

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	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sorted list/array	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
binary search tree	$\Theta(\text{height})$	$\Theta(\text{height})$	$\Theta(\text{height})$

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) $m = \lfloor \frac{l+r}{2} \rfloor$
- 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:

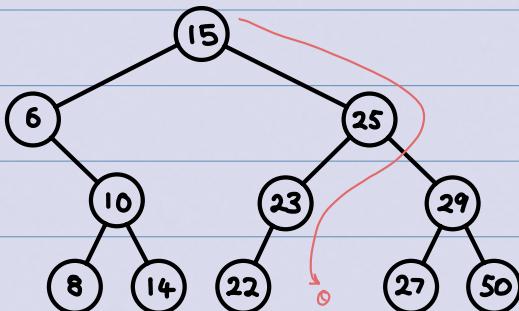
- all nodes have two (possibly empty) subtrees
- every node stores a KVP
- empty subtrees usually not shown

Ordering:

- every key k in $T.\text{left}$ is less than the root key
- every key k in $T.\text{right}$ is greater than the root key

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

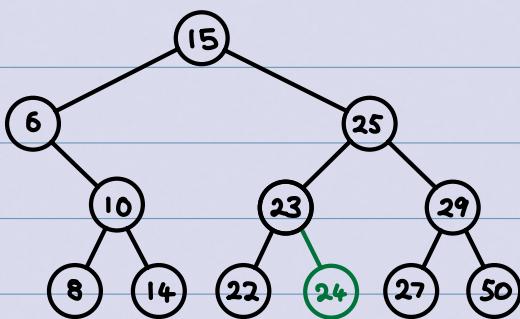
Eg: BST:: Search(24):



\therefore 24 not found in the BST!

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

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

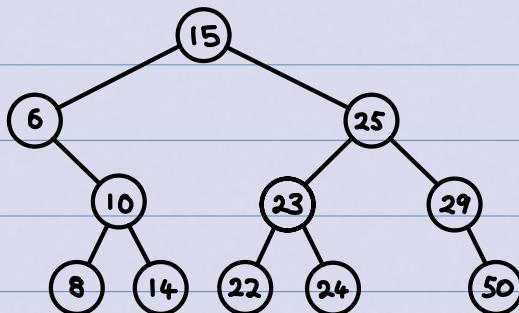
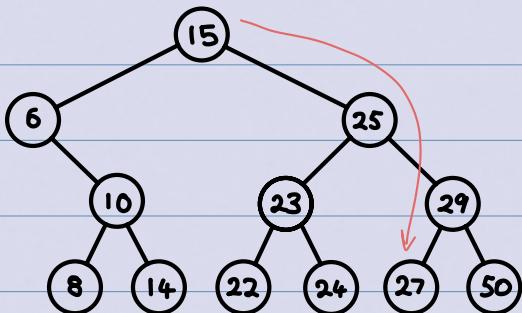


\therefore 24 inserted!

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

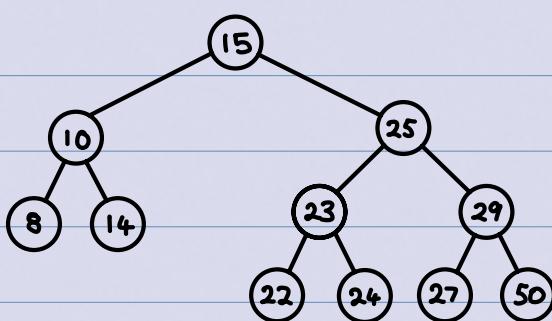
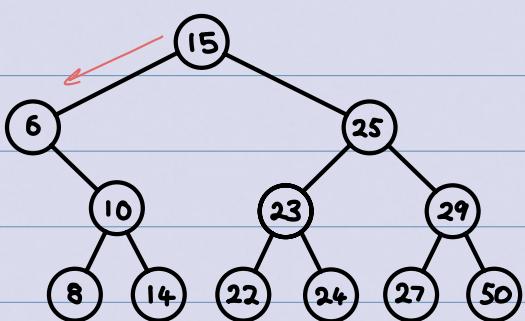
- If x is a leaf, delete it.
 - If x has one empty subtree, move the child up
 - Else, swap key at x with key at successor node and then delete that node
-
- Successor: next-smallest among all keys in the dict.

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



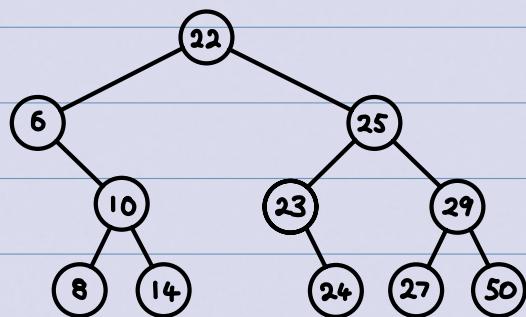
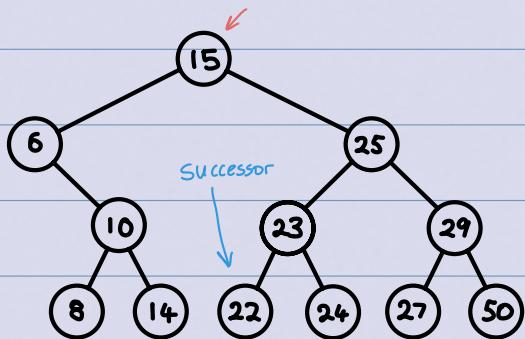
\therefore Deleted using case 1

Eg: BST:: Delete(6):



∴ Deleted using case 2

Eg: BST:: Delete(15)



∴ deleted using case 3

→ BST:: Search, BST:: insert, and BST:: delete all $\Theta(h)$, where $h = \text{maximum number for which level } h \text{ contains nodes}$.
 ↳ single-node tree has height 0, empty tree has height of -1.

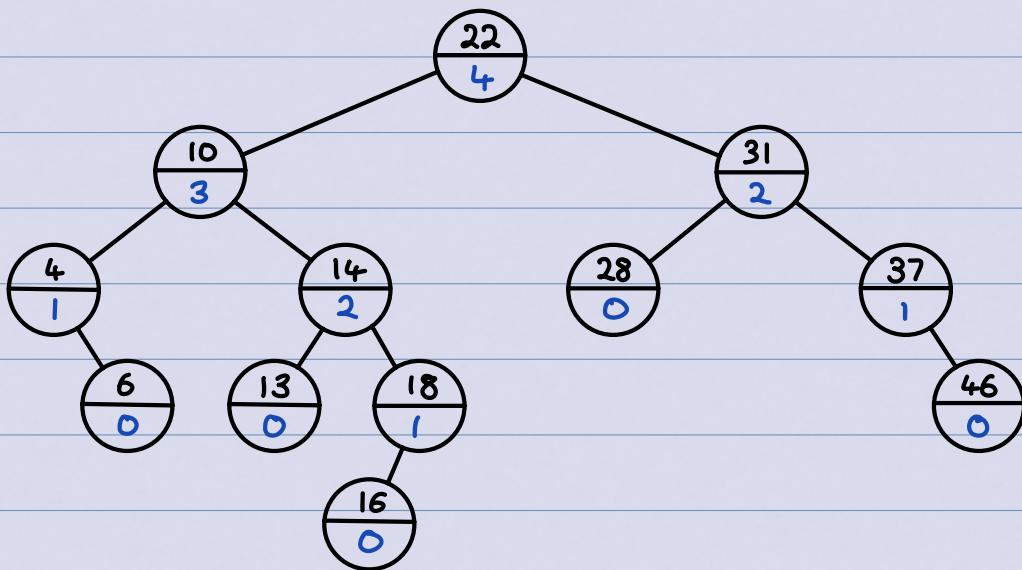
· if n items are inserted one-at-a-time, how big is h ?
 ↳ worst-case: $n-1 = \Theta(n)$
 ↳ best-case: $\Theta(\log n)$

AVL Trees

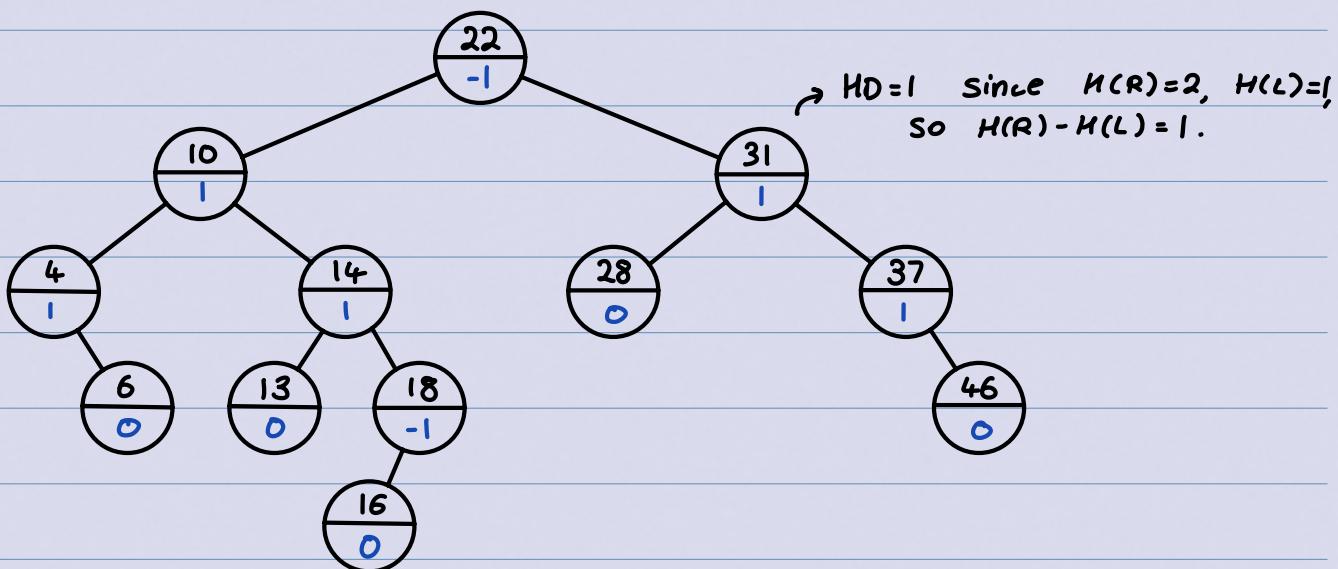
an AVL tree is a BST with an additional height-balance property at every node:
 The heights of the left and right subtree differ by at most 1.

↳ ie, if node z has left subtree L and right subtree R ,
then $\text{height}(R) - \text{height}(L)$ 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 $\mathcal{O}(\log n)$.

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

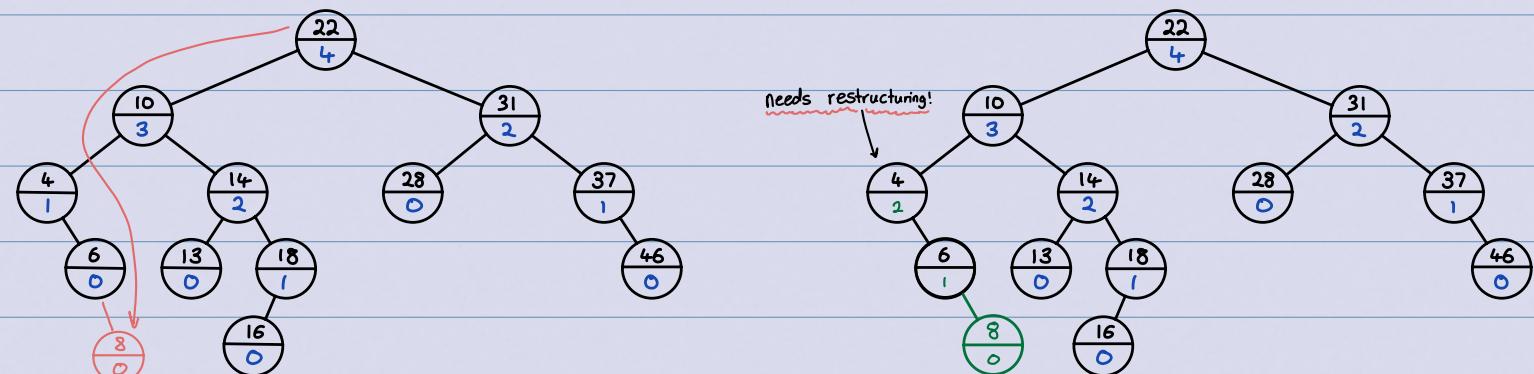
AVL::insert(R, v) \rightarrow first, insert (k, v) with usual BST

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:

