from "retrieval", but pronounced "try"

Example Trie:



Tries: Search

- Follow links that correspond to current bits in w, keeping track of current depth d
- Repeat until no such link or w found at a leaf
- Similar for skip lists, we find search-path P first.

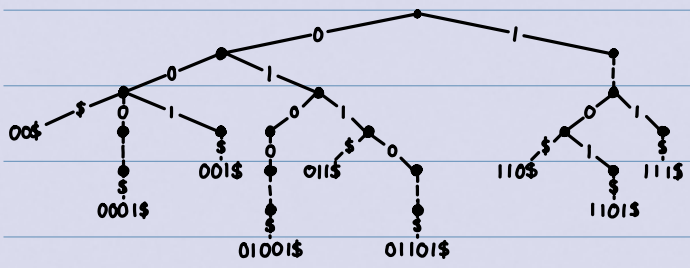
Trie::get-path-to(w)

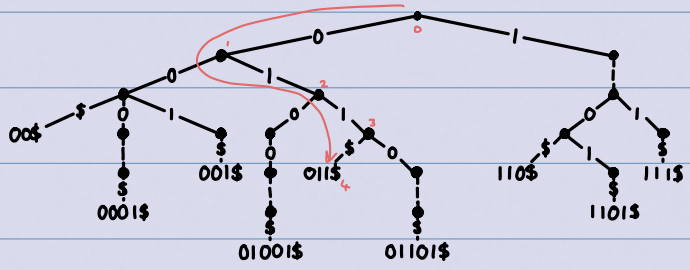
```
//Output: stack with all ancestors of where w would be
1. P= empty stack, z= root, d= 0, P.push(z)
2. while (d ≤ |w|) {
3.     if (z has a child-link labelled with w[d]) {
4.         z= child at this link, d++, P.push(z)
5.     }
6.     else { break }
7. }
8. return P
```

Trie::Search(w)

```
1. P= get-path-to(w), z= P.top
2. if (z is not a leaf) { return "not found, but would be in subtrie of z" }
3. return key-value pair at z
```

Tries search example: Trie::search(011$):

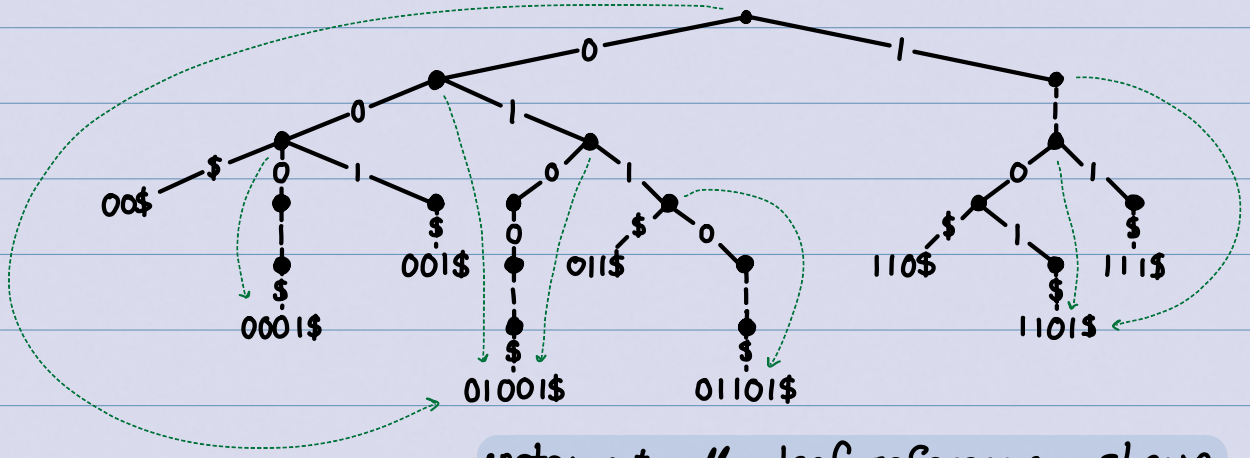




Tries: prefix search

- prefix-search(w): find extension (word for which w is a prefix)
- to find extensions quickly, we need leaf-references
  - ↳ every node z stores reference z.leaf to a leaf in subtree
  - ↳ convention: Store leaf with longest word

example of trie with leaf-references:

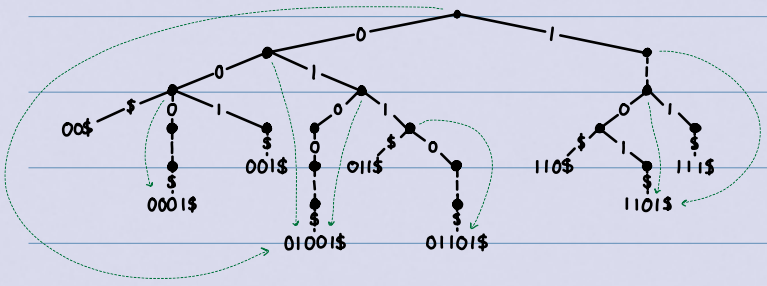


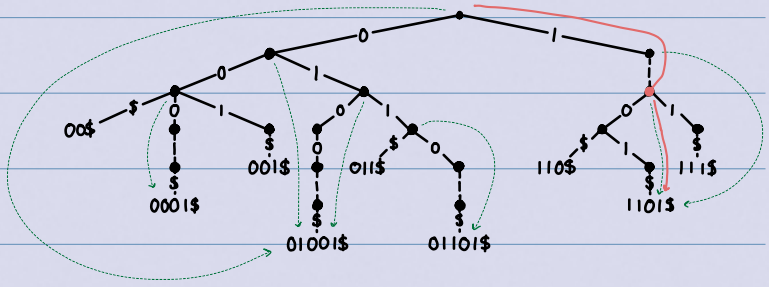
- note: not all leaf-references shown

Trie::prefix-search(w)

```
1. P= get-path-to(w[0, ..., |w|-1]) // ignore end-sentinel!
2. if (#nodes on P is at most |w|) {
3.     return "no extension of w found"
4. }
5. return P.top().leaf
```

Tries prefix-search example: Trie::prefix-search(11$):

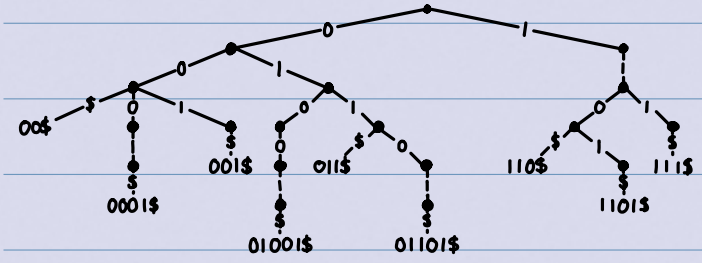


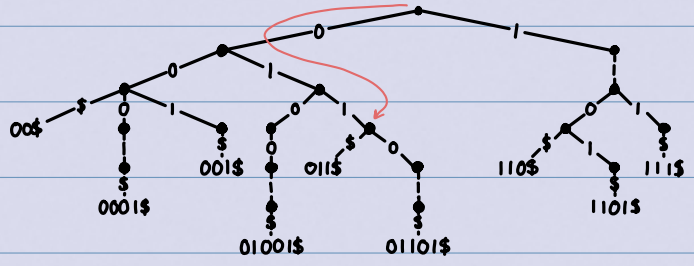


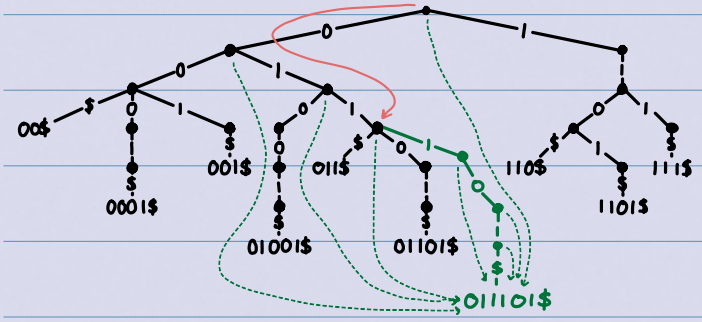
Tries:: Insert

- P = get-path-to(w) gives ancestors that exist already
- Expand the trie from P.top() by adding necessary nodes that correspond to extra bits of w.
- Update leaf-references (also at P if w is longer than previous leaves).

Example: Trie:: insert(01110 1$):







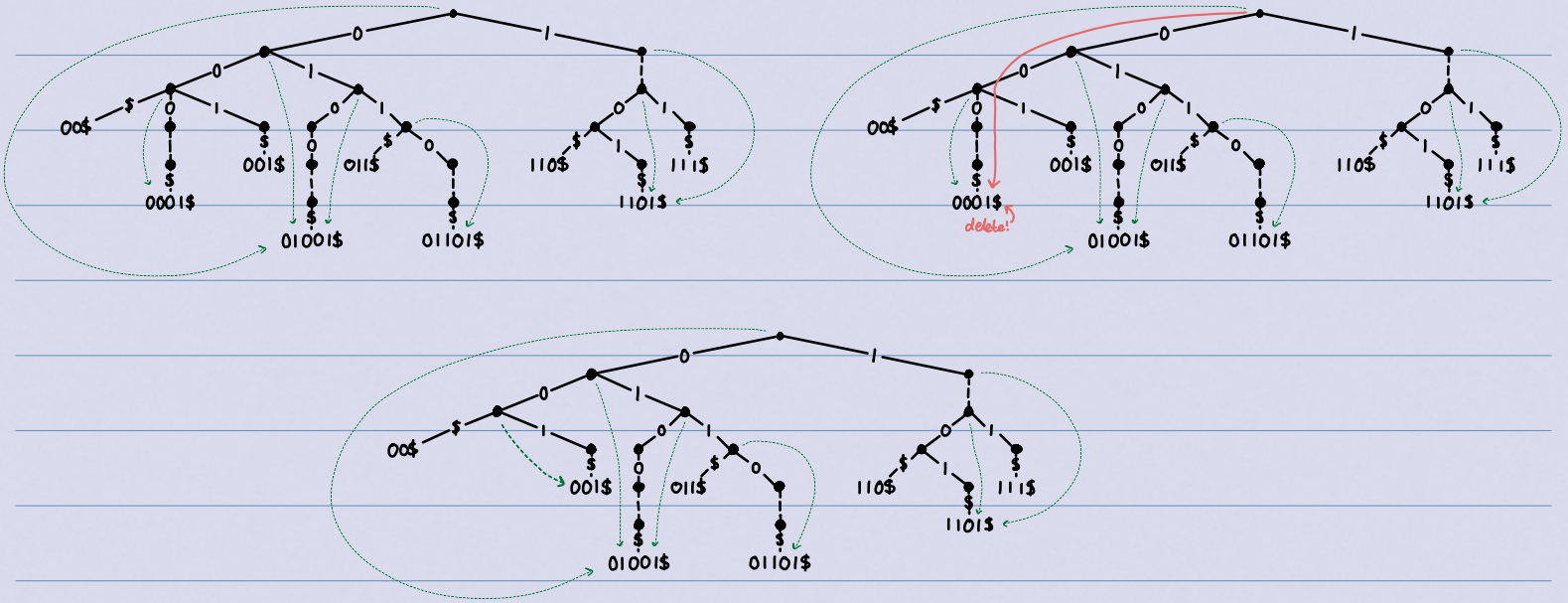
note: only updated leaf-references are shown!

