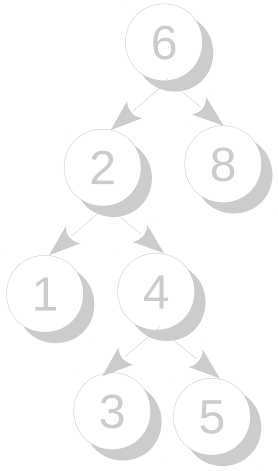
Dictionaries



Rupesh Nasre.
rupesh@iitm.ac.in

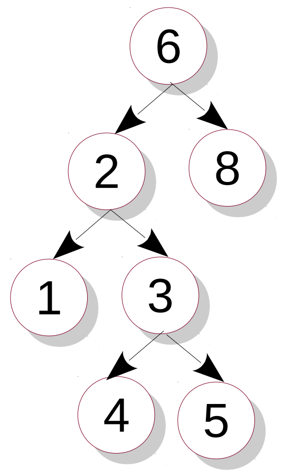
July 2019

Dictionary Instances

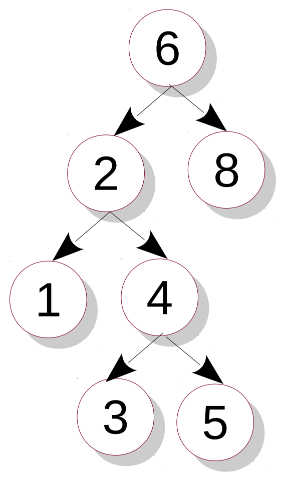
- Binary Search Trees
  - Balanced BST
- Hash Tables

Definition

- A BST is a binary tree.
- If it is non-empty, the value at the root is larger than any value in the left-subtree, and
- the value at the root is smaller than any value in the right-subtree.
- The left and the right subtrees are BSTs.
- Assumption: All values are unique.



Not a BST



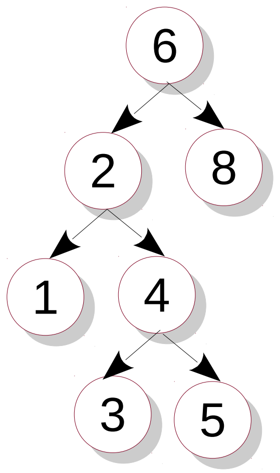
BST

BST ADT

```
class BST {
        …
public:
        …
        PtrToNode search(DataType element);
        PtrToNode findMin();
        PtrToNode findMax();
        PtrToNode insert(DataType element);
        void remove();
};
```

Search

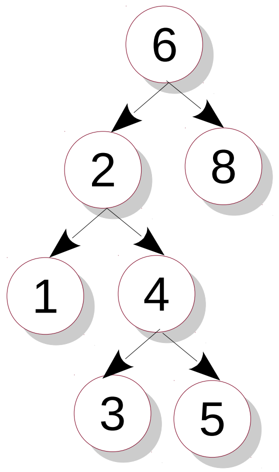
```
bool Tree::search(DataType data, PtrToNode rr) {
	if (rr == NULL) return false;
	if (data == rr->data) return true;
	if (data < rr->data) return search(data, rr->left);
	return search(data, rr->right);
}
bool Tree::search(DataType data) {
	bool present = search(data, root);
	if (present)
		std::cout << data << " present." << std::endl;
	else
		std::cout << data << " NOT present." << std::endl;
	return present;
}
```



Switch to bst.cpp.

FindMin

```
DataType Tree::findmin(PtrToNode rr) {
        if (rr) {
                if (rr->left) return findmin(rr->left);
                return rr->data;
        }
        return -1;
}
DataType Tree::findmin() {
        return findmin(root);
}
```

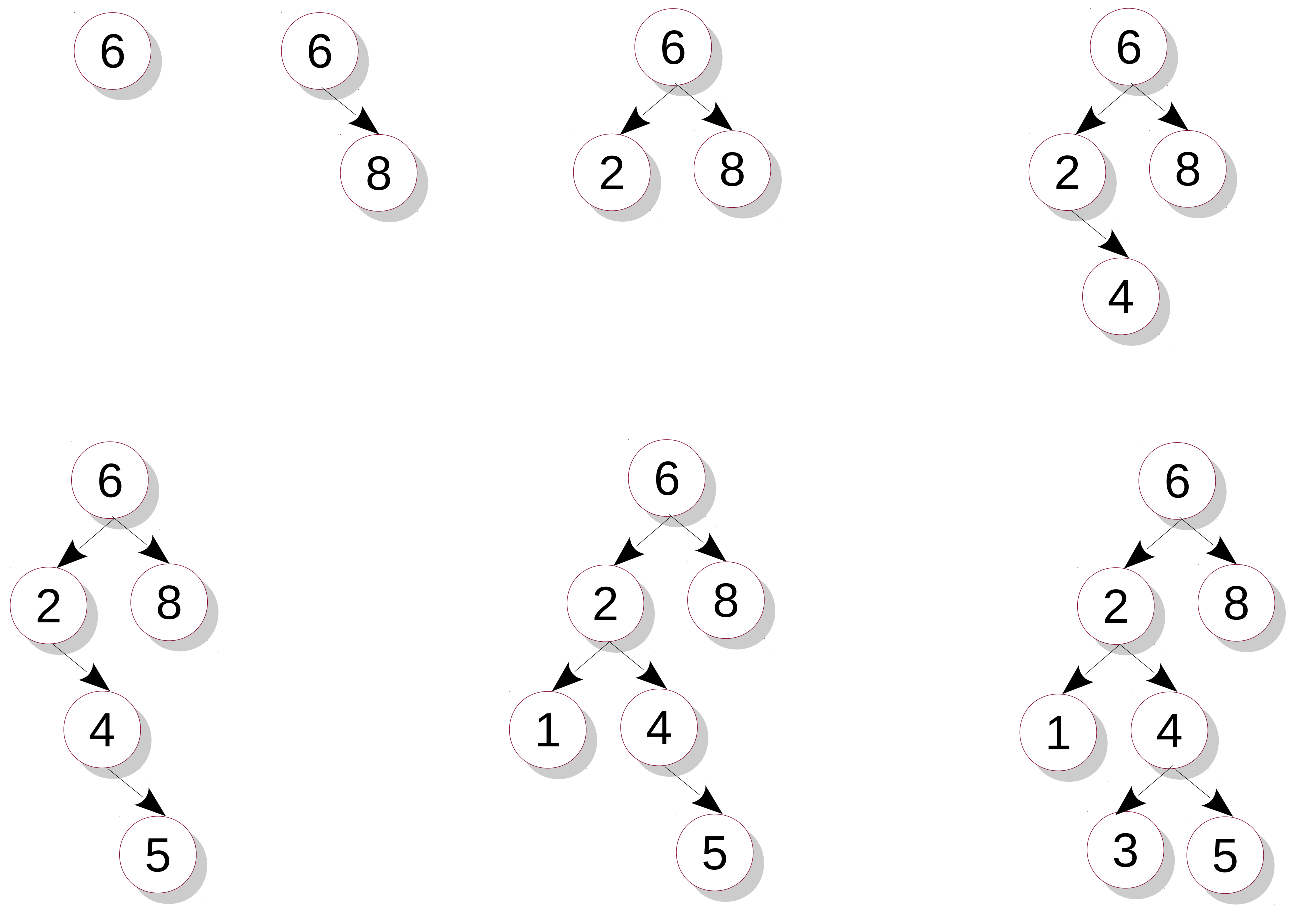


Write iterative findMax.

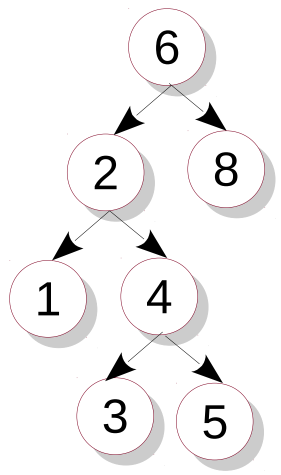
```
DataType Tree::findminiterative() {
        PtrToNode ptr = root;
        if (ptr) {
                while (ptr->left) ptr = ptr->left;
                return ptr->data;
        }
        return -1;
}
```

Insert

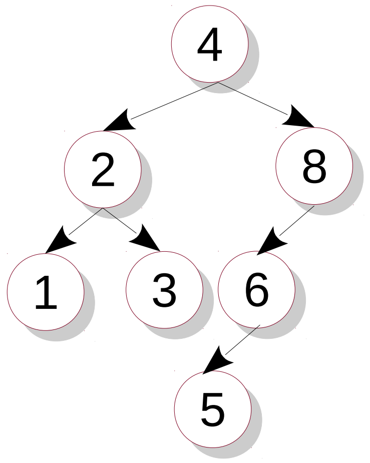
Insert 6, 8, 2, 4, 5, 1, 3



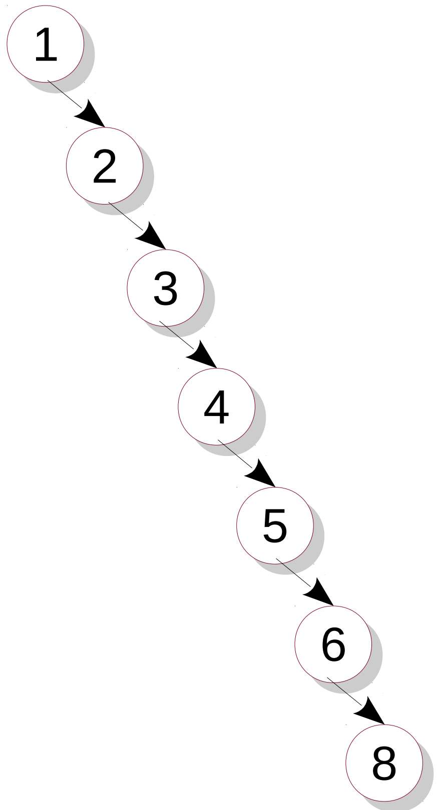
Insert 6, 8, 2, 4, 5, 1, 3



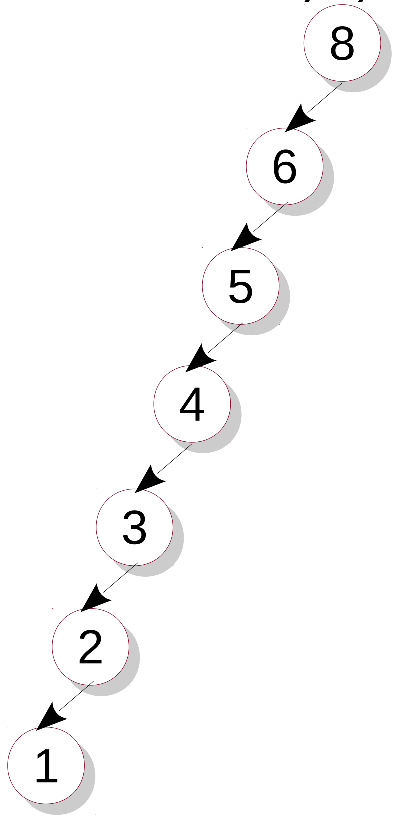
Insert 4, 8, 2, 6, 5, 1, 3



Insert 1, 2, 3, 4, 5, 6, 8



Insert 8, 6, 5, 4, 3, 2, 1



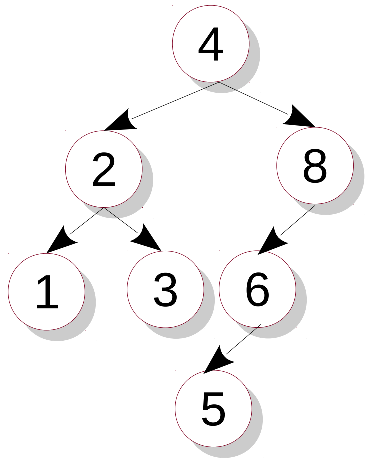
Perform inorder traversal on the last BST.

Insert

- Insertion order may change the tree structure.
  - Different insertion orders may form the same tree.
- Inorder traversal prints the values in exactly the same order, irrespective of the BST structure.
  - This is also the sorted order.
- The first insertion forms the root.
- Insertions always happen at leaves.
  - New node cannot be added as an intermediate node.
- Insertion order decides the tree height.
  - Tree height affects efficiency/complexity of operations.

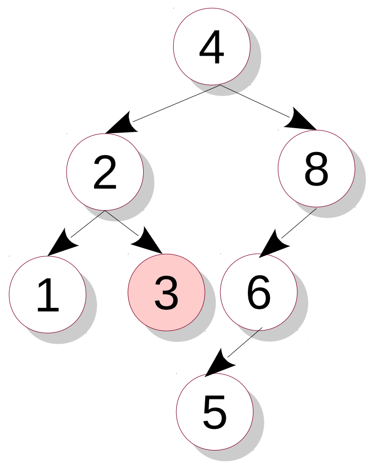
Insertion Orders

- For this BST, find three different insertion orders.
  - 4, 2, 8, 1, 3, 6, 5
  - 4, 8, 2, 6, 1, 3, 5
  - 4, 2, 1, 3, 8, 6, 5
  - ...

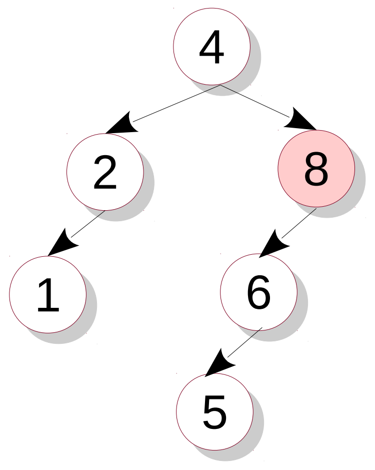


Remove(value)

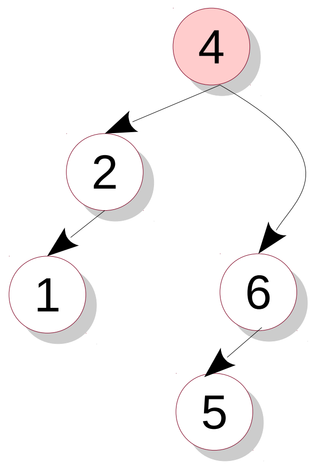
- Search for the node **n** to be removed.
- If **n** is a **leaf**, remove **n** from its parent.
- If **n** has **one** child **c**, make **c** the child of **n**'s parent.
- If **n** has **two** children, scratch your head.



Remove(3)



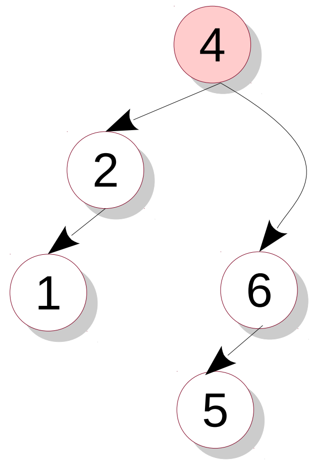
Remove(8)



Remove(4)

Remove(value)

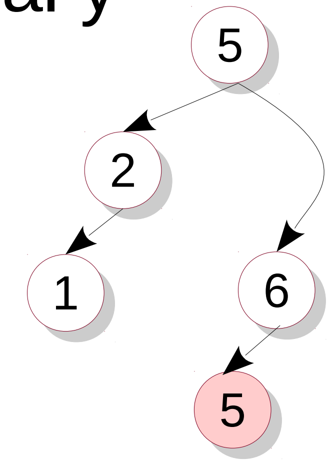
- We would like to convert this complicated remove into another simpler remove.
  - That is, convert this case of two children into a case of one child or zero children.
  - For instance, remove(4) can be converted to remove(5) or remove(6) or remove(2) or remove(1).
  - Which one would be the best, in general?
- **General strategy:**
  - Copy the smallest value from the right subtree here.
  - Recursively delete that smallest value.



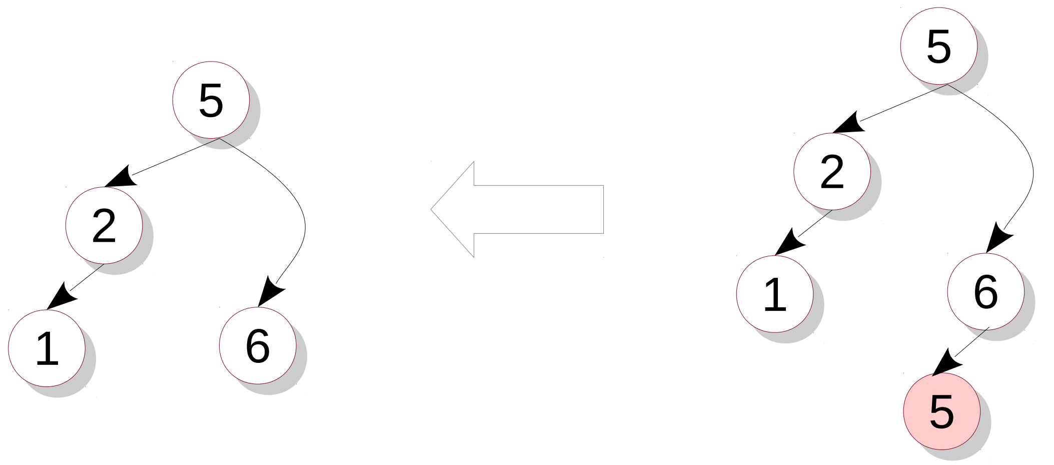
Remove(4)

Guarantees on the smallest value

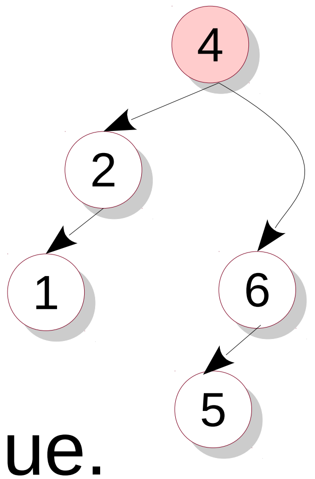
- If x is the value to be deleted, and y is the smallest value in its right subtree,
  - There are no values in the BST between x and y.
  - The node with y value cannot have two children.
  - In fact, y cannot have a left child.
  - When y replaces x, after removing original y node, the BST structure is not affected.
  - Removal of a node with two children does not result in further removal of another node with two children.



