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:

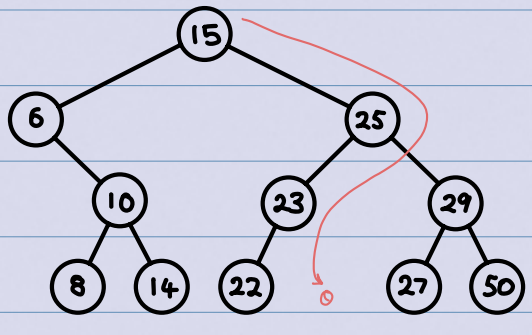
- all nodes have two (possibly empty) subtrees
- every node stores a KVP
- empty subtrees usually not shown

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.

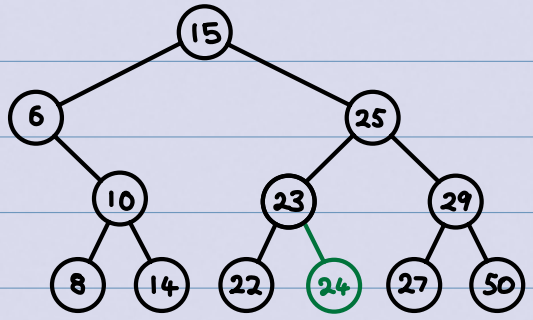
Eg: BST::Search(24):



∴, 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):



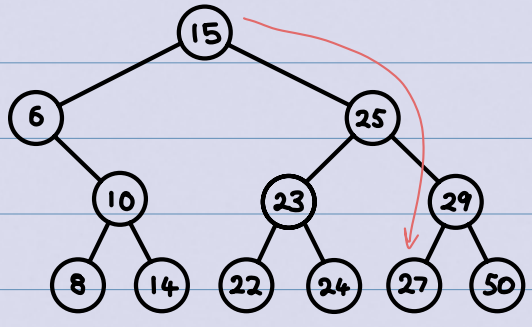
∴, 24 inserted!

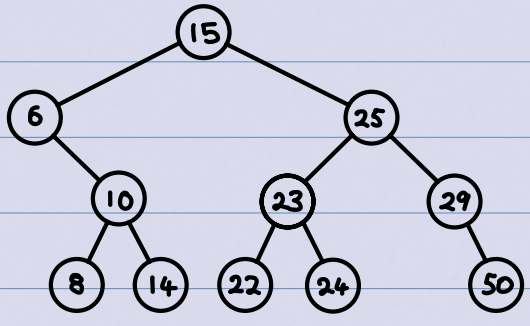
BST:: Delete(k) → first search for the node x that contains the key.

- 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.

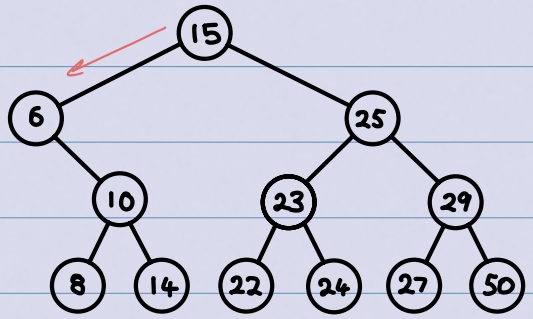
Eg: BST:: Delete(27): (leaf)

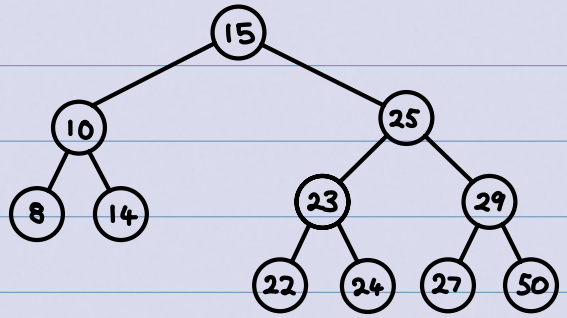




∴ Deleted using case 1

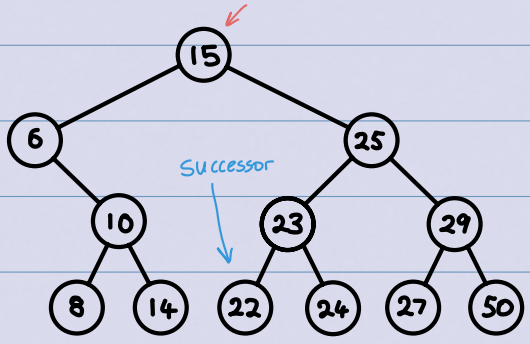
Eg: BST:: Delete(6):

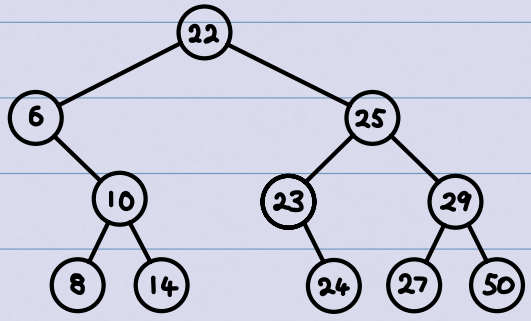




∴ Deleted using case 2

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: $O(\log n)$

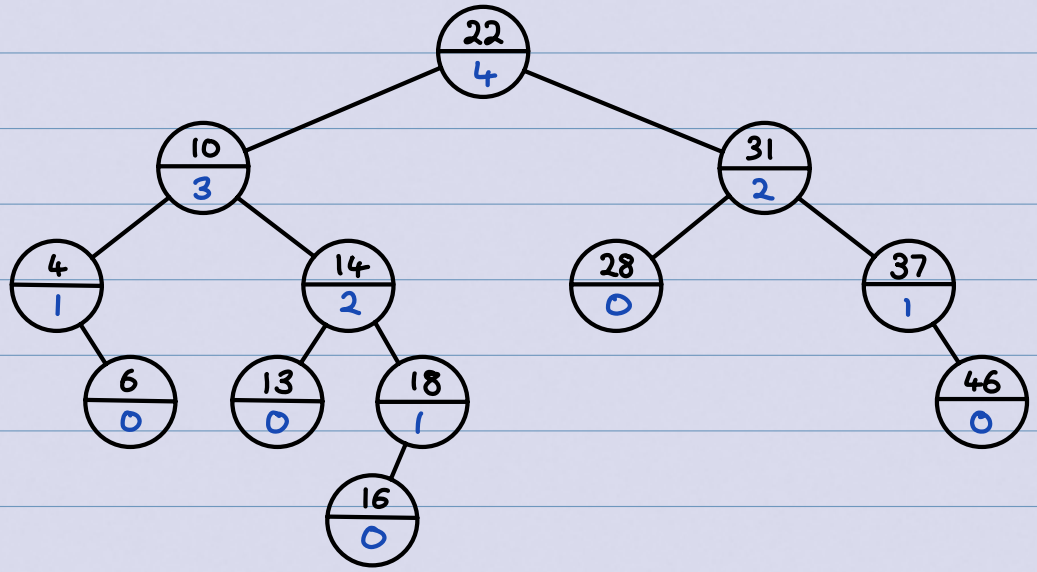
AVL Trees

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

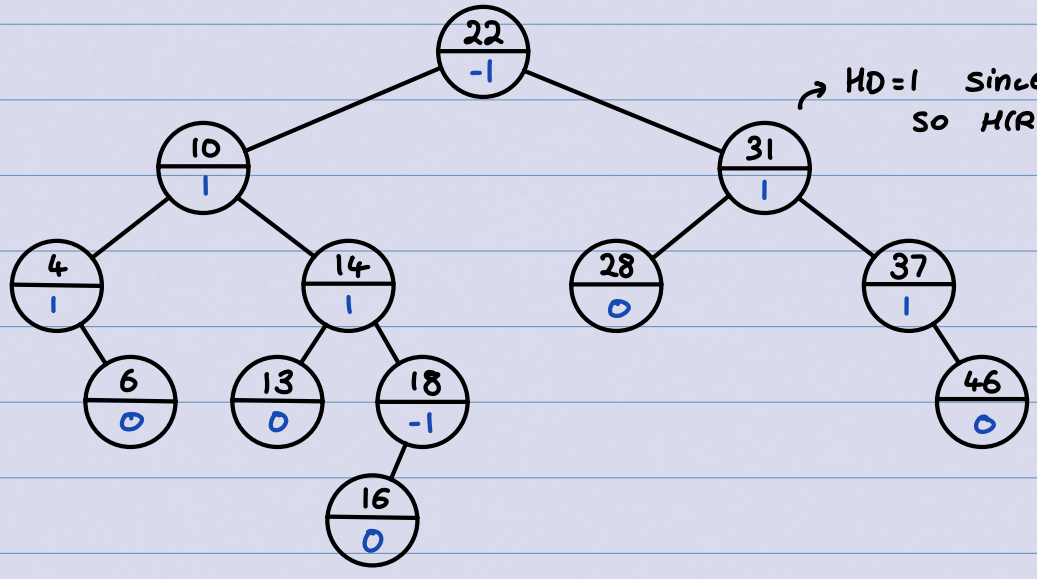
The heights of the left and right subtree differ by at most 1.