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

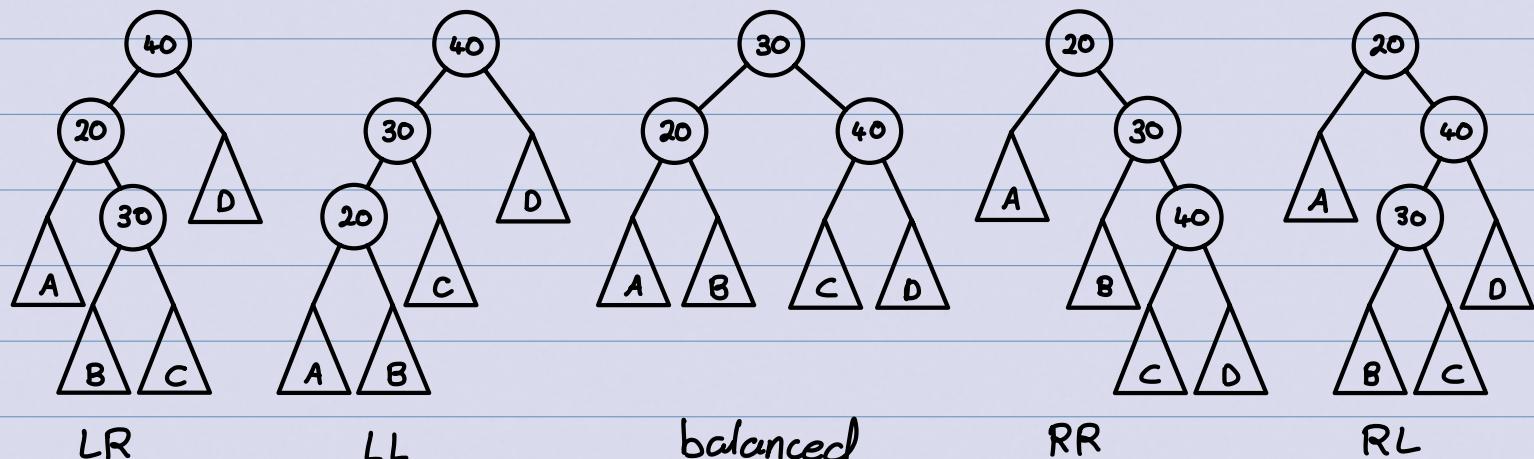
Eg: AVL: insert (8):