Remove(value)



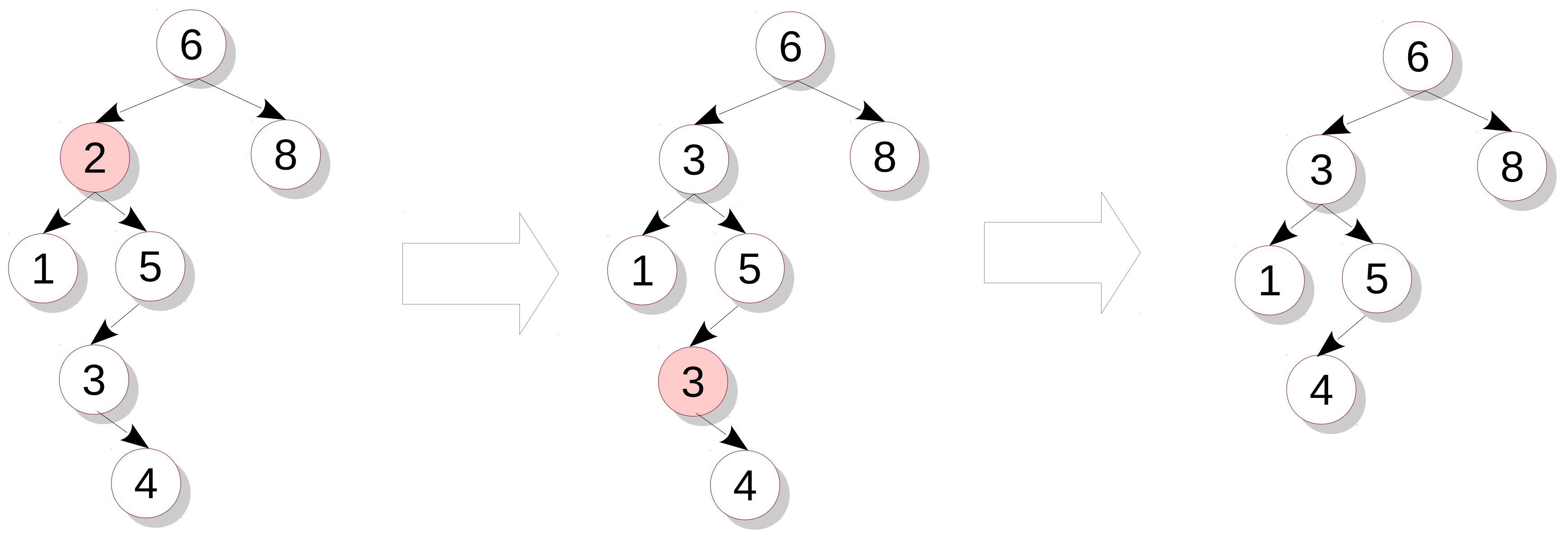
Remove(5)



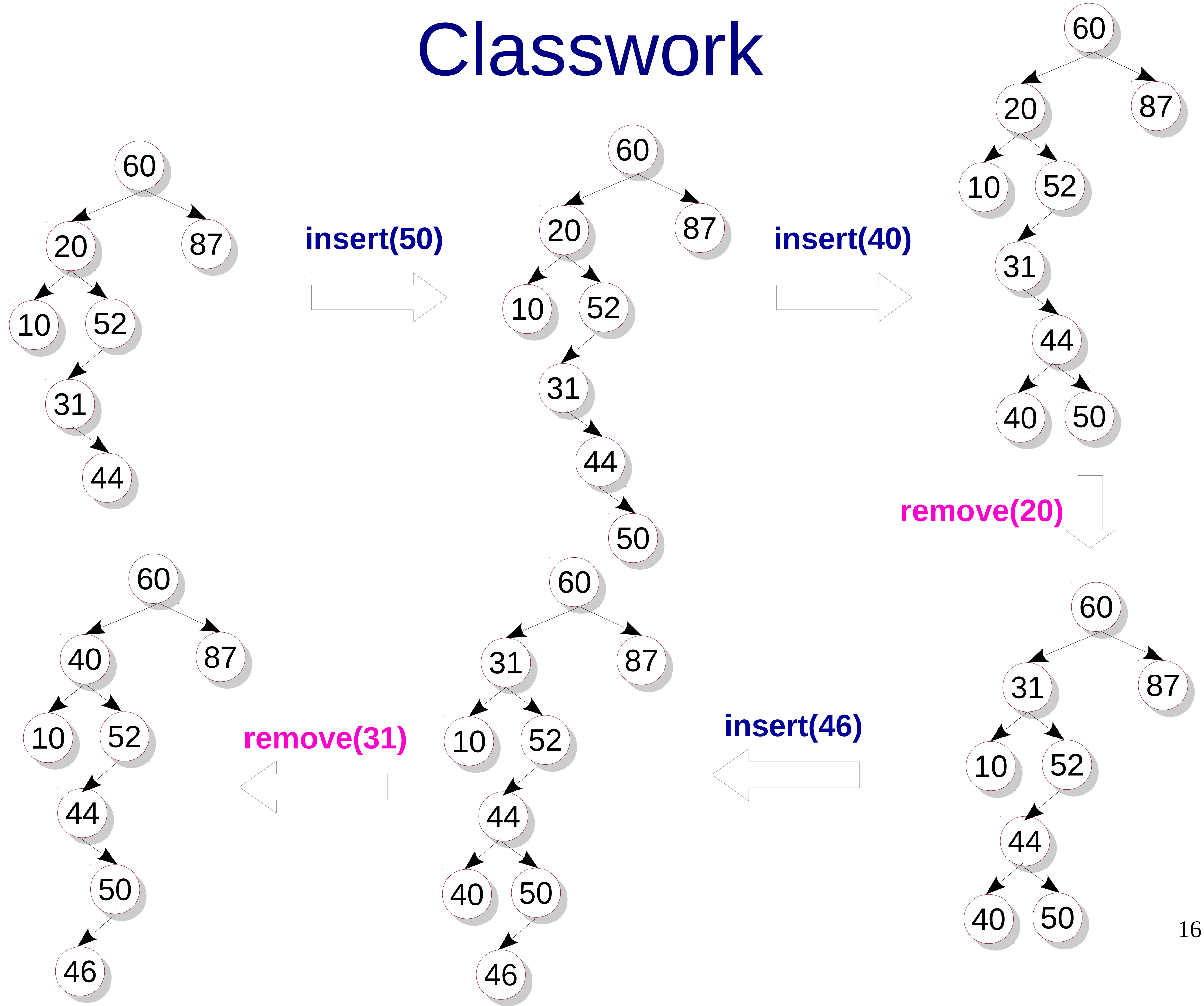
Remove(4)

- **General strategy:**
  - Copy the smallest value in the right subtree here.
  - Recursively delete that smallest value.

Remove(value)



Classwork

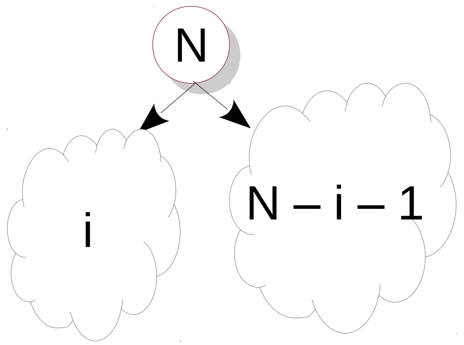

insert(50)
insert(40)
remove(20)
insert(46)
remove(31)
60
20
87
10
52
31
44
50
40
46

Time Complexity

- **Search**
  - One may get tempted to conclude it to be O(log n).
  - But it is O(tree height), which could be O(n).
- **Insert**
  - Same as that of search.
- **Remove**
  - There may be two remove calls.
  - Still the complexity does not change. It is same as that of search.
- All the complexities improve if the BST is height-balanced (AVL trees, Splay Trees, red-black trees, B trees, …).

Average Case Analysis

- Let's calculate the average height of the BST when elements are inserted in random order.
- Let $h(n1)$ denote height of a node $n1$.
- Let H(N) denote the sum of the heights of all the nodes in the N-node BST.
  - $H(N) = \sum_{i=0}^{N-1} h(i)$



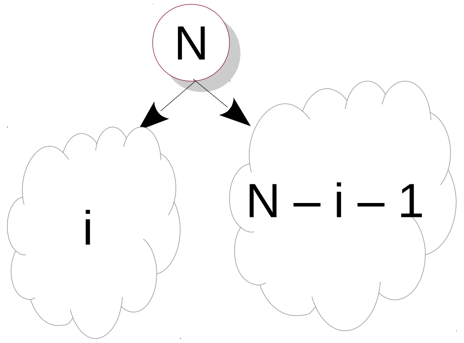
- H(N) = H(i) + H(N – i – 1) + N – 1
  - For each node in left and right subtrees, height increases by 1. Hence N – 1.

Average Case Analysis

- $H(N) = H(i) + H(N - i - 1) + N - 1$
- If the trees are random, each subtree height is equally likely.
- Thus, average value of $H(i)$ and $H(N - i - 1)$ is

  $$\frac{1}{N} \sum_{j=0}^{N-1} H(j)$$

- $H(N) = N - 1 + \frac{2}{N} \sum_{j=0}^{N-1} H(j)$



Average Case Analysis

$$H(N) = N - 1 + 2/N \sum_{j=0}^{N-1} H(j)$$

$$N\,H(N) = N\,(N - 1) + 2 \sum_{j=0}^{N-1} H(j) \quad \ldots\ldots\ldots\ldots (1)$$

Replacing N by N – 1

$$(N-1)\,H(N-1) = (N-1)\,(N - 2) + 2 \sum_{j=0}^{N-2} H(j) \ldots\ldots\ldots (2)$$

(1) – (2) gives

$$N\,H(N) - (N-1)\,H(N-1) = 2N - 2 + 2H(N-1)$$

Rearranging and ignoring 2

$$N\,H(N) = (N + 1)H(N - 1) + 2N$$

Average Case Analysis

$N\,H(N) = (N + 1)H(N - 1) + 2N$

Dividing by N(N+1)

$H(N) / (N+1) = H(N-1) / N + 2 / (N + 1)$ …… (3)

$H(N-1) / N = H(N-2) / (N-1) + 2 / N$ …… (4)

…

$H(2) / 3 = H(1) / 2 + 2 / 3$ …… (5)

Adding (3), (4), …, (5)

$$H(N) / (N+1) = H(1) / 2 + 2 \sum_{i=3}^{N+1} 1/i$$

Average Case Analysis

$$H(N) / (N + 1) = H(1) / 2 + 2 \sum_{i = 3}^{N + 1} 1/i$$

$$H(N) / (N + 1) = O(\log N)$$

Thus, $H(N) = O(N \log N)$

- This indicates that sum of the heights of the nodes is $O(N \log N)$ in an N-node random BST.
- Thus, average height of each node (in random BST) is $O(\log N)$.
- The worst case is still $O(N)$.

Some Questions?

- What if a BST has duplicates?
- Can a BST node contain strings? Other types?
- Can I store more pointers in a node?

Exercises

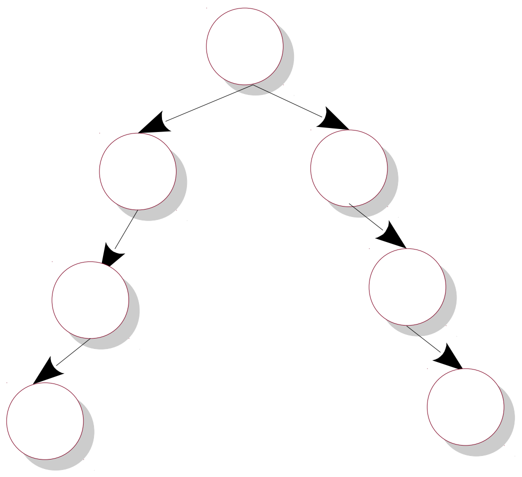
- Given a binary tree, find out if it is BST.
- Given an insertion sequence, how would you permute it to achieve the minimum height of the resultant BST?
  - See if your answer has a resemblance with binary search in a sorted array.
- Count the number of leaves in a BST.
- Print a BST in a level-order (breadth-first) manner.
- Write a program to print values in a BST in reverse-sorted order.

AVL Trees

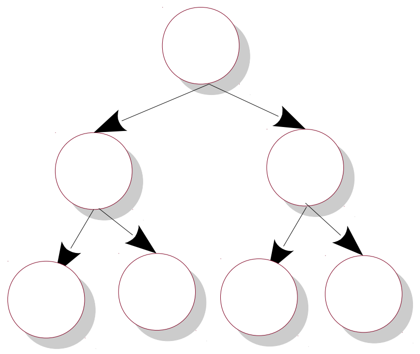
- Normal BSTs may have height O(N).
- As long as BST property is satisfied, the BST can be restructured to maintain $O(\log N)$ height.
- Invented by **two** researchers Georgy Adelson-Velsky and Evgenii Landis from Russia in 1962.
- Often called height-balanced trees or self-balancing BSTs.

What Doesn’t Work

- Ensure that at the root, the left and the right subtrees have the same heights.
  - Doesn’t guarantee height balance.

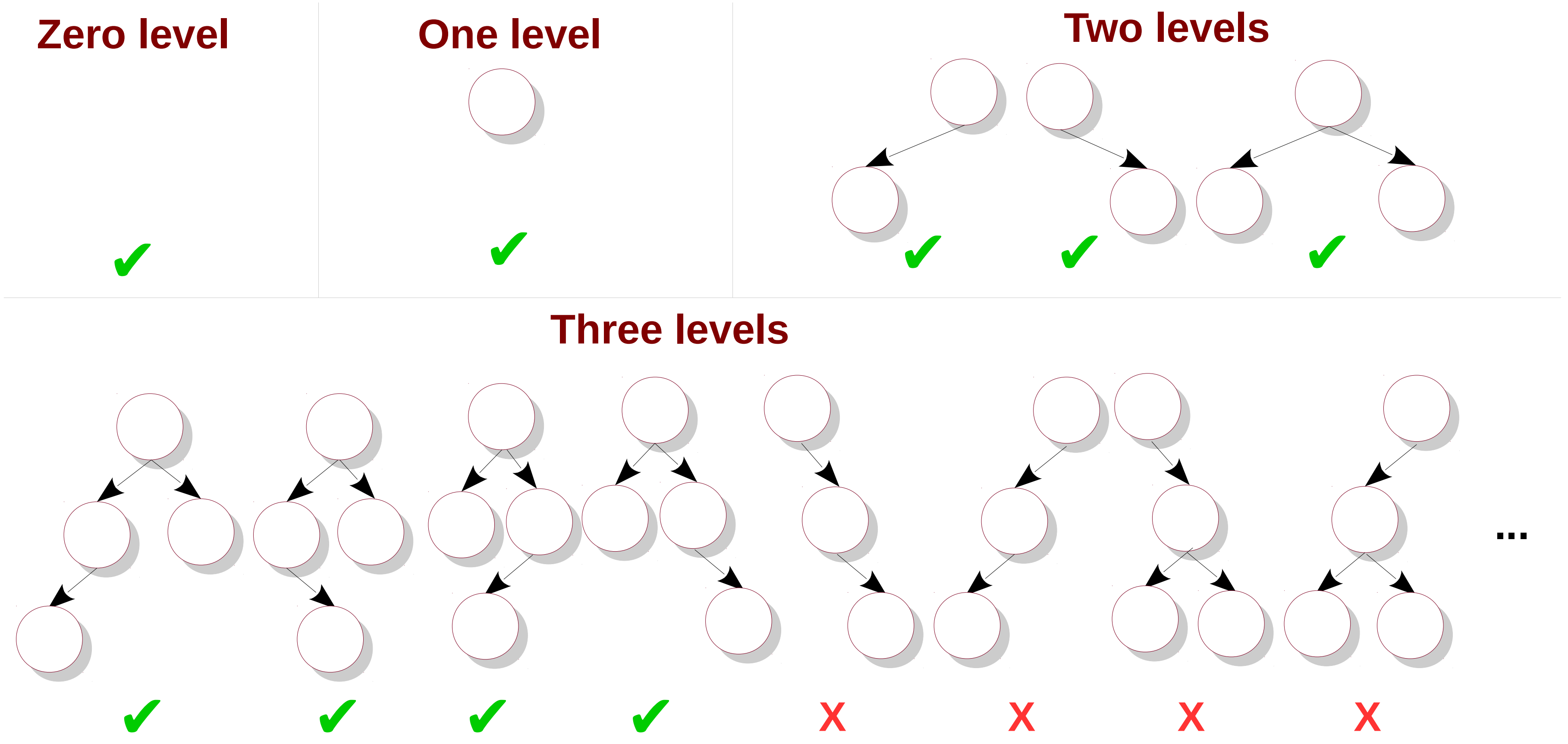


- Ensure the above at every node in the BST.
  - Allows only a few BSTs (number of nodes $2^K - 1$)



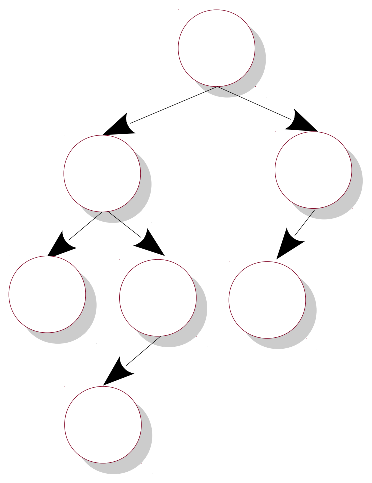
AVL Property

- At every node, the height-difference must not exceed 1.

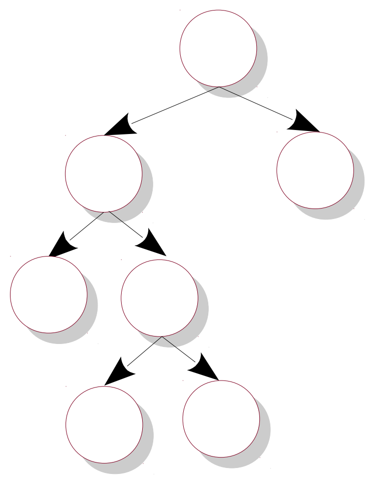


AVL Property

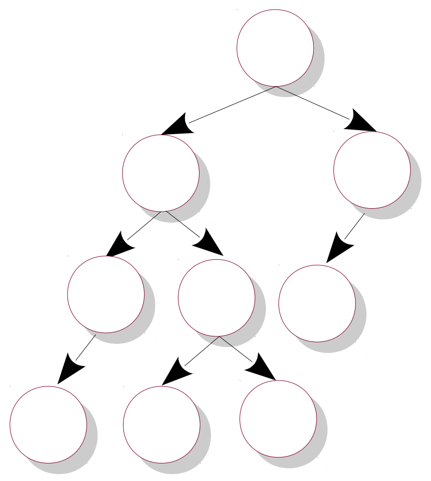
- At every node, the height-difference must not exceed 1.
- **Classwork**: Which of these have AVL structure?