↳ 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.

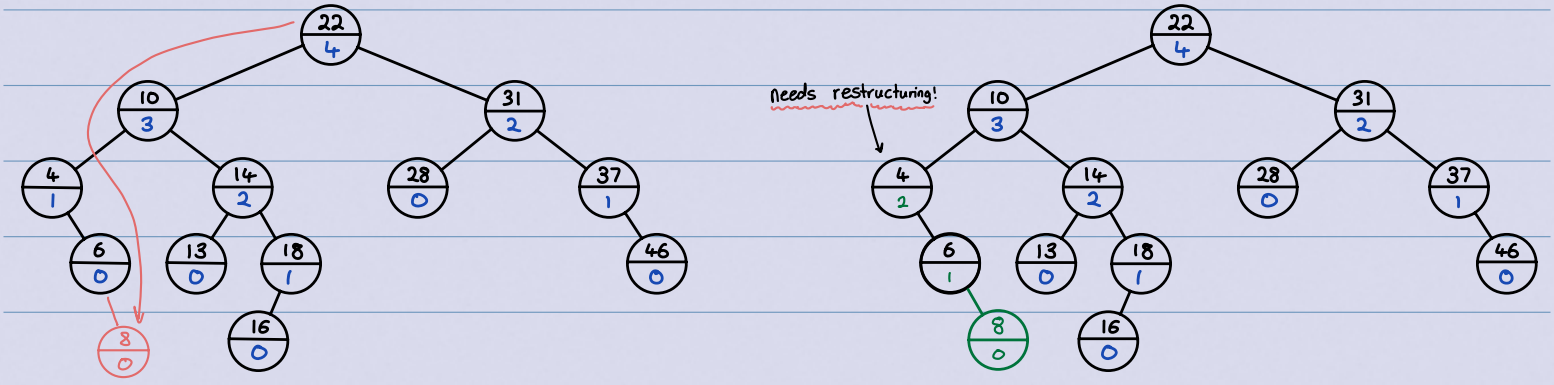
AVL::insert(k,v) → first, insert(k,v) with usual BST

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}

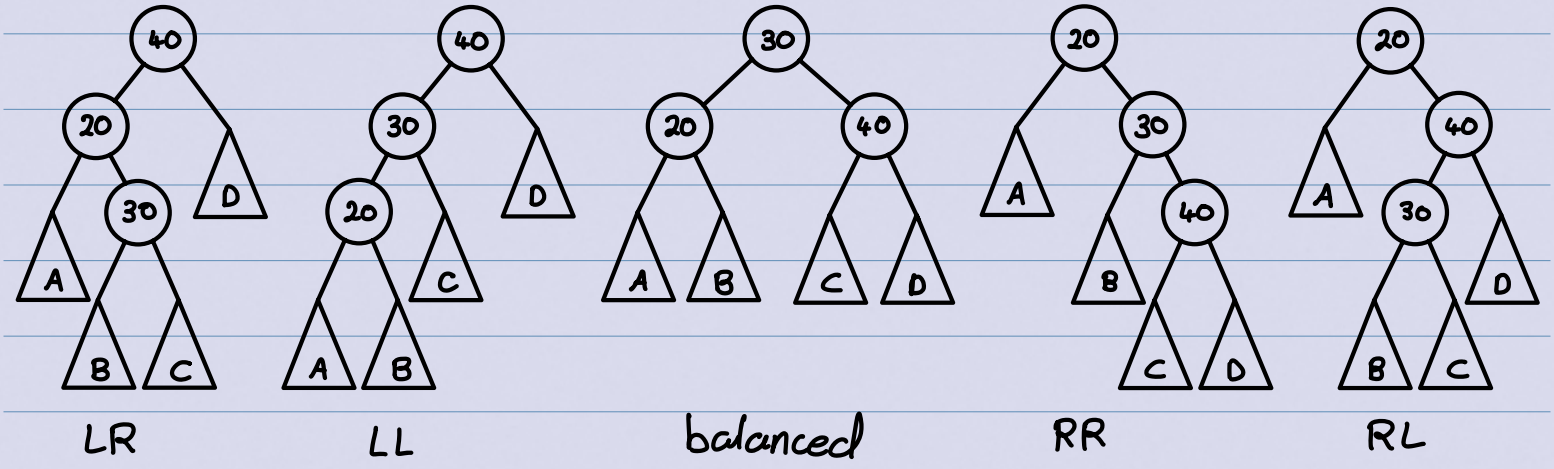
Eg: AVL: insert (8):



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

Restructuring in a BST: Rotations

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

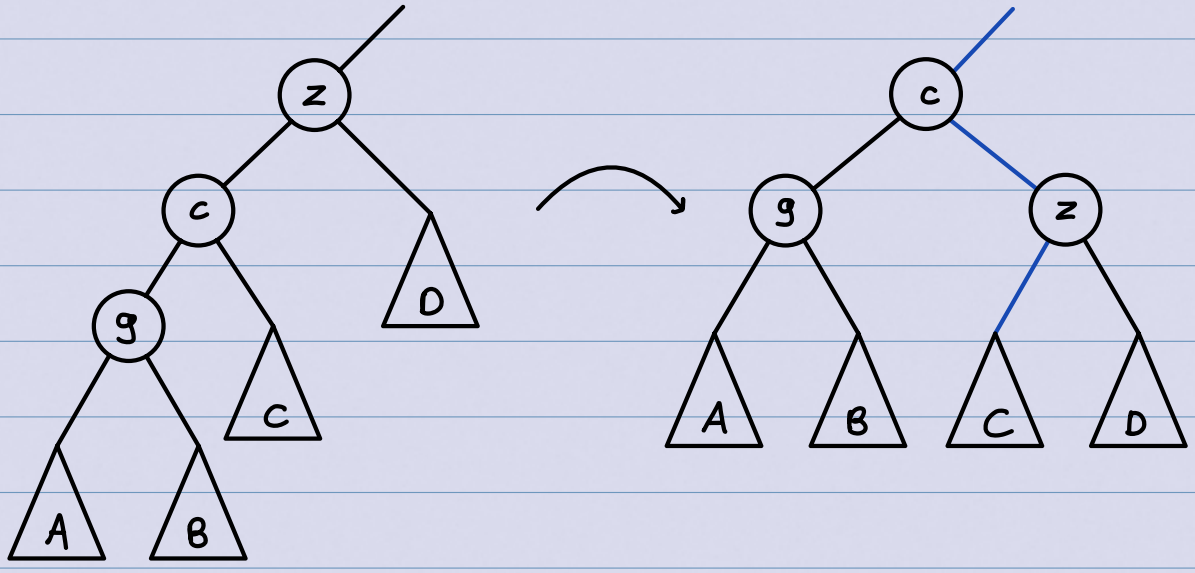


→ goal: change the structure without changing the order, and

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:

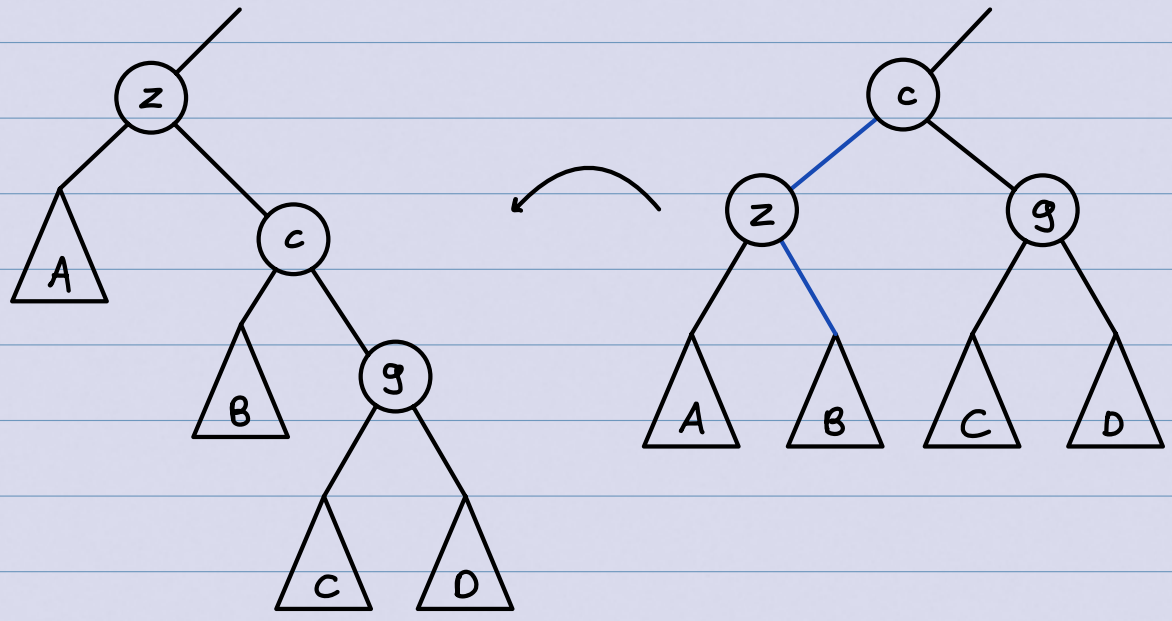
```
rotate-right(z)
1) c = z.left
2) // fix links connecting to above
3) c.parent = (p = z.parent)
4) if (p == null) { root = c }
5) else {
6)     if (p.left == z) { p.left = c }
7)     else { p.right = c }
8) }
9) // actual rotation
10) z.left = c.right,    c.right.parent = z
11) c.right = z,    z.parent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

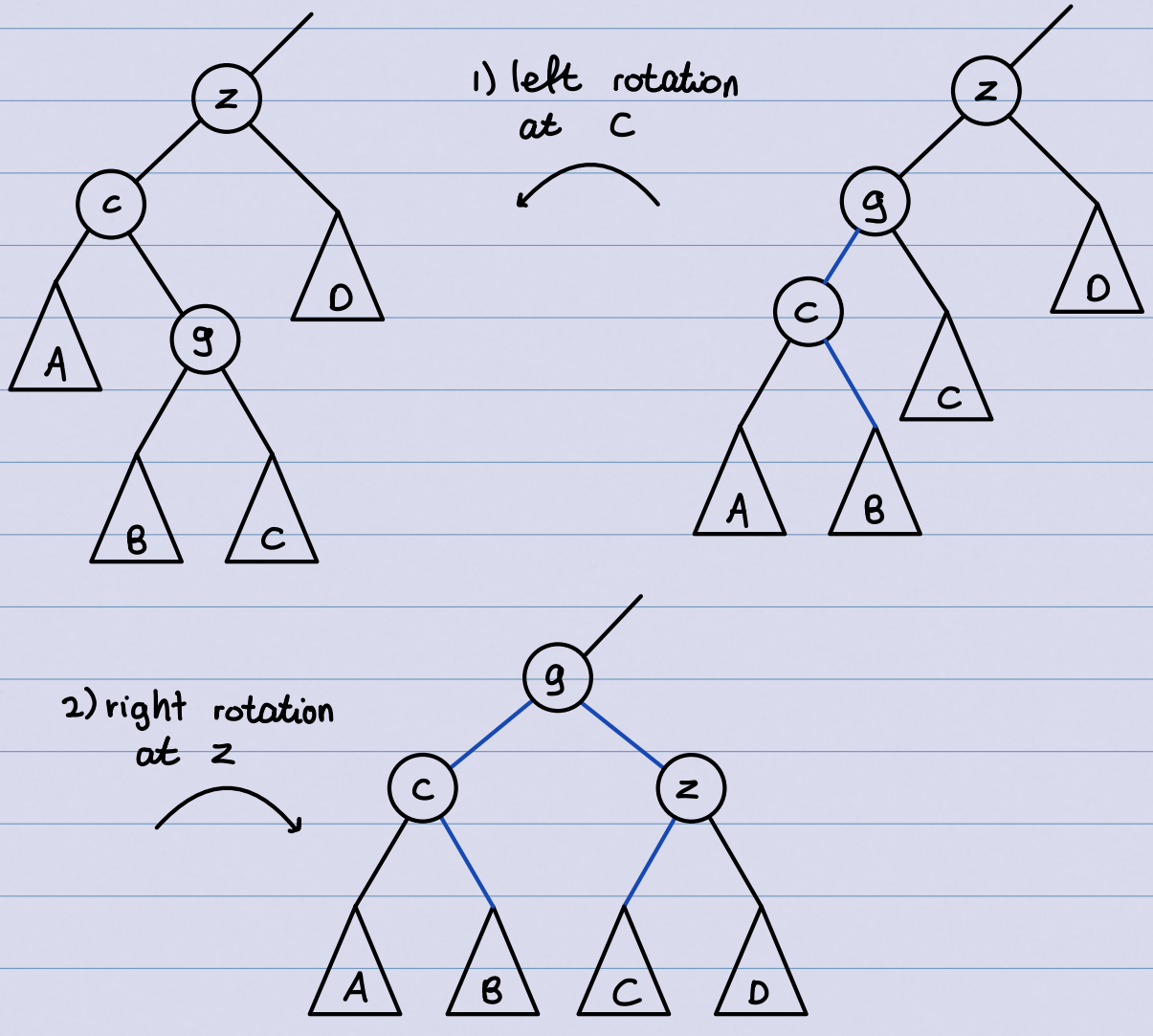
11) c.left = z, z.parent = c

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

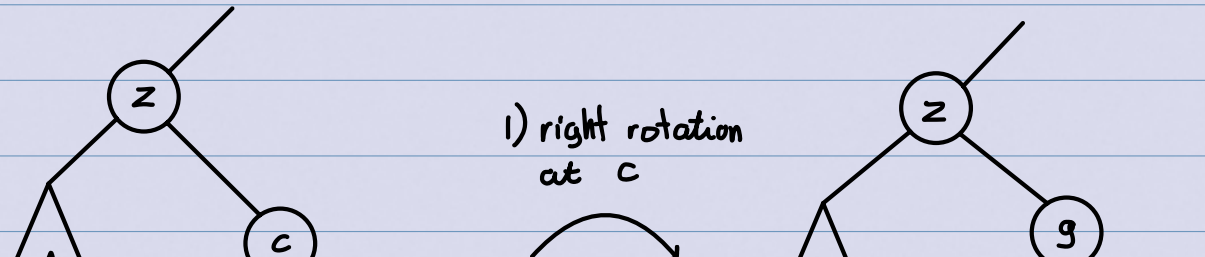
13) return c // returns new root of subtree

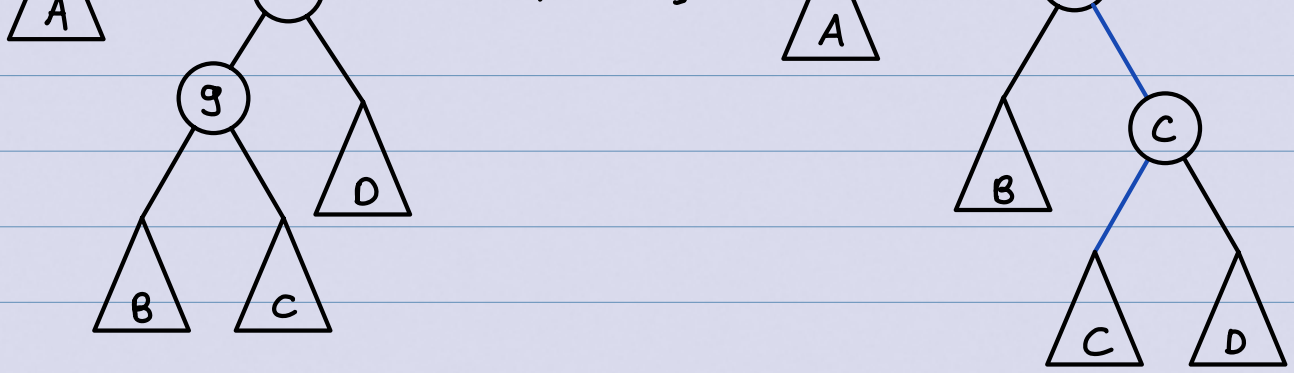
↳ runs in O(1)!

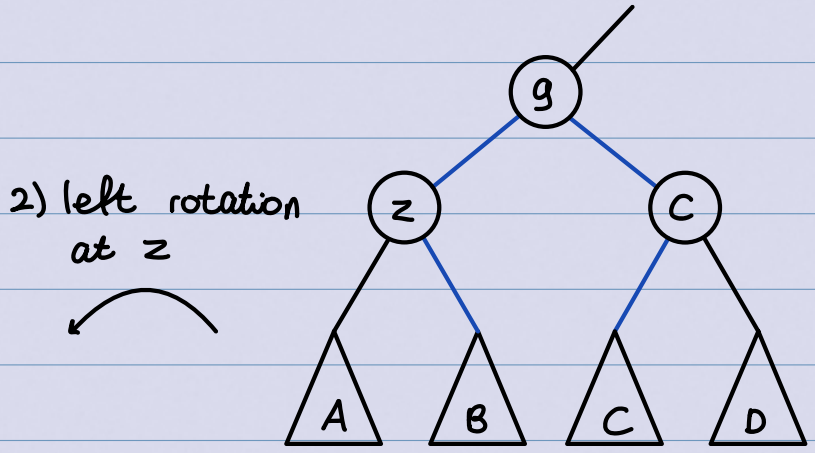
Double Right Rotation