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

Restructuring in a BST: Rotations

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

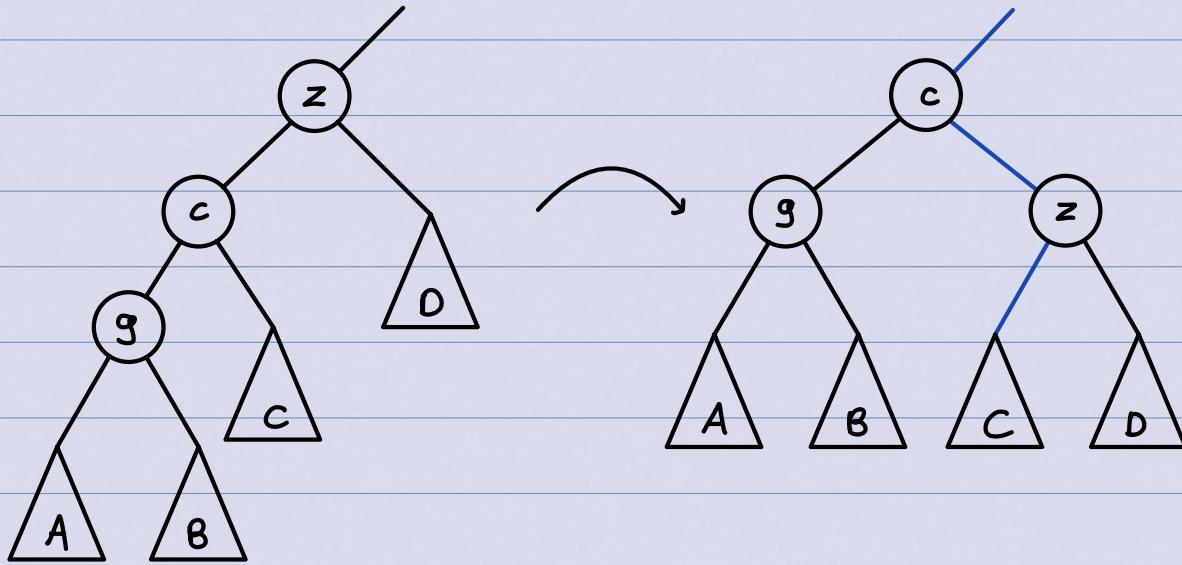


→ goal: change the structure without changing the order, and

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( $z$ )
```

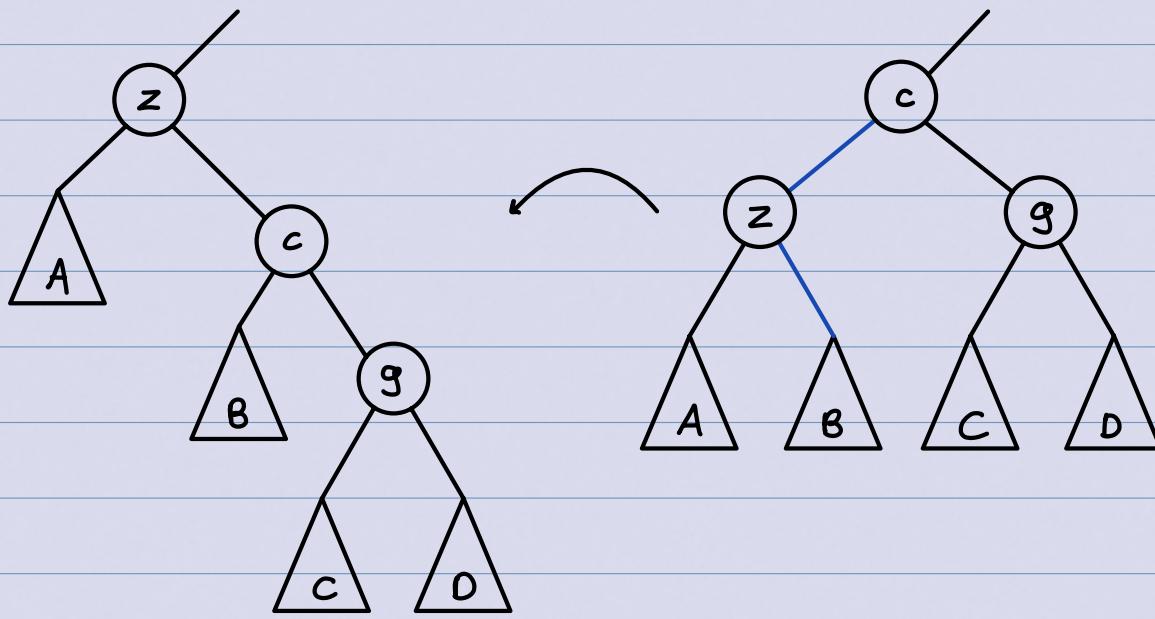
- 1) $c = z.\text{left}$
- 2) //fix links connecting to above
- 3) $c.\text{parent} = (p = z.\text{parent})$
- 4) if ($p == \text{null}$) { $\text{root} = c$ }
- 5) else {
- 6) if ($p.\text{left} == z$) { $p.\text{left} = c$ }
- 7) else { $p.\text{right} = c$ }
- 8) }
- 9) //actual rotation
- 10) $z.\text{left} = c.\text{right}, \quad c.\text{right.parent} = z$
- 11) $c.\text{right} = z, \quad z.\text{parent} = c$
- 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) $C = z.\text{right}$
- 2) // fix links connecting to above
- 3) $C.\text{parent} = (p = z.\text{parent})$
- 4) if ($p == \text{null}$) { $\text{root} = C$ }
- 5) else {
- 6) if ($p.\text{right} == z$) { $p.\text{right} = C$ }
- 7) else { $p.\text{left} = C$ }
- 8) }
- 9) // actual rotation

10) $z.\text{right} = c.\text{left}$, $c.\text{left.parent} = z$

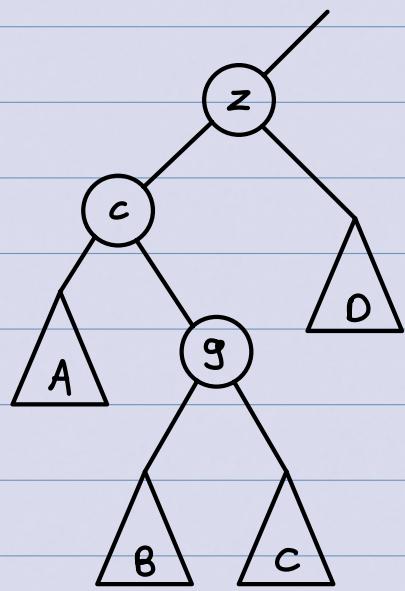
11) $c.\text{left} = z$, $z.\text{parent} = c$

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

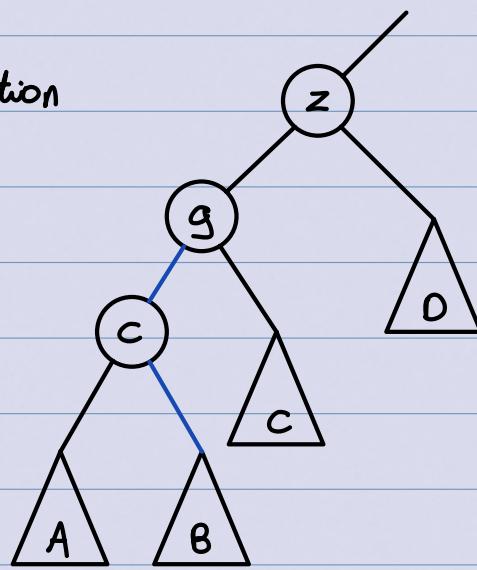
13) return c // returns new root of subtree

↳ runs in $O(1)$!

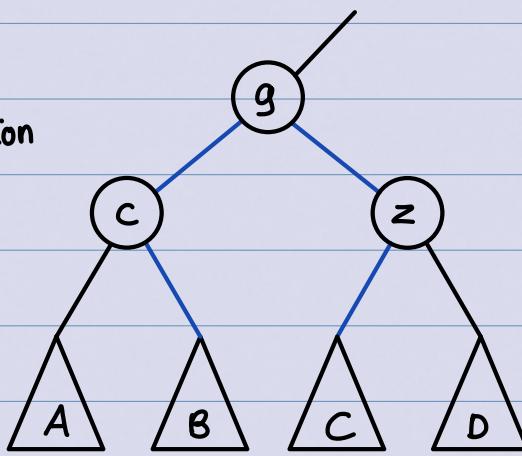
Double Right Rotation



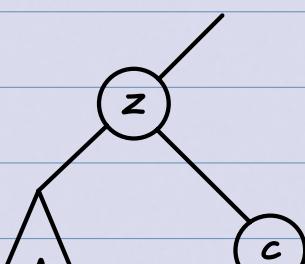
1) left rotation
at c



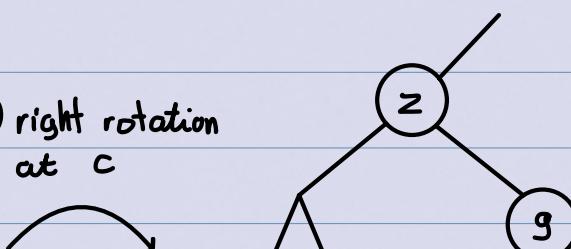
2) right rotation
at z

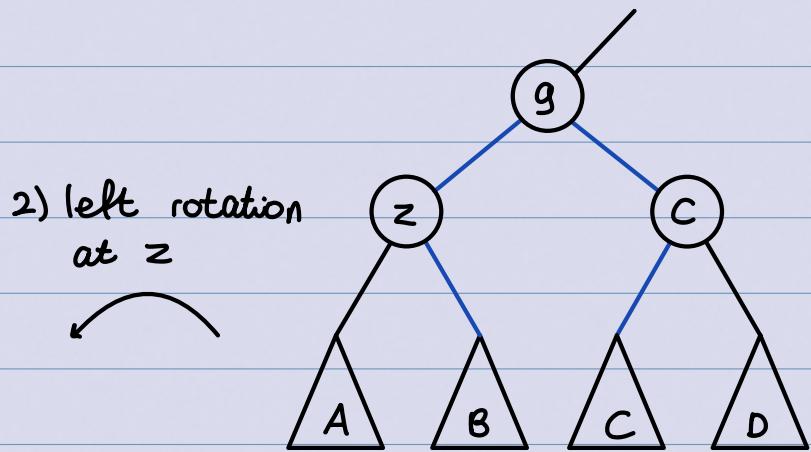
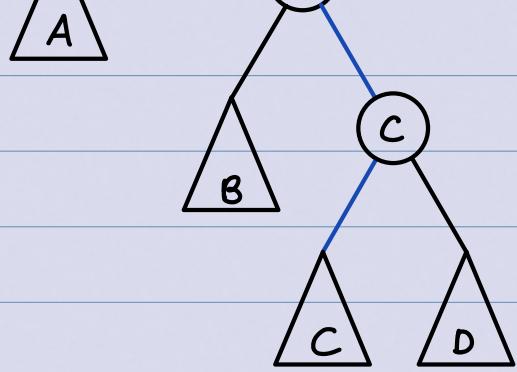
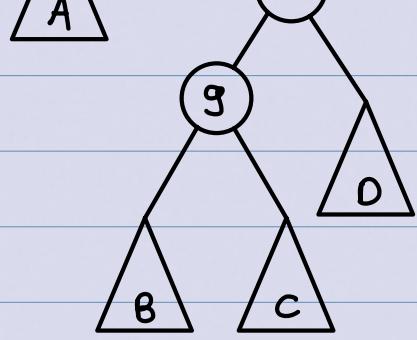


Double Left Rotation



1) right rotation
at c





AVL Insertion Revisited

Imbalance at z : do (single or double) rotation

- Choose c as child where subtree has bigger height.

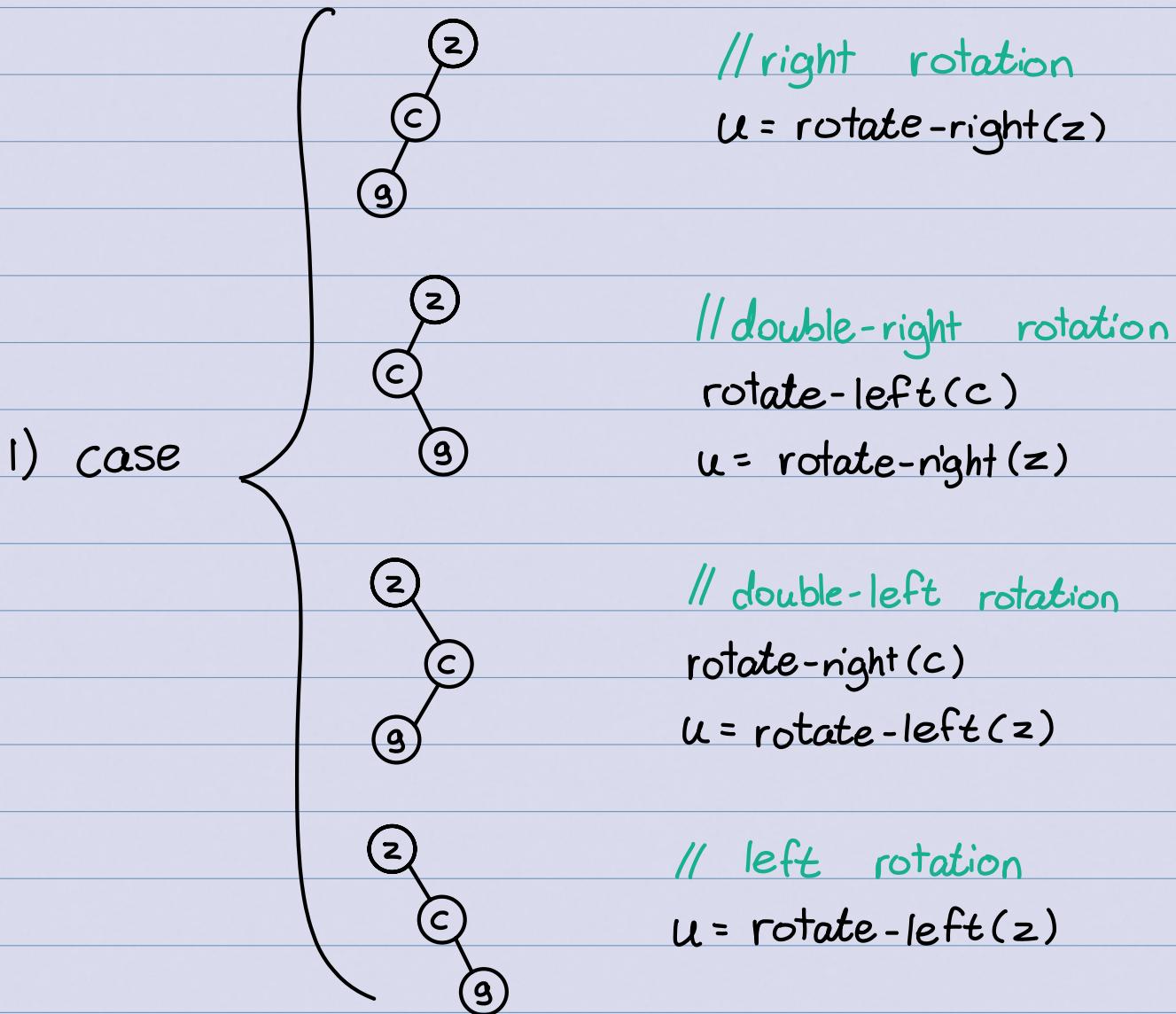
AVL:: insert(k, v)

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

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

Fixing an slightly-unbalanced AVL tree:

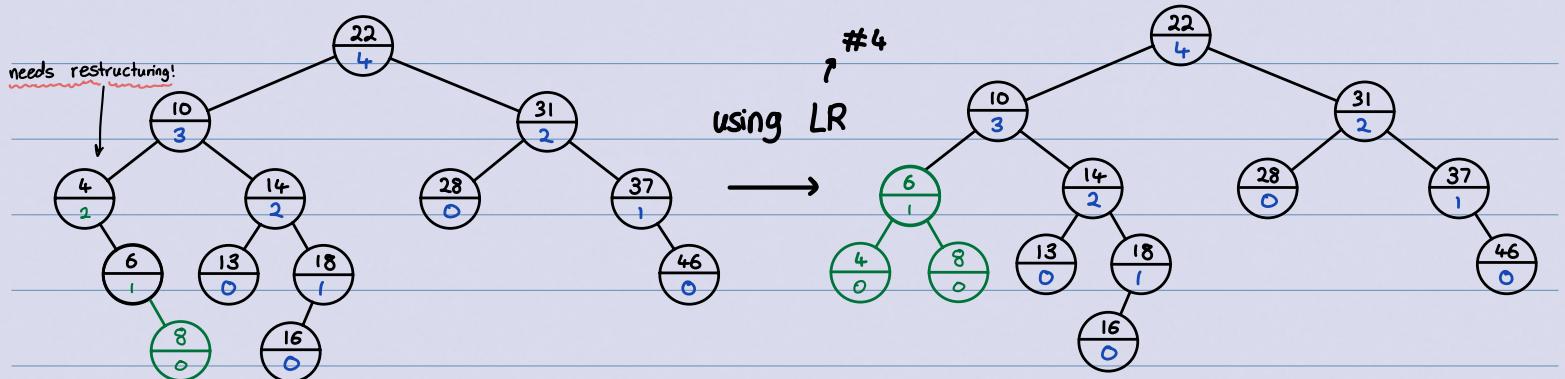
`restructure(g, c, z) → node g is child of c which is child of z`



2) return u

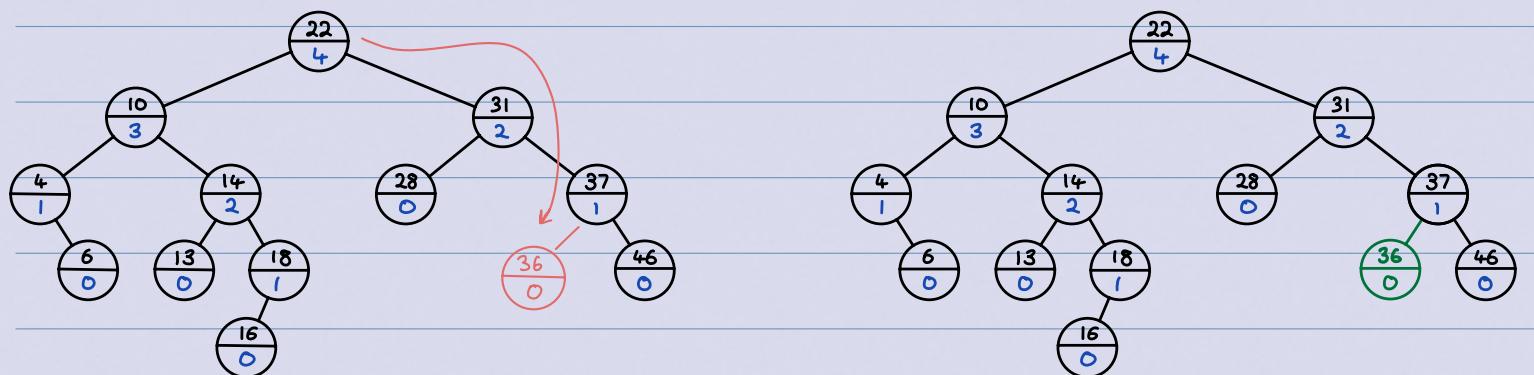
Eg: revisiting `AVL::insert(8)`

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

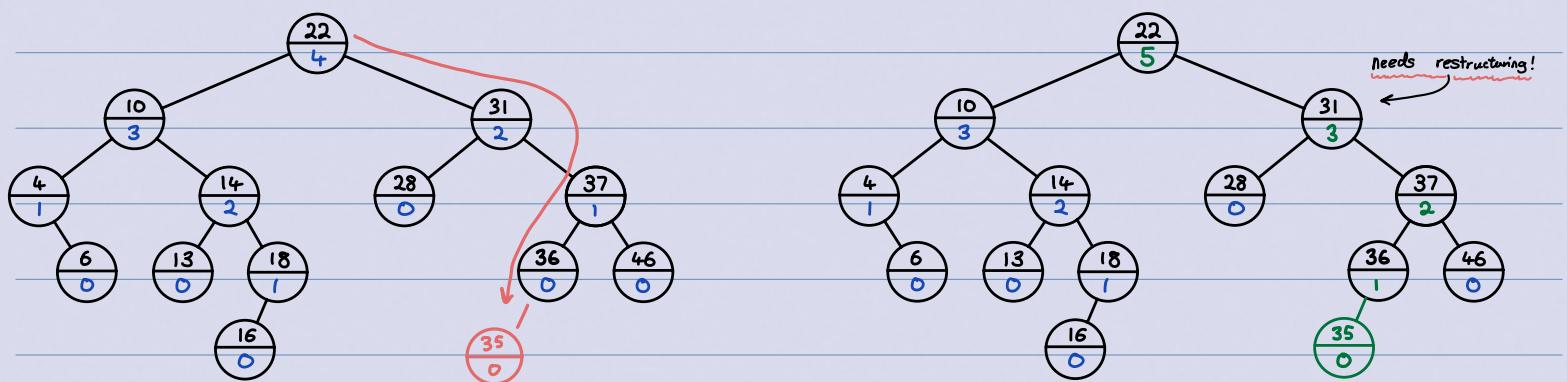