✓



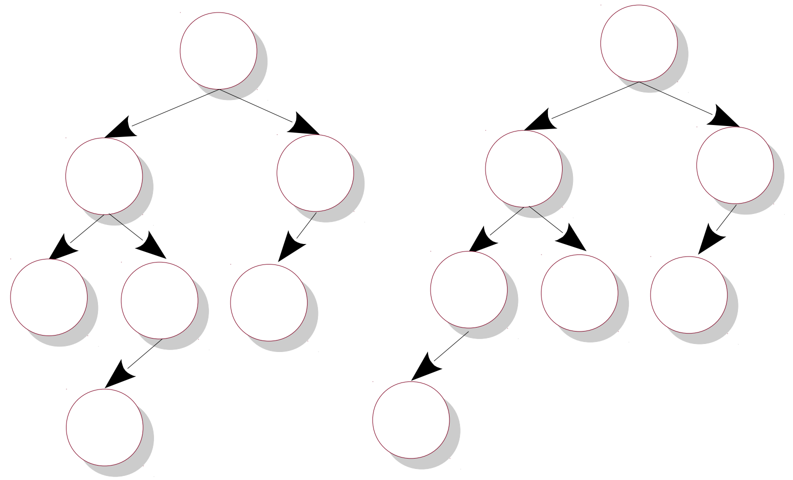
x



✓

AVL Property

- **Classwork**: Find ~~the~~ an AVL tree having four levels and fewest number of nodes.

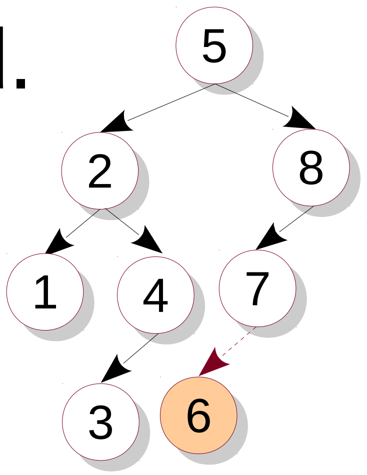


...

Minimum number of nodes in an h-height AVL tree:
S(h) = S(h-1) + S(h-2) + 1

- **Classwork**: Find the maximal AVL tree having four levels (such that addition of any edge makes it a height-imbalanced BST).

Insertion may violate AVL property



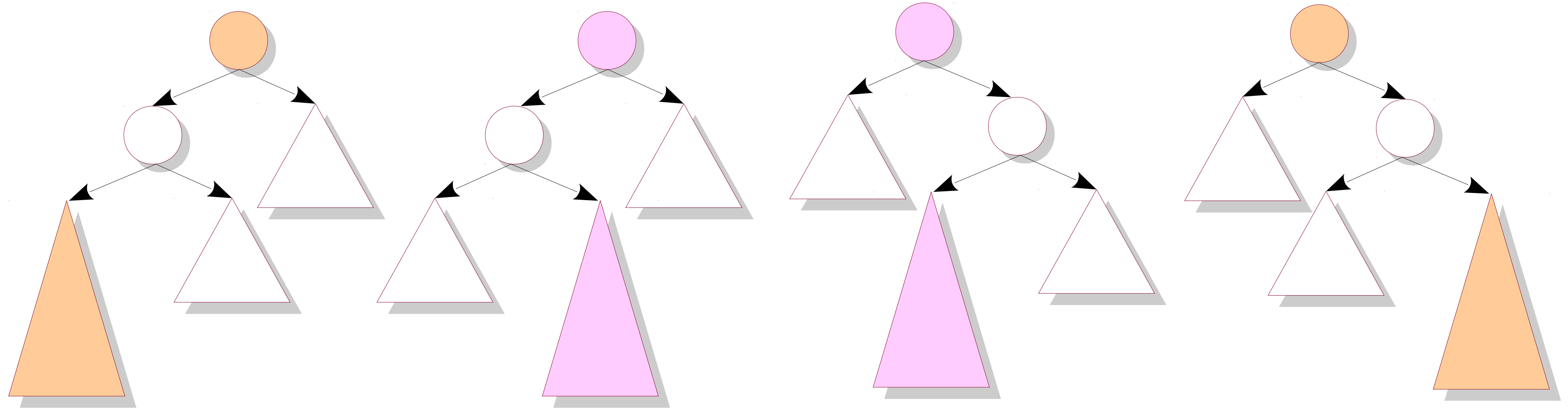
- Originally, the BST is height-balanced.
- Insert(6) violates AVL property
  - at node 8
- This is handled using *rotations*.
  - Exploits BST property which allows multiple structures for the same set of keys.
- **Observation**: Only nodes along the path from root to the new node have their subtrees altered.
  - Only these nodes may be checked for imbalance.
  - This means O(log N) rotations may be required.
  - We will show that only 2 rotations are sufficient.

AVL insert

- **Four cases:**
  1. insert into left subtree of left child
  2. Insert into right subtree of left child
  3. Insert into left subtree of right child
  4. Insert into right subtree of right child

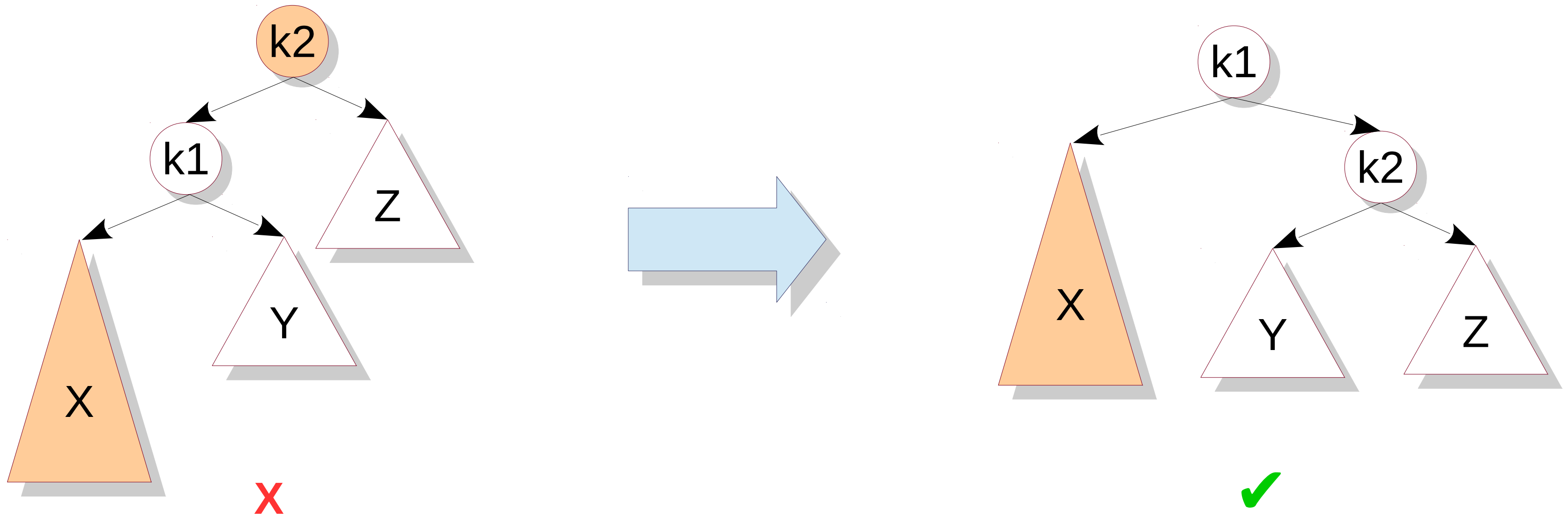
Double rotation

Single rotation



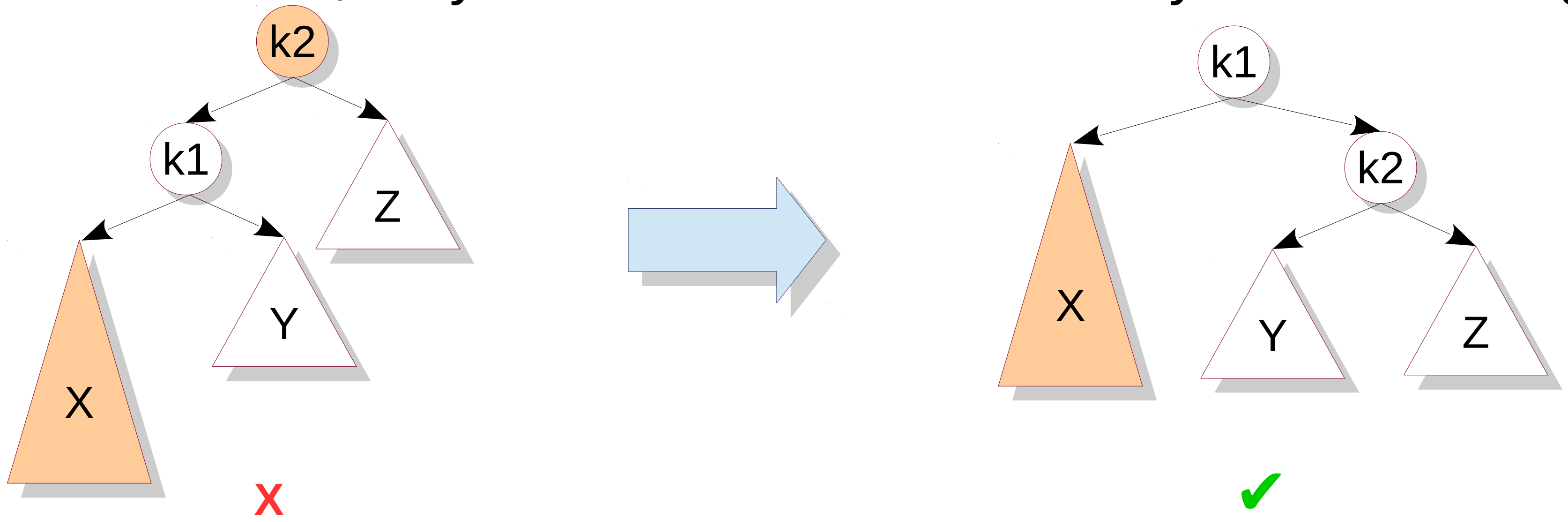
AVL insert: Case 1

- Insert into left subtree of left child
  - Before insertion into X, AVL property was satisfied.
  - Let k2 be the first node upward with imbalance.
  - Height(k2→left) – Height(k2→right) > 1
  - k1 continues to be height-balanced.
  - Can Y and Z be at the same level?

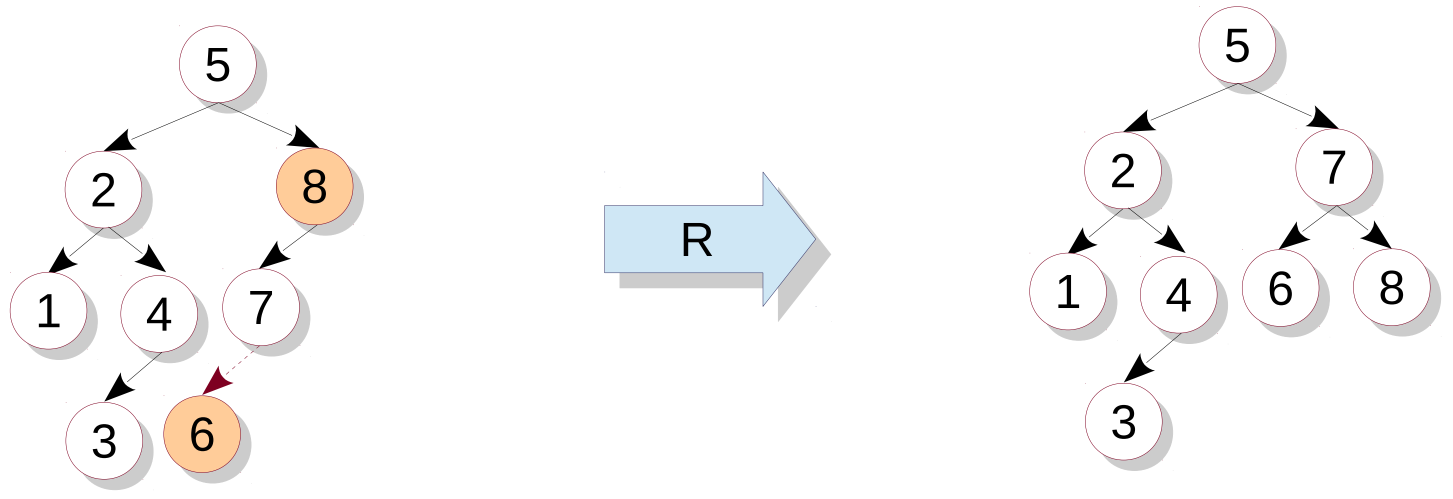


AVL insert: Case 1

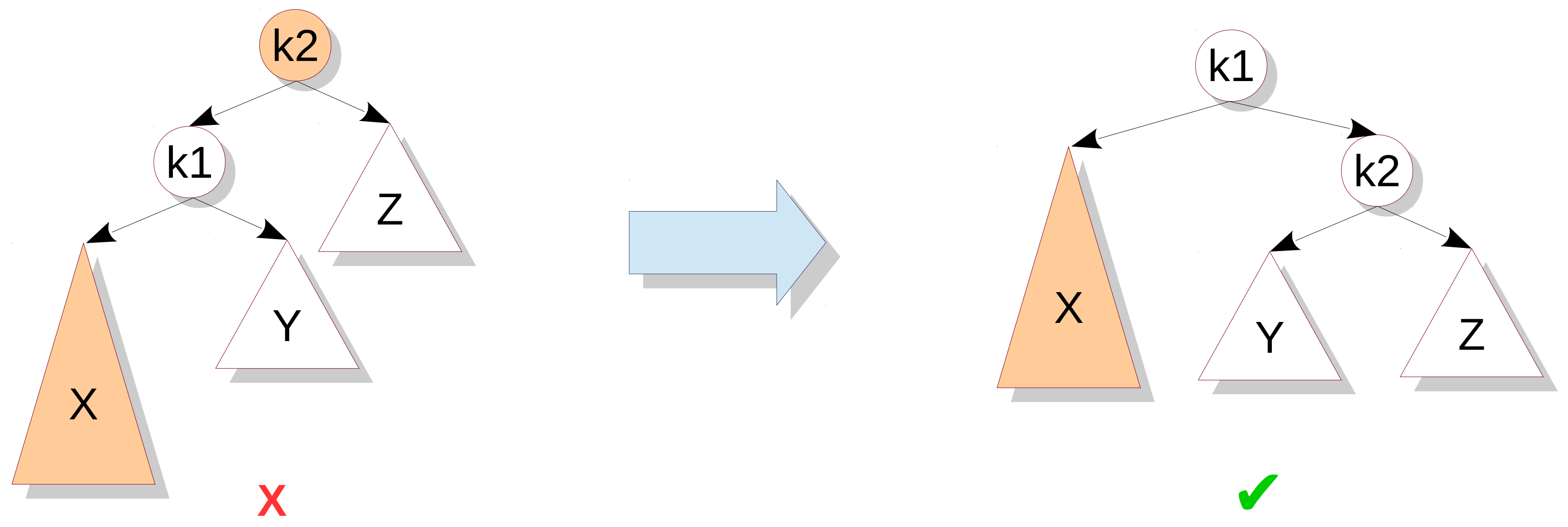
- Insert into left subtree of left child
  - Original order: X k1 Y k2 Z
  - New order: X k1 Y k2 Z
  - X moves up one level, Y stays at the same level, and Z moves down one level.
  - k1 and k2 satisfy AVL property.
    - In fact, they have subtrees with exactly the same height.



AVL insert: Case 1

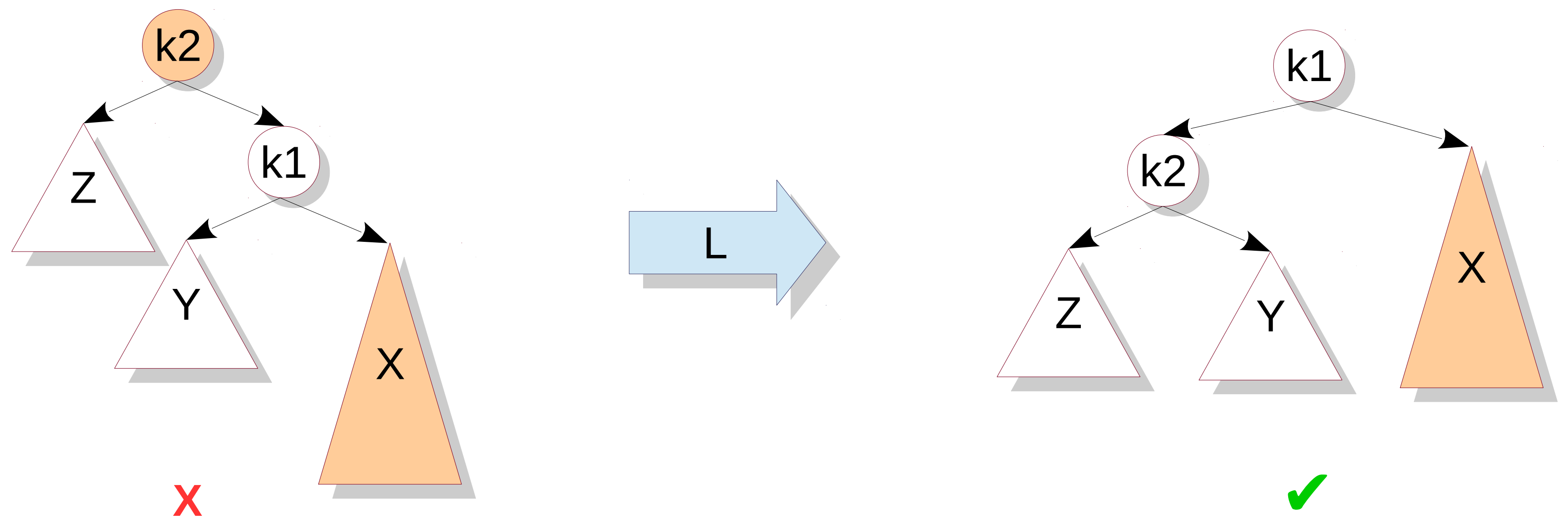


K1 == 7, k2 == 8, X is subtree rooted at 6, Y is empty, Z is empty.