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)

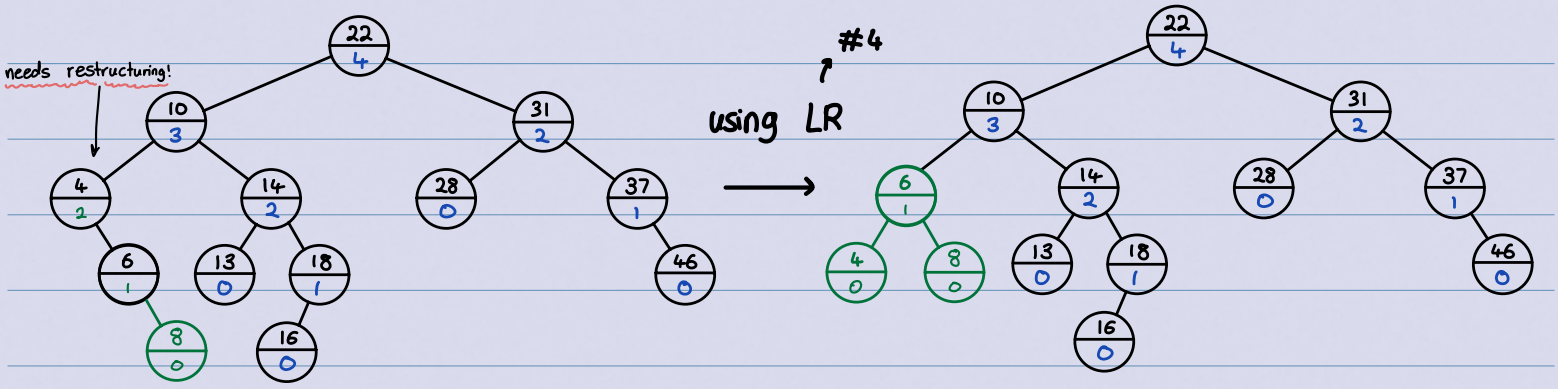
// double-left rotation
rotate-right(c)
u = rotate-left(z)

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

2) return u

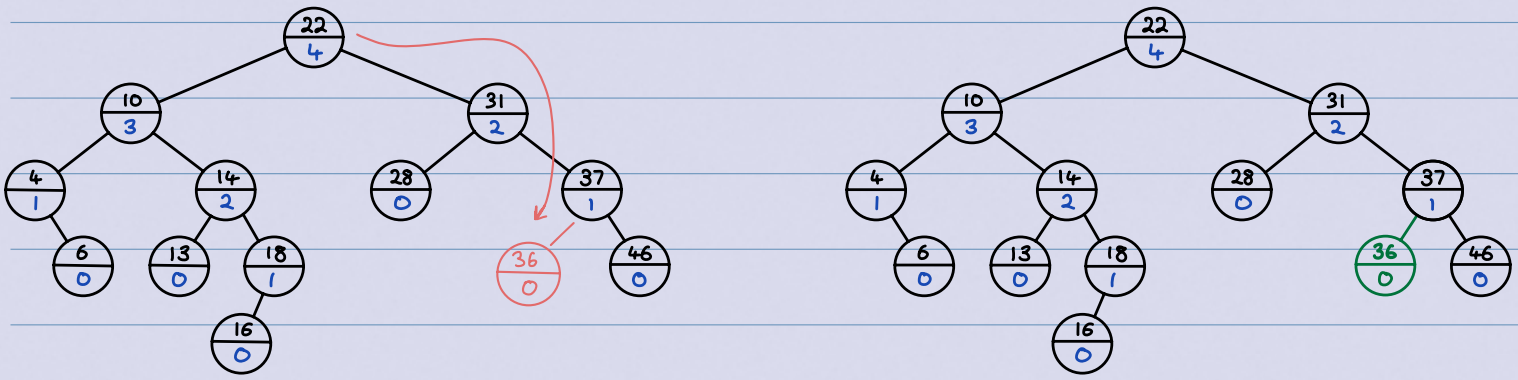
Eg: revisiting AVL::insert(8)

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

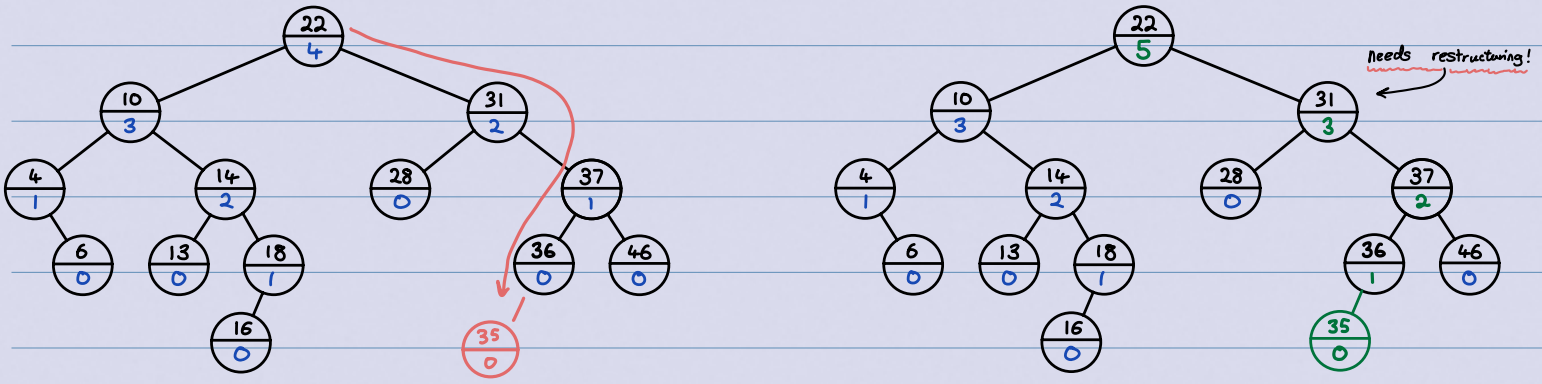


∴, we have correctly restructured the subtree using a left rotation.

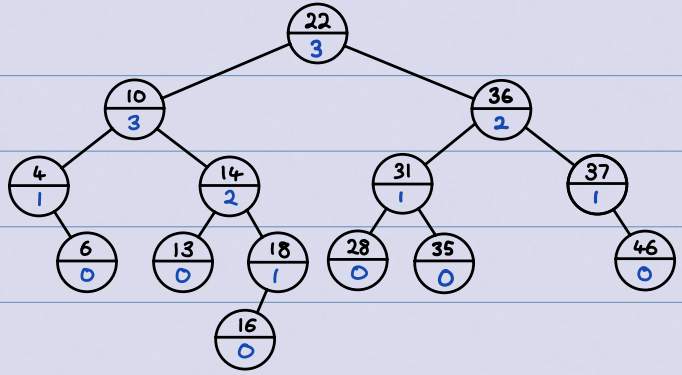
Eg: AVL::insert(35), AVL::insert(36):



→ 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!

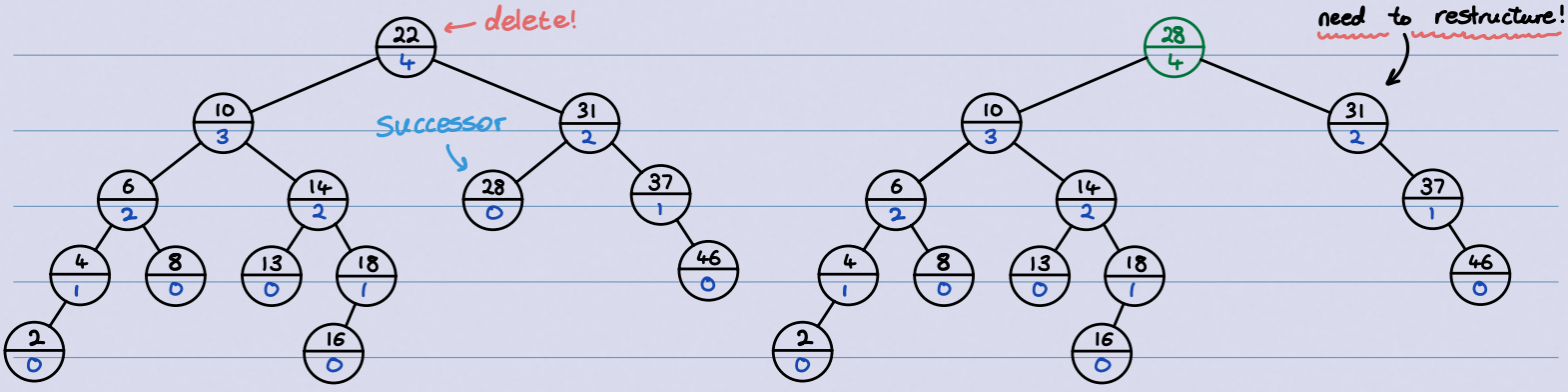
Deletion in AVL Trees

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.

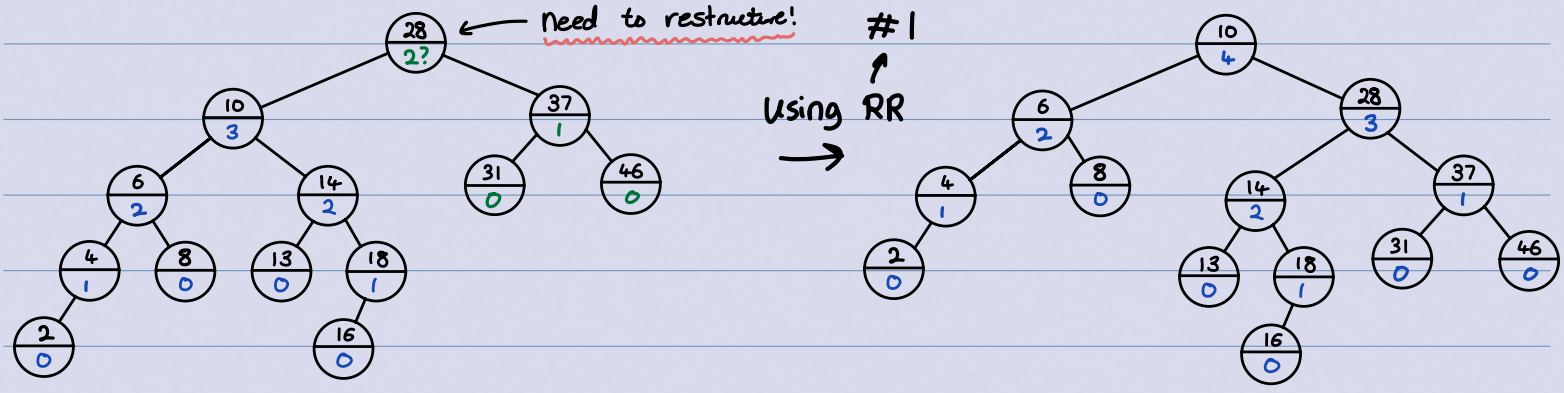
AVL::delete(k)