∴ we have correctly restructured the subtree using a left rotation.

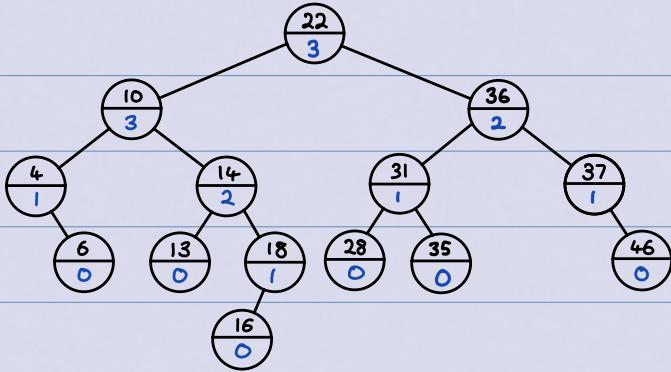
Eg: AVL::insert(35), AVL::insert(36):



→ 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

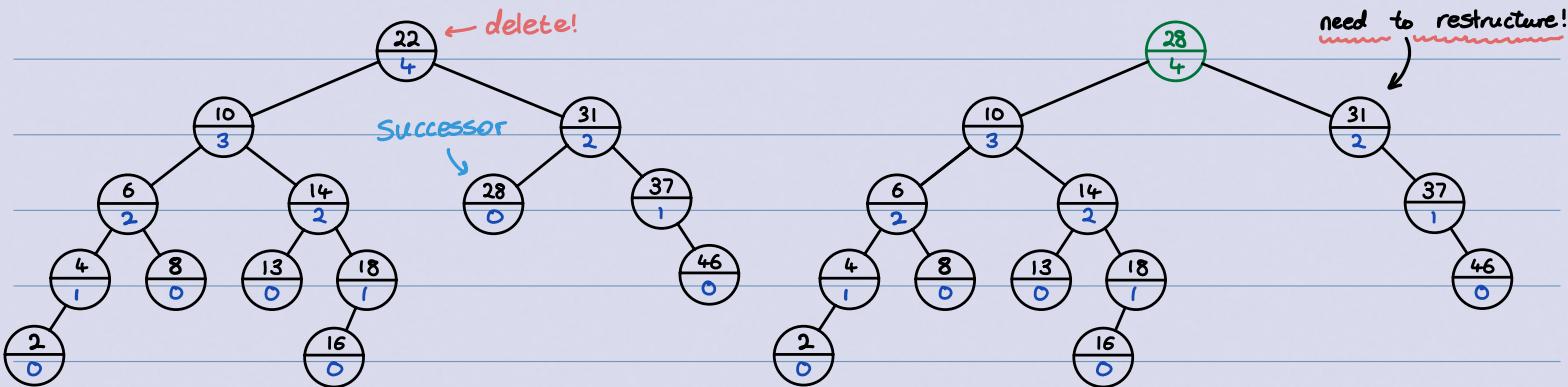
`AVL::delete(k)` → first, remove the key k with `BST::delete`.

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.

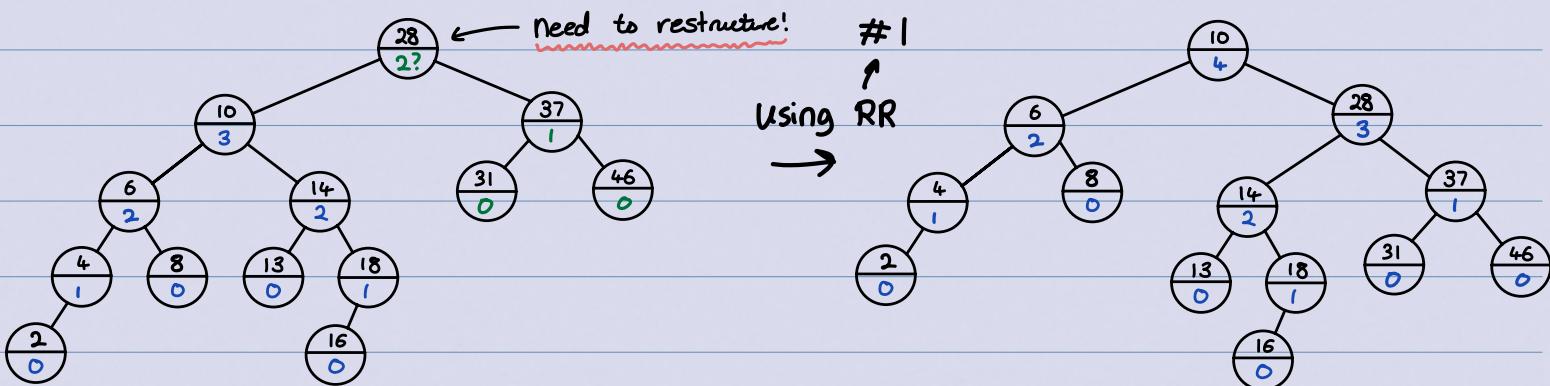
`AVL:: delete(k)`

- 1) $z = \text{BST}::\text{delete}(k)$
- 2) // Assume z is the parent of the BST node that was removed
- 3) while (z is not null) {
 - 4) if ($|z.\text{right}.\text{height} - z.\text{left}.\text{height}| > 1$) {
 - 5) $C = \text{taller child of } z$
 - 6) $g = \text{taller child of } C$ // break ties → avoid double rotation
 - 7) $z = \text{restructure}(g, C, z)$
 - 8) }
 - 9) // always continue up the path
 - 10) $\text{set-height-from-subtrees}(z)$
 - 11) $z = z.\text{parent}$
 - 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 deleted node 22 and restructured accordingly!

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

AVL Trees - Summary

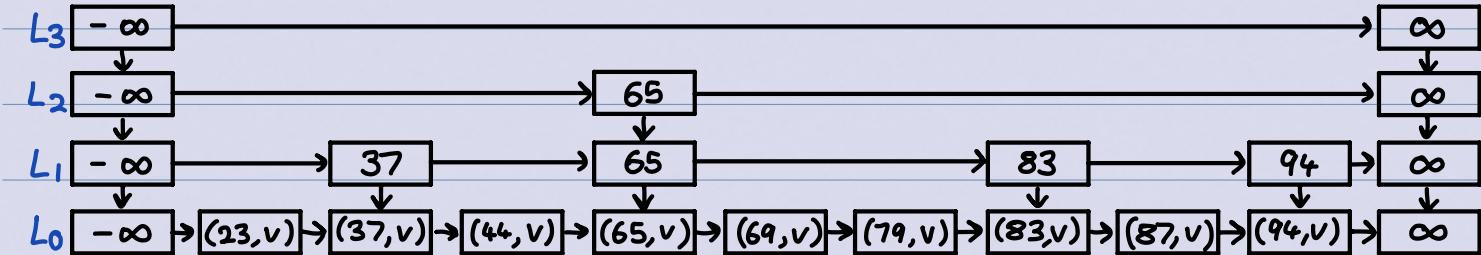
- Search → just like BSTs, costs $\Theta(\text{height})$
- Insert → BST::insert, then check & update along path to new leaf
 - total cost $\Theta(\text{height})$
 - restructure will be called at most once!
- Delete → BST::delete, then check & update along path to deleted node

- total cost $\Theta(\text{height})$
 - restructure may be called $\Theta(\text{height})$ times!
 - Worst-case for all operations is $\Theta(\text{height}) = \Theta(\log n)$

Skip Lists

A hierarchy 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,
$$L_0 \supseteq L_1 \supseteq \dots \supseteq L_k$$
 - list L_k 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 Lo
 - there are (usually) more nodes than KVPs (above example has 14 non-sentinel 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 $R \in L_i$
 - ↳ 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).

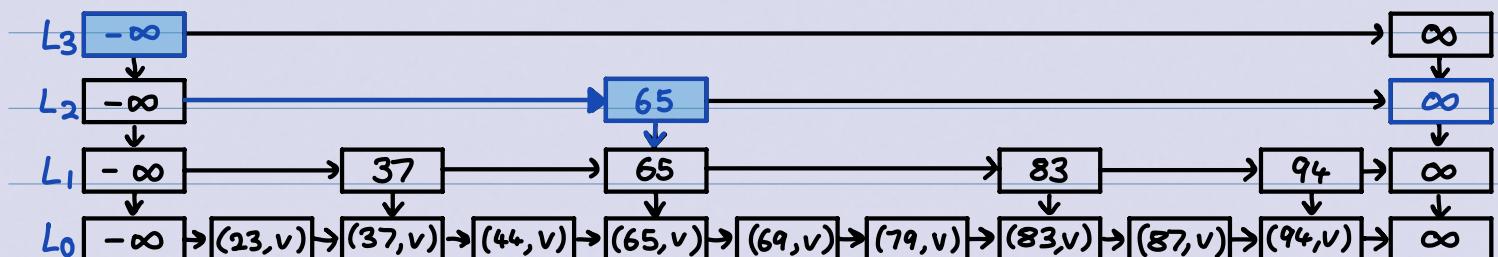
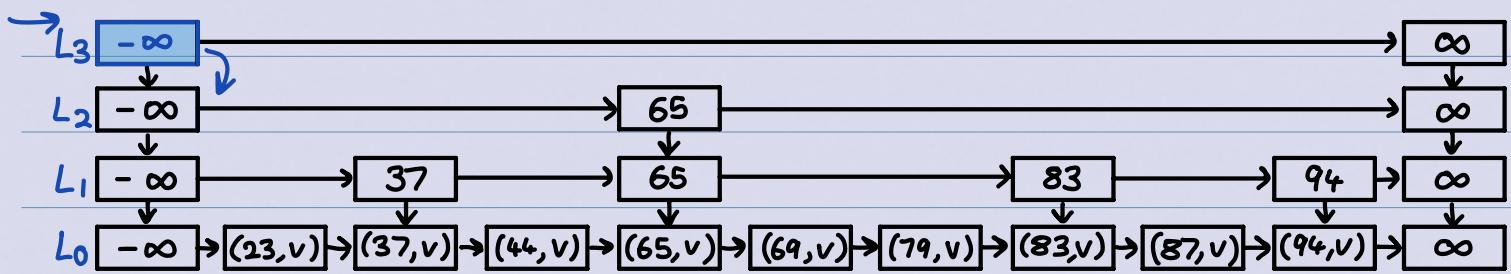
get-predecessors(k)

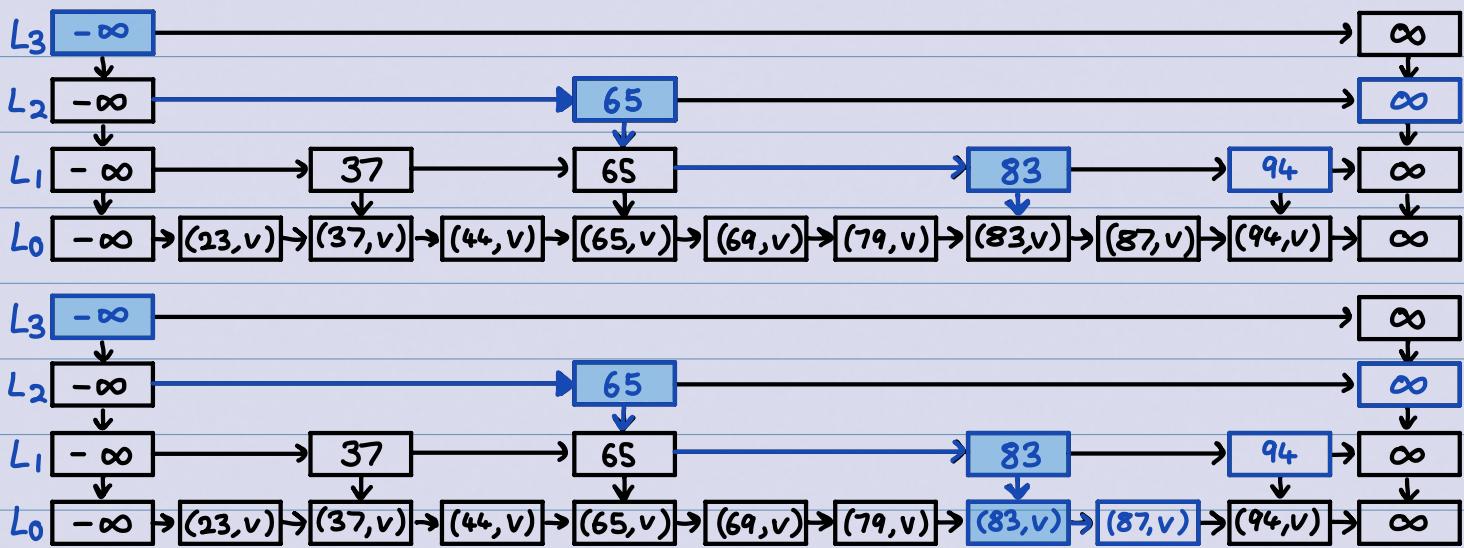
- 1) p = root
- 2) P = stack of nodes, initially containing p
- 3) while (p.below ≠ null) {
- 4) p = p.below
- 5) while (p.after.key < k) { p=p.after }
- 6) P.push(p)
- 7) }
- 8) return P

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/ R

final stack:
(83, v)

 = added to P

83

→ = path taken by p

65

-∞

∴ 83 was found in only 7 comparisons!