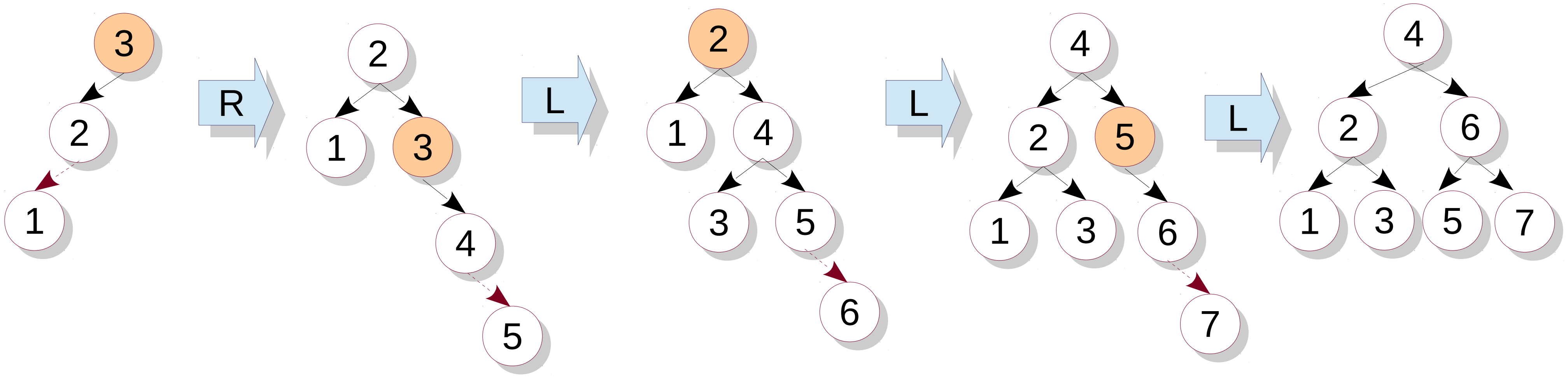
AVL insert: Case 4

- Scenario is symmetric to Case 1.
- Case 1 had a right rotation.
- Case 4 needs a left rotation.



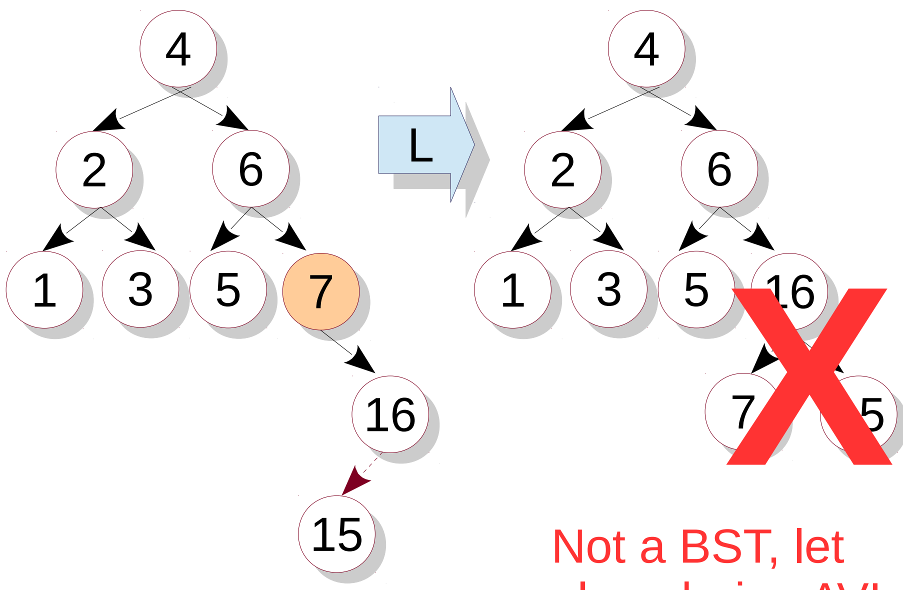
Example

- Start with an empty AVL tree.
- Insert 3, 2, 1, 4, 5, 6, 7.
- Then insert 16, 15, 14, 13, 12, 11, 10, 8, 9.



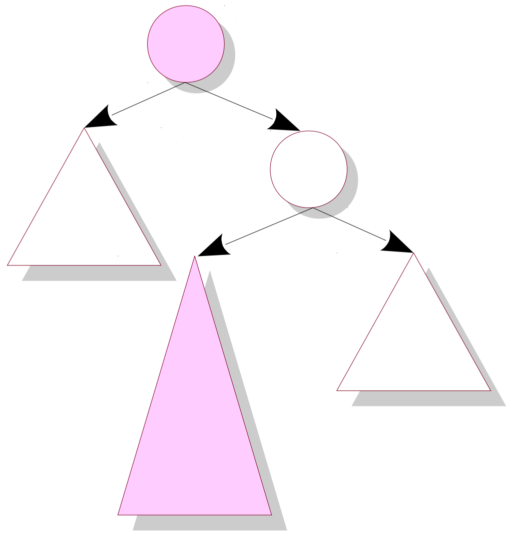
Example

- Start with an empty AVL tree.
- Insert 3, 2, 1, 4, 5, 6, 7.
- Then insert 16, 15, 14, 13, 12, 11, 10, 8, 9.



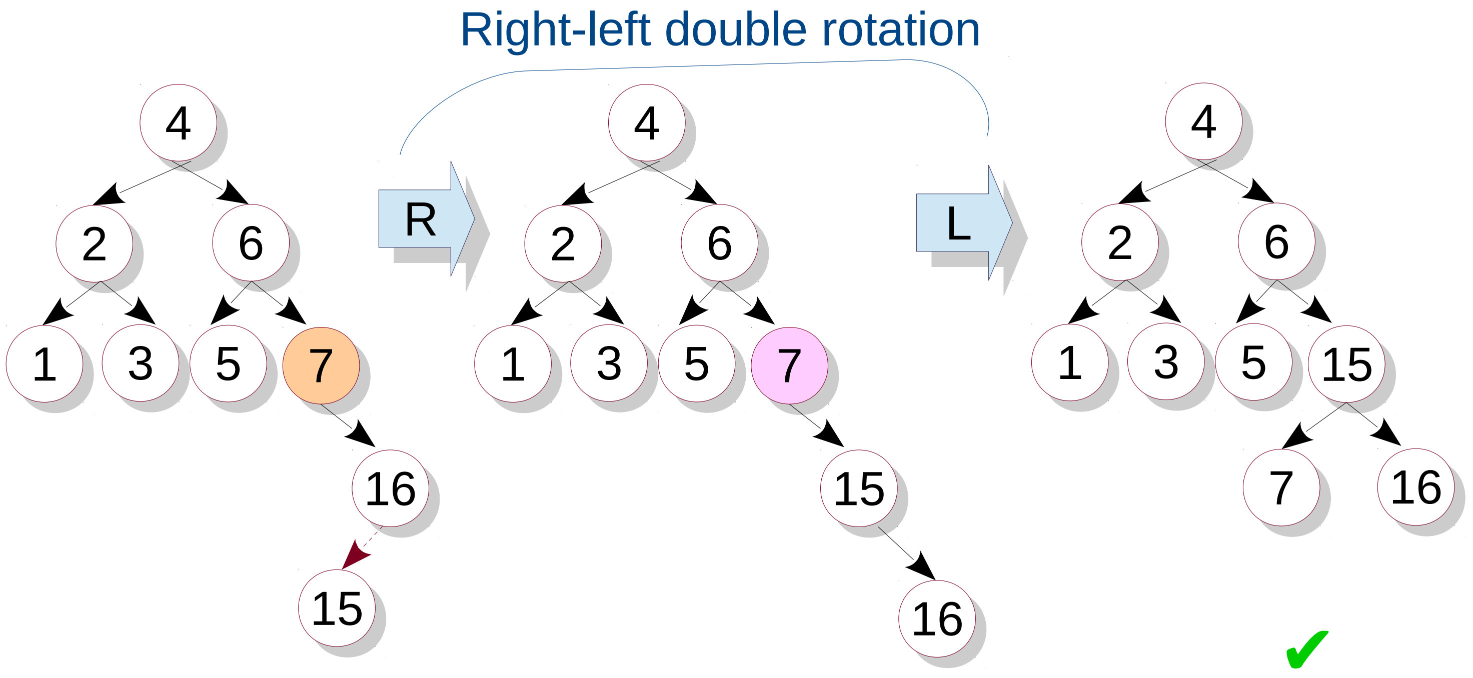
Not a BST, let
alone being AVL

The case is neither Case 1 (left-left)
nor Case 4 (right-right).
It is Case 3 (right-left).



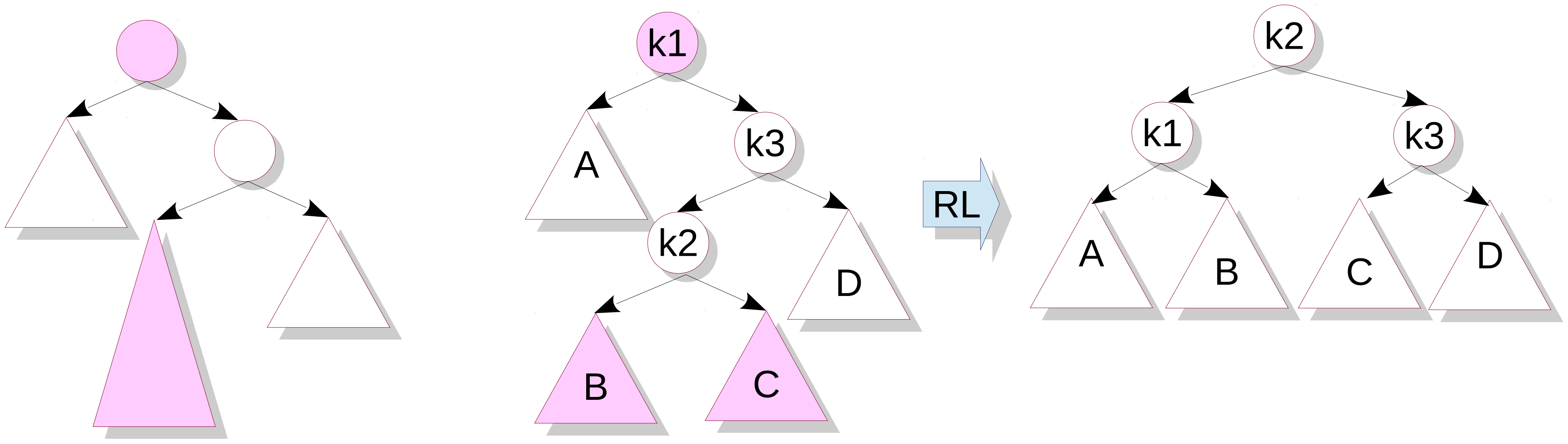
Example

- Start with an empty AVL tree.
- Insert 3, 2, 1, 4, 5, 6, 7.
- Then insert 16, 15, 14, 13, 12, 11, 10, 8, 9.



Example

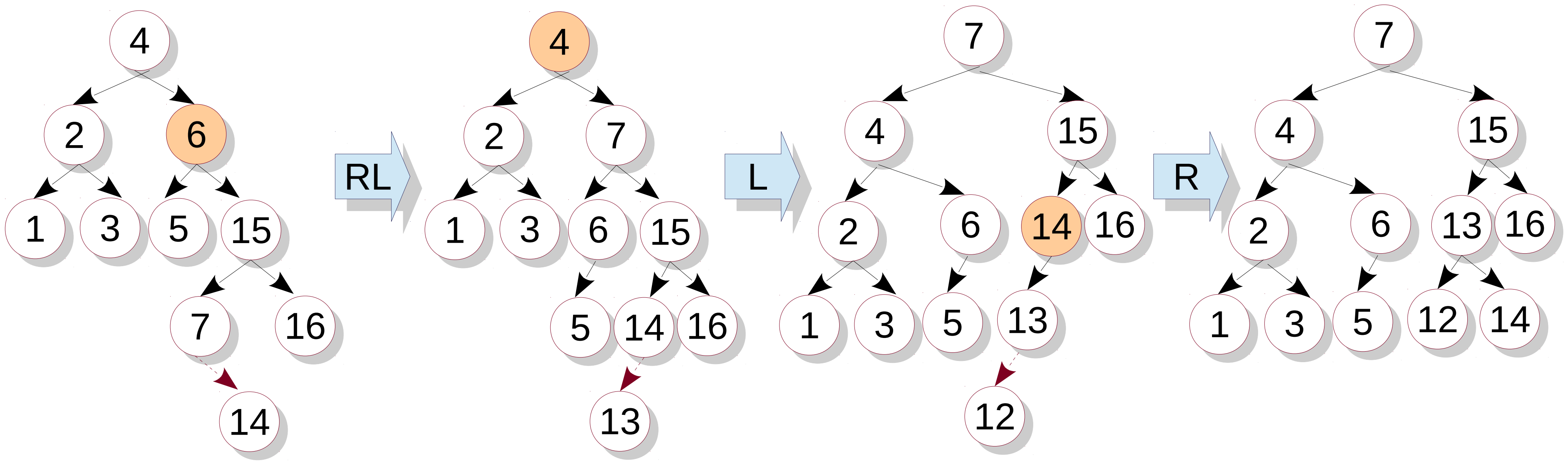
- Start with an empty AVL tree.
- Insert 3, 2, 1, 4, 5, 6, 7.
- Then insert 16, 15, 14, 13, 12, 11, 10, 8, 9.



BST Sequence: A k1 B k2 C k3 D

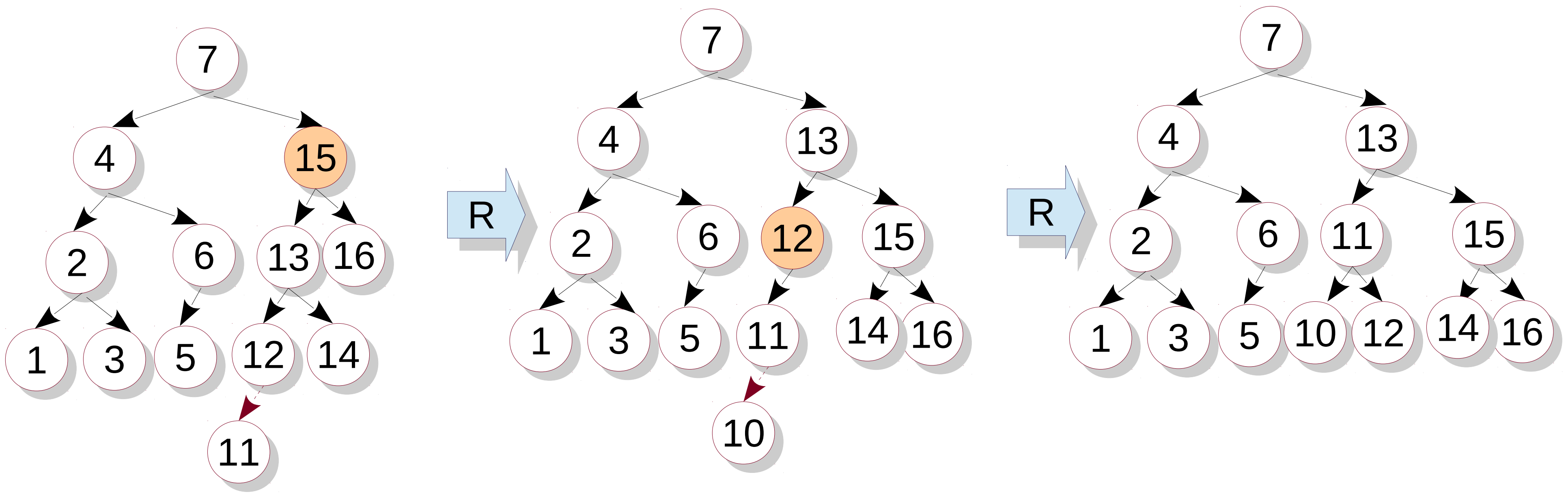
Example

- Start with an empty AVL tree.
- Insert 3, 2, 1, 4, 5, 6, 7.
- Then insert 16, 15, 14, 13, 12, 11, 10, 8, 9.