```
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

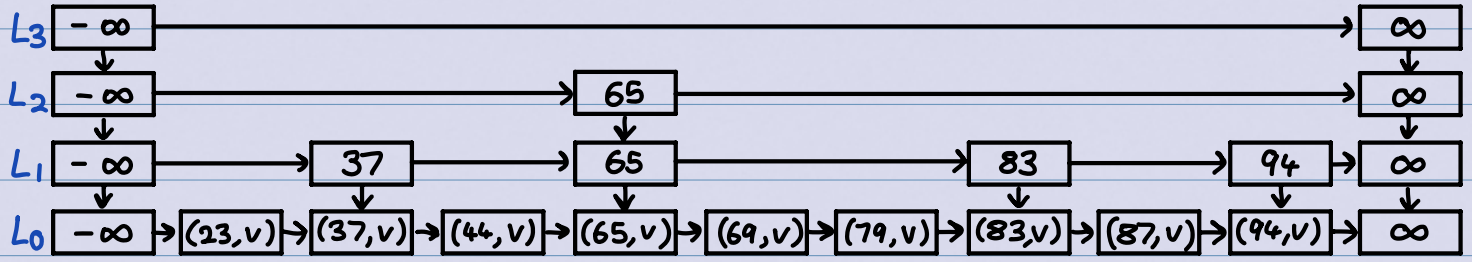
- total cost $\mathcal{O}$(height)
- restructure may be called $\mathcal{O}$(height) times!