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(R)

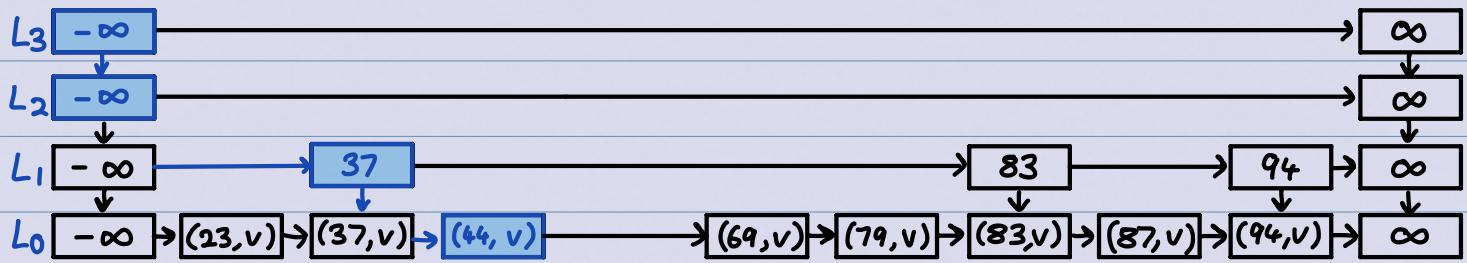
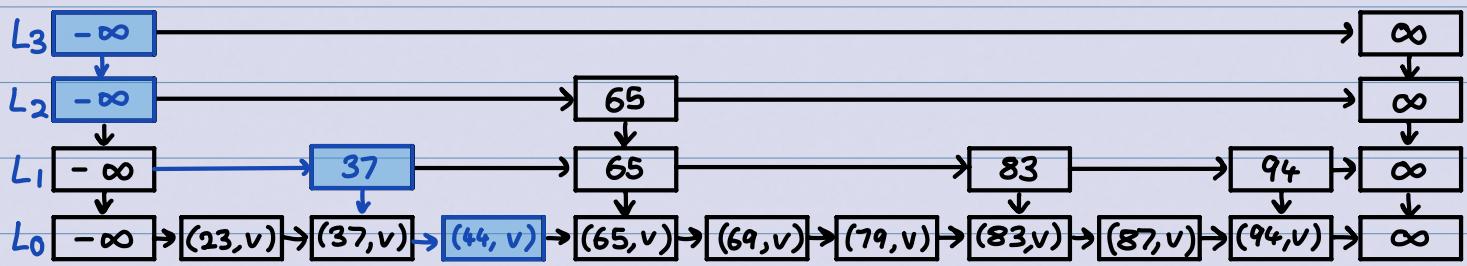
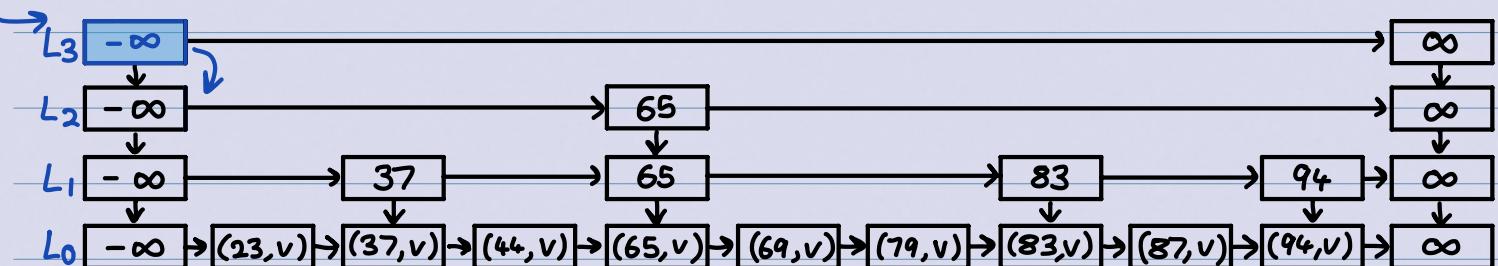
- 1) P = get-predecessors(R)
- 2) while (P is not empty) {
- 3) p = P.pop() // predecessor of R in some list
- 4) if (p.after.key = R) { p.after = p.after.after }
- 5) else { break } // no more copies of R
- 6) }
- 7) p = left sentinel of the root list
- 8) while (p.below.after is the ∞-sentinel) {
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

9) p. below = p. below. below

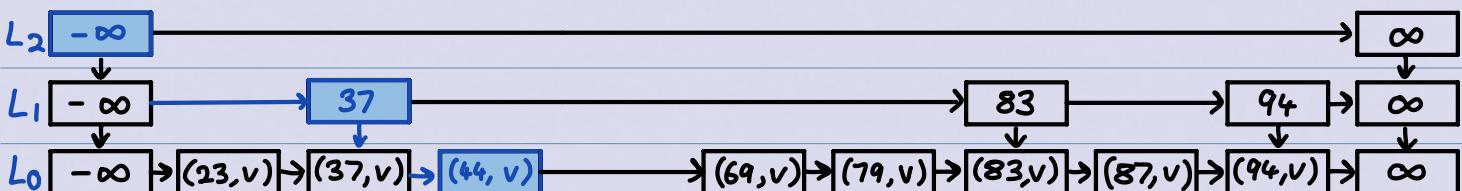
10) p. after. below = p. after. below. below

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 successfully 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
 - $\therefore \Pr(\text{tower of key } k \text{ has height } i) = (\frac{1}{2})^i$
- 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)

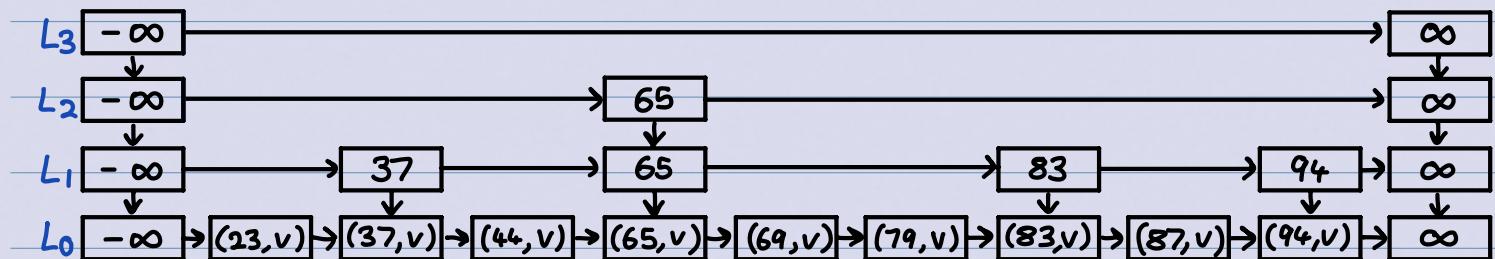
- for ($i=0; \text{random}(2)=1; i++$) {} // random tower height
- for ($h=0; p=\text{root}.below; p \neq \text{null}; p=p.below; h++$) {}
- while ($i \geq h$) {
 - Create new sentinel-only list; link it in below topmost level
 - $h++$
 - }
 - $P = \text{get-predecessors}(k)$
 - $p = P.pop()$ // insert (k, v) in L_0
 - $Z_{\text{below}} = \text{new node with } (k, v)$
 - $Z_{\text{below}}.after = p.after, p.after = Z_{\text{below}}$
 - while ($i > 0$) { // insert k in L_1, \dots, L_i
 - $p = P.pop()$
 - $z = \text{new node with } k$

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

15) $i = i + 1$

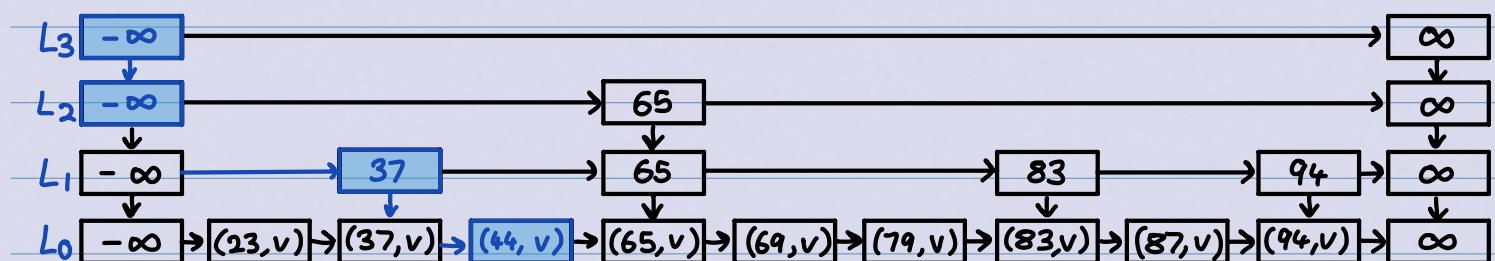
16) }

Example (1): `skipList::insert(52, v)`. coin tosses: H, T → $i = 1$

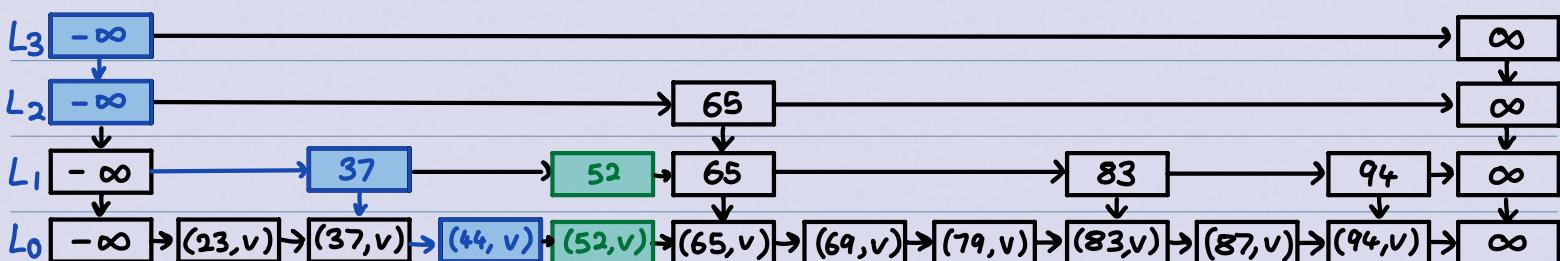


Since $h=3 > i=1$, we don't need to insert any sentinel-only lists!

`get-predecessors(52)`:

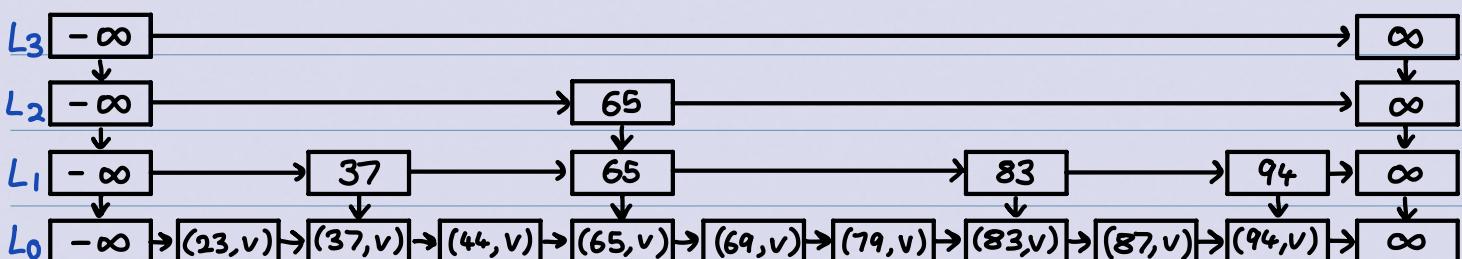


insert 52 in lists L_0, \dots, L_i (ie, L_0 and L_1):

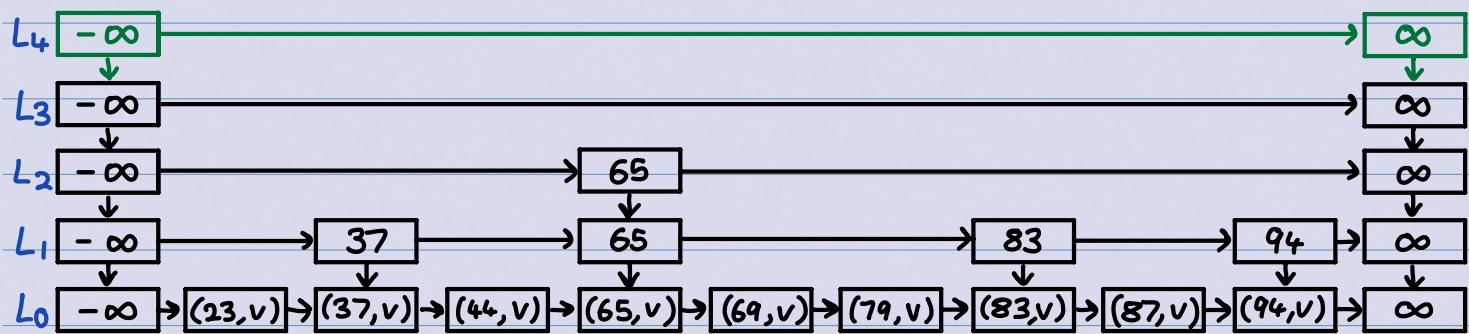


∴ we have successfully inserted node with key = 52 !

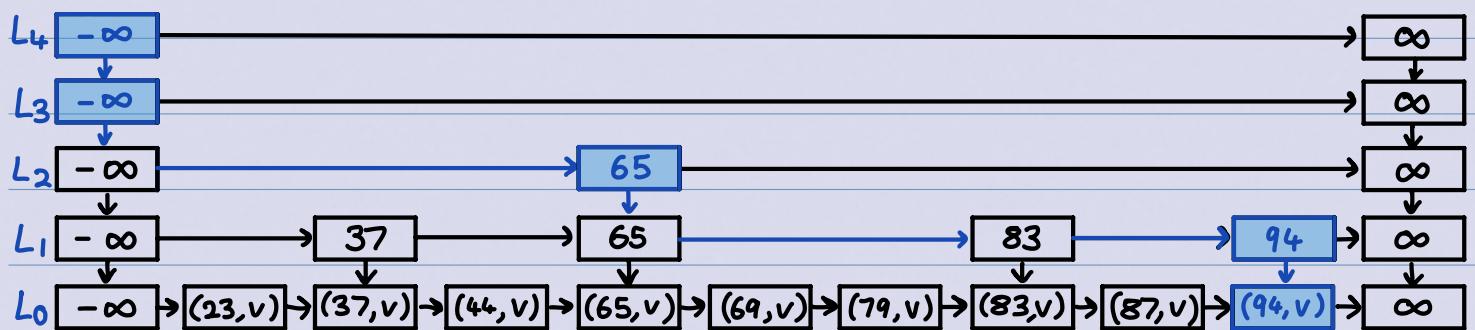
Example (2): `skipList::insert(100, v)`. coin tosses: H, H, H, T → $i = 3$.



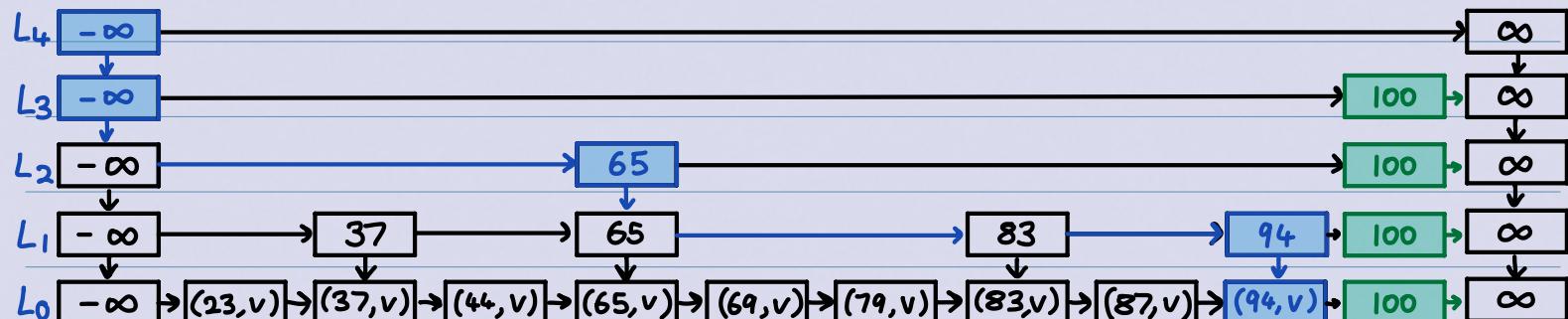
Since $h=3$ is not greater than $i=3$, we must add another level:



now, we call `get-predecessors(100)`:



then, we insert 100 in lists L_0, \dots, L_i



\therefore we have successfully 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 $O(n)$

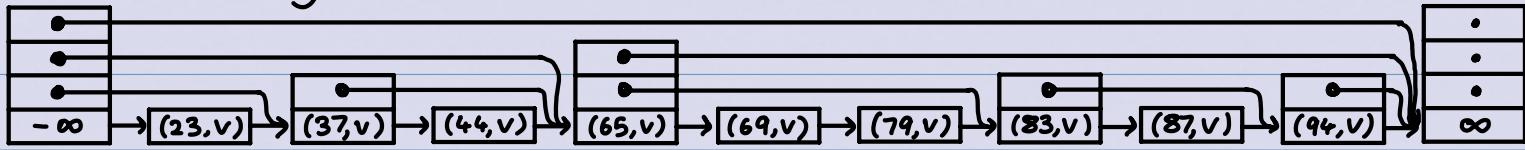
- run-time of operations is dominated by `get-predecessors`:

- ↳ how often do we drop down (execute $p=p.\text{below}$)? height.
- ↳ how often do we step forward (execute $p=p.\text{after}$)?
- 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 / successful 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 front (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 = $\sum_{i=1}^n i \cdot (\text{access probability of } k \text{ at position } i)$.

Example:

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

the current order has expected access cost:

$$\frac{2}{26} \cdot 1 + \frac{8}{26} \cdot 2 + \frac{1}{26} \cdot 3 + \frac{10}{26} \cdot 4 + \frac{5}{26} \cdot 5 = \frac{86}{26} \approx 3.31$$

the order $D \rightarrow B \rightarrow E \rightarrow A \rightarrow C$ is better!

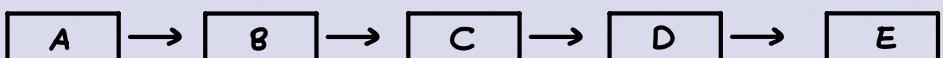
$$\frac{10}{26} \cdot 1 + \frac{8}{26} \cdot 2 + \frac{5}{26} \cdot 3 + \frac{2}{26} \cdot 4 + \frac{1}{26} \cdot 5 = \frac{66}{26} \approx 2.54.$$

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).

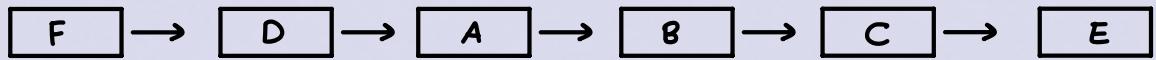
Move-To-Front Heuristic (MTF): upon a successful search, move the accessed item to the front of the list.



↓ search (D)



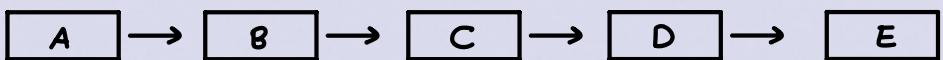
↓ insert (F)



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 preceding it.



↓ search (D)



↓ insert (F)



↳ 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 $\Theta(\log n)$ time for search?

- No: comparison-based searching lower bound is $\Omega(\log n)$.
- Yes: non-comparison-based searching can achieve $\sigma(\log n)$ (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 ($l \leq r$) {
3. $m = \lfloor \frac{l+r}{2} \rfloor$
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 $l + \lceil \frac{r - A[l]}{A[r] - A[l]} \cdot (r - l - 1) \rceil$.

- $k - A[l]$ → 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(), R)

1. $l = 0, r = n-1$
 2. while ($l \leq r$) {
 3. if ($k < A[l]$ or $k > A[r]$) { return "not found" }
 4. if ($k = A[r]$) { return "found at $A[r]$ " }
 5. $m = l - \left\lceil \frac{k - A[l]}{A[r] - A[l]} \cdot (r - l - 1) \right\rceil$
 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):

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

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

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

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

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

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

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

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

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

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

Tries

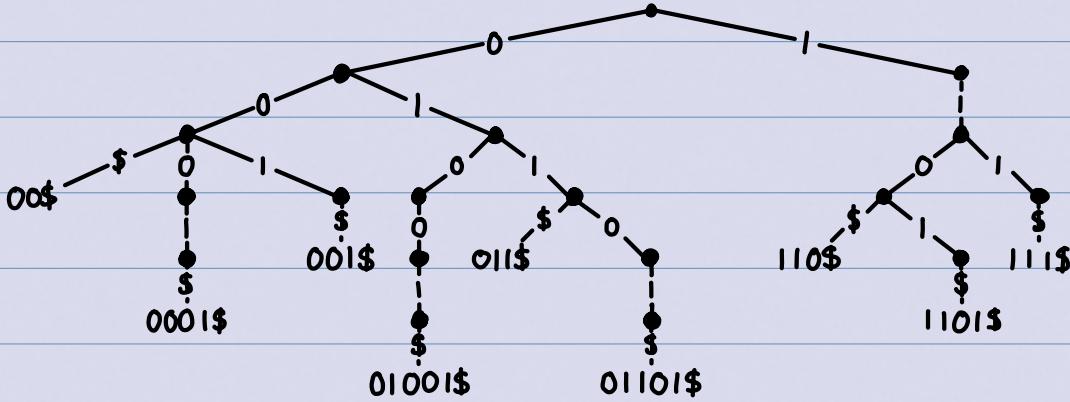
Words (=strings) are sequences of characters over alphabet Σ .

- Typical Alphabets: $\{0, 1\}$ (bitstrings), ASCII, $\{C, G, T, A\}$.
- Stored in an array: $w[i]$ gets i th character (for $i=0, 1, \dots$).
- Words have end-sentinel \$: 
- $w.\text{Size} = |w| = \# \text{non-sentinel characters}$ ($|bear\$| = 4$)
- Sort words lexicographically: $be\$ <_{\text{lex}} bear\$ <_{\text{lex}} beer\$$

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 ω , keeping track of current depth d
- Repeat until no such link or ω found at a leaf
- Similar for skip lists, we find search-path P first.

Trie:: get-path-to(ω)

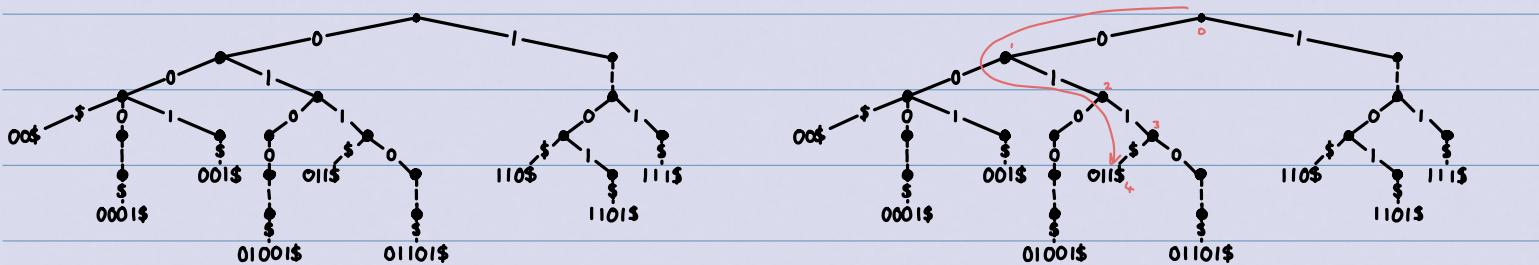
//Output: stack with all ancestors of where ω would be

- $P = \text{empty stack}$, $z = \text{root}$, $d = 0$, $P.\text{push}(z)$
- while ($d \leq |\omega|$) {
 - if (z has a child-link labelled with $\omega[d]$) {
 - $z = \text{child at this link}$, $d++$, $P.\text{push}(z)$
 - else { break }
- }
- else { break }
- }
- return P

Trie:: Search(ω)

- $P = \text{get-path-to}(\omega)$, $z = P.\text{top}$
- if (z is not a leaf) { return "not found, but would be in subtrie of z " }
- return key-value pair at z

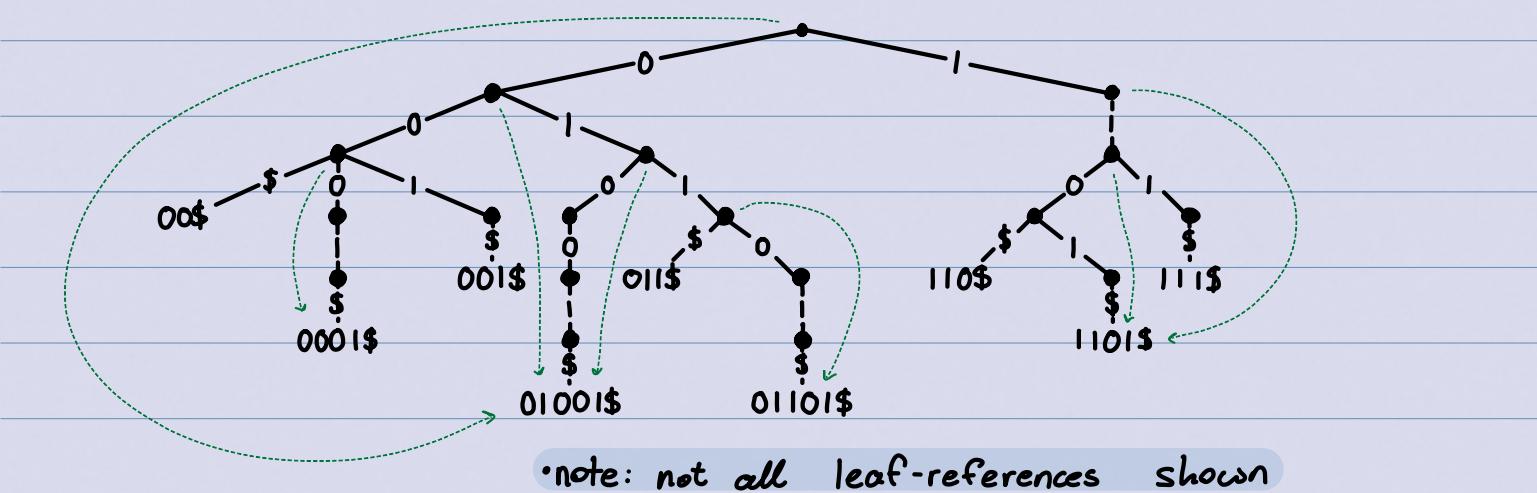
Tries search example: Trie::Search(011\$):



Tries: prefix search

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

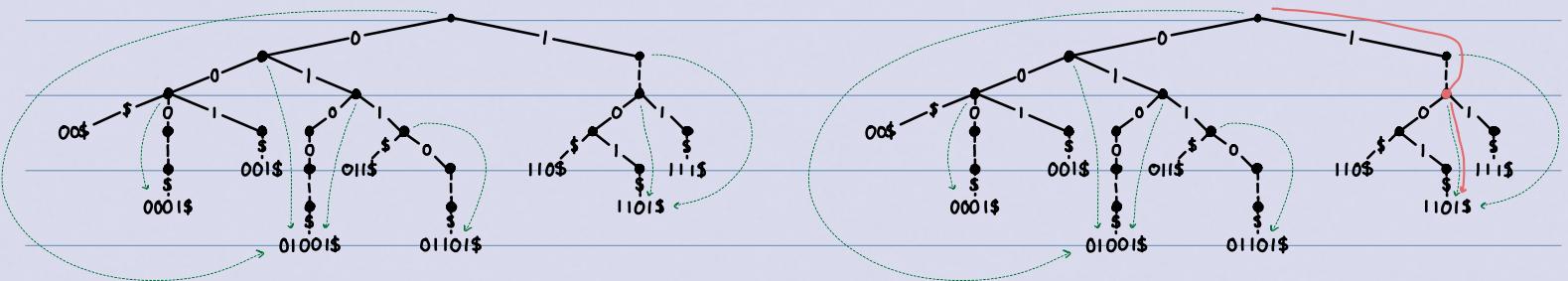
example of trie with leaf-references:



Trie::prefix-Search(ω)

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

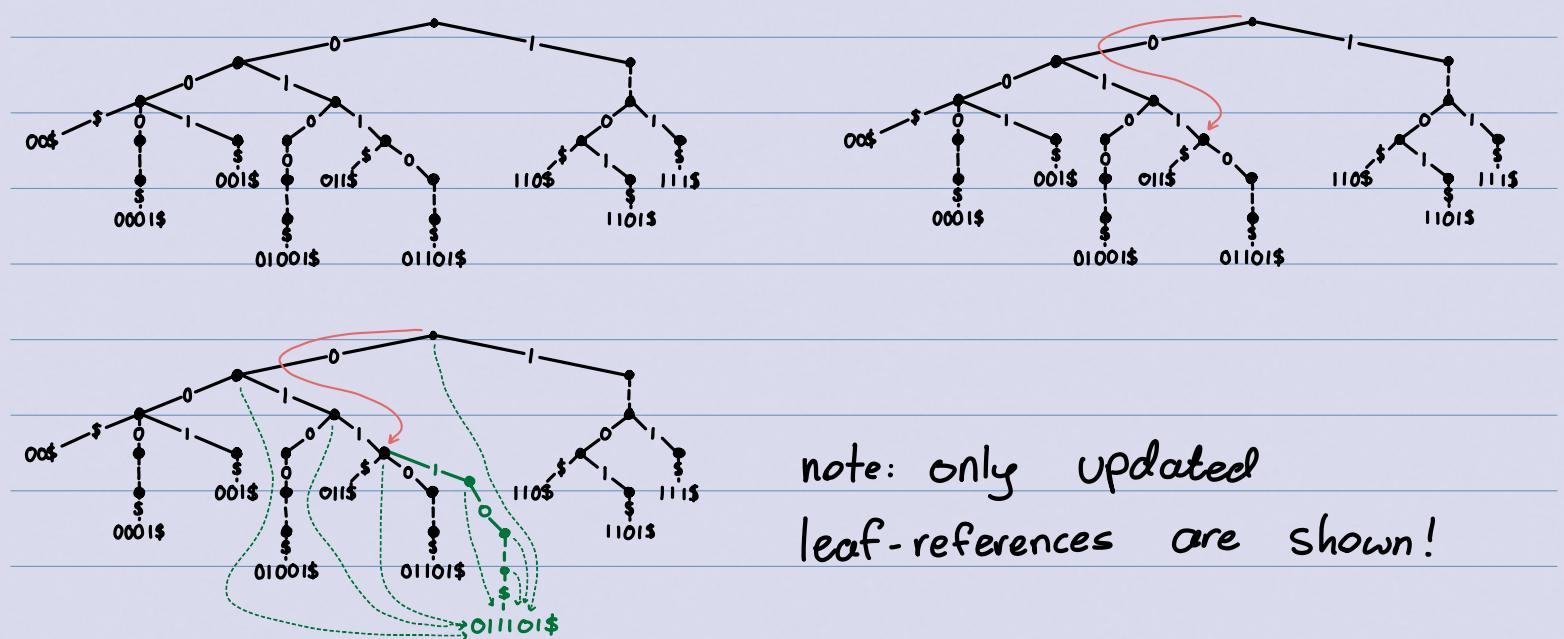
Tries prefix-search example: Trie::prefix-search(11\$):



Tries:: Insert

- $P = \text{get-path-to}(\omega)$ gives ancestors that exist already
- Expand the trie from $P.\text{top}()$ by adding necessary nodes that correspond to extra bits of ω .
- Update leaf-references (also cut P if ω is longer than previous leaves).

Example: Trie:: insert(01101\$):

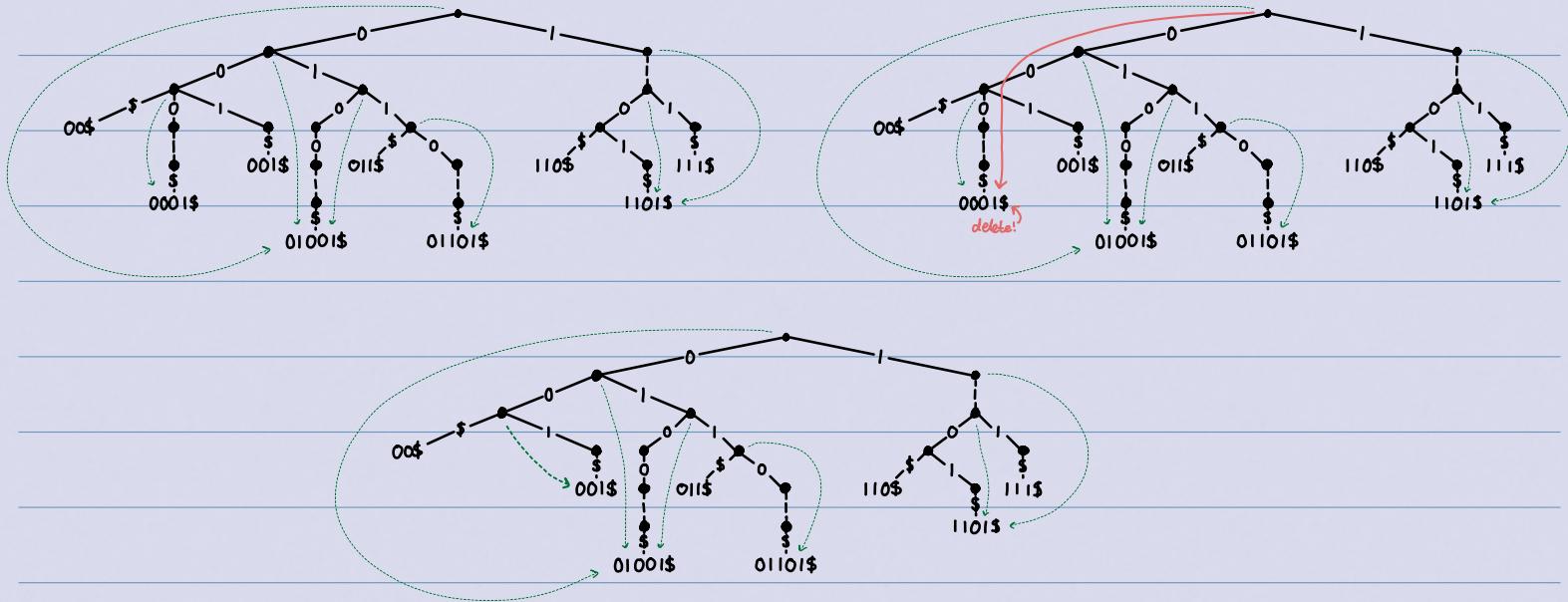


Tries: Delete

- $P = \text{get-path-to}(\omega)$ gives all ancestors
- Let l be the leaf where ω is stored
- Delete l and nodes on P until ancestor that had two or more children

- Update leaf-references on rest of P
(if $z \in P$ referred to l , find new $z.\text{leaf}$ from other children)

Example: Trie:: delete(0001\$):



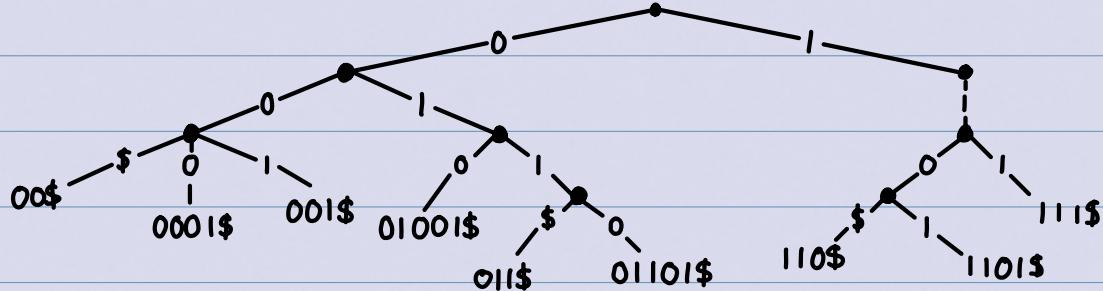
Binary Tries: Summary

- Search(w), prefix-search(w), insert(w), and delete(w) all take $\Theta(|w|)$ 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 $\Theta(n \cdot \text{maximum length of a word})$
 - ↳ what can we do to save space?



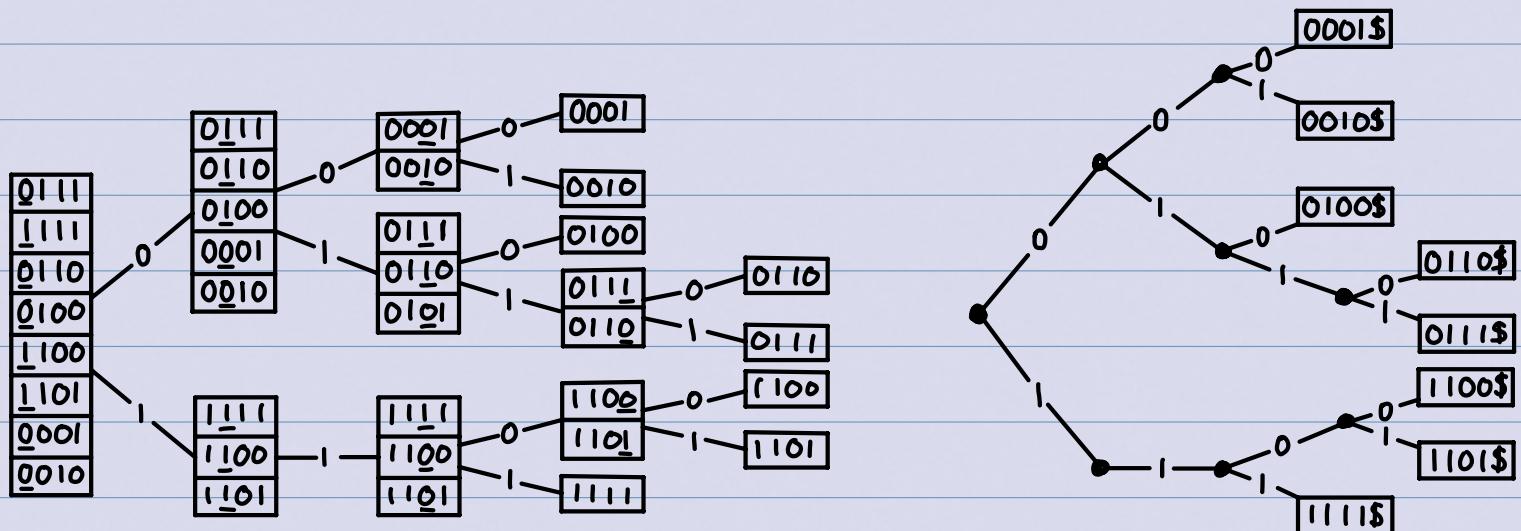
Pruned Trees : Stop adding nodes to trie as soon as the key is unique

↳ 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!

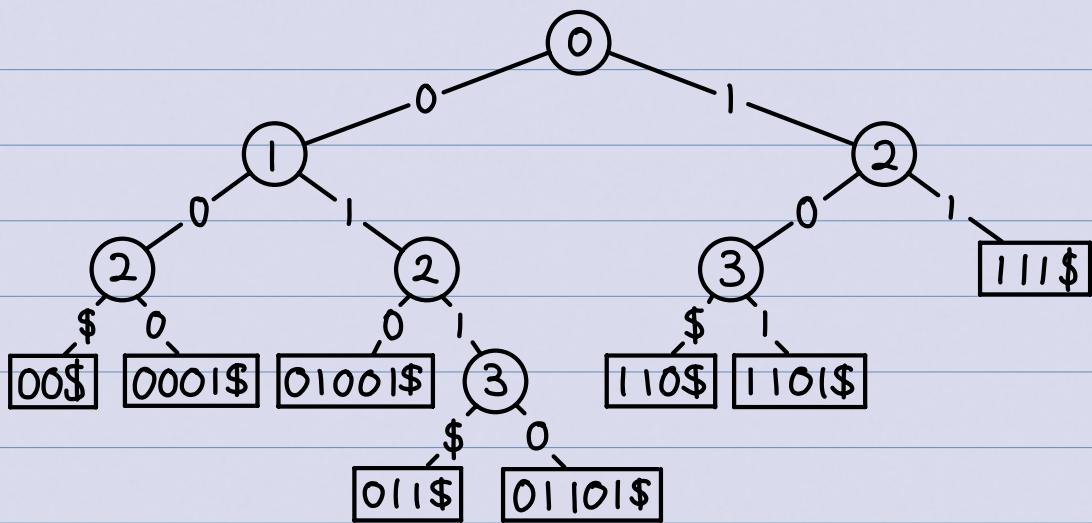


Compressed Tries

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 $\omega[d]$).

Example Compressed Trie:



- **Invariant:** At any node z where $d = z.\text{index}$, 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.

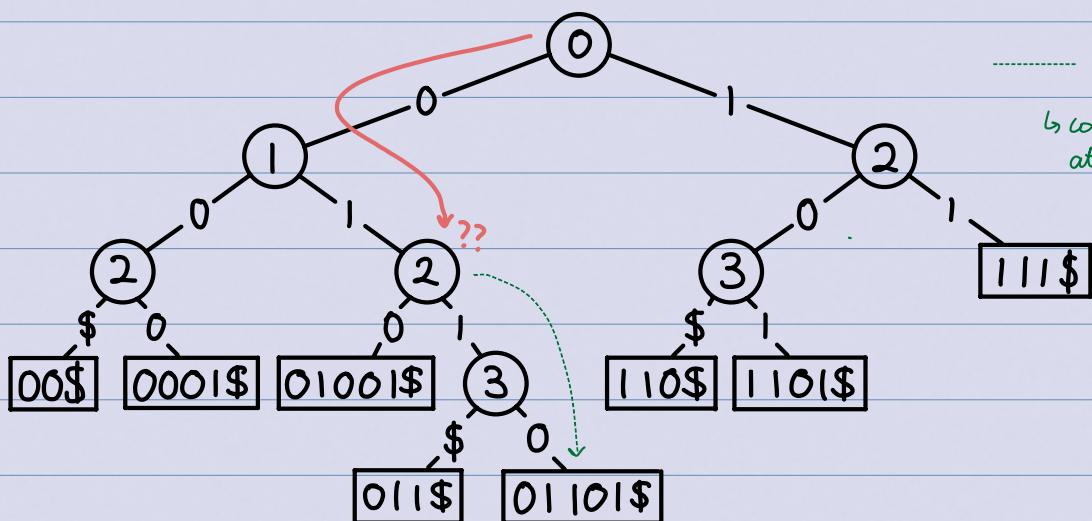
Compressed Trie :: get-path-to(ω)

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

Compressed Trie :: Search(ω)

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

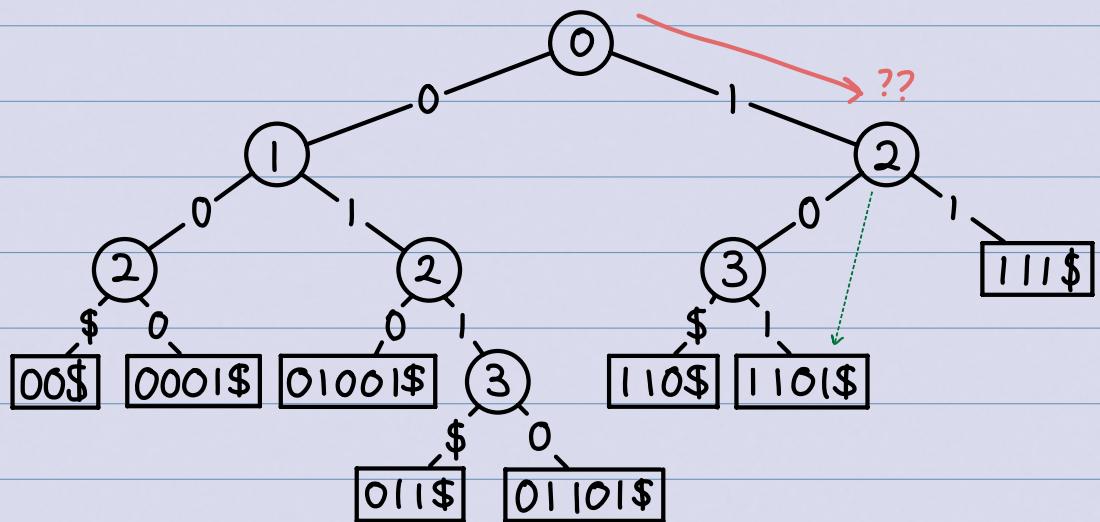
Compressed Tries :: Search ($\boxed{0\ 1\ \$}$)



----- = prefix-search(ω)
↳ compare ω to $z.\text{leaf}()$ at last visited node z .

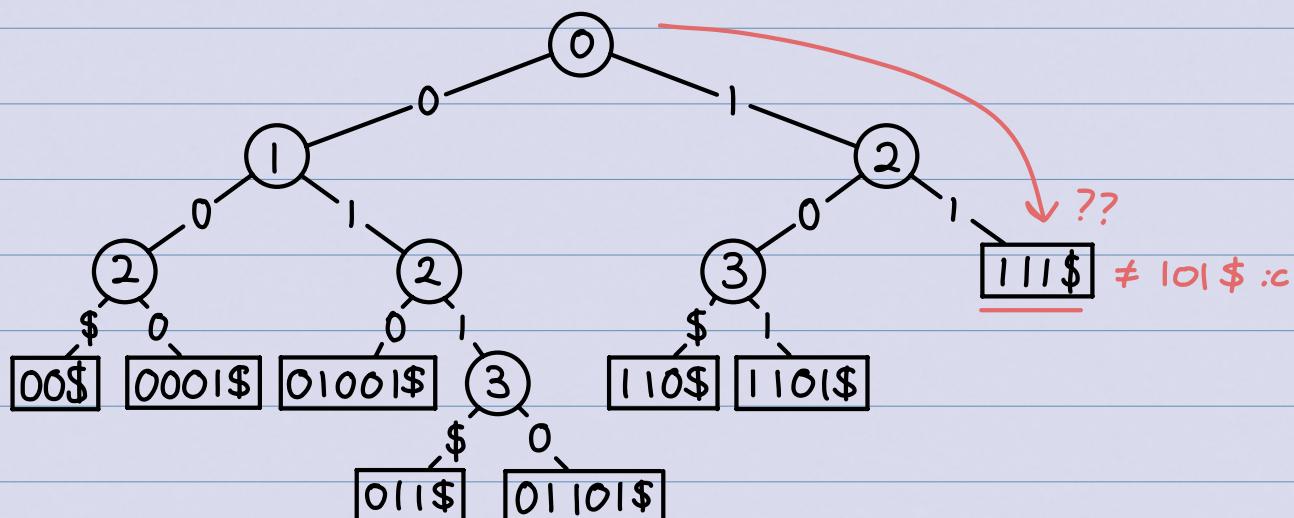
unsuccessful search! (no \$-child)

Compressed Tries :: Search ($\boxed{1\$}$)



unSuccessful! d too big

Compressed Tries :: Search ($\boxed{101\$}$)



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

Search unsuccessful! 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

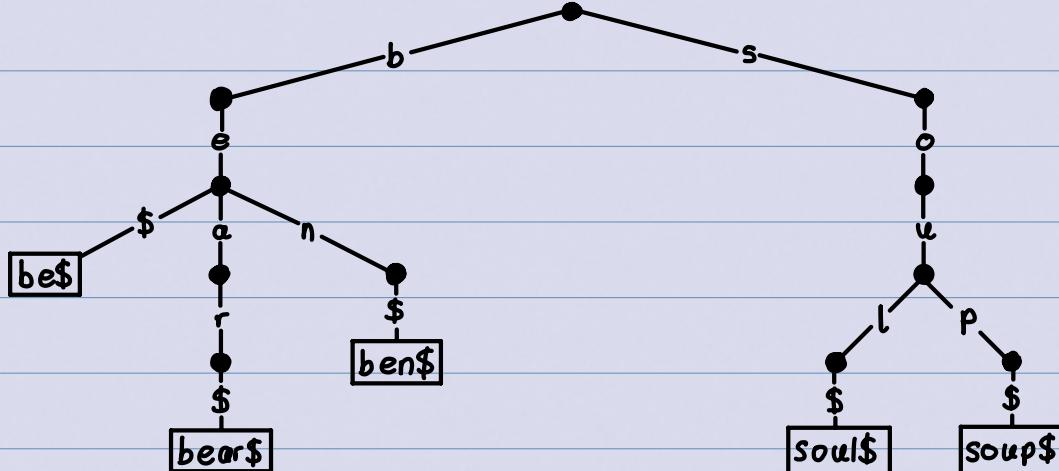
Uncompressing:

- Search for path P to word w (say we reach node z)
- Uncompress this path (using characters of $z.\text{leaf}$)
- 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

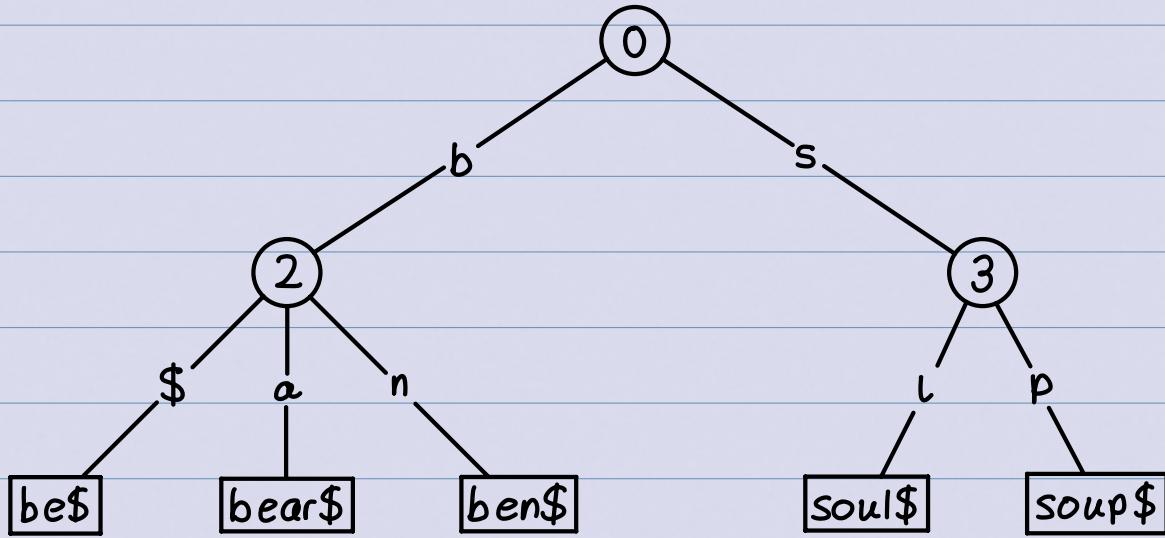
- to represent strings over any fixed alphabet Σ
- any node will have at most $|\Sigma| + 1$ children (one child for the $\$$ character)

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



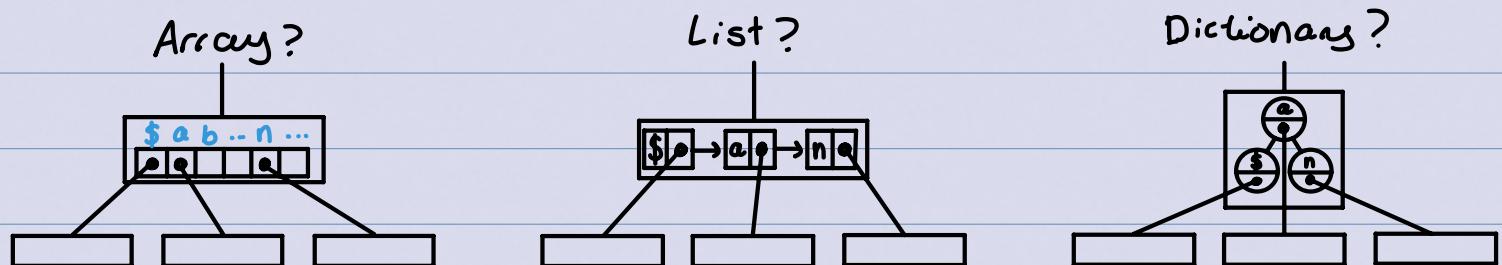
Variation: Compressed multiway tries: compress paths as before!

Example using same Σ :



Multiway Tries: Summary

- Operations $\text{search}(\omega)$, $\text{prefix-search}(\omega)$, $\text{insert}(\omega)$, and $\text{delete}(\omega)$ are exactly as for tries for bitstrings.
- Runtime $O(|\omega| \cdot (\text{time to find appropriate child}))$
- 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!