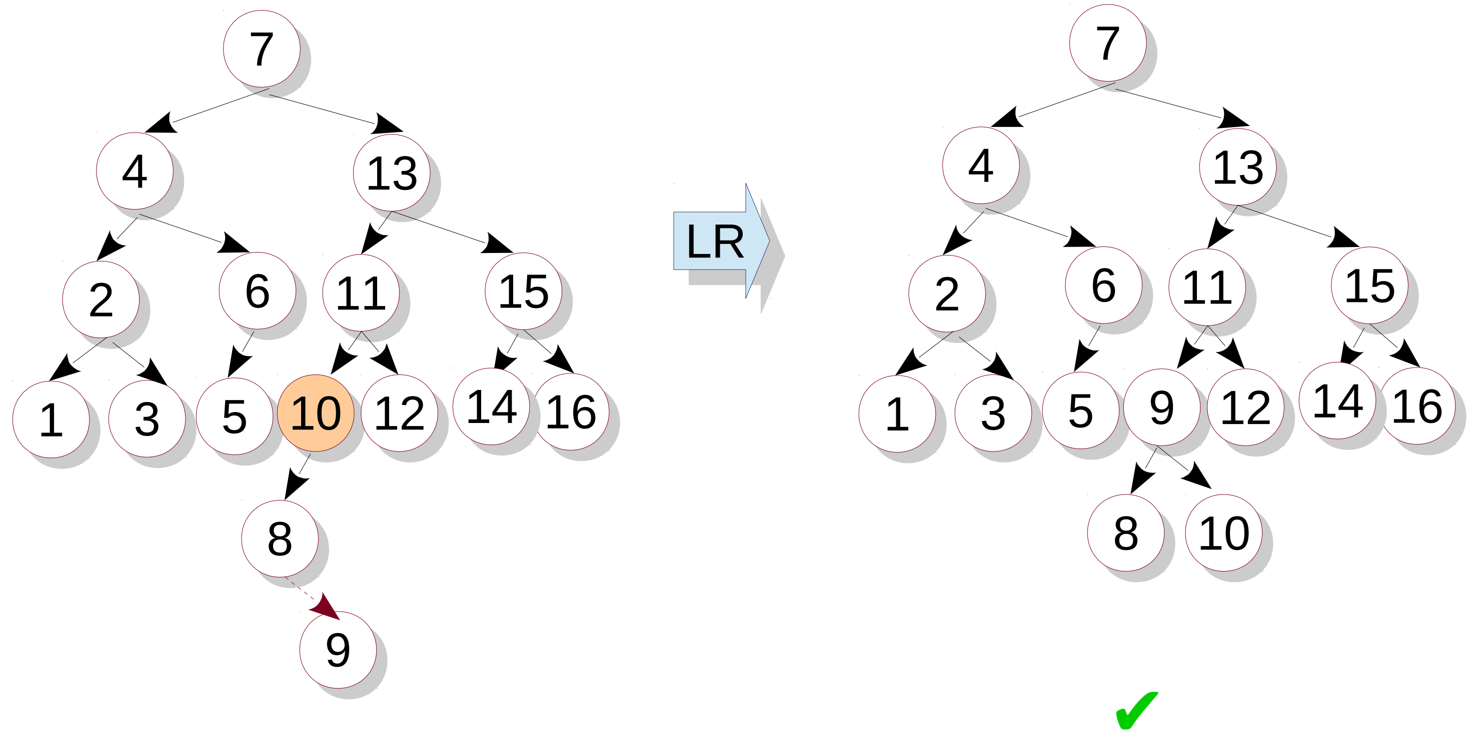
Example

- Start with an empty AVL tree.
- Insert 3, 2, 1, 4, 5, 6, 7.
- Then insert 16, 15, 14, 13, 12, 11, 10, 8, 9.



Example

- Start with an empty AVL tree.
- Insert 3, 2, 1, 4, 5, 6, 7.
- Then insert 16, 15, 14, 13, 12, 11, 10, 8, 9.



Source: avl.cpp

Classwork

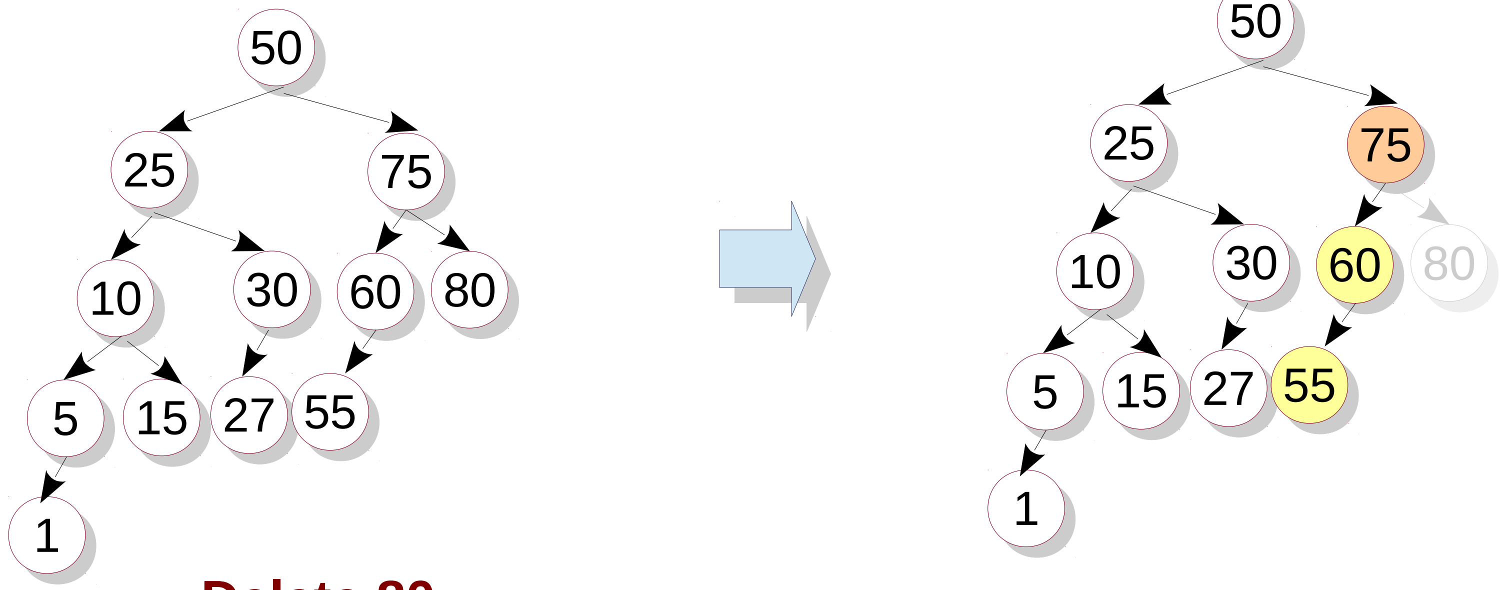
- In an empty AVL tree,

  insert 4, 10, 1, 2, 5, 7, 3, 6, 8, 9.

Deletion in AVL

- Can be implemented lazily (marking).
  - Does not maintain AVL property.
- Inverse of insertion, so need to think backwards.
- But rotations would help us rebalance.
  - The four rotations can be called as primitives.
  - Deletion at internal node starts similar to as in BST. Gets converted to deletion at the leaf.
- Need to find taller of the two subtrees (left or right).
  - We can then rotate that subtree to rebalance.
- Unlike insertion, deletion may need to be repeated for all the ancestors.

Deletion Example

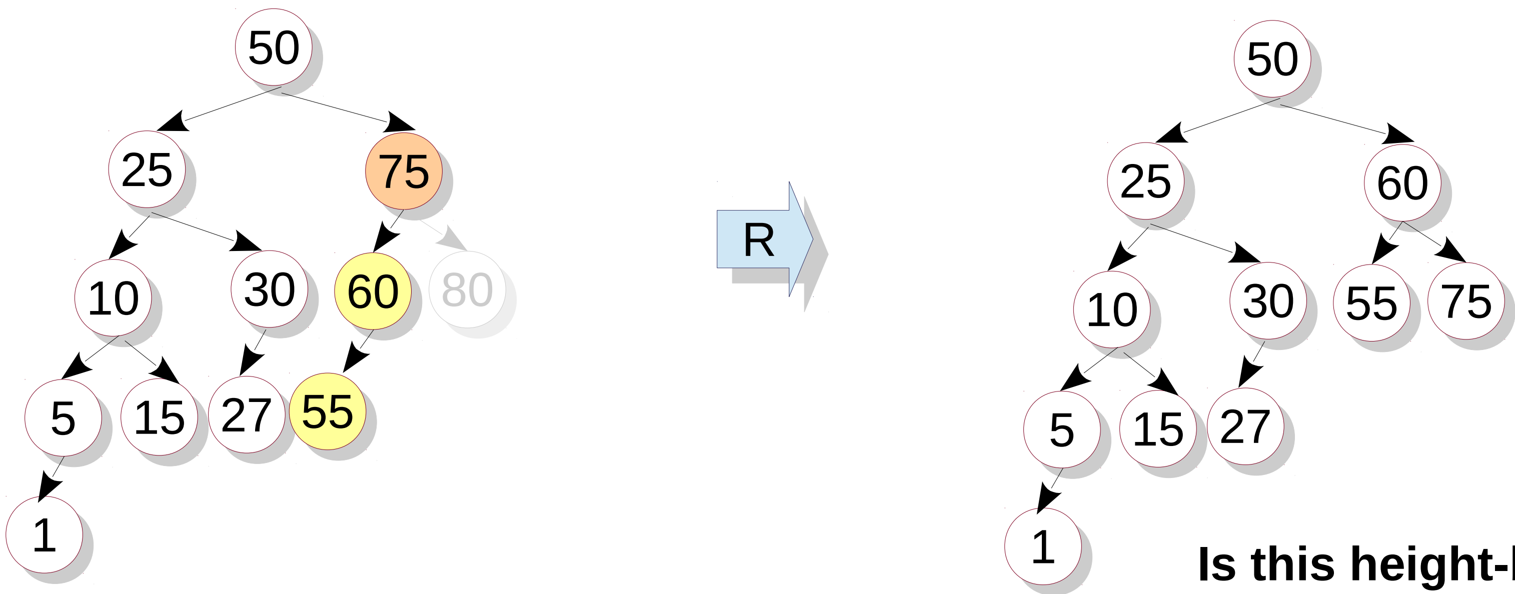


Delete 80

- Remove the node as in BST.
- Update heights of ancestors upward along the path.
- Find the first (deepest / lowest) imbalanced node x (75).
- Find x's tallest child y (60). Find y's tallest child z (55).
- Rotate x, y, z.

http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Trees/AVL-delete.html

Deletion Example

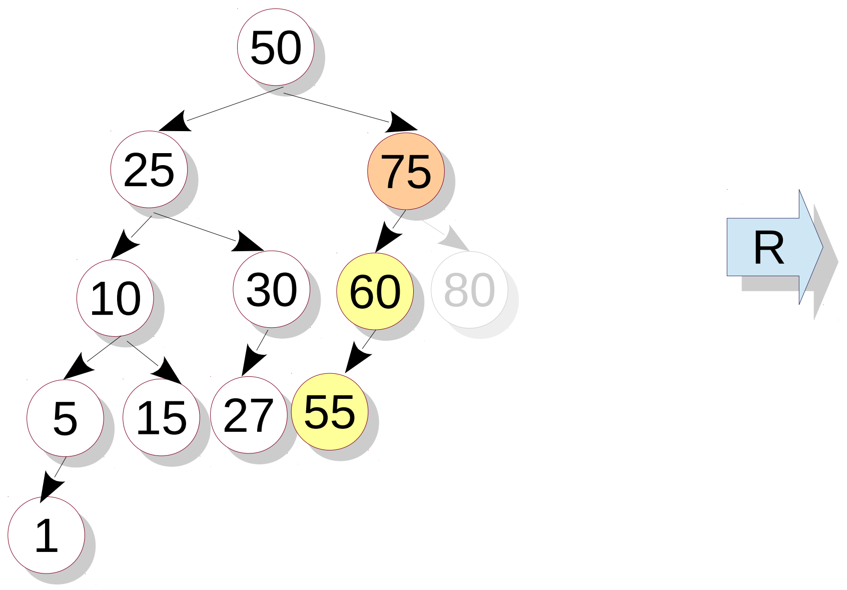


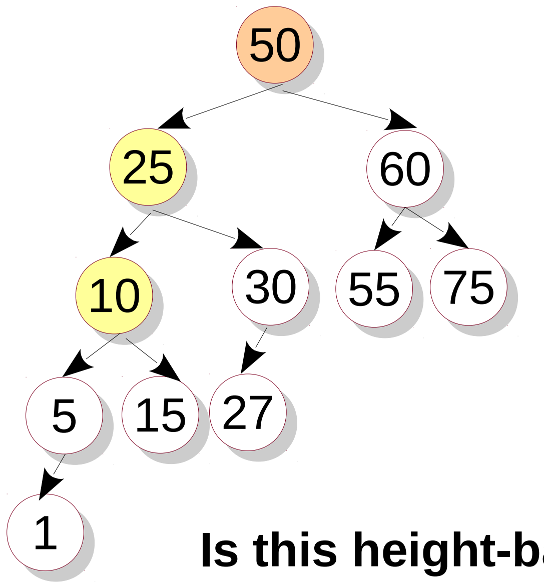
Is this height-balanced?
- Yes at 60, No at 50.

Delete 80

- Remove the node as in BST.
- Update heights of ancestors upward along the path.
- Find the first (deepest / lowest) imbalanced node x (75).
- Find x's tallest child y (60). Find y's tallest child z (55).
- Rotate x, y, z.

Deletion Example





Is this height-balanced?
- Yes at 60, No at 50.

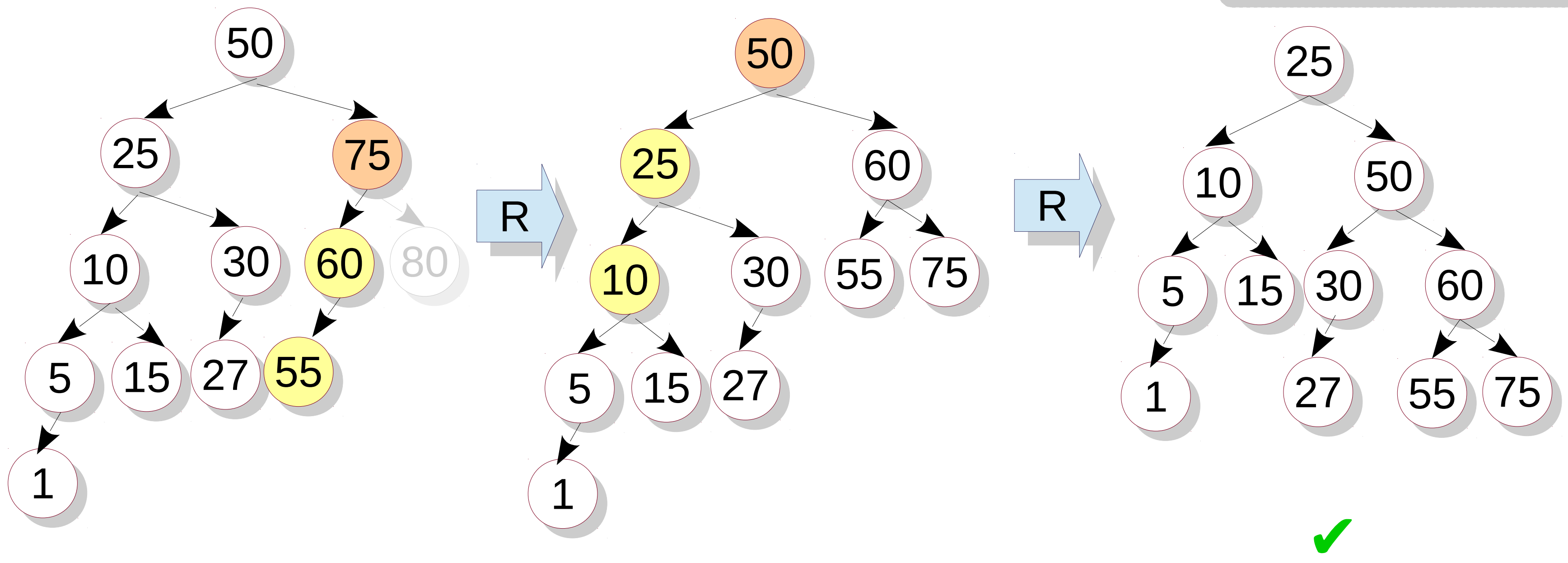
Delete 80

- Remove the node as in BST.
- Update heights of ancestors upward along the path.
- Find the first (deepest / lowest) imbalanced node x (50).
- Find x's tallest child y (25). Find y's tallest child z (10).
- Rotate x, y, z.

Deletion Example

Classwork: Delete 75, 55, 60, 50, 27.

Source: avl.cpp



Delete 80

- Remove the node as in BST.
- Update heights of ancestors upward along the path.
- Find the first (deepest / lowest) imbalanced node x (50).
- Find x's tallest child y (25). Find y's tallest child z (10).
- Rotate x, y, z.

http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Trees/AVL-delete.html

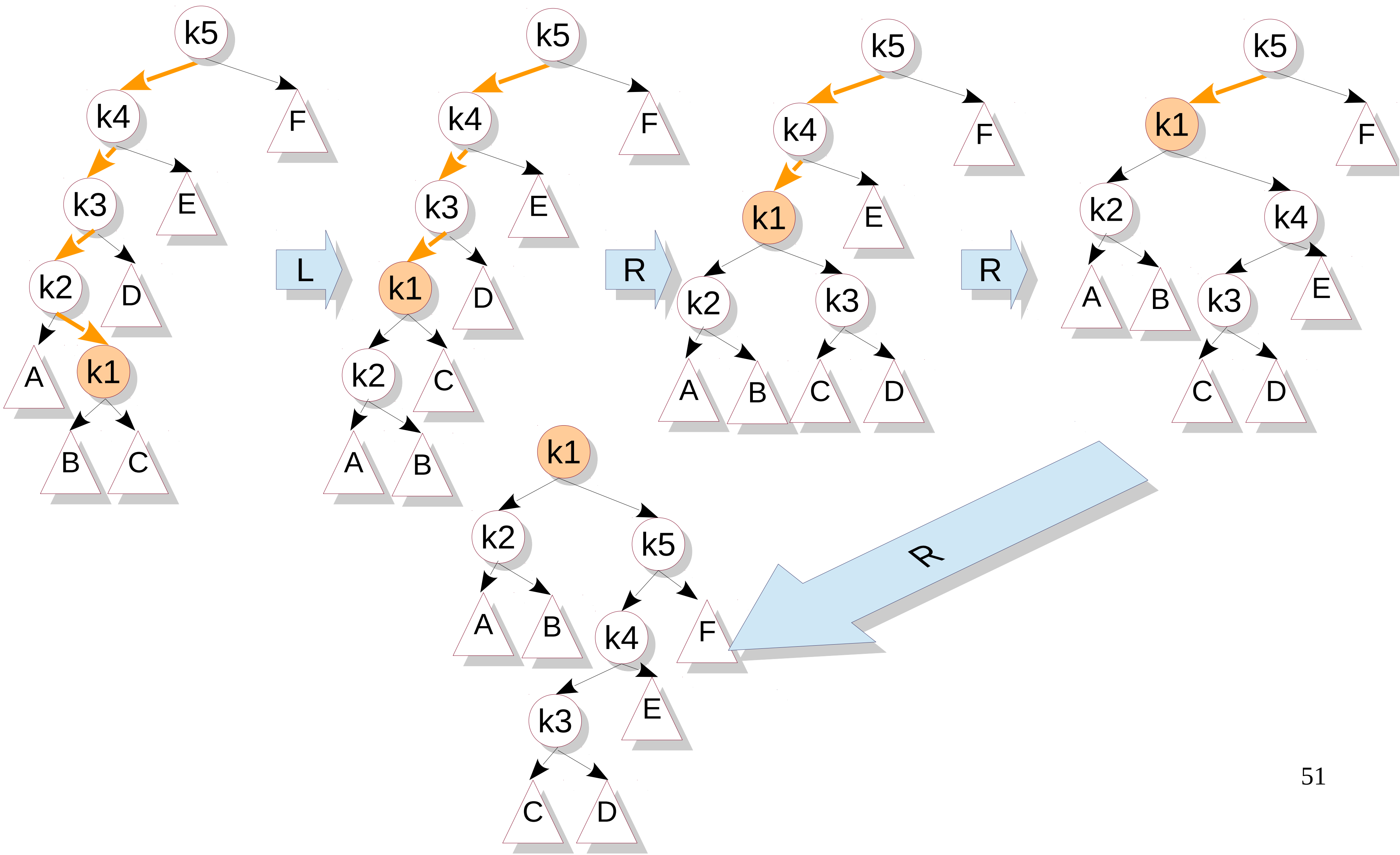
Splay Trees

- BSTs provide O(N) search.
- AVLs provide O(log N) search.
  - Too strict about height-balancing.
  - Can be expensive in practice.
- Can we construct a lightweight mechanism?
  - Splay Trees provide O(log N) amortized search.
  - Splaying is widening of the road at junctions to improve visibility. In this case, we improve visibility of accessed nodes.
  - The BST need not be height-balanced.