- 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

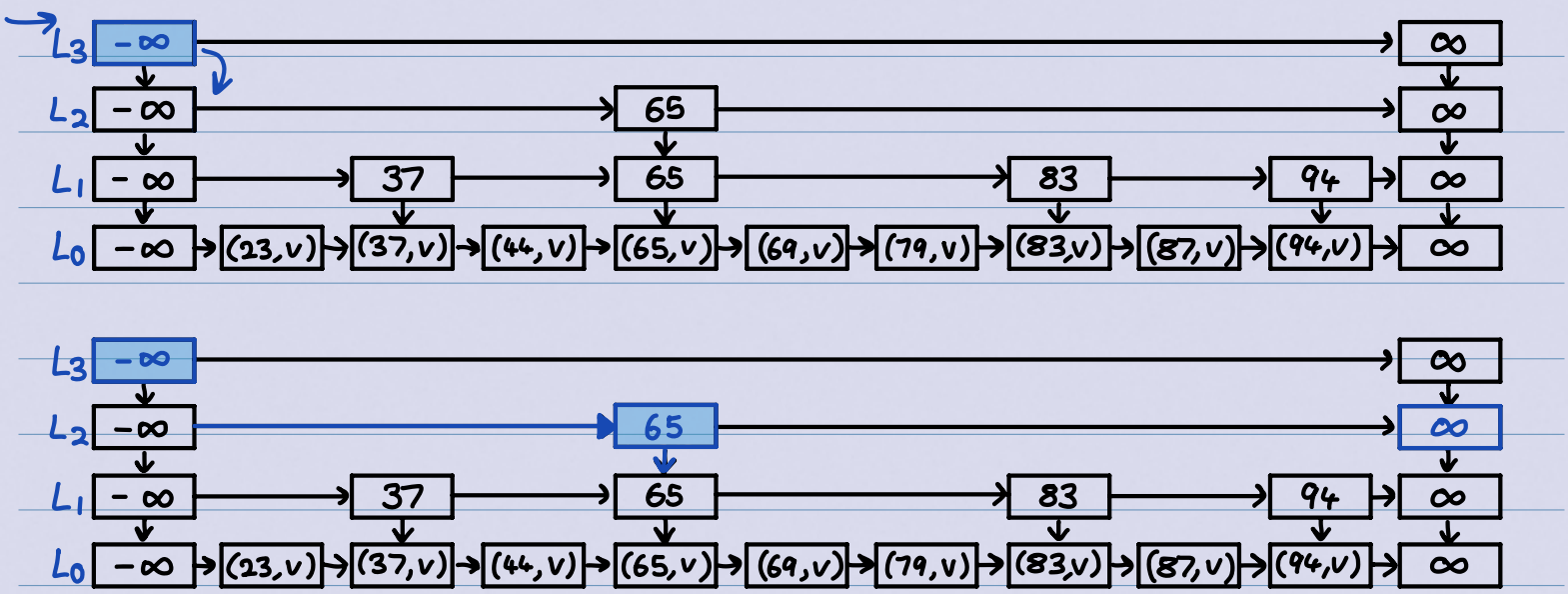
Skip Lists: Search

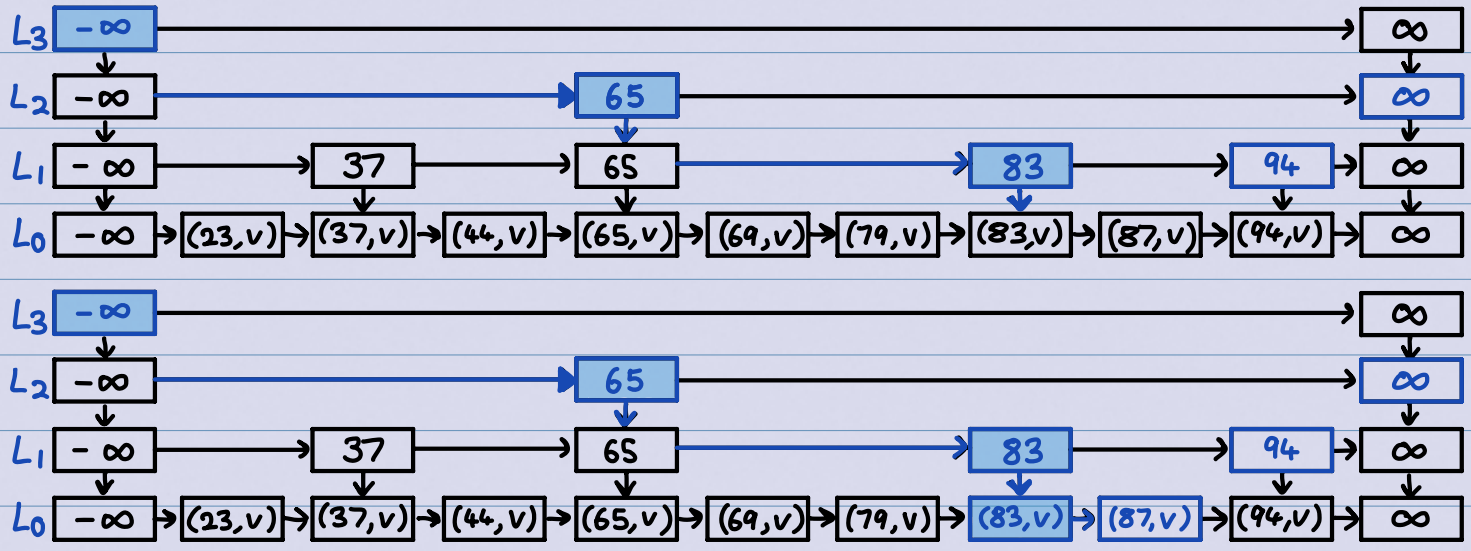
for each list, find predecessor (node before where k would be).

```
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
-∞

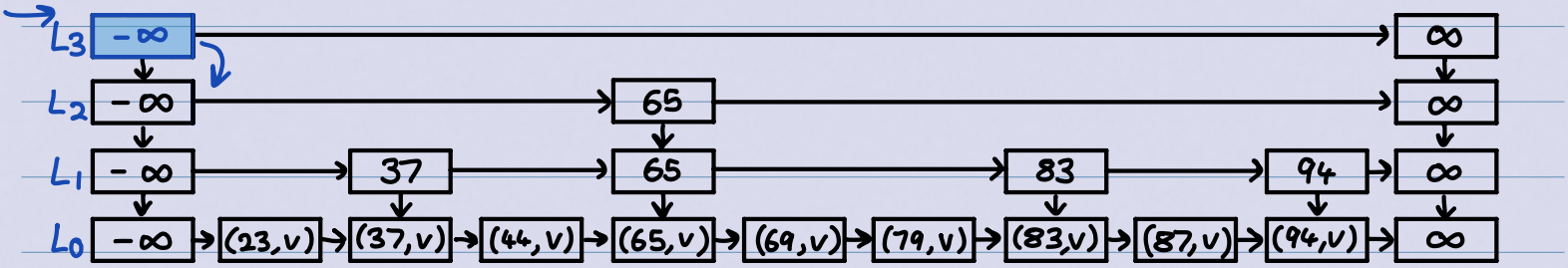
Skip List: Deletion

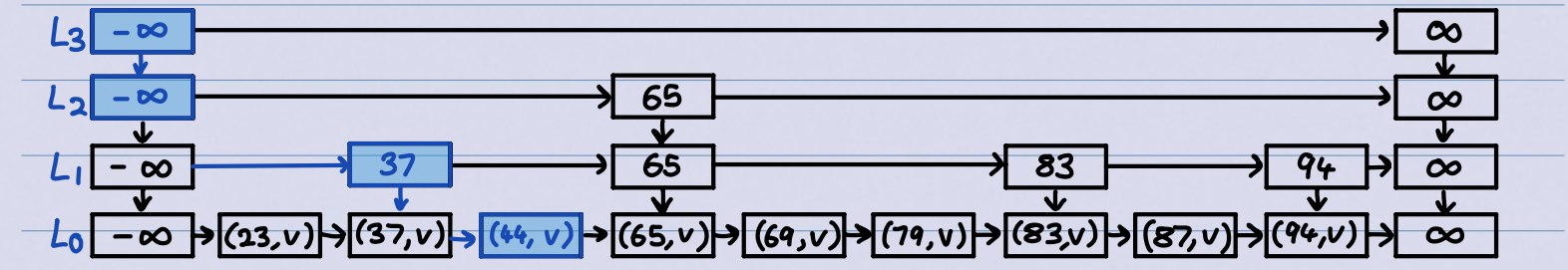
it's easy to remove a key since we can find all predecessors. Then eliminate lists if there are multiple ones will only sentinels.

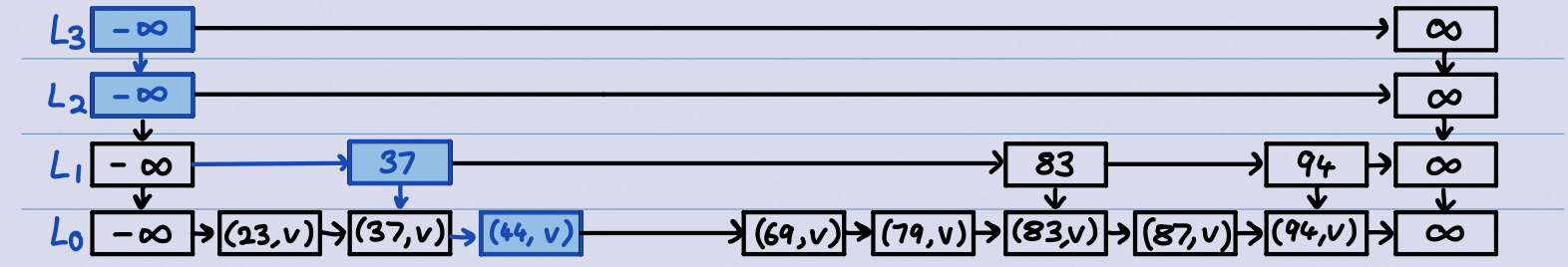
```
skipList:: deletion (k)
1) P = get-predecessors (k)
2) while (P is not empty) {
3)     p = P.pop()  // predecessor of k in some list
4)     if (p.after.key = k) { p.after = p.after.after }
5)     else { break }  // no more copies of k
6) }
7) p = left sentinel of the root list
8) while (p.below.after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

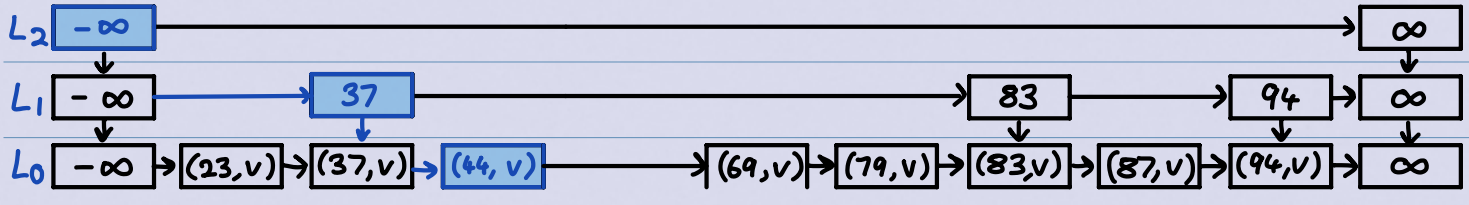
Example: SkipList::delete(65):







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 $1 \leq j \leq i$.