Tries: Delete

- P = get-path-to(w) gives all ancestors
- Let l be the leaf where w is stored
- Delete l and nodes on P until ancestor that had two or more children

- Update leaf-references on rest of P
  (if z ∈ P referred to l, find new z.leaf from other children)

Example: Trie::delete(0001$):



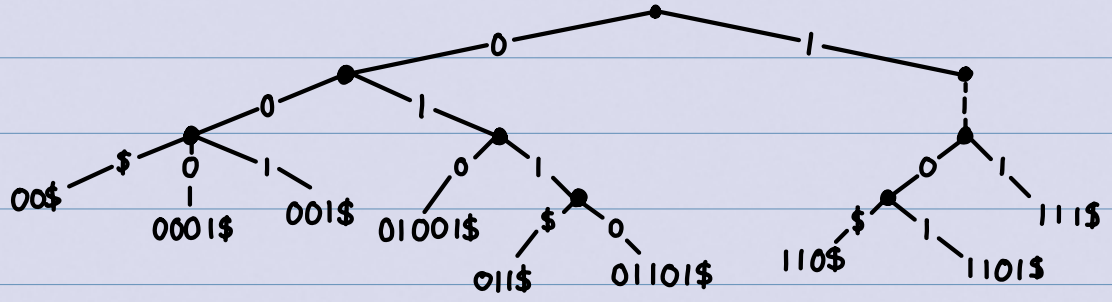
Binary Tries: Summary

- Search(w), prefix-search(w), insert(w), and delete(w) all take $O(|w|)$ time.
- Search time is independent of number of words stored in the trie!
  - ↳ ∴, search time is fast for small words.
- The trie for a given set of words is unique
  - ↳ except for order of children and ties among leaf-references
- But, tries can be wasteful with respect to space
  - ↳ worst-case space is $O(n \cdot$ maximum length of a word$)$
  - ↳ what can we do to save space?

↙

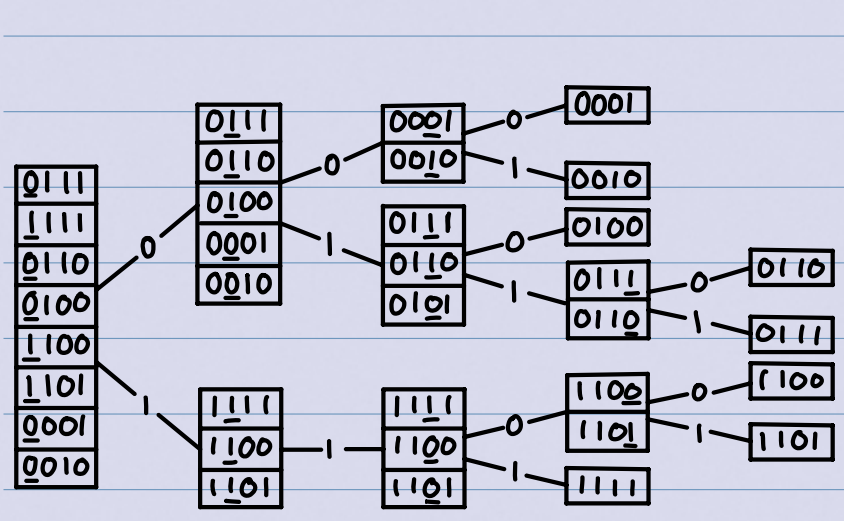
Pruned Tries: Stop adding nodes to trie as soon as the key is unique

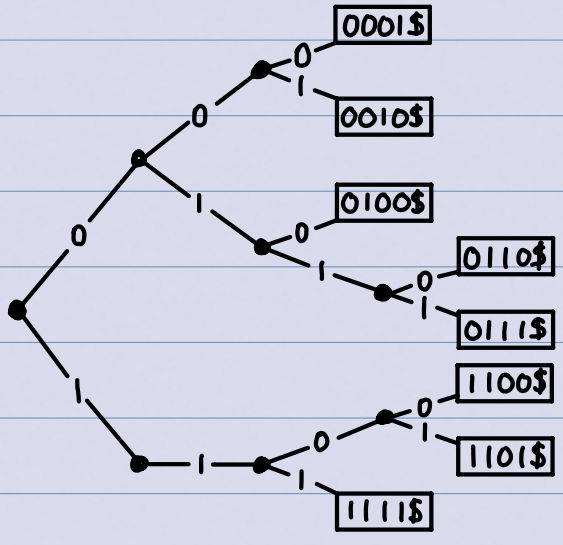
↳ saves space if there are only a few bitstrings that are long!



↳ space can still be bad, but better than regular tries!

NOTE: we have seen pruned trees before! For equal length bitstrings, pruned tree = recursion tree of MSD-radix-sort!



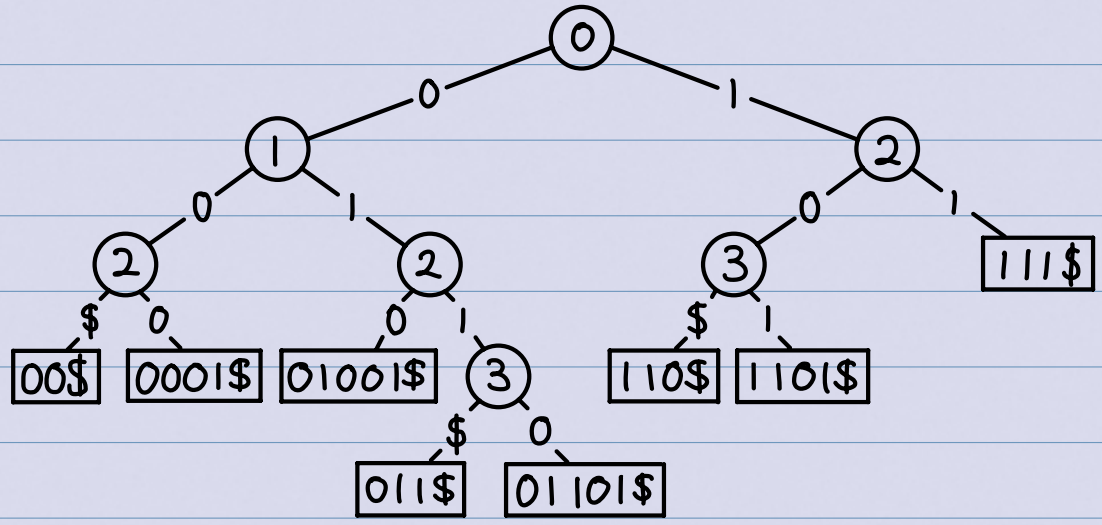


Compressed Tries

Another (important) variation:

- Compress paths of nodes with only one child.
- Each node stores an index, corresponding to the level of the node in the uncompressed trie.
  (on level d, we searched for link with $w[d]$).

Example Compressed Trie:



- **Invariant:** At any node z where $d = z.index$, all words in the subtree rooted at z have the same initial d bits.

- **Observe:** any compressed trie with n words has $O(n)$ nodes!
  - ↳ # nodes = # leaves + # internal nodes
  - ↳ every internal node has 2 or more children.
    - ↳ ∴, more leaves than internal nodes
  - ↳ So, # nodes $\leq 2n-1$.
  - ↳ We use $O(n)$ auxiliary space.

Compressed Tries: Search

- As for tries, follow links that correspond to current bits in w
- Main difference: stored indices say which bits to compare!
- Also: must compare w to word found at leaf.

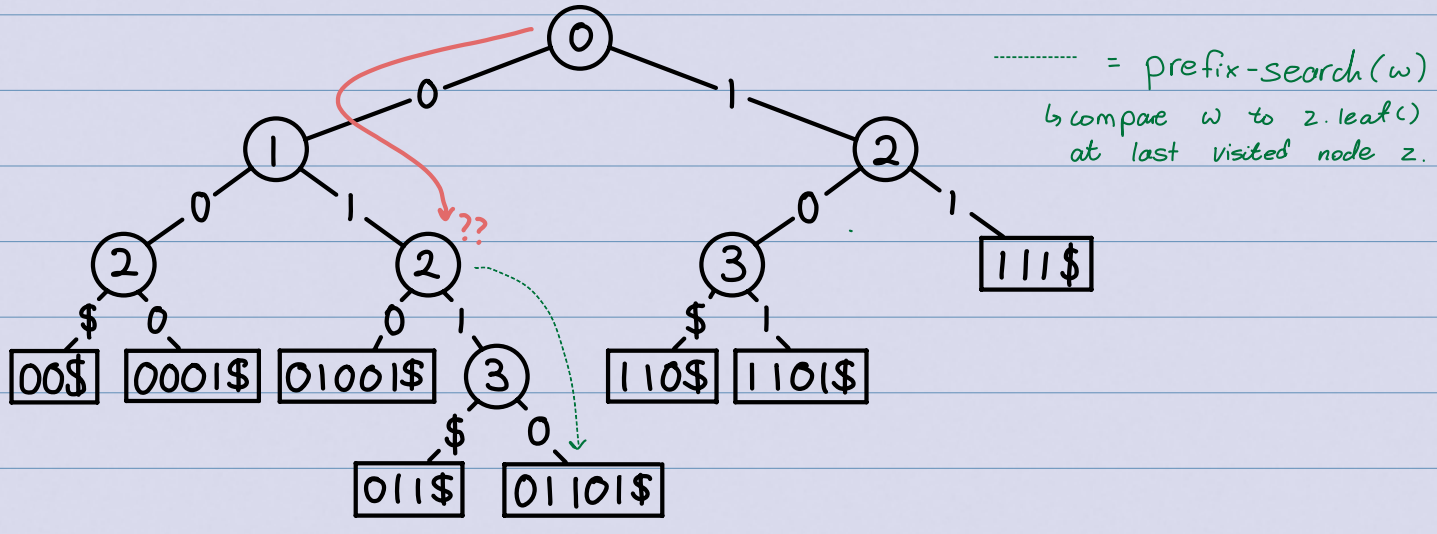
Compressed Trie :: get-path-to (w)

```
1. P= empty-stack(), z= root(), P.push(z);
2. while (z is not a leaf and d= z.index ≤ |w|) {
3.     if (z has a child-link labelled with w[d]) {
4.         z= child at this link;
5.         P.push(z);
6.     }
7.     else { break; }
8. }
9. return P;
```

Compressed Trie :: Search (w)

```
1. P= get-path-to (w); z= P.top();
2. if (z is not a leaf or word stored at z is not w){
3.     return "not found :c";
4. }
5. return key-value pair at z;
```

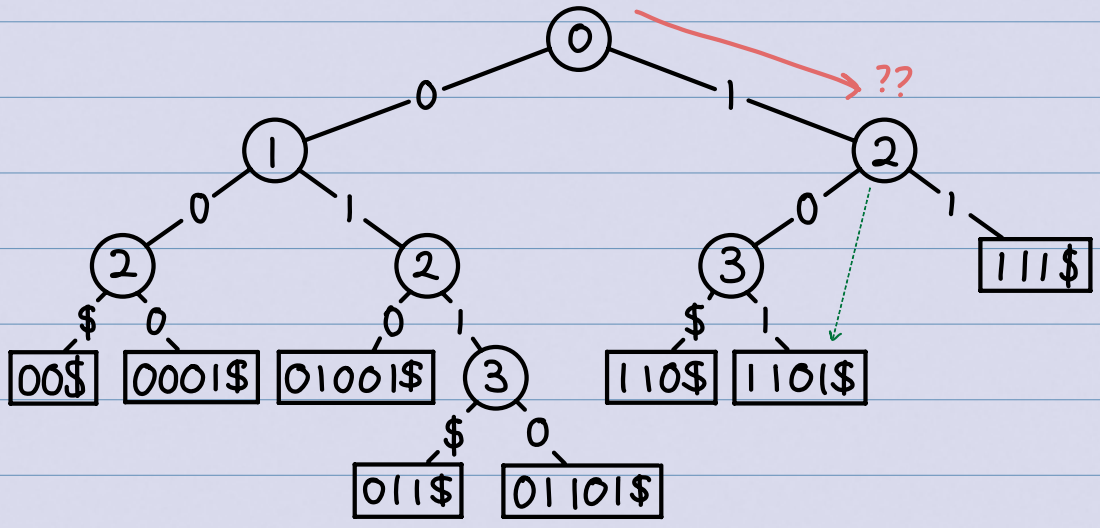
Compressed Tries :: Search (0 1 2 | 0 | 1 | $)



..... = prefix-search(w)
↳ compare w to z.leaf() at last visited node z.

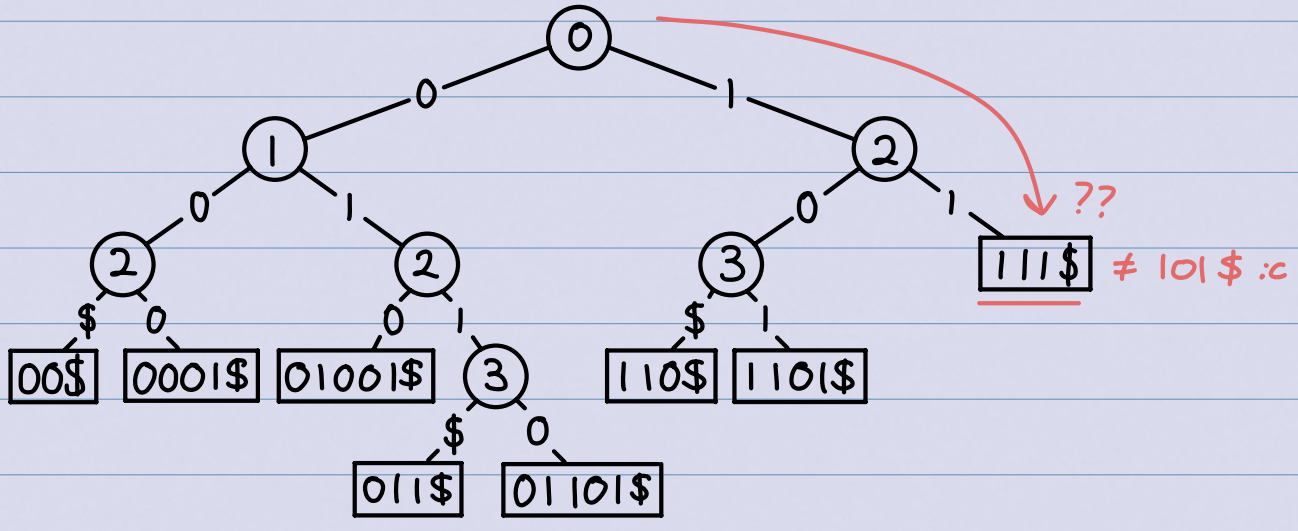
unsuccessful search! (no $-child)

Compressed Tries::Search(0 1 | 1 | $)



unsuccesful! d too big

Compressed Tries::Search(0 1 2 3 | 1 | 0 | 1 | $)



- at root, d=0. w[0]=1, so we go right.
- at right child, d=2. w[2]=1, so we go right.

Search unsuccesful! wrong word at leaf.

Compressed Tries: Summary

- search(w) and prefix-search(w) are fairly easy
- insert(w) and delete(w) are conceptually simple by

Uncompressing:

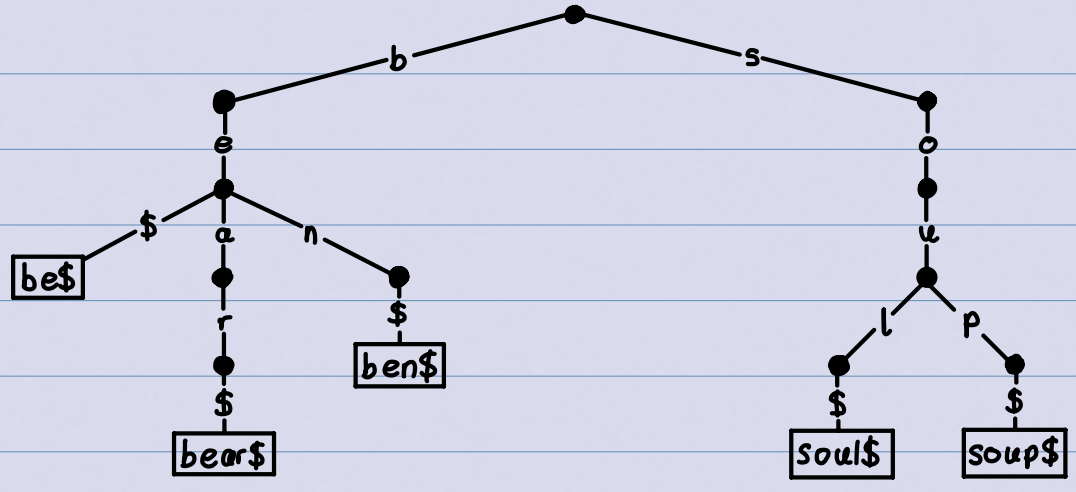
- Search for path P to word w (say we reach node z)
- Uncompress this path (using characters of z.leaf)
- Insert/Delete w as in uncompressed trie
- Compress path from root to where the change happened.

- All operations take $O(|w|)$ time for a word w!
- Compressed tries use $O(n)$ space (better than other trie-variants)
- Overall, code is more complicated, but space-savings are worth it if words are unevenly distributed.

Multiway Tries for Larger Alphabets

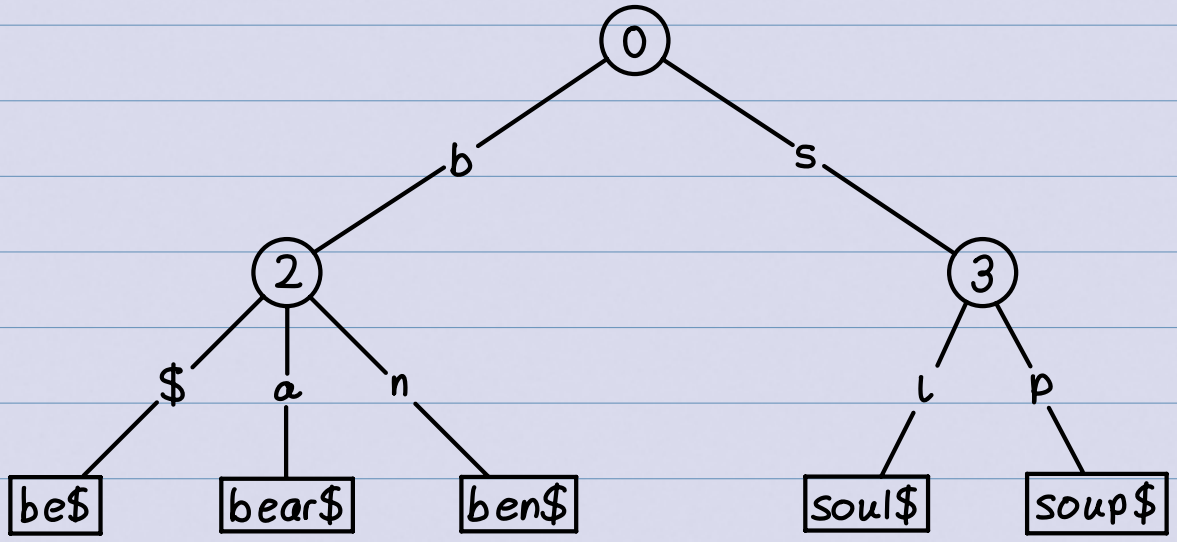
- to represent strings over any fixed alphabet Σ
- any node will have at most $|\Sigma|+1$ children (one child for the $ character)

Example: a trie holding strings {bear$, ben$, be$, soul$, soup$}



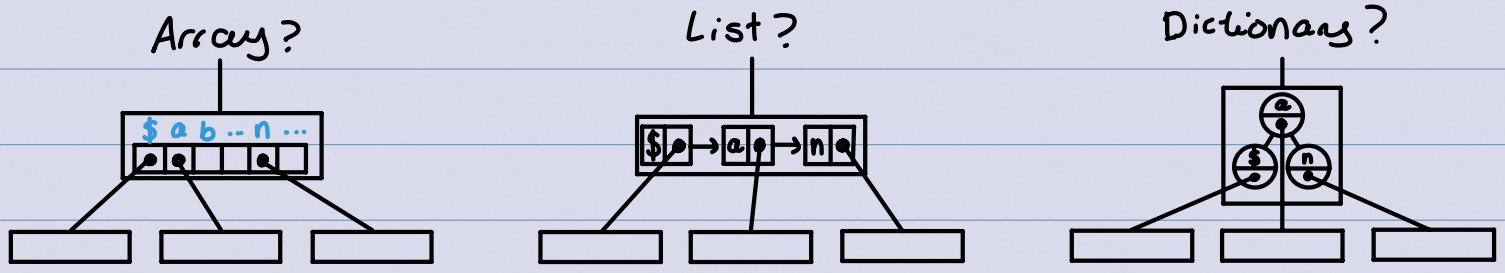
Variation: Compressed multiway tries: compress paths as before!

Example using same Σ:



Multiway Tries: Summary

- Operations search(w), prefix-search(w), insert(w), and delete(w) are exactly as for tries for bitstrings.
- Runtime $O(|w| \cdot$ (time to find appropriate child $))$
- Each node now has $|\Sigma|+1$ children. How should they be stored?



- Time/space tradeoff: arrays are fast, but lists are more space-efficient
- Dictionary is best in theory, but not really worth in practice unless $|\Sigma|$ is huge
- In practice, use direct addressing or hashing!

Hashing

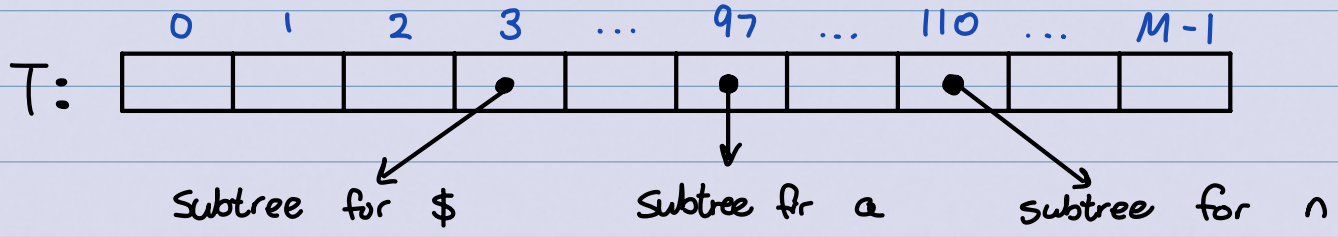
Hashing Introduction:

Direct Addressing

Assume, for some $M \in \mathbb{N}$, every key k is an integer with $0 \leq k < M$.

↳ example: to store child-links in multiway tries, the keys are characters in $ASCII = \{0, \ldots, 127\}$.

we can then implement a dictionary easily: use an array T of size m that stores (k, v) via $T[k] = v$.

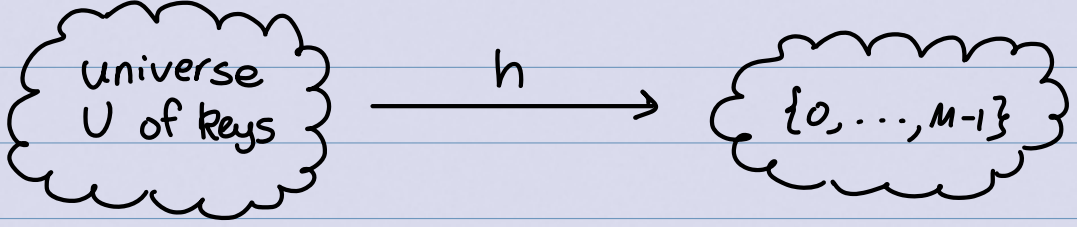


- need $\mathcal{O}(M)$ space, but each operation takes only $\mathcal{O}(1)$ time
  - ↳ Search(k): check whether T[k] is null
  - ↳ insert(k, v): T[k] = v
  - ↳ delete(k): T[k] = null

But, two disadvantages of direct hashing:

1. it cannot be used if the keys are not integers
2. it needs $\mathcal{O}(M)$ space, which is wasteful if M is unknown or if $M >> n$.

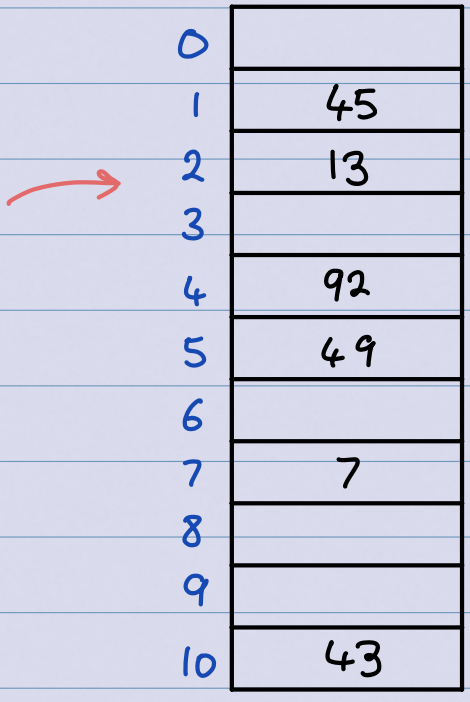
So, Hashing Idea: Map (arbitrary) keys to integers in range $\{0, \ldots, M-1\}$ (for an integer M of our choice), and then use direct hashing.



Assumption: we know that all keys come from some universe U. (Typically, U = non-negative integers, sometimes $|U|$ is finite)

- We pick a **table size** M
- We pick a **hash function** $h: U \to \{0, \ldots, M-1\}$
  - ↳ Commonly used: $h(k) = k \bmod M$.
- **Store dictionary in a hash table** (ie, an array of size M)
- An item with **key** k wants to be **stored in** slot $h(k)$, ie, at $T[h(k)]$.

Hashing: example: $U = \mathbb{N}$, $M = 11$, $h(k) = k \bmod 11$.

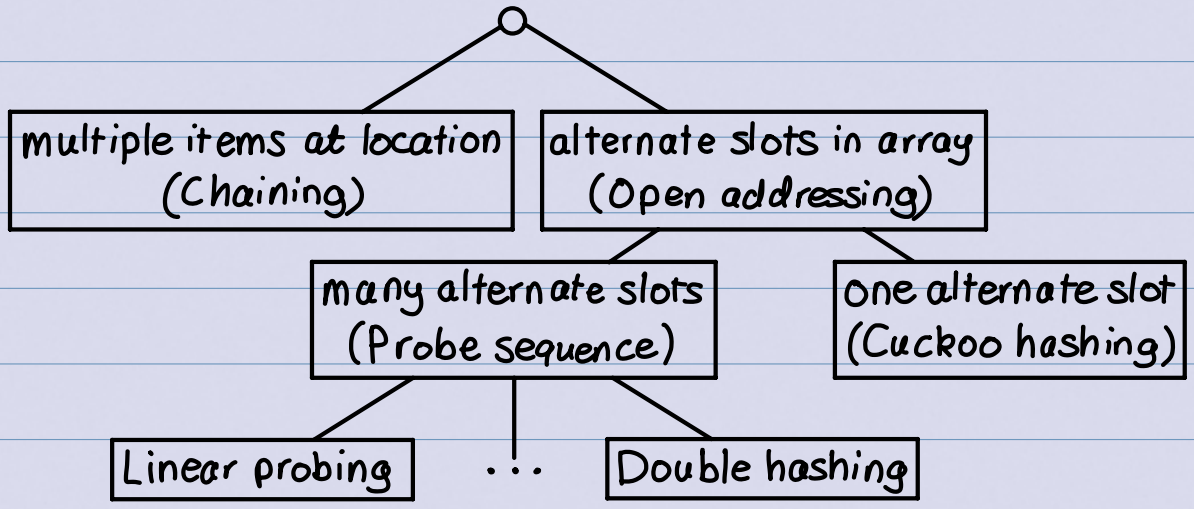


the hash table stores keys
7, 13, 43, 45, 49, 92
(values not shown)

Collisions:

- Generally, hash function k is not injective, so many keys can map to the same integer
  - ↳ For example, $h(46) = 2 = h(13)$ if $h(k) = k \bmod 11$.
- So, we get collisions: we want to insert (k, v) into the table, but $T[h(k)]$ is already occupied.

There are many **strategies to resolving collisions**.



Hashing with Chaining

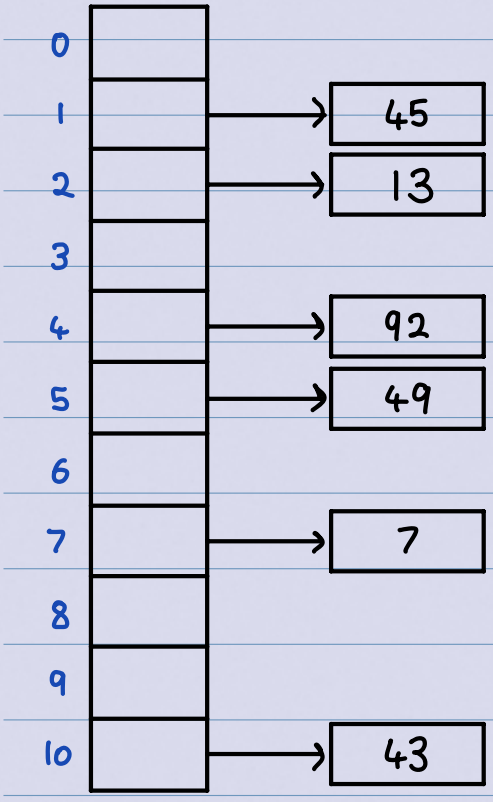
Simplest collision-reduction strategy: **each slot stores a bucket** containing 0 or more KVPs.

- A bucket could be implemented by any dictionary realisation (even another hash table!)
- The simplest approach is to use **unsorted lists with MTF for buckets.** This is called collision resolution by chaining.

- **insert(k, v):** add (k, v) to the front of the list at T[h(k)]
- **search(k):** look for key k in the list at T[h(k)]. Apply MTF-heuristic!
- **delete(k):** perform a search, then delete from the linked list

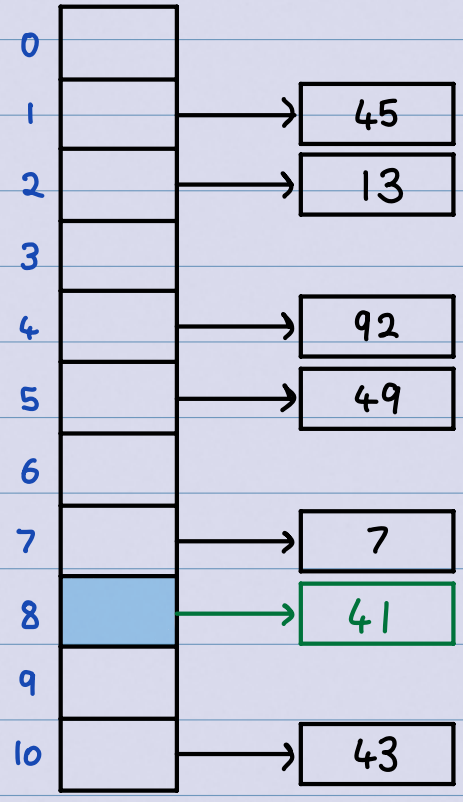
- **insert** takes time **O(1)**, and **search + delete** have runtime **O(1 + length of list at T[h(k)])**.

Hashing with Chaining: Example: $M = 11$, $h(k) = k \bmod 11$.



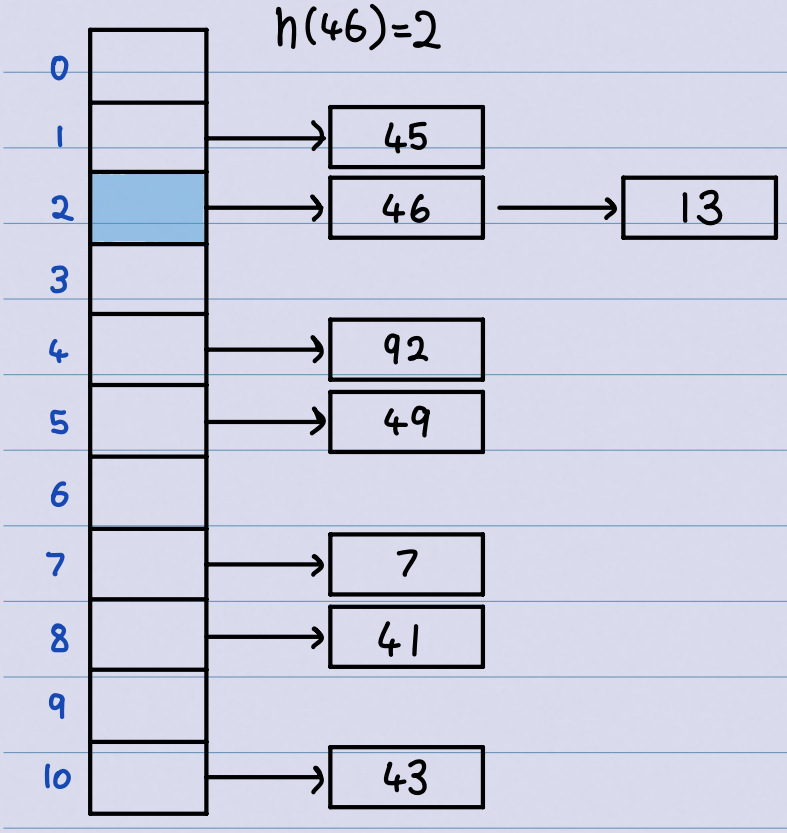
insert(41)

h(41) = 8



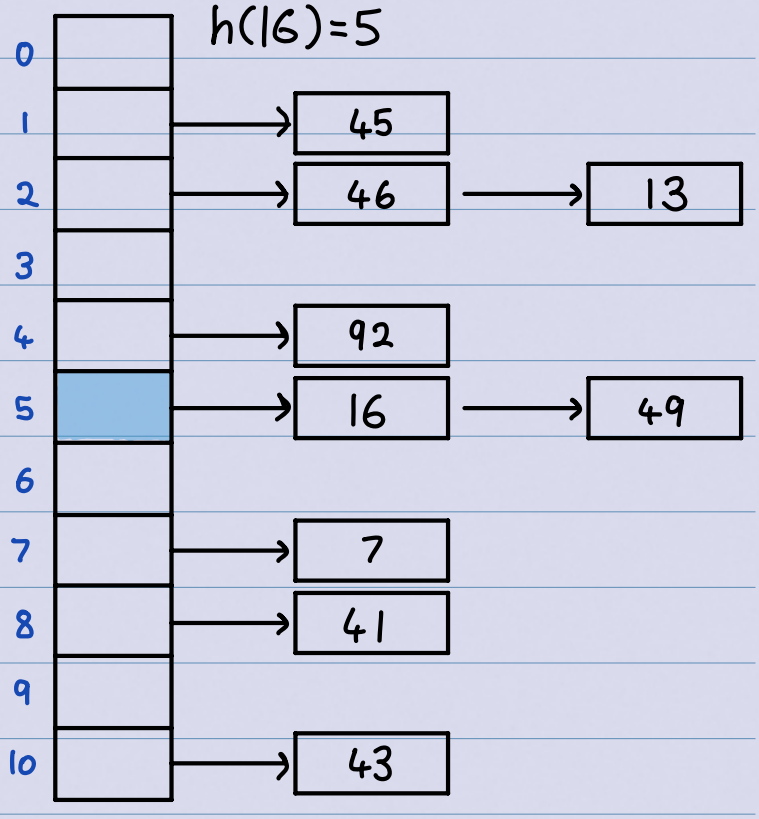
insert(46)

h(46)=2



insert(16)

h(16)=5



Hashing with Chaining: Analysis

- Runtimes:
  - insert takes O(1)

- search / delete take $O(1 + \text{size of bucket at } T[h(k)])$.

- Average bucket size is $\frac{n}{m} := \alpha$ (α is also called the load factor)

- But, this does not imply that the average-case cost of search/delete is $O(1+\alpha)$
  ↳ consider the case where all keys hash to the same slot. Average bucket size is still α, but operations take $O(n)$ on average!
- So, to get meaningful average-case bounds, we need some assumptions on the hash functions and the keys.

↓

The Uniform Hashing Assumption (UHA): Any possible hash function is equally likely to be chosen as hash-function.
↳ this is not at all realistic, but the assumption makes analysis possible.

- To analyse what happens "on average", switch to randomised hashing! We randomise by randomly picking a hash function.

Hashing with Chaining (with UHA) Analysis:
UHA implies that the distribution of keys is unimportant.
- Claim: Hash-values are uniform.
  ↳ $\Pr(h(k) = i) = 1/m$ for any key k and slot i.
- Similar: two keys collide with probability $1/m$.

How big is the bucket of key k?

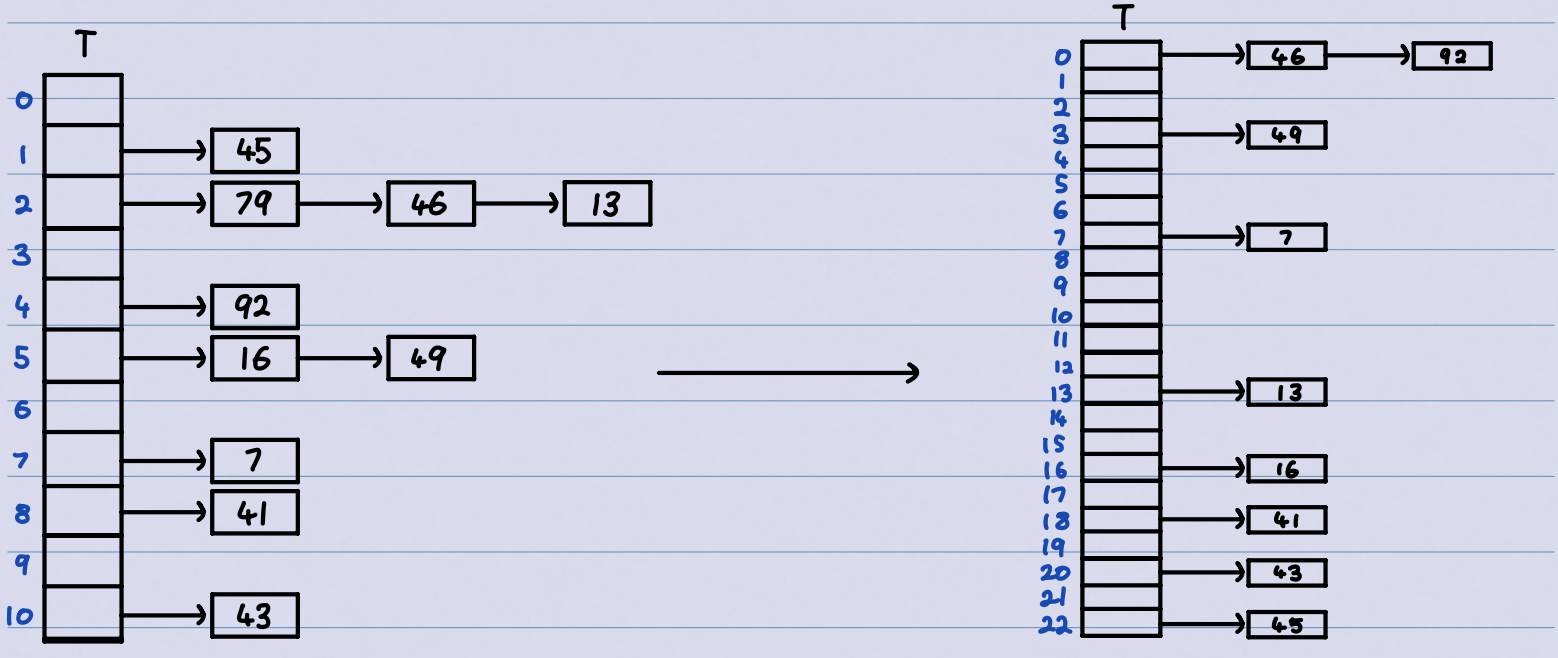
- If key k is not in dictionary
  - n keys hash to this bucket with probability $1/M$ each.
  - So, bucket $T[h(k)]$ has expected length $\alpha = n/M$
- If key k is in dictionary (eg during delete):
  - Key k is definitely in this bucket
  - Each of the other $n-1$ keys collide with probability $1/M$
  - So, bucket $T[h(k)]$ has expected length $1 + \frac{n-1}{M} \leq 1 + \alpha$.

- Therefore, expected cost of search/delete is $O(1+\alpha)$

Load Factor and Re-hashing

For hashing with chaining (and also other collision-reduction strategies), the runtime bound depends on $\alpha = n/M$.

So, we keep the load factor small by rehashing when needed:



- Keep track of n and M throughout operations
- If α gets too large, create new (roughly 2x) hash-table, new hash function(s), and re-insert all items.

Hashing with Chaining: Summary

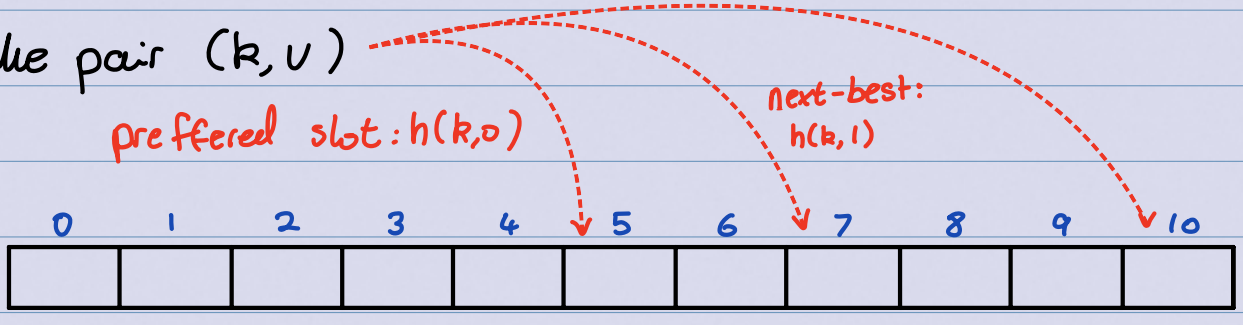
- For hashing with chaining, rehash so that $\alpha \in O(1)$ throughout
- Rehashing costs $O(n+m)$ time (+ time to find a new hash function)
- Rehashing happens rarely enough that we can ignore this term when amortizing over all operations.
- If space is critical: also rehash when α gets too small, so that $M \in O(n)$ throughout, and space is always $O(n)$.

- Summary: The amortized expected cost for hashing with chaining is $O(1)$ and the space is $O(n)$ (Assuming uniform hashing and $\alpha \in O(1)$ throughout).
  ↳ Theoretically perfect, but slow in practice.

Open Addressing

Main Idea: Avoid the links needed for chaining by permitting only one item per slot, but allowing a key k to be in multiple slots.

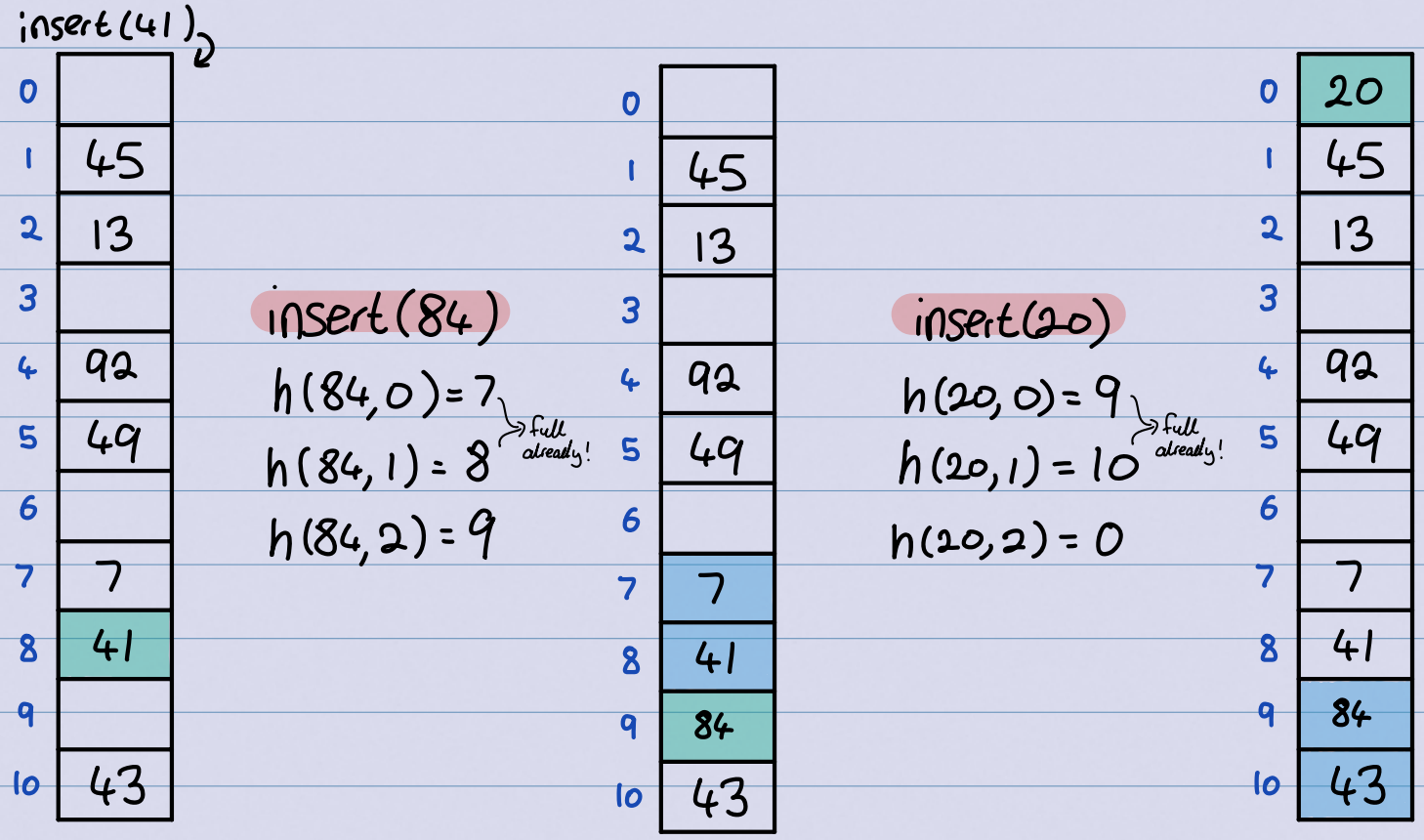
search and insert follow a probe sequence of possible locations for key k: $\langle h(k,0), h(k,1), \ldots, h(k, M-1) \rangle$ until an empty slot is found.



Simplest method for open addressing: linear probing

↳ $h(k,j) = (h(k)+j) \bmod M$, for some hash function h.

Hashing with Linear Probing: $M=11$, $h(k)=k \bmod 11$, $h(k,j)=(h(k)+j) \bmod 11$.



But, delete becomes problematic:

- Cannot leave an empty spot behind: the next search might otherwise not go far enough
- We could try to move later items in probe sequence forward (but it's non-trivial to find one that can be moved)
- Better idea: lazy deletion
  - Mark spot as DELETED (rather than NULL)
  - Search continues past deleted slots
  - Insertion re-uses deleted slots
- Keep track of how many items are DELETED and rehash (to keep space at $O(n)$) if there are too many.

Hashing w/ Linear Probing: $M=11$, $h(k) = k \bmod 11$, $h(k,j) = (h(k)+j) \bmod 11$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	43

delete(43)

$h(43,0) = 10$

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	DELETED

Search(63)

$h(63,0) = 8$

$h(63,1) = 9$

$h(63,2) = 10$

$h(63,3) = 0$

$h(63,4) = 1$

$h(63,5) = 2$

$h(63,6) = 3$

not found!

0	20
1	45
2	13
3	
4	92
5	49
6	
7	7
8	41
9	84
10	DELETED

Hashing w/ Probe Sequences: Operations

```
probe-sequence:: insert(T, (k,v))
1. for (j=0; j< M; j++) {
2.     if (T[h(k)] is NULL or "deleted") {
3.         T[h(k)] = v;
4.         return "success! :) ";
5.     }
6. }
7. return "failed insert: need to rehash!";
```

```
probe-sequence:: search(T, k)
1. for (j=0; j< M; j++) {
2.     if (T[h(k)] is NULL) { return "not found"; }
3.     if (T[h(k)] has key k) { return T[h(k,j)]; }
```

```
4.  // key is incorrect/deleted, so try next probe (i.e., j++)
5. }
6. return "not found";
```

Independent Hash Functions

- Some hashing methods require **2 hash functions, h_0 and h_1.**
- These hash functions **should be independent** in the sense that the random variables **$Pr(h_0(k)=i)$ and $Pr(h_1(k)=j)$ are independent.**
- Using **two modular hash-functions often leads to dependencies!**
- Better idea: use **multiplication method** for second hash function:
  - fix some floating-point number A with $0 < A < 1$

$$h(k) = \lfloor M \cdot (\underbrace{A \cdot k}_{\text{multiply}} - \underbrace{\lfloor A \cdot k \rfloor}_{\text{integral part}}) \rfloor$$

$(A \cdot k - \lfloor A \cdot k \rfloor)$: fractional part, in $[0, 1)$

$\lfloor M \cdot (\text{fractional part}) \rfloor$: integer in $[0, M)$

Our examples will use $\varphi = \frac{\sqrt{5}-1}{2} \approx 0.618$ as A.

Double Hashing

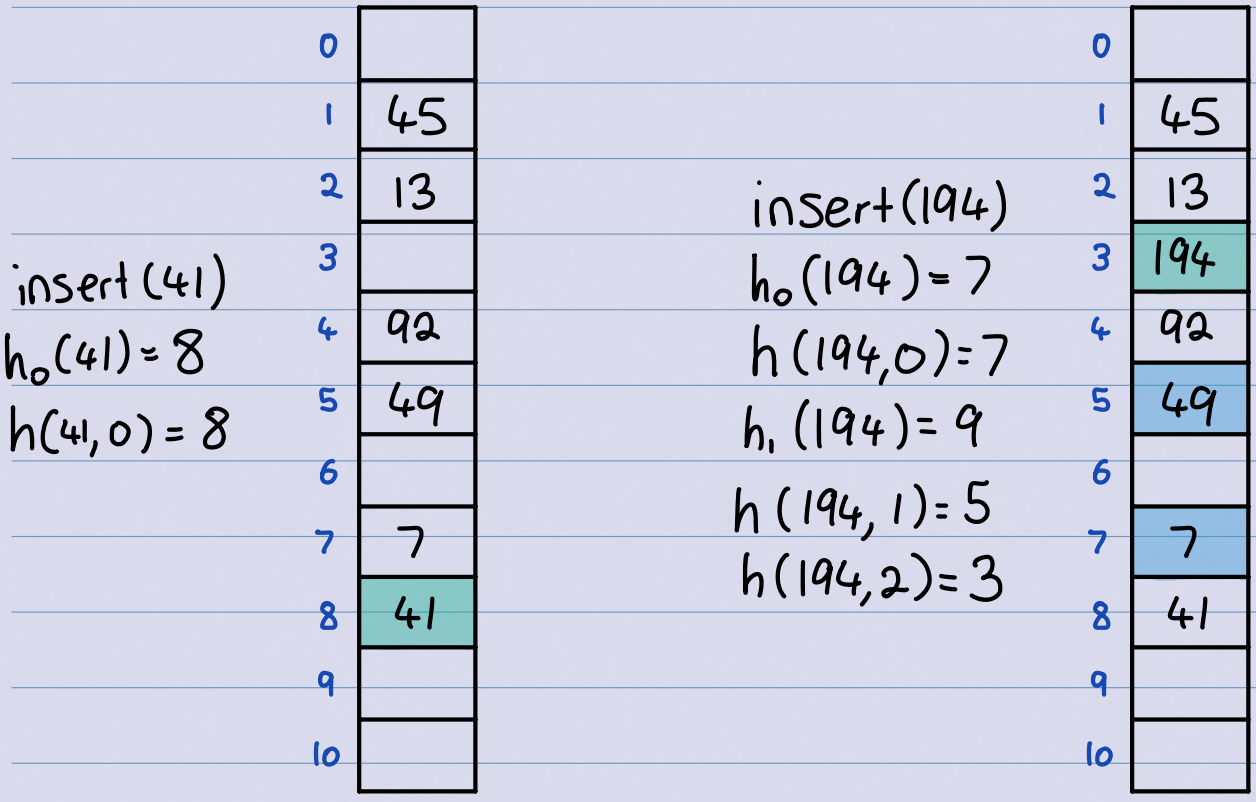
Open addressing with probe sequence $h(k,j) = (h_0(k) + j h_1(k)) \bmod M$, where h_0 and h_1 are independent functions.

We require ($\forall$ keys k):

- $h_1(k) \neq 0$
  - ↳ can modify standard hash functions to ensure this
    - eg, modified multiplication method: $h(k) = 1 + \lfloor (M-1)(kA - \lfloor kA \rfloor) \rfloor$
- $h_0(k)$ is relative prime w/ table size M.

↳ so choose a prime M

Double Hashing: Example, $M=11$, $h_0(k)=k \bmod 11$, $h_1(k)=\lfloor 10(\varphi k - \lfloor \varphi k \rfloor)\rfloor + 1$



Cuckoo Hashing

We use 2 independent hash functions h_0 and h_1, and 2 tables T_0 and T_1.

Main idea: an item with key k can only be at $T_0[h_0(k)]$ or $T_1[h_1(k)]$.

Search and delete then always take constant time!

Cuckoo Hashing: Insertion

insert always initially puts the new item at $T_0[h_0(k)]$

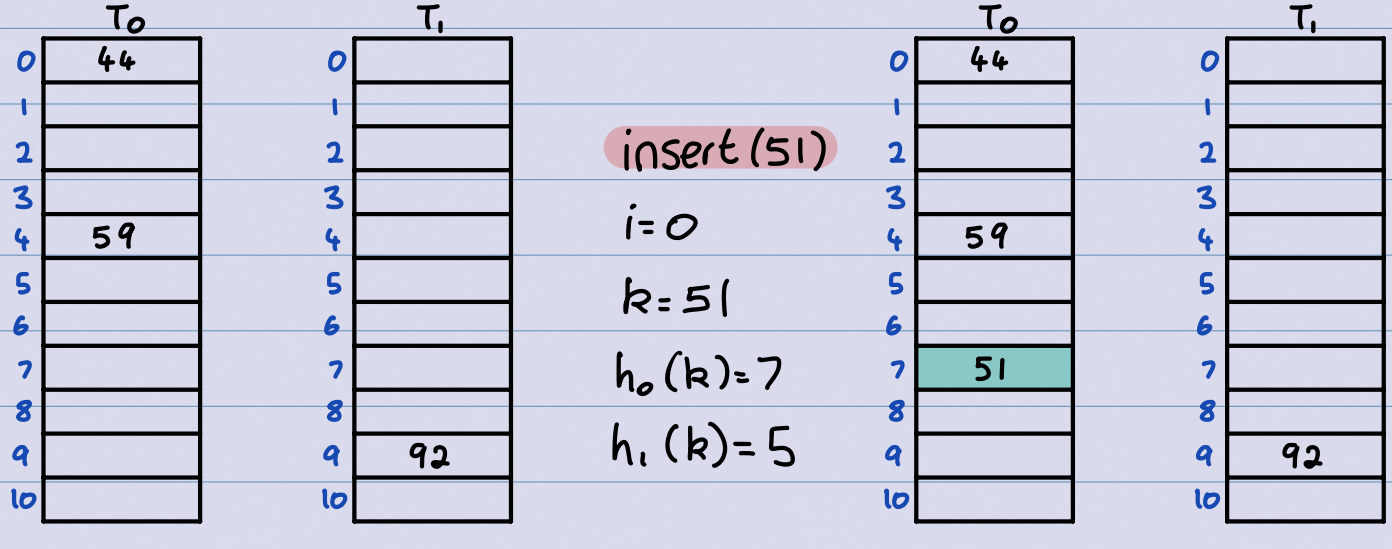
- evict item that may have been there already
- if so, evicted item inserted at alternate position

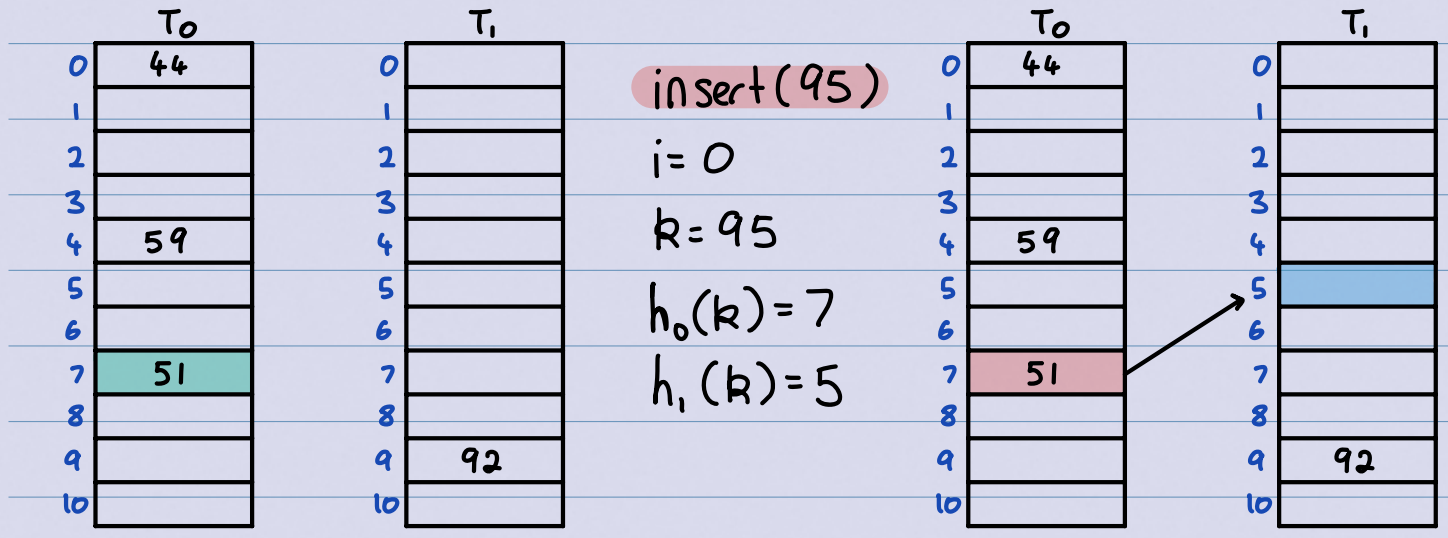
- this may lead to a loop of evictions
  - ↳ can show: if insertion is possible, there's max $2n$ evictions
    so abort after too many attempts

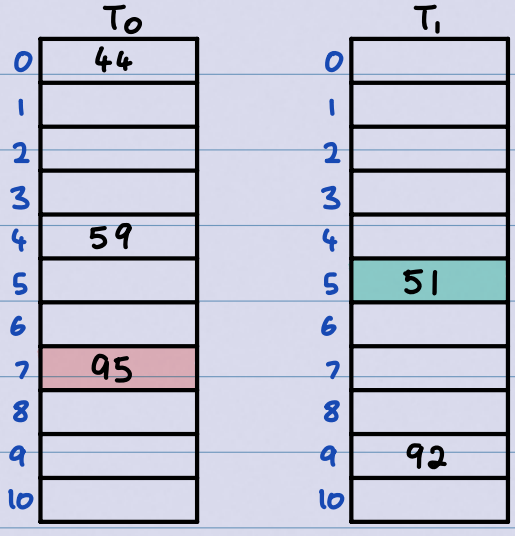
```
Cuckoo::insert(k, v)
1. (k_insert, V_insert) = new KVP with (k, v)
2. i = 0
3. do at most 2n times {
4.     (k_evict, V_evict) = T_i[h_i(k_insert)]   //save old KVP
5.     T_i[h_i(k_insert)] = (k_insert, V_insert)   //put in new KVP
6.     if ((k_evict, V_evict) is NULL) { return "success"; }
7.     else {                                   //repeat in other table
8.         (k_insert, V_insert) = (k_evict, V_evict);
9.         i = 1 - i;
10.    }
11. }
12. return "failed to insert";   //need to rehash!
```

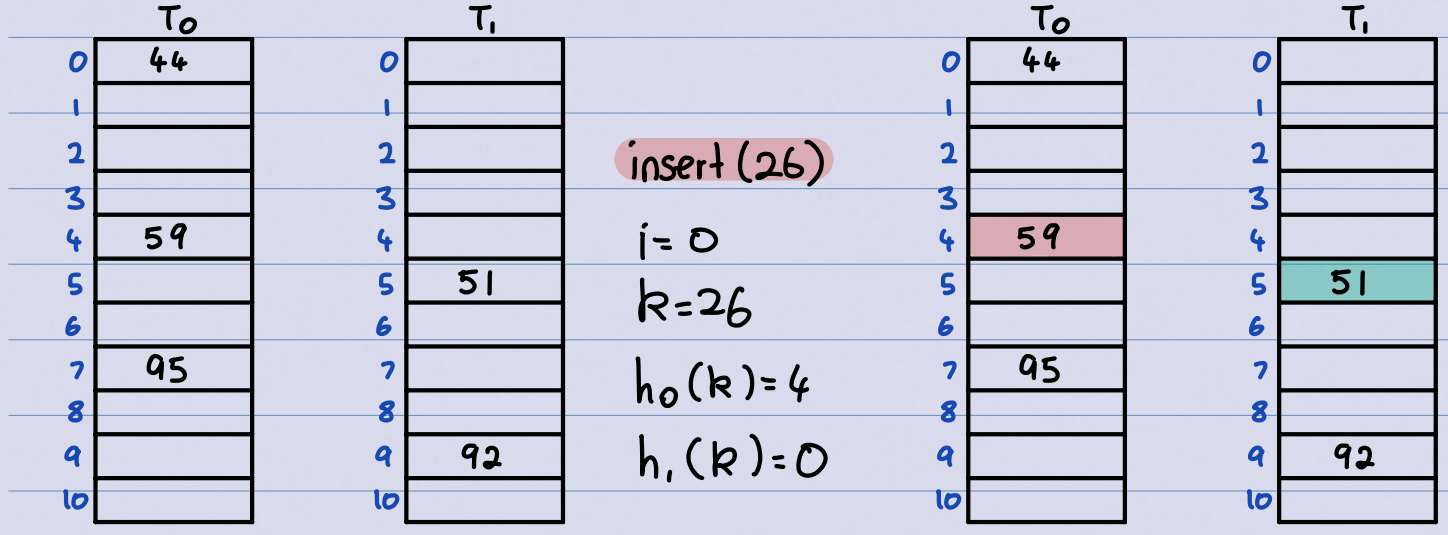
Cuckoo Hashing Insertion: Example

$M = 11$, $h_0(k) = k \bmod 11$, $h_1(k) = \lfloor 11(\varphi k - \lfloor \varphi k \rfloor) \rfloor$

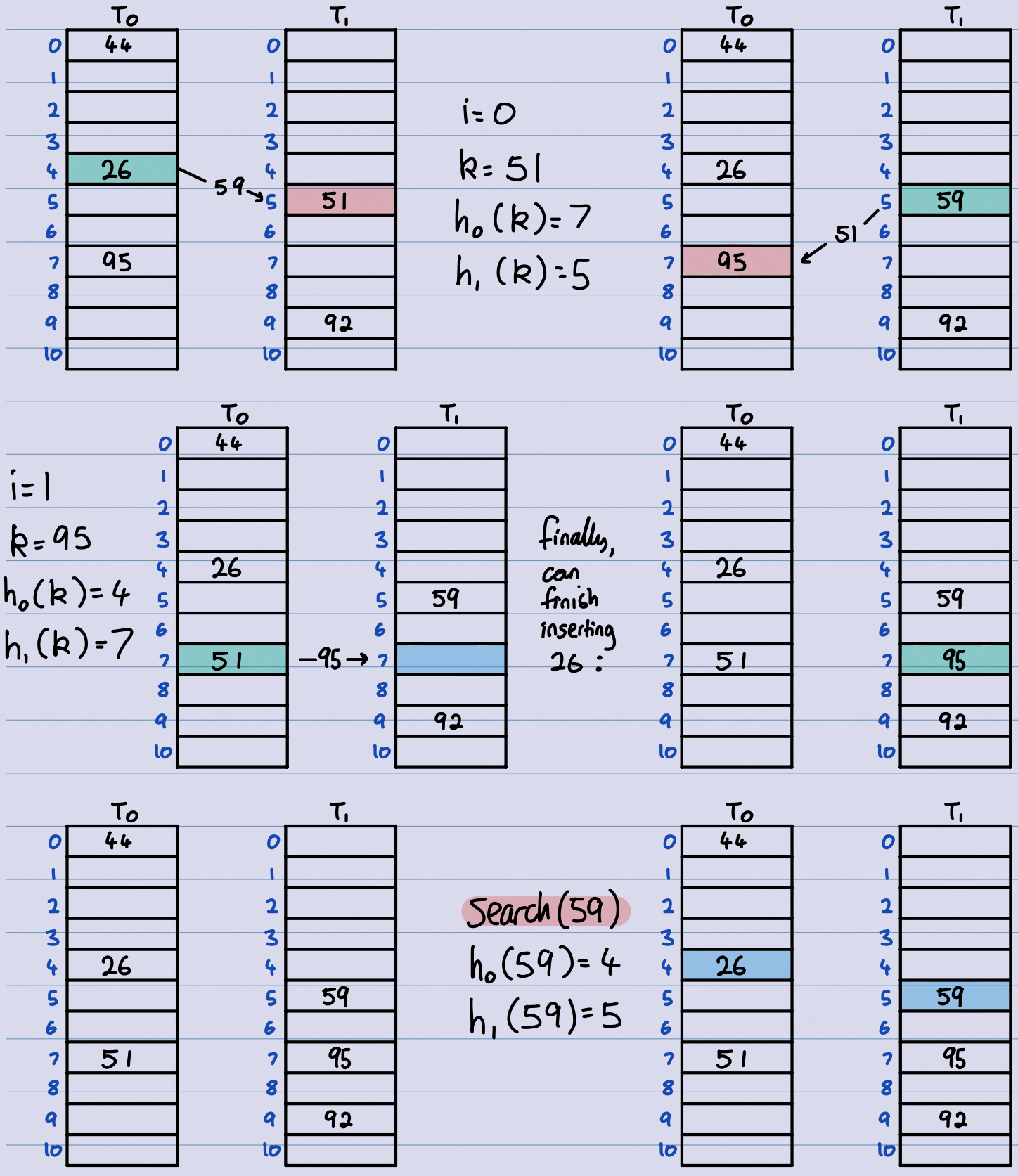








but now, i=1 and k=59. $h_0(k)=4$, $h_1(k)=5$.
So, we try to put the evicted 59 at slot $h_i(k)$ in T_i (ie, slot 5 in T_1). But, that's taken too! :



Cuckoo Hashing: Summary

- Expected number of evictions is $O(1)$.
  - ↳ so, in practice, stop evictions much earlier than $2n$
- This crucially requires a load factor $\alpha < 1/2$
  - ↳ Here, $\alpha = n / (\text{size of } T_0 + \text{size of } T_1)$

- So, Cuckoo hashing is wasteful of space
  ↳ In fact, space is $\omega(n)$ if insert forces lots of rehashing
- Expected space is $\Theta(n)$

There are many possible variations of cuckoo hashing. Eg:
- Combine T_0 and T_1 into one table
- Be more flexible when inserting: always consider both possible positions
- Use $k > 2$ allowed locations (ie, k hash functions)

Hashing with Open Addressing: Summary

For any open addressing scheme, we must have $\alpha \leq 1$
For the analysis, we need $0 < \alpha < 1$
Cuckoo hashing requires $0 < \alpha < 1/2$

Under these restrictions (and UHA):
- All strategies have $O(1)$ expected time for search, insert, and delete
- Cuckoo hashing has $O(1)$ worst case time for search/delete
- Probe sequences use $O(n)$ worst-case space, cuckoo hashing uses $O(n)$ expected space

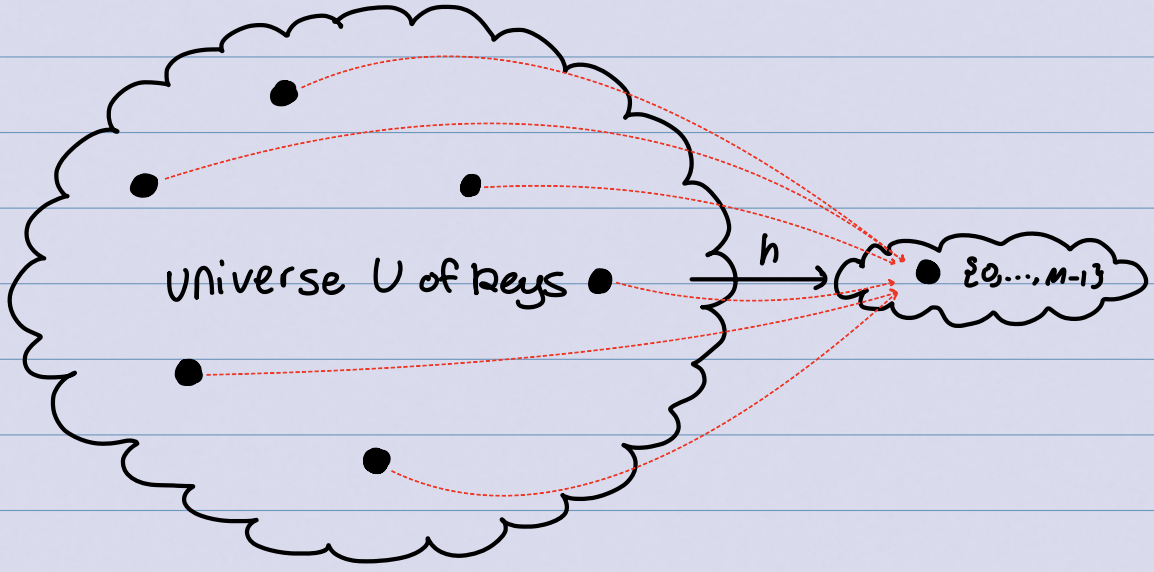
But for any hash-function, the worst-case runtime is $\Theta(n)$ for insert.

Note: in practice, double hashing seems most popular, or cuckoo hashing if there's many more searches than insertions

Hash Functions

Every hash function must do badly for some inputs:

- if the universe is big enough ($|U| \geq M(n-1)+1$), then there are n keys that all hash to the same value



- if we insert into this set of keys, then we have $O(n)$ runtime.

Choosing a Good Hash Function

- Analysis works only under UHA!
- Satisfying this is impossible: there are too many hash functions, we wouldn't know how to compute $h(k)$ efficiently.

Two ways to compromise:

1. Deterministic: hope for good performance by choosing a hash-function that is:
   - unrelated to any possible patterns in the data
   - depends on all parts of the key
2. Randomised: choose randomly among a limited set of functions
   - But aim for $Pr(\text{two keys collide}) = 1/M$ wrt key-distribution
   - This is enough to prove the expected runtimes for chaining

Deterministic Hash Functions

We've already seen the two basic methods for integer keys:

- **Modular method:** $h(k) = k \bmod M$
  - ↳ we should choose M to be a prime
  - ↳ this means finding a suitable prime quickly when rehashing
  - ↳ this can be done in $O(M \log\log M)$ time

- **Multiplication method:** $h(k) = \lfloor M(Ak - \lfloor Ak \rfloor) \rfloor$, for some floating point number A with $0 < A < 1$.
  - ↳ Multiplying with A is used to scramble the keys. So A should be irrational to avoid patterns in the keys.
  - ↳ Experiments show that good scrambling is achieved by the golden ratio $\varphi = \frac{\sqrt{5}-1}{2} \approx 0.61803\ldots$
  - ↳ We should use at least $\log|U| + \log|M|$ bits of A.

Carter-Wegmen's Universal Hashing

Better idea: choose hash function randomly!

- Requires: all keys are in $\{0, \ldots, p-1\}$ for some (big) prime p
- At initialisation, and whenever we rehash:
  - Choose $M < p$ arbitrarily, power of 2 is okay
  - Choose (and store) two random numbers (a, b):
    - $b = \text{random}(p)$
    - $a = 1 + \text{random}(p-1)$ (so $a \neq 0$)
  - Use as hash-function $h_{a,b}(k) = ((ak + b) \bmod p) \bmod M$
- Observe: hash-value can be computed in constant time

Analysis of these Carter-Wegmen hash function:

- Choosing h in this way doesn't satisfy the UHA
- But can show: two keys collide with probability $1/M$
- This suffices to prove the runtime bounds for hashing w/ chaining

Multi-Dimensional Data

What if the keys are multi-dimensional, like strings?

Standard approach is to flatten string w to integer $f(w) \in \mathbb{N}$

Eg: APPLE → (65, 80, 80, 76, 69) → $65R^4 + 80R^3 + 80R^2 + 76R^1 + 69R^0$

for some radix R, eg, R = 255.

we combine this with a modular hash function: $h(w) = f(w) \bmod M$

to compute this in $O(|w|)$ time without overflow, use Horner's rule and apply mod early.

Eg, h(APPLE) is:

$((((((65R + 80) \bmod M)R + 80) \bmod M)R + 76) \bmod M)R + 69) \bmod M$

Hashing vs Balanced Search Trees

Advantages of Balanced Search Trees

- $O(\log n)$ worst case operation cost
- Doesn't need special functions, assumptions, or known distributions
- Predictable space usage (exactly n nodes)
- Never need to rebuild the entire structure
- Supports ordered dictionary operations (successor, select, rank, etc)

Advantages of Hash Tables

- $O(1)$ operation cost (if hash function random and α is small)
- We can choose space / time tradeoff via α

- Cuckoo hashing achieves O(1) worst-case for search/delete

Range-Searching in Dictionaries for Points

Range Searches