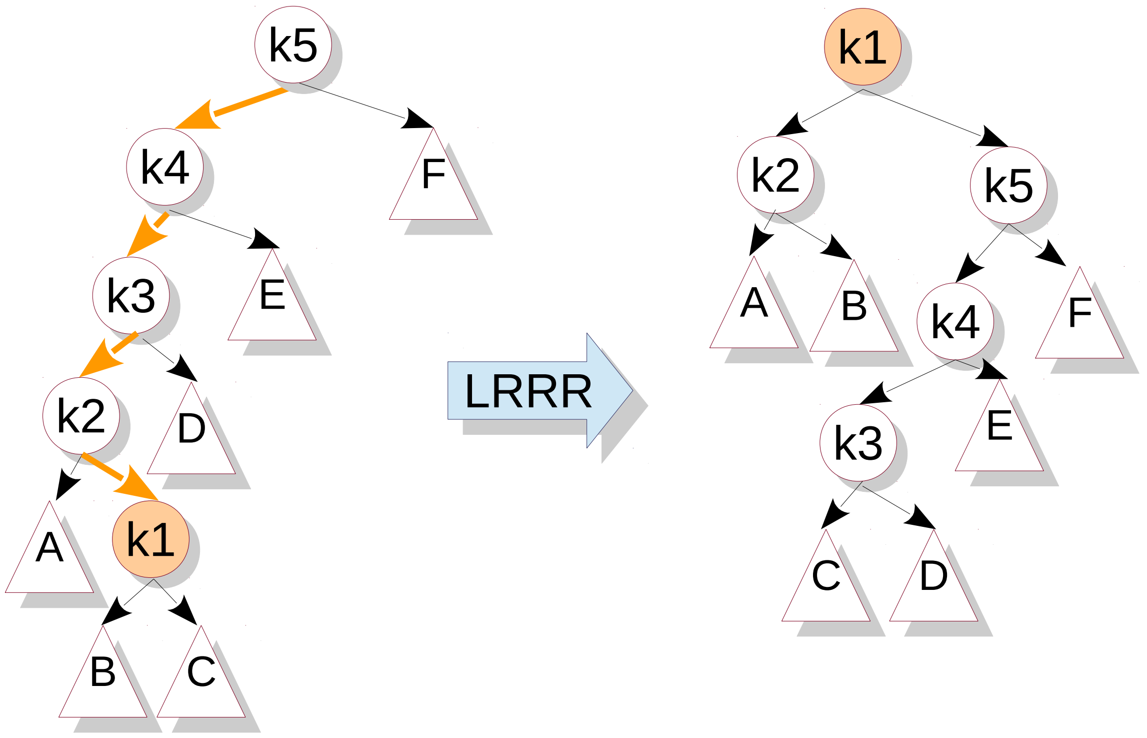
Splay Trees

- Worst-case time to find an element is still O(N).
  - Amortization: Over a set of M operations, the execution time is O(M log N).
- This means, a deep node cannot retain its position after access.
  - Else, an adversary can come up with a sequence of M accesses requiring O(M N) time.
- On an access, the element is pushed to the root.
  - This uses a series of rotations.
  - In the process, some other nodes also change their heights (resulting in some balancing).

Moving $\mathbf{k1}$ up on $\mathbf{search(k1)}$



Moving k1 up on search(k1)



What did this move achieve?

- Next access to k1 is fast.
- Next access to k2 is also faster.
- Next accesses to k3, k4, k5 are slower.
- Overall tree height has reduced.

Let's play adversarial:

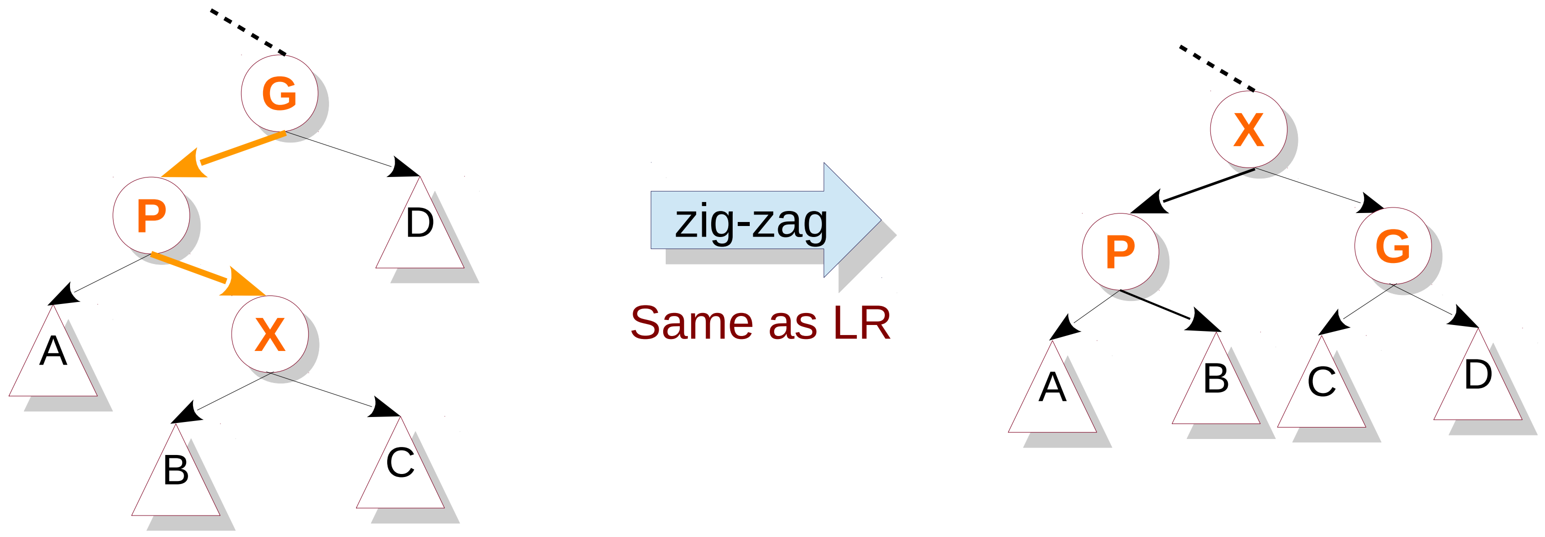
- Construct a left-skewed BST with insertion order N..1.
- Access 1 (traverses N-1 links). Pushes 1 to root.
- Access 2 (traverses N-2 links). Pushes 2 to root.
- Access 3 (traverses N-3 links). Pushes 3 to root.
- …
- Access N (traverses 1 link). Pushes N to root.
- We get the original left-skewed BST. **Repeat**.

$\Omega(N^2)$

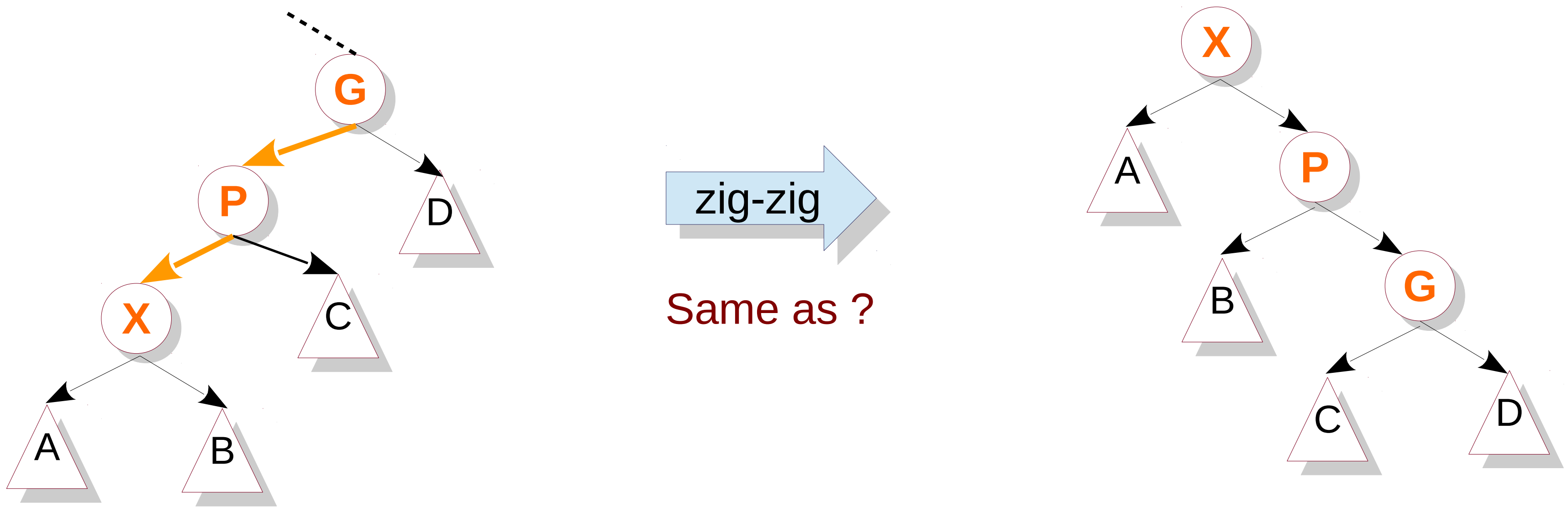
Splaying

- Splaying on node X
  - X must be a non-root; else no splaying is required.
- If X’s parent is the root, rotate X and root.
- Otherwise, X has a parent P and grandparent G.
  - Again, four cases.
  - Transform each of the cases as on the next slide.

Splaying

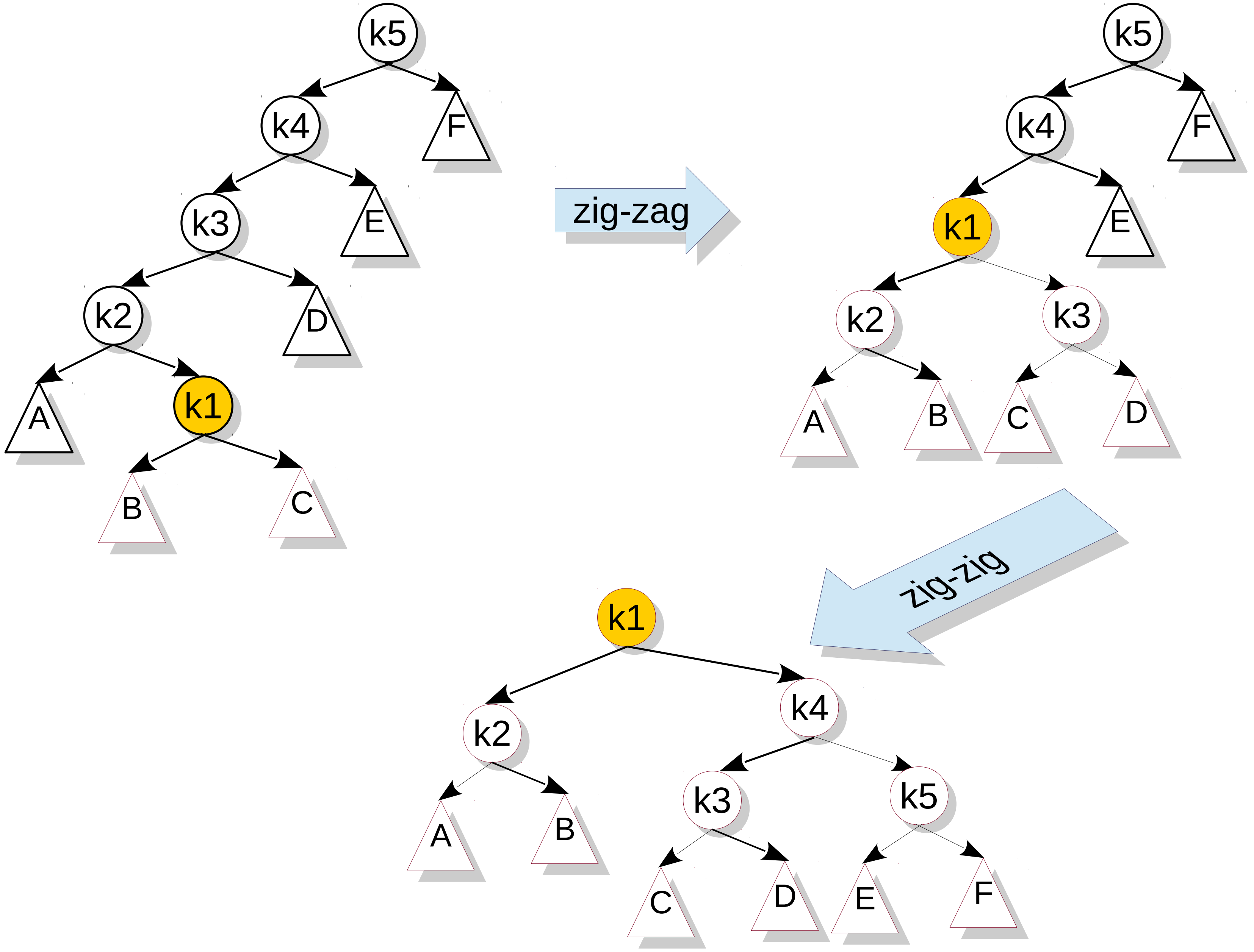


Same as LR

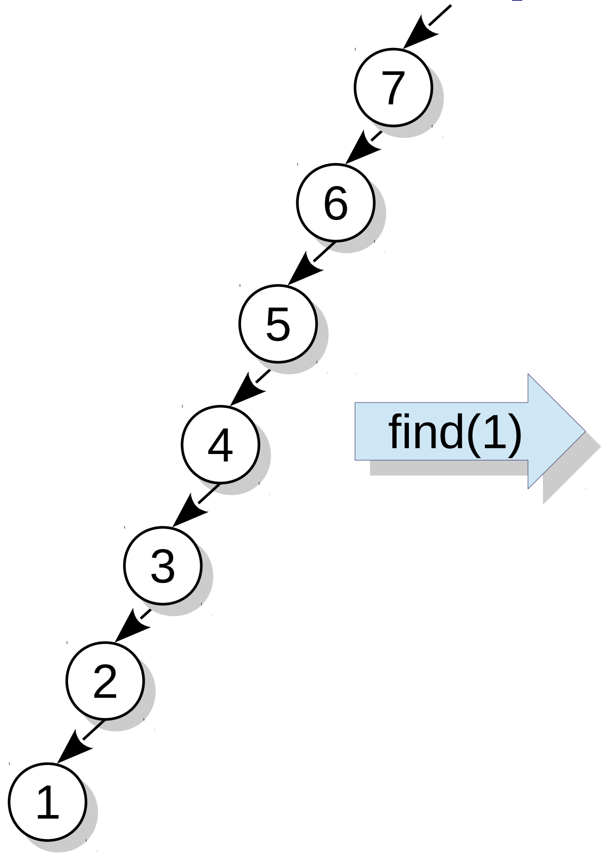


Same as ?

Splaying Example: find(k1)



Splaying on Adversary

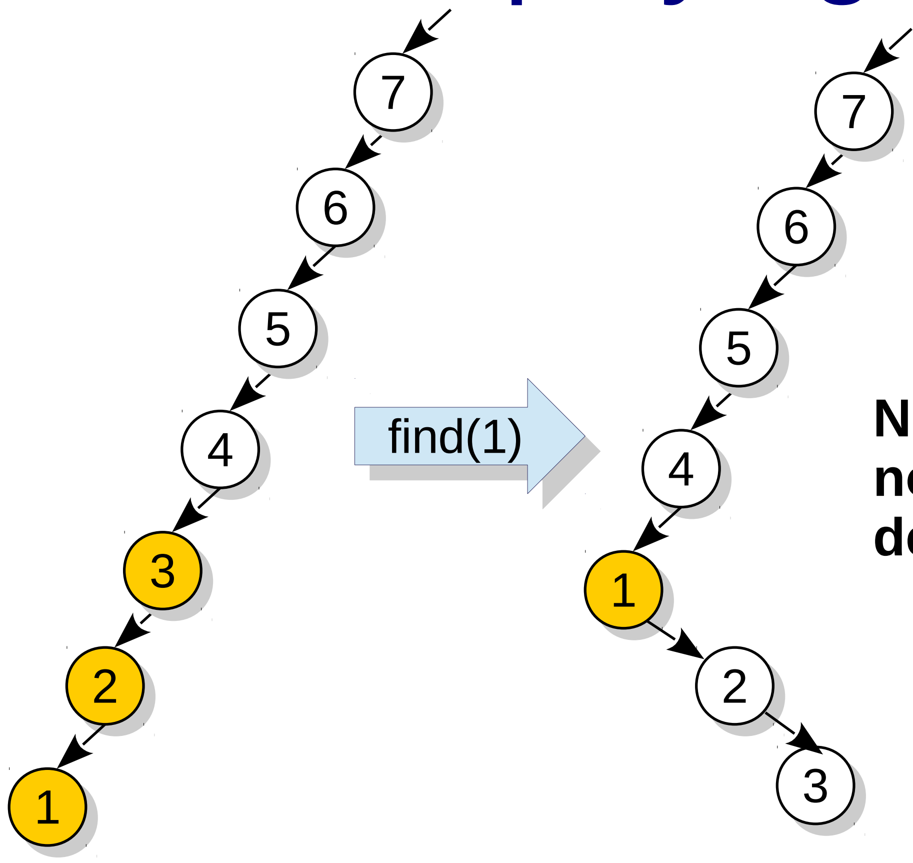


Let’s play adversarial:

- Construct a left-skewed BST with insertion order N..1.
- Access 1 (traverses N-1 links). Pushes 1 to root.
- Access 2 (traverses N-2 links). Pushes 2 to root.
- Access 3 (traverses N-3 links). Pushes 3 to root.
- …
- Access N (traverses 1 link). Pushes N to root.
- We get the original left-skewed BST. **Repeat**.

$\left.\right\}$ **$\Omega(N^2)$**

Splaying on Adversary



Notice that no nodes are as deep as before.

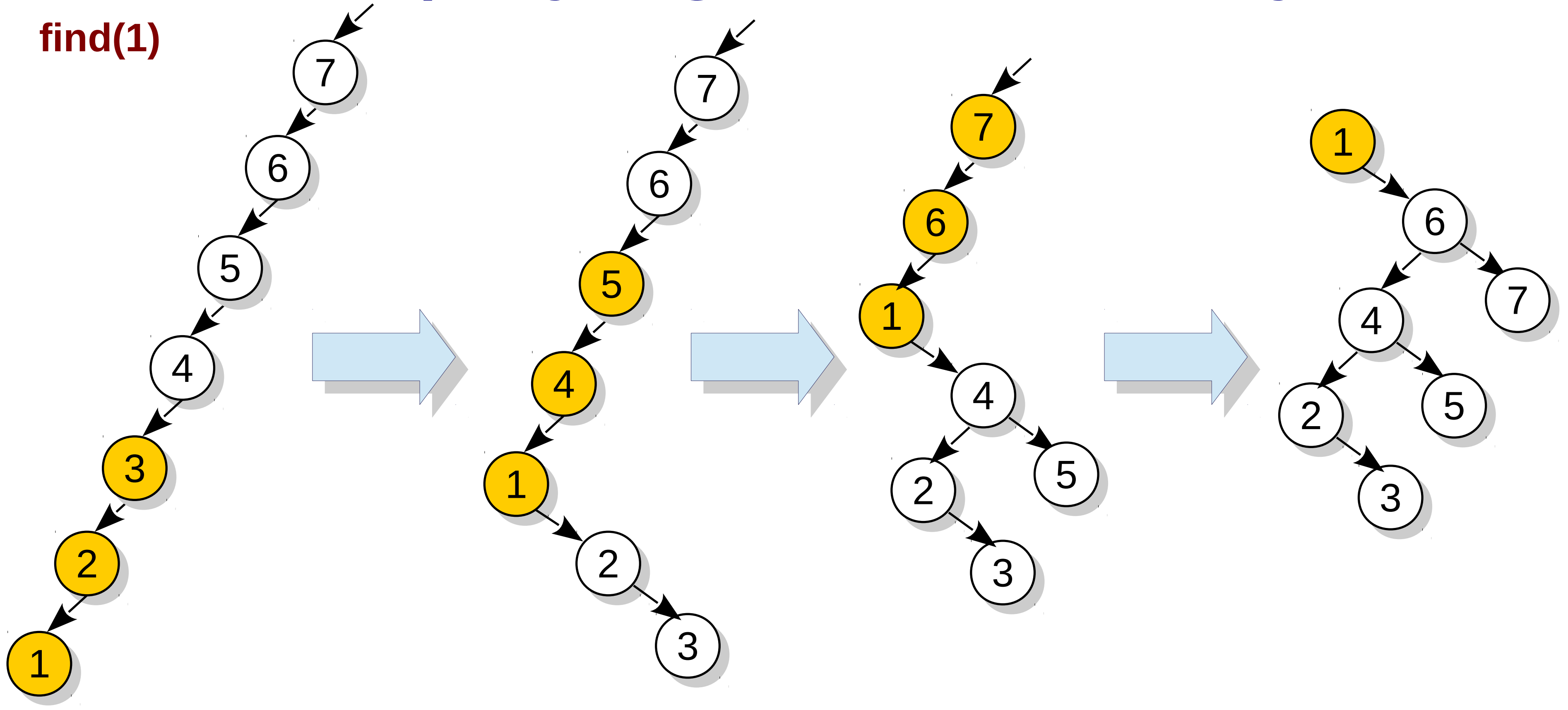
Let's play adversarial:

- Construct a left-skewed BST with insertion order N..1.
- Access 1 (traverses N-1 links). Pushes 1 to root.
- Access 2 (traverses N-2 links). Pushes 2 to root.
- Access 3 (traverses N-3 links). Pushes 3 to root.
- …
- Access N (traverses 1 link). Pushes N to root.
- We get the original left-skewed BST. **Repeat**.

$\Omega(N^2)$

Splaying on Adversary

find(1)



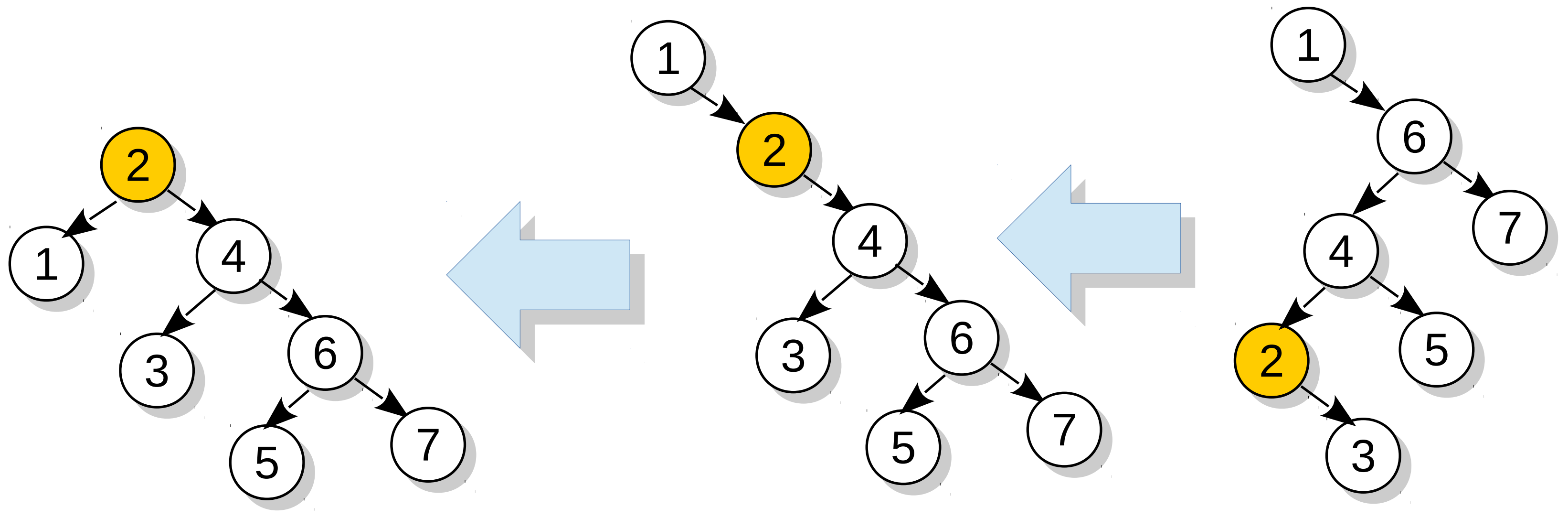
Let’s play adversarial:

- Construct a left-skewed BST with insertion order N..1.
- Access 1 (traverses N-1 links). Pushes 1 to root.
- Access 2 (traverses N-2 links). Pushes 2 to root.
- Access 3 (traverses N-3 links). Pushes 3 to root.
- …
- Access N (traverses 1 link). Pushes N to root.
- We get the original left-skewed BST. **Repeat.**

$\Omega(N^2)$

Splaying on Adversary

find(2)



Splaying Analysis

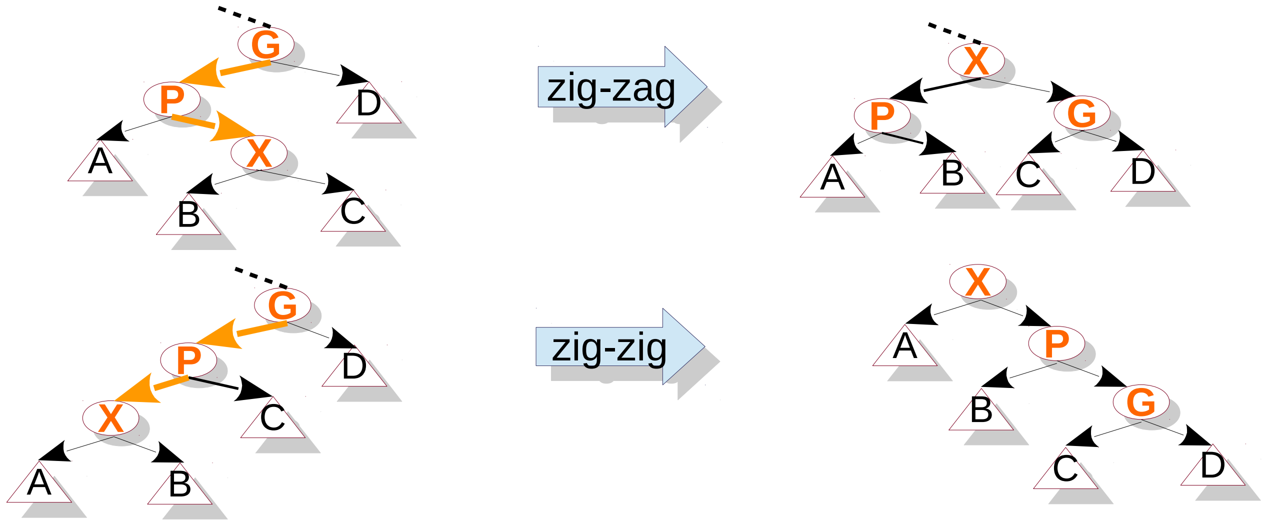
- Many of you noted that
  - Left-skewed may not be the worst-case sequence.
  - Height of a node may be above log(N) for every step in splaying.
- How can we guarantee O(M log N) complexity for M operations?
- Need to depend upon amortized analysis.
  - Will use potential function.

Potential Function

- We intelligently guess a potential, and
  - Show that it is maintained across operations.
- In case of splaying, since we want an amortized bound of O(log N) per operation, the potential function is naturally O(log N).
  - This means, potential of the data structure (splay tree here) increases by at max. O(log N).
- A good potential function:
  - Clearly, node height doesn’t help.
  - Thus, sum of the node heights also does not help.
  - We will use sum of log(S_i) where S_i is the subtree size rooted at node i.

Potential Function

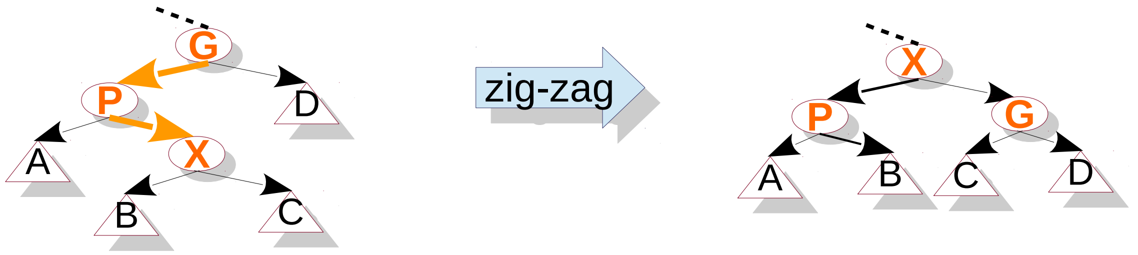
- $PF(i) = \Sigma_i R(i)$ where $R(i) = \log(S_i)$ for node i.
  - R(i) is often called the rank of node i.
  - Root has a rank of log(N).
  - We need to show that PF is bounded for zig-zig and zig-zag rotations.
  - **Important**: rotation may change heights of many nodes, but only X, P, G may change their ranks.



Zig-zag Step's Analysis

- Let R_i be initial rank and R_f be the rank after each step of splaying.
- Cost of only zig-zag = 2
- Potential change =

  $R_f(X) + R_f(P) + R_f(G) – (R_i(X) + R_i(P) + R_i(G))$
- Now, $S_f(X) = S_i(G)$, so $R_f(X) = R_i(G)$.
- Also, $S_i(P) >= S_i(X)$. Thus, $R_i(P) >= R_i(X)$.



Zig-zag Step's Analysis

$AT_{\text{zig-zag}} <= 2 + R_f(P) + R_f(G) – 2R_i(X)$ ………..(1)

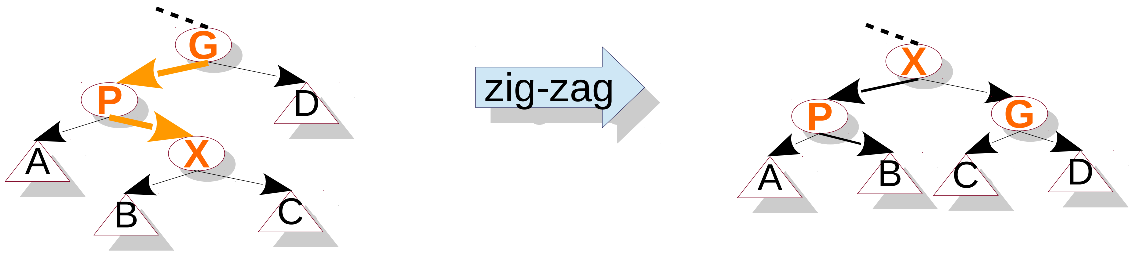
Now, $S_f(P) + S_f(G) <= S_f(X)$.

Hence, $\log S_f(P) + \log S_f(G) <= 2 \log S_f(X) – 2$

Thus, $R_f(P) + R_f(G) <= 2R_f(X) – 2$

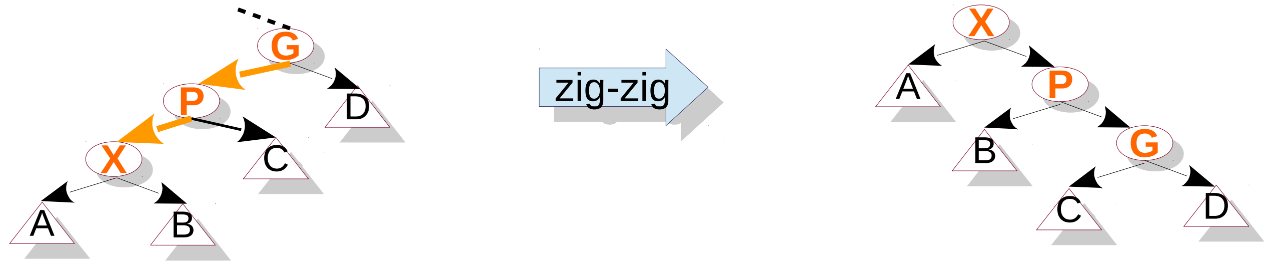
Substituting in (1)

$AT_{\text{zig-zag}} <= 2R_f(X) – 2R_i(X) <= 3R_f(X) - 3R_i(X)$



Zig-zig Step's Analysis

- Similar to zig-zag, left as a homework.
  - $R_f(X) = R_i(G)$
  - $R_f(X) >= R_f(P)$
  - $R_i(X) <= R_i(P)$
  - $S_i(X) + S_f(G) <= S_f(X)$



Splaying Analysis

- $AT_{zig\text{-}zag} <= 3R_f(X) – 3R_i(X)$
  - This is for one step of splaying.
  - For the next step, $R_f(X)$ would be the initial rank.
- Summing up across steps:
  - Total cost <= $3\ R_{final}(X) – 3R_{initial}(X) + 1$ (last 1 for zig)
  - But finally, X becomes the root.
  - Total cost <= $3\ R(root) – 3\ R_{initial}(X) + 1$
  - Total cost <= O(log N) (root has a rank of log N)
- Thus, splaying is amortized O(log N).

Hash Tables

- Hashtable is also a dictionary.
  - Dictionaries map keys to values.
  - Hashtables also do the same in a specific way.
- Hashing is indexing elements in a (typically) fixed length array.
- Element's pattern (name, bit-pattern, …) is used for computing the index.

Sorted Array	Akshat	Krishna	Monisha	Naveen	Pavan
Hash Table	Pavan	Monisha	Akshat	Naveen	Krishna

Hash Functions

- Identity function: f(x) = x
  - simple
  - needs space equal to the maximum value of x
  - may leave large set of holes
  - applicable only to integral keys
- General function
  - may use space judiciously
  - array size not dependent on the element values
  - may result in collisions
  - Applicable to arbitrary types

```
int hashfun(char *key, int H) {
        int val = 0;
        while (*key) val += *key++;
        return val % H;
}
```

Hash Functions

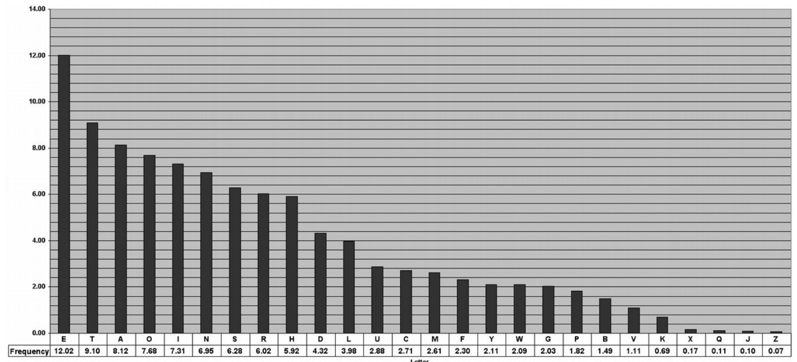
- Various hash functions
  - Integral value % H *(H is the hash table size)*
  - Sum of all the characters / bytes
  - ExOR of odd-indexed bits
  - Integral value / M *(similar values in nearby buckets)*
  - …
- Several hash functions are implemented at Arash Partow's webpage.
  - Bitwise (! | & ^ << >>)
  - Mathematical (+ *)
  - Lookup (prime numbers, magic numbers)

Hash Functions

- Application needs to choose the right functions.
  - using last two bits of a pointer’s address
  - using % 10 for values with least count of 10
  - using % 123456789 for student’s roll-number
  - using first character of a name to choose the bucket
  - …
- Index values must distribute across the array.
  - reduces number of conflicts
  - improves overall execution time
- But, hash functions should also be fast.

Index Distribution

- For equitable distribution, H is often a prime.
- Consider the following hash function:
  - `return (key[0] + 27 * key[1] + 729 * key[2]) % H;`
- Assumes at least two characters in the key
- Works well for several strings
- Skewed by first three characters
- Skewed if characters are not random



	E	T	A	O	I	N	S	R	H	D	L	U	C	M	F	Y	W	G	P	B	V	K	X	Q	J	Z
Frequency	12.02	9.10	8.12	7.68	7.31	6.95	6.28	6.02	5.92	4.32	3.98	2.88	2.71	2.61	2.30	2.11	2.09	2.03	1.82	1.49	1.11	0.69	0.17	0.11	0.10	0.07

A Better Hash Function

```
int hashfun(char *key, int H) {
    int val = 0;
    while (*key) val = (val << 5) + *key++;
    return val % H;
}
```

- Uses all the characters
- Distributes better
- Exploits Horner’s rule for faster computation
- Is no longer O(1), but proportional to key length.

Hash Table ADT

```
struct Hashtable {
    void insert(KeyType key, ElementType value);
    void remove(KeyType key);
    ElementType find(KeyType key);
};
```

In some applications, value may not be required.

If you get a *deja vu* feeling by looking at the ADT, don’t worry; that’s natural.

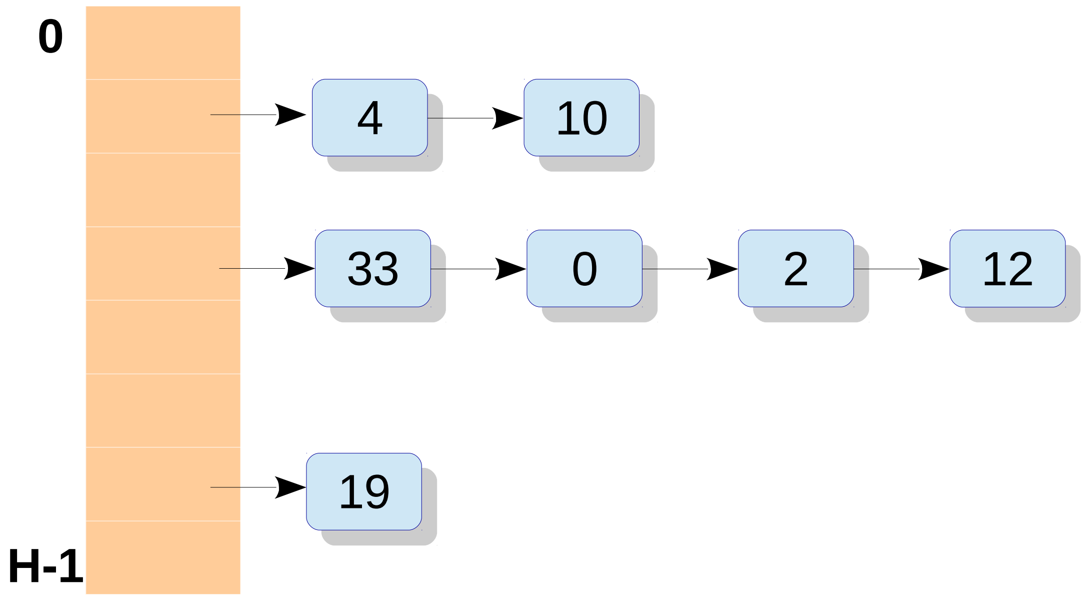
Pitfalls

- Prime numbers provide equitable distribution.
- Data distribution does not matter for hashing.
- Hashing is O(1).
- There exists a hash function which works well across all applications.
- It is always possible to increase the hash table size to reduce collisions.
- Hash function must contain modulus operator.

Collisions

- Without collisions, key need not be stored.
- A good distribution reduces collisions.
- Multiple ways to handle collision:
  - Chaining
  - Open Addressing (linear probing, quadratic probing, double hashing)
  - Rehashing

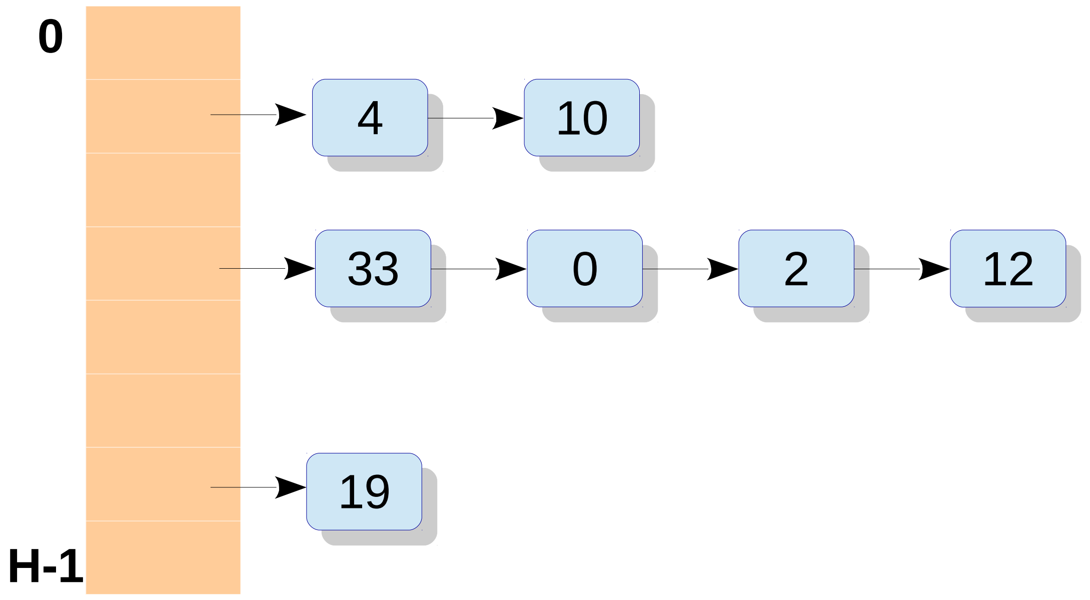
Chaining



- Allows arbitrary number of elements
- Can be used to cluster elements
- Can use List ADT
- List can actually be replaced by other data structures such as trees, or even hash tables!

Source: hash.cpp

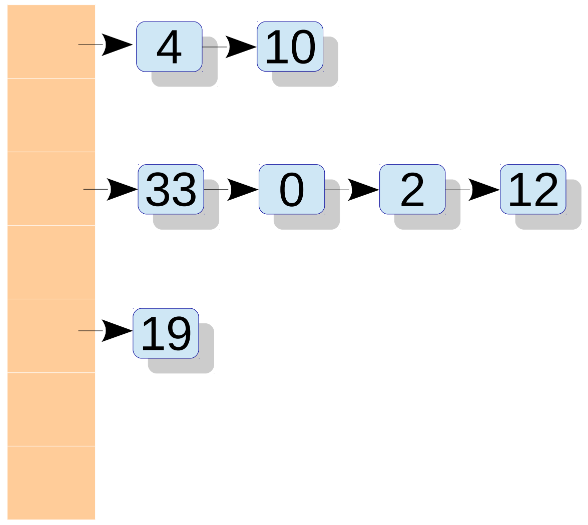
Chaining: Operations



- **Insert**: hash, then list insert.
  - May have to apply **find**.
- **Remove**: hash, then list remove.
- **Find**: hash, then list search.

Load Factor

- Load factor indicates fullness of a hashtable.
- LF = number of elements inserted / table size
  - In our example, LF = 7/7
- In case of chaining, load factor is the average chain-length.
  - Useful for calculating the average hashtable search complexity.
- Table size should be chosen to have LF close to 1.
  - But LF alone is not the precise measure.



Issues with Chaining

- Has low cache efficiency
- Uses pointers (makes code complicated)
- Need to maintain two data structures

Can we maintain elements in the array itself?

Open Addressing

- Has only the hashtable array.
- On collision, try some other position in the hashtable array.
- Formally, positions $(\mathrm{hash}(X) + F(i)) \% H$ are tried in succession.
  - $F(0) = 0$
- Function F is the collision resolution strategy.
  - $F(i) = i$ is linear probing
  - $F(i) = i^2$ is quadratic probing
  - $F(i) = i * \mathrm{hash}_2(i)$ is double hashing

Linear Probing

- $(\text{hash}(X) + F(i)) \% H$ and $F(i) = i$
- One may have a different linear function of i.
  - Identity function is the most common.
- Let H = 7, hash(X) = X % 7
- Elements: 4, 12, 70, 11, 20
- Elements: 20, 11, 4, 12, 70
- Elements: 11, 20, 12, 70, 4
  - Check number of elements colliding
  - Check total number of collisions

70	12	70
20	70	4
4	11	11
12	4	12
11	20	20

Classwork: Write codes for insert, remove and find.

Linear Probing: Operations

- insert

```
int index = hash(e);
for (int ii = 0; ii < H; ++ii) {
        if (arr[index] == emptycell) {
                arr[index] = e;
                return true;
        }
        index = (index + 1) % H;
}
std::cerr << "Hashtable is full.” << std::endl;
return false;
```

O(H)

Linear Probing: Operations

- remove

```
int index = hash(e);
 for (int ii = 0; ii < H; ++ii) {
        if (arr[index] == e) {
                arr[index] = deletedcell;
                return true;
        } else if (arr[index] == emptycell) {
                std::cout << e << " not present.\n";
                return false;
        }
        index = (index + 1) % H;
 }
 std::cerr << e << " not present.\n";
 return false;
```

O(H)

Linear Probing: Operations

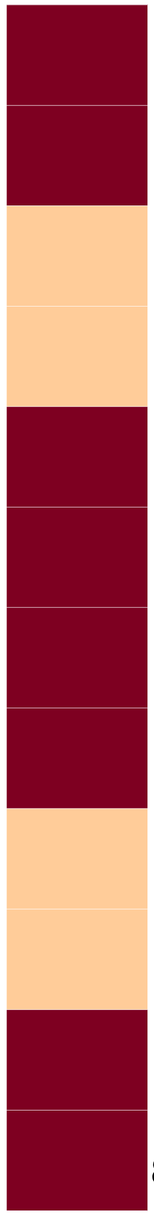
- find

```
int index = hash(e);
for (int ii = 0; ii < H; ++ii) {
        if (arr[index] == e) {
                std::cout << e << " is found” << std::endl;
                return true;
        } else if (arr[index] == emptycell) {
                std::cout << e << " not present.\n";
                return false;
        }
        index = (index + 1) % H;
}
std::cerr << e << " not present.\n";
return false;
```

O(H)

Primary Clustering

- Due to linear probing, groups of elements may get formed in the array.
- This phenomenon is called clustering.
- Clustering increases the number of collisions.
  - in turn, the execution time.
- Intuitively, this works against uniform distribution.
- Deletions are useful in such a scenario.
- Alternatively, probing should leave holes.



Quadratic Probing

- (hash(X) + F(i)) % H   and $F(i) = i^2$
- Distributes the indices better
  - Needs H to be prime
  - Practically needs load factor to be at max. 0.5
- Avoids primary clustering issue
  - Leaves holes
- Not guaranteed to cover all the indices

Quadratic Probing Guarantee

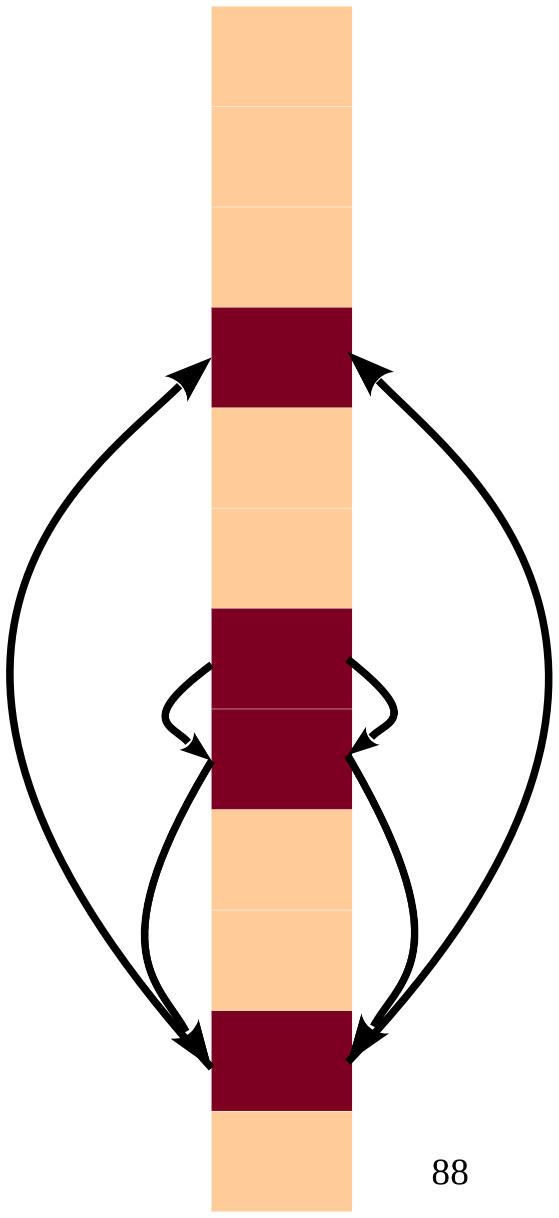
Theorem: *If table size is prime, then using quadratic probing, a new element can* <u>*always*</u> *be inserted if the table is at least half empty.*

Proof: The theorem means that when tried for indices hash(X) + i^2, we get at least H/2 different indices. At least one index is guaranteed to be empty since the hashtable is at least half empty.

Consider the first H/2 probes ($0 < i, j$). Let two arbitrary indices for different probes be hash(X) + i^2 and hash(X) + j^2, both (% H).
For the sake of contradiction, assume that two locations are the same.
Thus, i != j and hash(X) + i^2 == hash(X) + j^2 (% H)
i^2 == j^2 (% H)
(i – j)(i + j) == 0 (% H)
Thus, (i – j) == 0 or (i + j) == 0 (% H)
(i – j) == 0 is not possible since i != j.
(i + j) == 0 is not possible since $0 < i, j < H/2$.
Thus, first H/2 locations are distinct; and pigeon-hole principle allows the insertion.

Secondary Clustering

- Does quadratic probing avoid primary clustering?
- Does it lead to any issue?
  - Two elements mapping to the same initial index continue to conflict.
  - This is called secondary clustering
- Can be resolved using double hashing



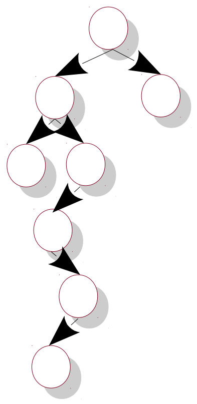
Double Hashing

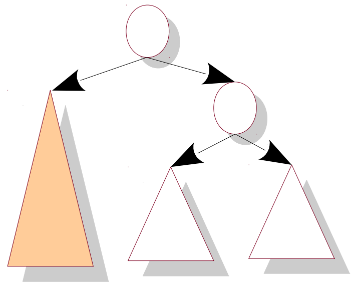
- $(\text{hash}(X) + F(i)) \% H$ and $F(i) = i * \text{hash}_2(i)$
- Distributes elements further
- Avoids primary as well as secondary clustering
- hash_2 must not evaluate to zero
- H being prime becomes more important from linear to quadratic, and from quadratic to double.
- Needs another hash function

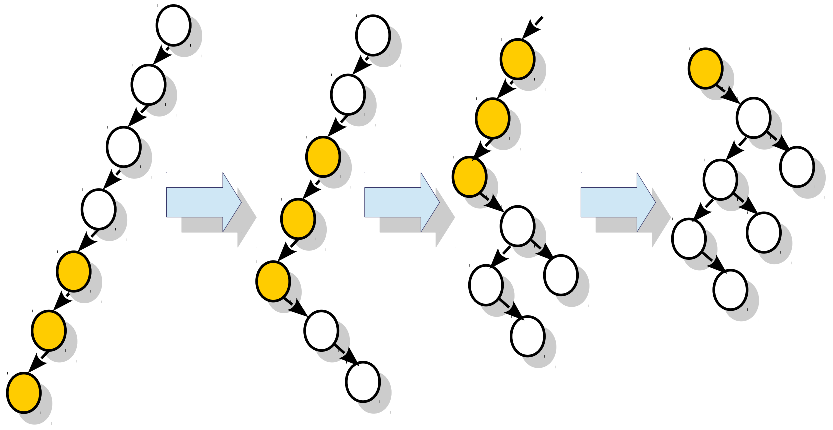
Source: hash-open.h and .cpp

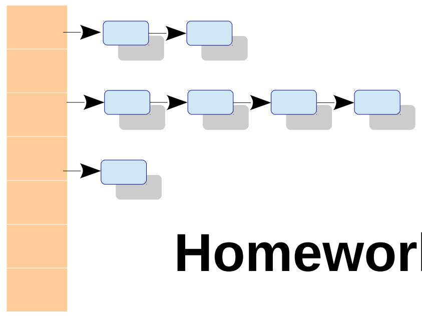
Learning Outcomes

- BST: add, remove, find
  - Implementation, complexity
- AVL: add, remove, find
- Splay Trees: splaying
- Hash Tables
  - Chaining
  - Linear probing
  - Quadratic probing
  - Double hashing









Homework: Exercises in Chapter 5.