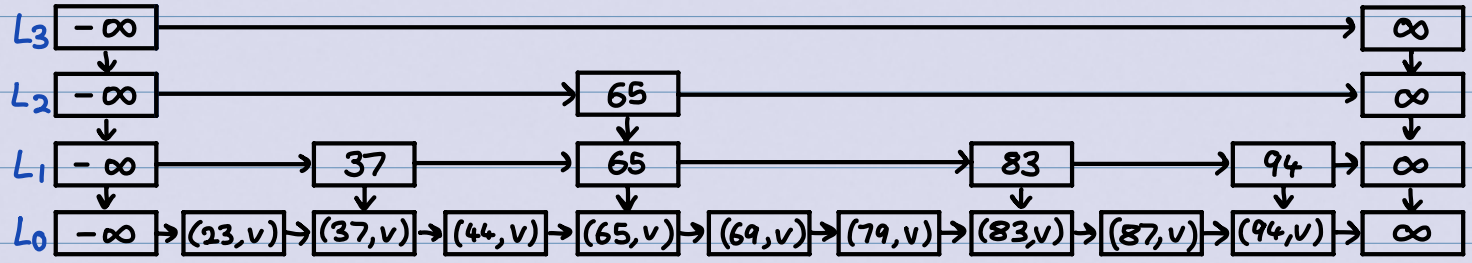
```
SkipList::insert(k, v)
1) for (i=0; random(2)=1; i++) {}   // random tower height
2) for (h=0; p=root.below; p≠null; p=p.below; h++) {}
3) while (i ≥ h) {
4)   create new sentinel-only list; link it in below topmost level
5)   h++
6) }
7) P = get-predecessors(k)
8) p = P.pop()   // insert (k, v) 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

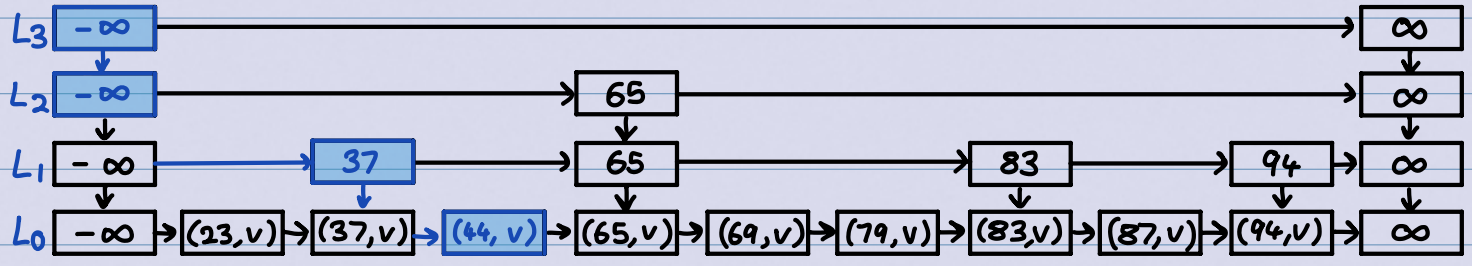
16) }

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

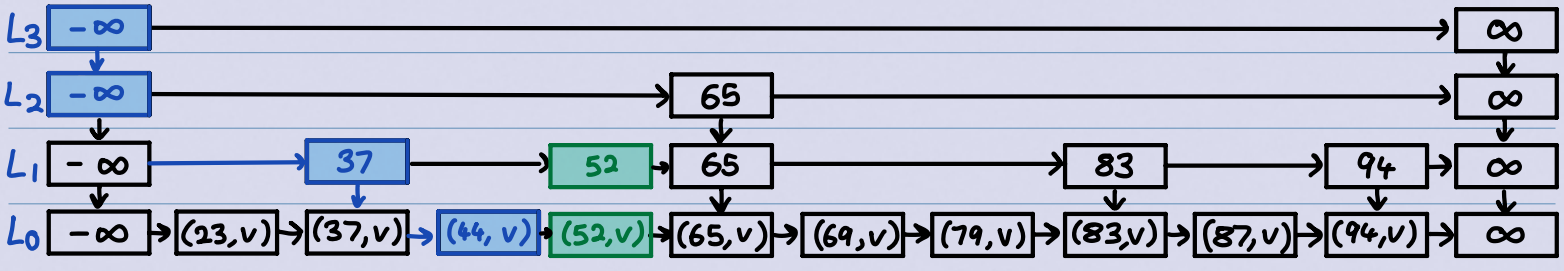


Since h = 3 > i = 1, we don't need to insert any sentinel-only lists!

get-predecessors(52):

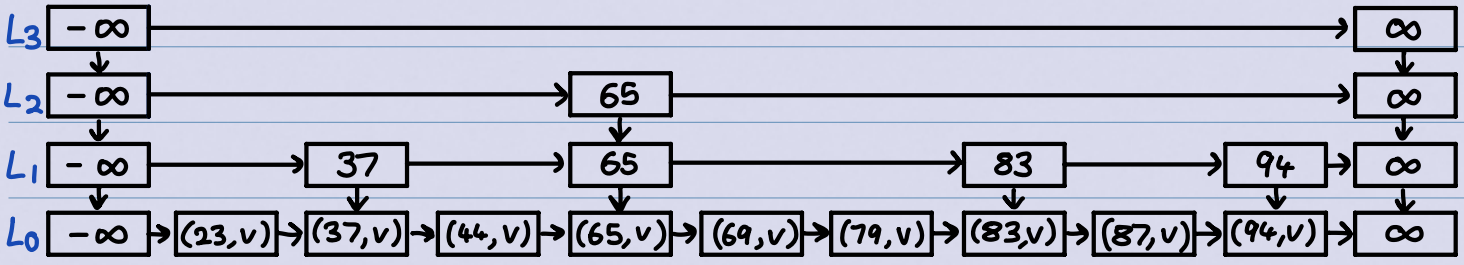


insert 52 in lists $L_0, \ldots, L_i$ (ie, L_0 and L_1):

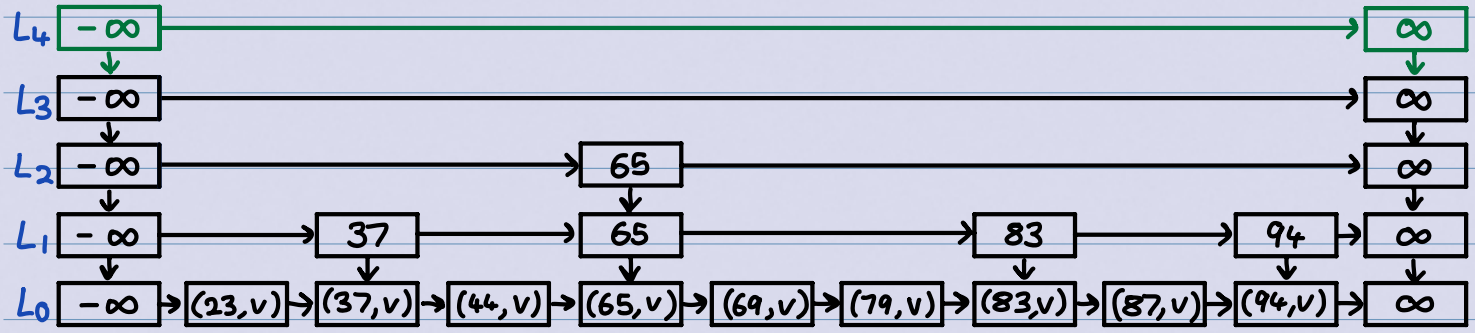


∴, we have succesfully inserted node with key = 52!

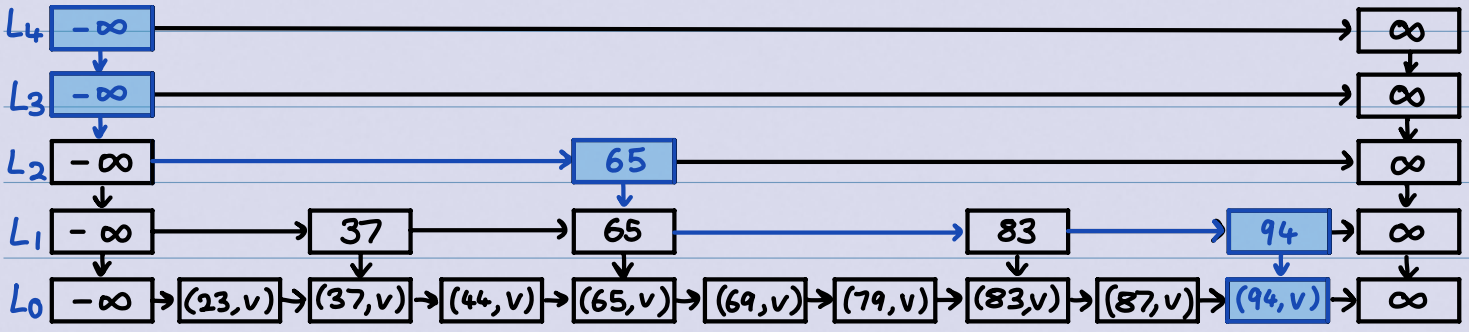
Example (2): SkipList::insert(100, v). Coin tosses: H, H, H, T → i = 3.



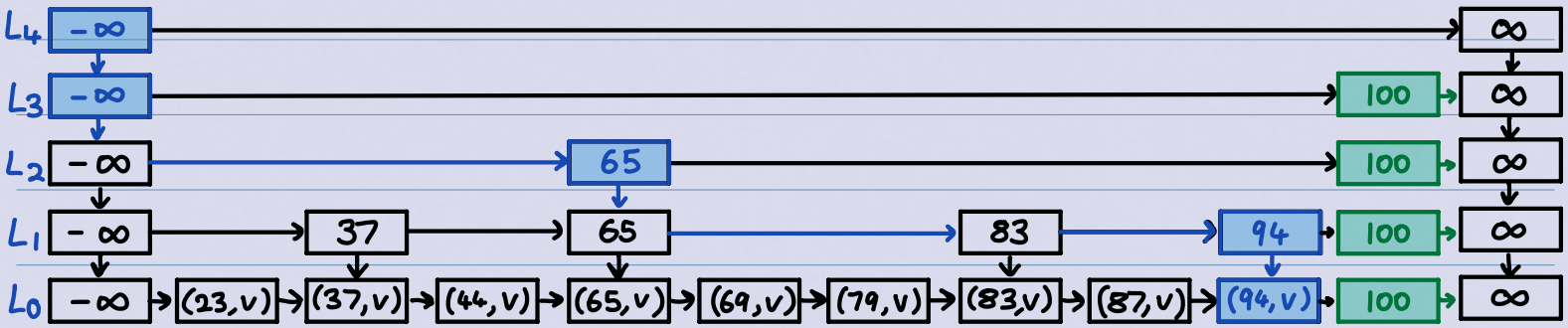
Since $h=3$ is not greater than $i=3$, we must add another level:



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.

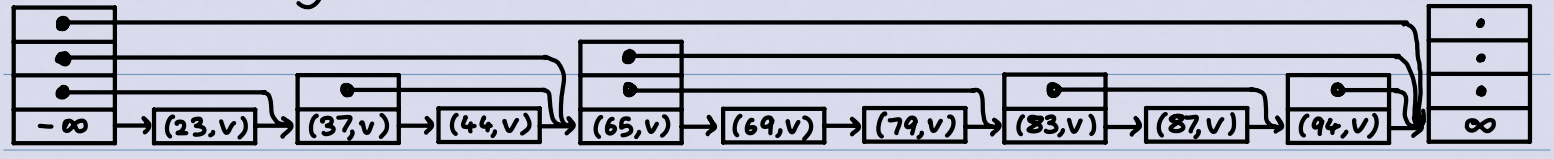
↳ how often do we step forward (execute p=p.after)?

- 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:
$2/26 \cdot 1 + 8/26 \cdot 2 + 1/26 \cdot 3 + 10/26 \cdot 4 + 5/26 \cdot 5 = 86/26 \approx 3.31$

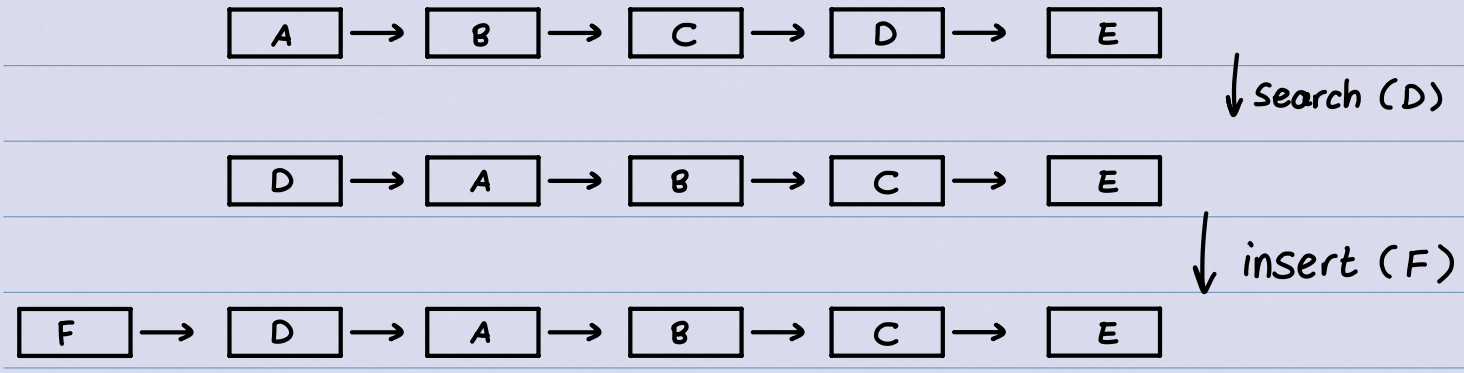
the order [D] → [B] → [E] → [A] → [C] is better!
$10/26 \cdot 1 + 8/26 \cdot 2 + 5/26 \cdot 3 + 2/26 \cdot 4 + 1/26 \cdot 5 = 66/26 \approx 2.54$.

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).

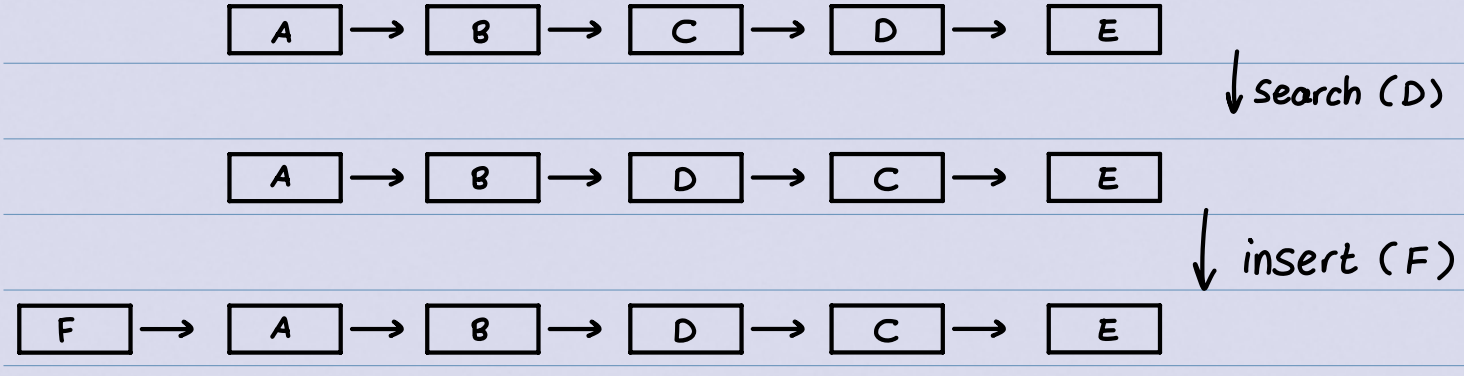
Move-To-Front Heuristic (MTF): upon a successful search, move the accessed item to the front of the list.



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: