Balanced Trees

Computing 2 COMP1927 16x1

BST Performance Issues

- Performance of insert and search good on average, but poor for some (common) special cases (items inserted in sorted order O(n))
- Goal:
  - build binary search trees with worst case performance of O(log n)
- A perfectly balanced binary tree (weight/size balanced)
  - |size(LeftSubtree) - size(RightSubtree)| < 2 for every node
- A less stringent definition (height balanced)
  - |height(leftsubtree) - height(rightSubtree)| < 2 for every node

Are these Trees Balanced?

```
    4
   / \
  2   5
 / \
1   3
```

```
  4
 / \
2   5
 \   \
  3   6
```

```
      4
     / \
    3   5
   /     \
  2       6
 /         \
1           7
```

```
    4
   /
  2
 / \
1   3
```

Approach 1: Global Rebalancing

- Insert nodes normally
- Have a function to rebalance the whole tree
  - The tree becomes perfectly balanced
  - How? The best key to have at the root of a tree is
    - The median
    - Will partition all the keys equally into left and right sub-trees

Approach 1: Basic Idea

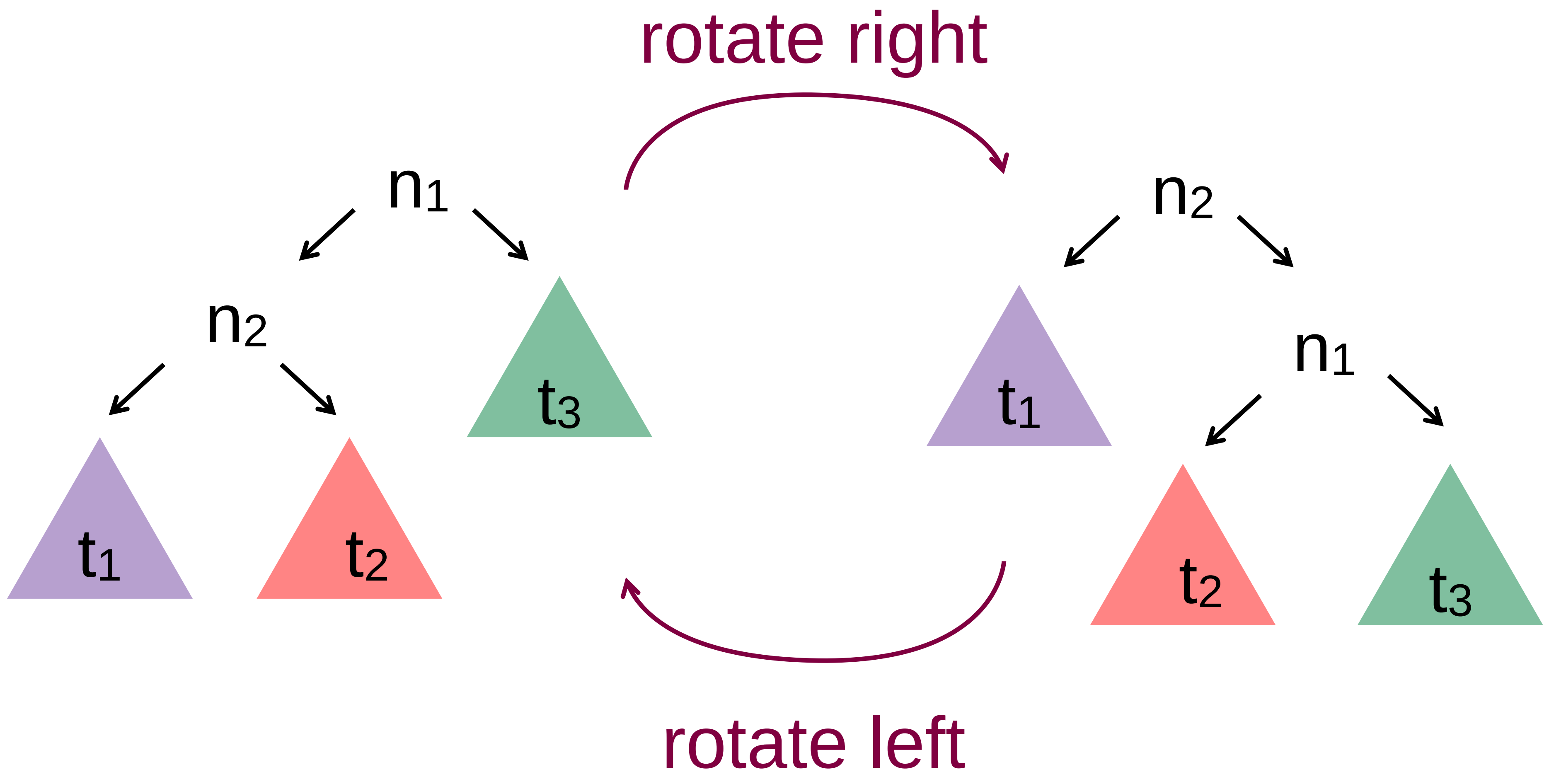
- Move median to the root of the tree
  - Get the median of the left sub-tree and move it to the root of the left sub-tree
  - Get the median of the right sub-tree and move it to the root of the right sub-tree
- How can we even find the median in a size n tree?
  - Median will be the (n/2th) node - similar to finding the ith item in a BST
  - Need to find n/2th node and move it to the root

ROTATIONS

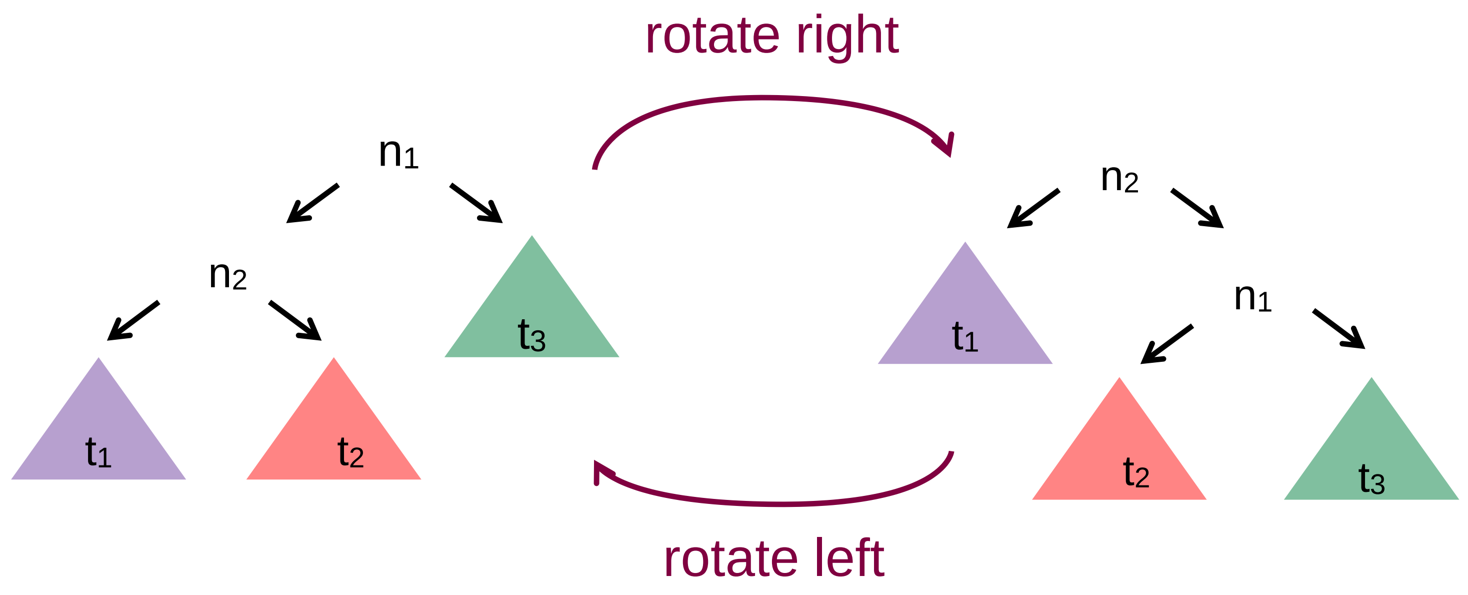
- We can move nodes up to the root using rotations
- Left rotation
  - Makes the original root the LEFT sub-child of the new root
- Right rotation
  - Makes the original root the RIGHT sub-child of the new root

MOVING NODES THROUGH ROTATION

- Move node n_2 up
  - $t_1 < n_2 < t_2 < n_1 < t_3$
- rotation leaves the relative order of the nodes intact!
- we can use it to successively move a node up to the root



MOVE NODES THROUGH ROTATION

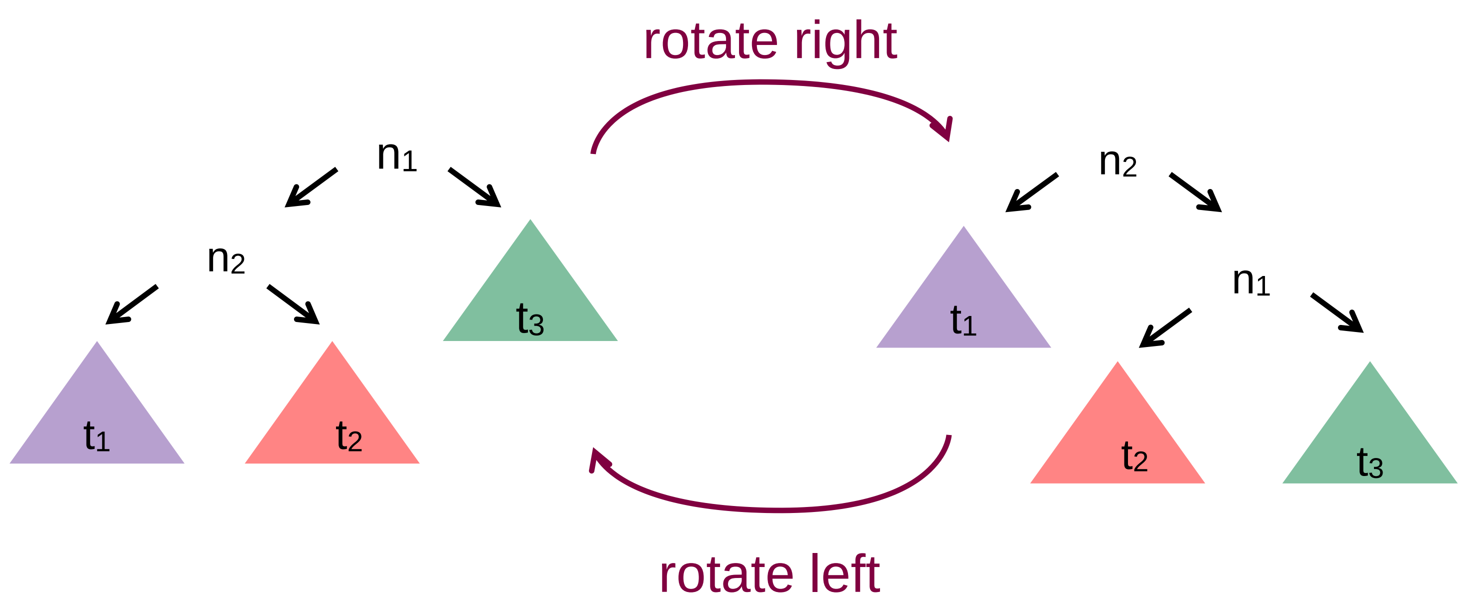


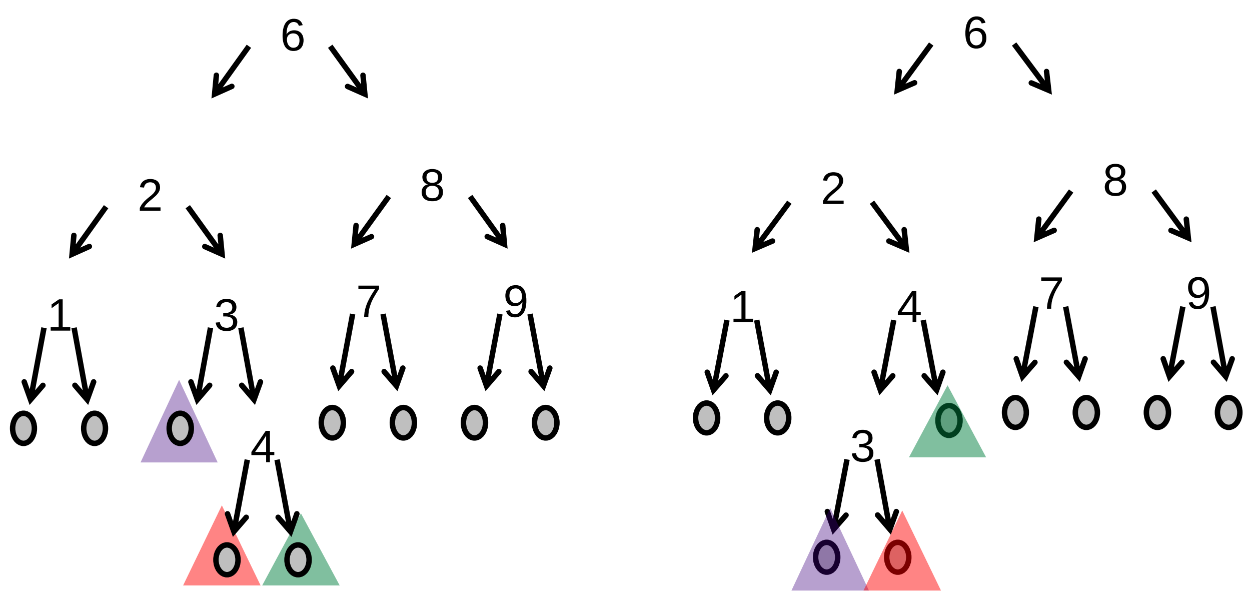
```
//Beware: this does not update the size fields in the nodes
link rotateRight(link n1) {
      if(n1 == NULL || n1 == emptyTree) return n1;
      link n2 = n1->left;
      if(n2 == emptyTree) return n1;
      n1->left = n2->right;
      n2->right = n1;
      return n2;
}

//Left rotation is similar with n1/n2 switched and left/right
switched.
```

Partitioning

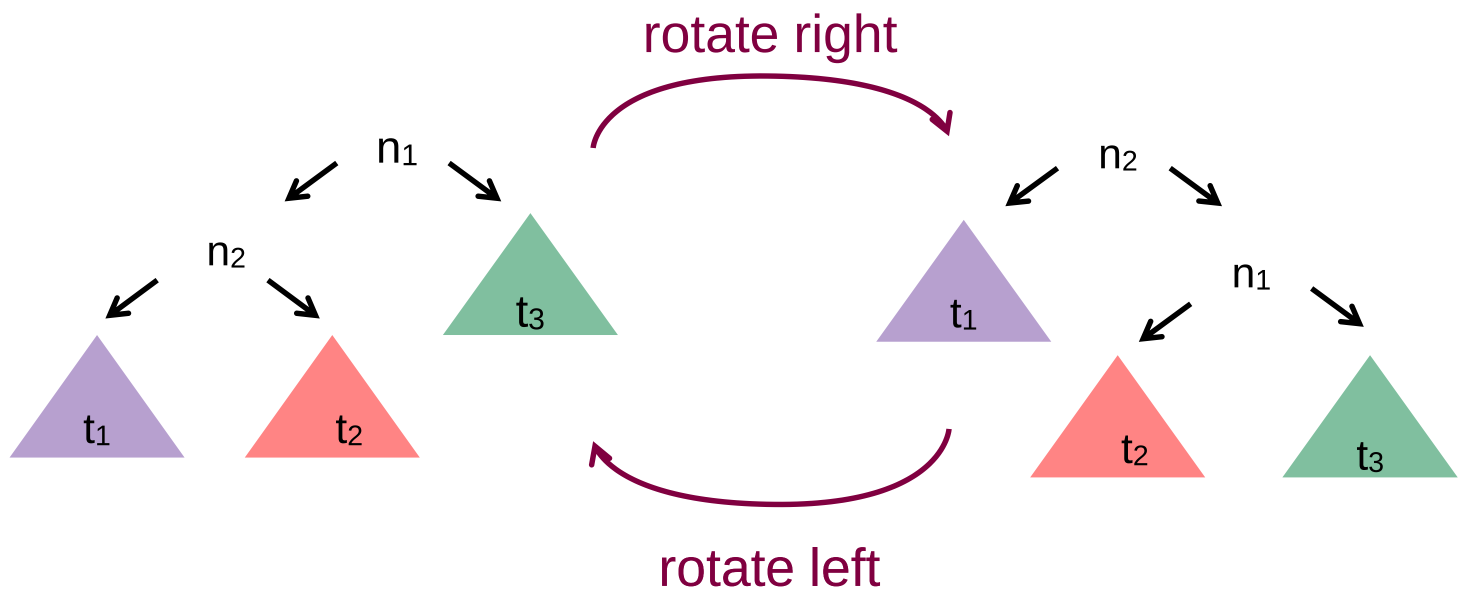
- Partition:
  - move the *k*th element of a tree up to the root
  - similar to select:

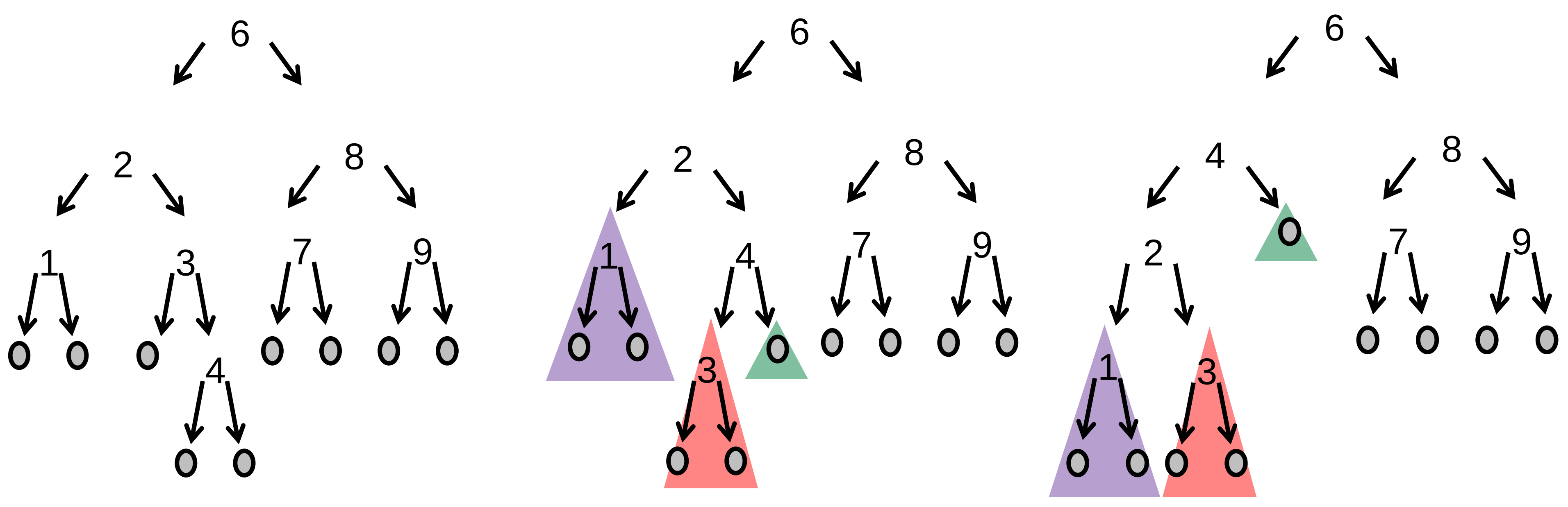
```
link partitionR (link tree, int k) {
  if (tree == emptyTree) {
        return tree;
  }
  int leftSize = tree->left->size;
  if (leftSize > k) {
      tree->left = partitionR (tree->left, k));
      tree = rotateR (tree);
  }
  if (leftSize < k) {
     tree->right = partitionR (tree->right, k - 1 -leftSize);
     tree = rotateL (tree);
  }
  return (tree);
}
```


rotate right
n_1
n_2
t_1
t_2
t_3
n_2
t_1
n_1
t_2
t_3
rotate left

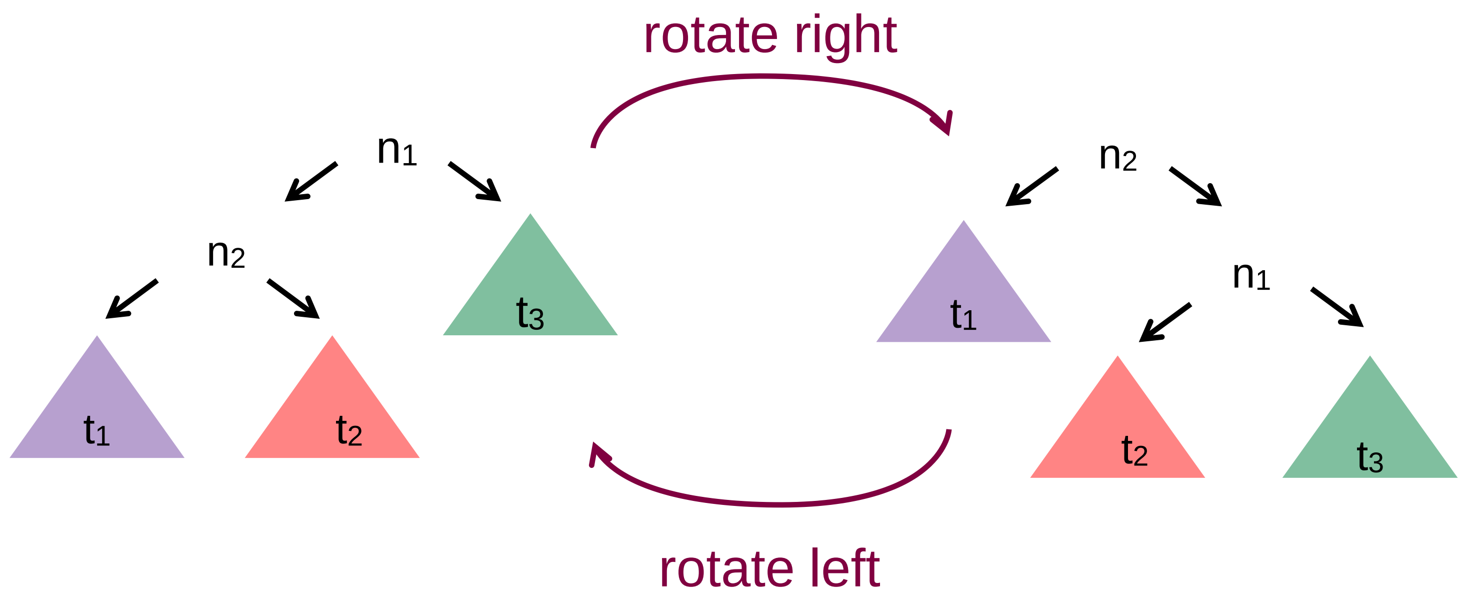

6
2
8
1
3
7
9
4
6
2
8
1
4
7
9
3

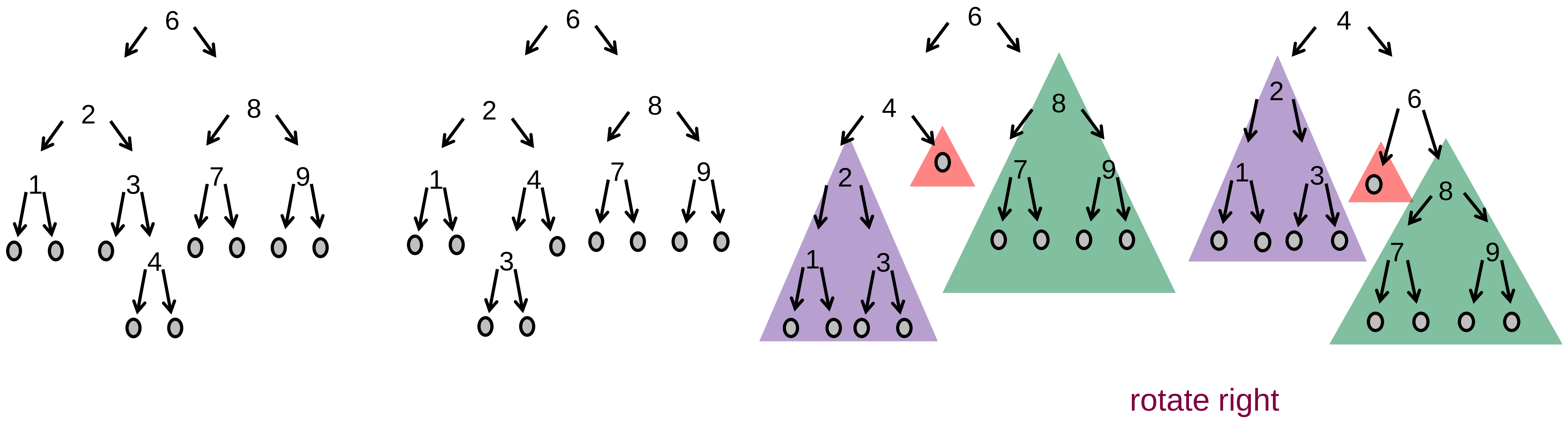
rotate left


rotate right
n1
n2
t1
t2
t3
n2
t1
n1
t2
t3
rotate left


6
2
8
1
3
7
9
4
6
2
8
1
4
7
9
3
6
4
8
2
7
9
1
3

rotate left


rotate right
n₁
n₂
t₁
t₂
t₃
n₂
n₁
t₁
t₂
t₃
rotate left


6
2
8
1
3
7
9
4
6
2
8
1
4
7
9
3
6
4
8
2
1
3
7
9
4
2
6
1
3
8
7
9
rotate right

Approach 1: Global Rebalancing

- Move the median node to the root by partitioning on size/2
  - Balance the left sub-tree
  - Balance the right sub-tree

```
link balance(link tree){
    if(tree != emptyTree){
        if(tree->size >= 2){
            tree = partition(tree,tree->size/2);
            tree->left = balance(tree->left);
            tree->right = balance(tree->right);
        }
    }
    return tree;
}
```

Approach 1: Problems

- Cost of rebalancing – O(n) for many trees or O(nlogn) for degenerate trees
- What if we insert more keys?
  - Rebalance every time – too expensive
  - Rebalance periodically
    - Every ‘k’ insertions
    - Rebalance when the “unbalance” exceeds some threshold
- Either way, we tolerate worse search performance for periods of time. Does it solve the problem for dynamic trees? ... Not really.

Approach 2: Local Rebalancing

- Global approach walks through every node of the tree and balances its sub-trees
  - Perfectly balanced tree
- Local approach
  - do incremental operations to improve the balance of the over-all tree
  - Tree may not end up perfectly balanced

LOCAL APPROACHES TO REBALANCING

- Randomisation:
  - the worst case for binary search trees occurs relatively frequently (partially sorted input)
  - use random decision making to dramatically reduce chance of worst case scenario
- Amortisation:
  - do extra work at one time to avoid more work later
- Optimisation:
  - maintain structural information to be able to provide performance guarantees

Randomised BST

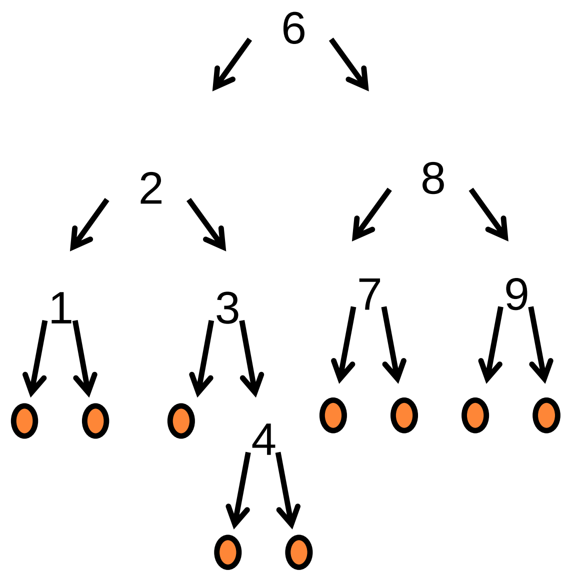
- BST ADT typically has no control over the order keys are supplied.
- To minimise the probability of ending up with a degenerate tree, we make a randomised decision at which level to insert a node.
  - at each level, the probability depends on the size of the remaining tree
    - Do normal leaf insertion most of the time
    - Randomly do insertion at **root**.
      - Insert new item at the root of the appropriate sub-tree
      - Rotate it to the root of the main tree

BST: INSERTING AT THE ROOT

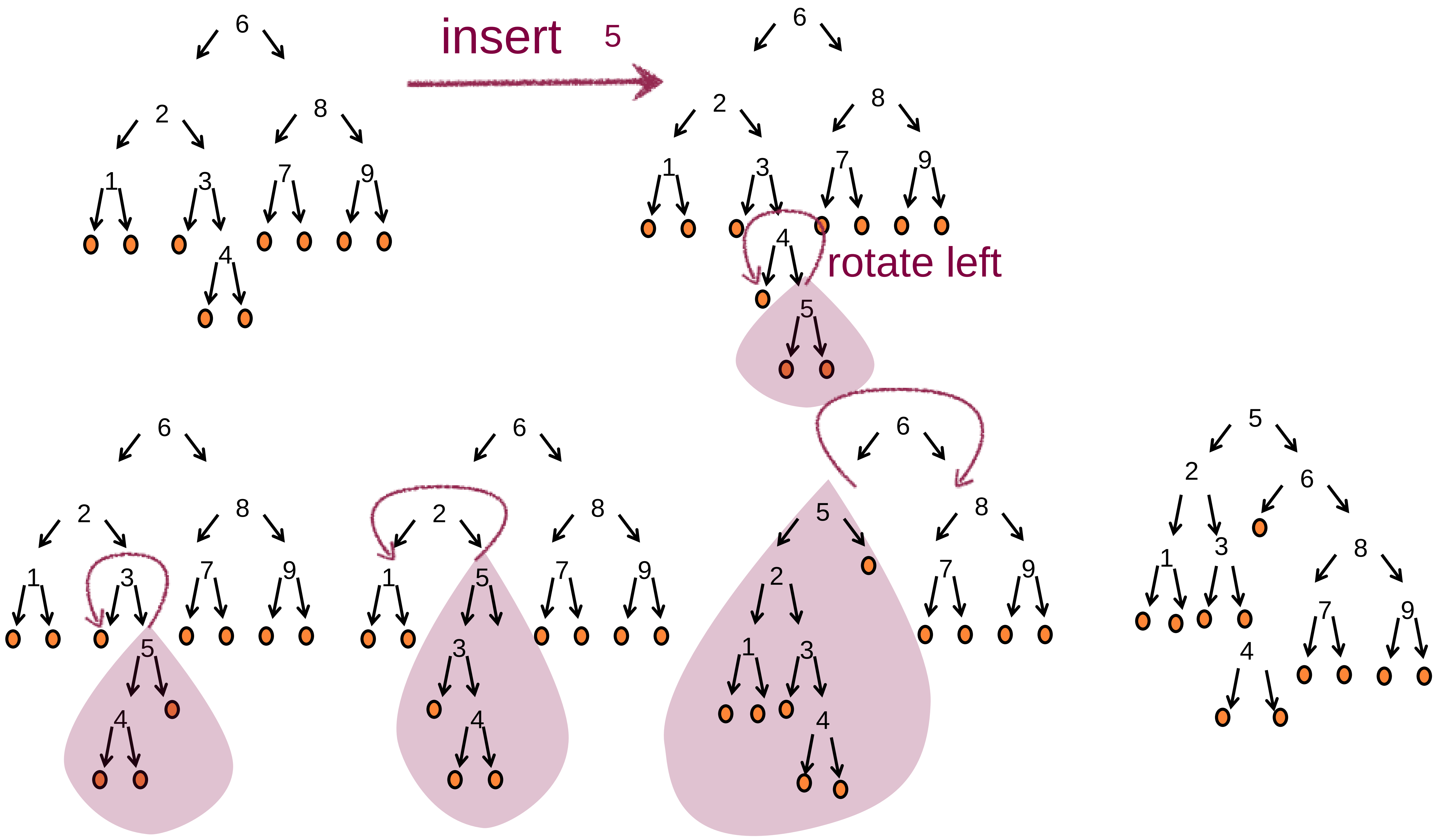
- Let us start with a simpler version of the problem:
  - How can we insert a node at the root instead of the leaves?
  - Problem:
    - this potentially already requires to re-arrange nodes in the whole tree
  - Solution:
    - we insert at the leaf position and move it up the tree without changing the relative order of the items

INSERTING AT ROOT

- Inserting at the root:
  - base case:
    - tree is empty
  - recursive case:
    1. insert it at root of appropriate subtree
    2. lift root of subtree by rotation



◦Inserting at Root



INSERTING AT ROOT

- Almost like insert at leaf:

```
link insertAtRootR (link currentLink, Item item) {
   if (currentLink == emptyTree) {
       return (NEW (item, emptyTree, emptyTree, 1));
   }

   if (less (key (item), key (currentLink->item))) {
       currentLink->left = insertAtRootR (currentLink->left, item);

       currentLink = rotateRight (currentLink);
   } else {

       currentLink->right = insertAtRootR (currentLink->right, item);

       currentLink = rotateLeft (currentLink);
   }

   return (currentLink);
}
```

- What is the work complexity of root insertion?
  - same as insertion at leaf: O($log\ n$)

INSERTING AT ROOT

- same work complexity as insertion at leaf, but extra actual work done for each insertion
- recently inserted items are close to the root
  - access time less for items inserted most recently
  - depending on the application. this might be a significant advantage

Randomised BST

- Randomised insertion:

```
link insertRand (link currTree, Item item) {
   Key currKey = key (currTree->item);

   if (currTree == emptyTree) {
       return NEW (item, emptyTree, emptyTree, 1);
   }

   if (rand () < RAND_MAX/(currTree->size+1)) {
       return (insertRootR (currTree, item));
   } else if (less (key (item), currKey) {
       currTree->left = insertRand (currTree->left, item);
   } else {
        currTree->right = insertRand (currTree->right, item);
   }

   currTree->size++;
   return currTree;
}
```

Randomised Trees

- Properties:
  - Building a randomised BST is equivalent to building a standard BST from a random initial permutation of keys
  - Worst, best and average case performance are the same as for standard BST, but no penalty if initial sequence is ordered or partially ordered

Randomised Trees Deletion

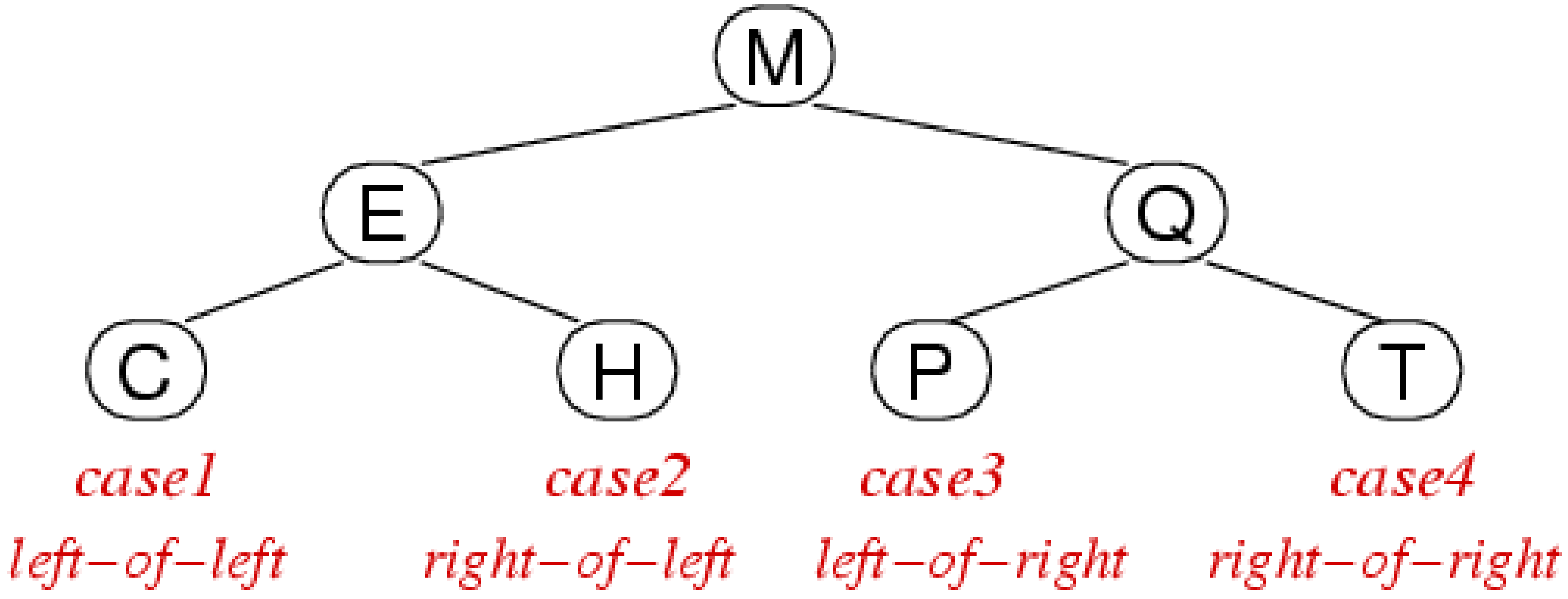
- Can use a similar approach for deletion
  - Make randomised decision whether to replace the deleted node with
    - in-order successor from right sub-tree
    - in-order predecessor from left sub-tree

Amortisation: Splay Trees

- Idea:
  - Whenever an operation has to `walk down` the spine of a tree, improve balance of tree
  - Use root insertion, but with a slight twist: whenever a node has to move either two successive left or two right rotations to move up, move the parent first
    - considers parent-child-grandchild orientation
- Some splay tree implementations also do rotation-in-search:
  - The node of the most recently searched for item (or last node in path of a dead end search) becomes the new root.
  - can improve balance of tree, but makes search more expensive

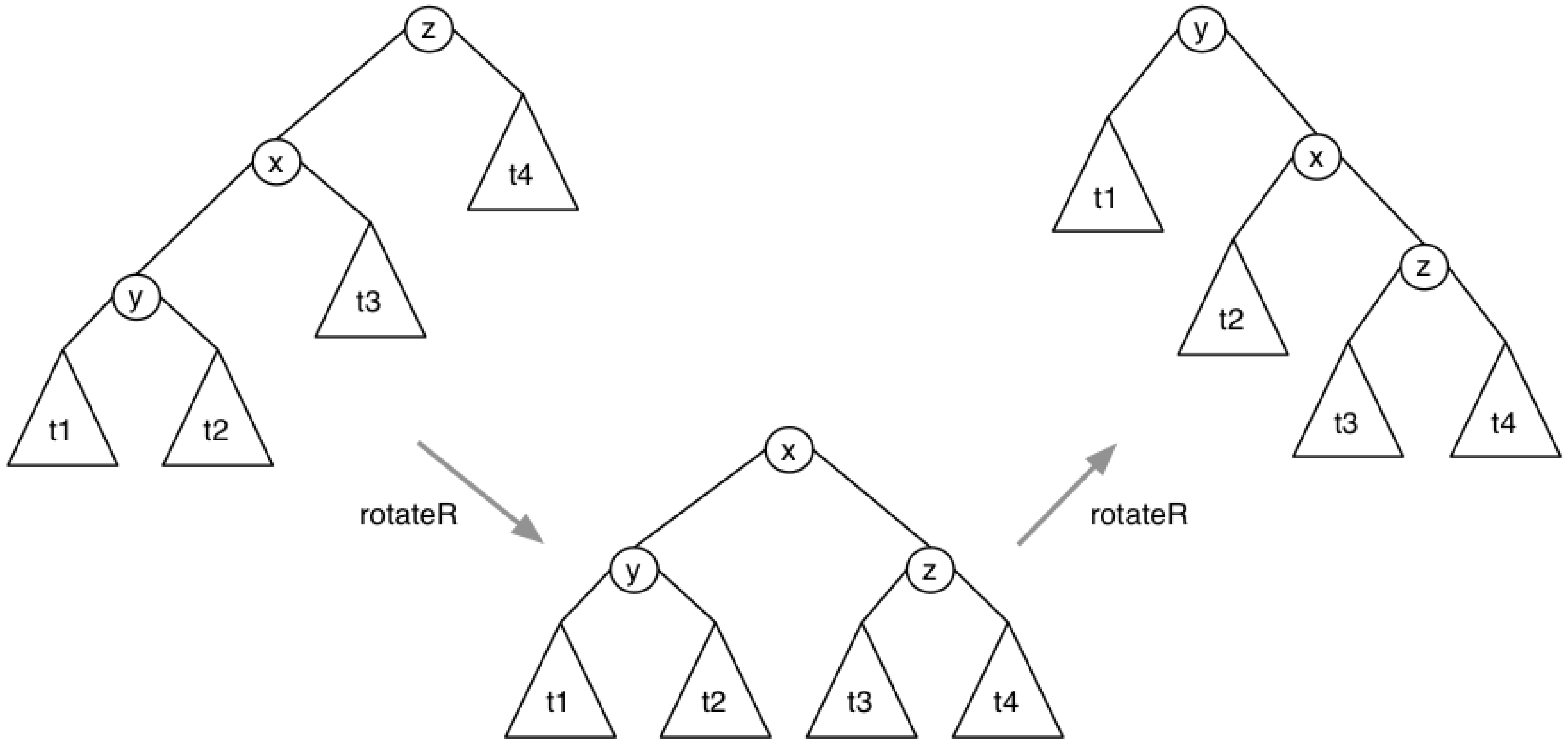
SPLAY TREE DOUBLE ROTATION CASES

- Cases for splay tree double-rotations:
  - case 1: grandchild is left-child of left-child
  - case 2: grandchild is right-child of left-child
  - case 3: grandchild is left-child of right-child
  - case 4: grandchild is right-child of right-child



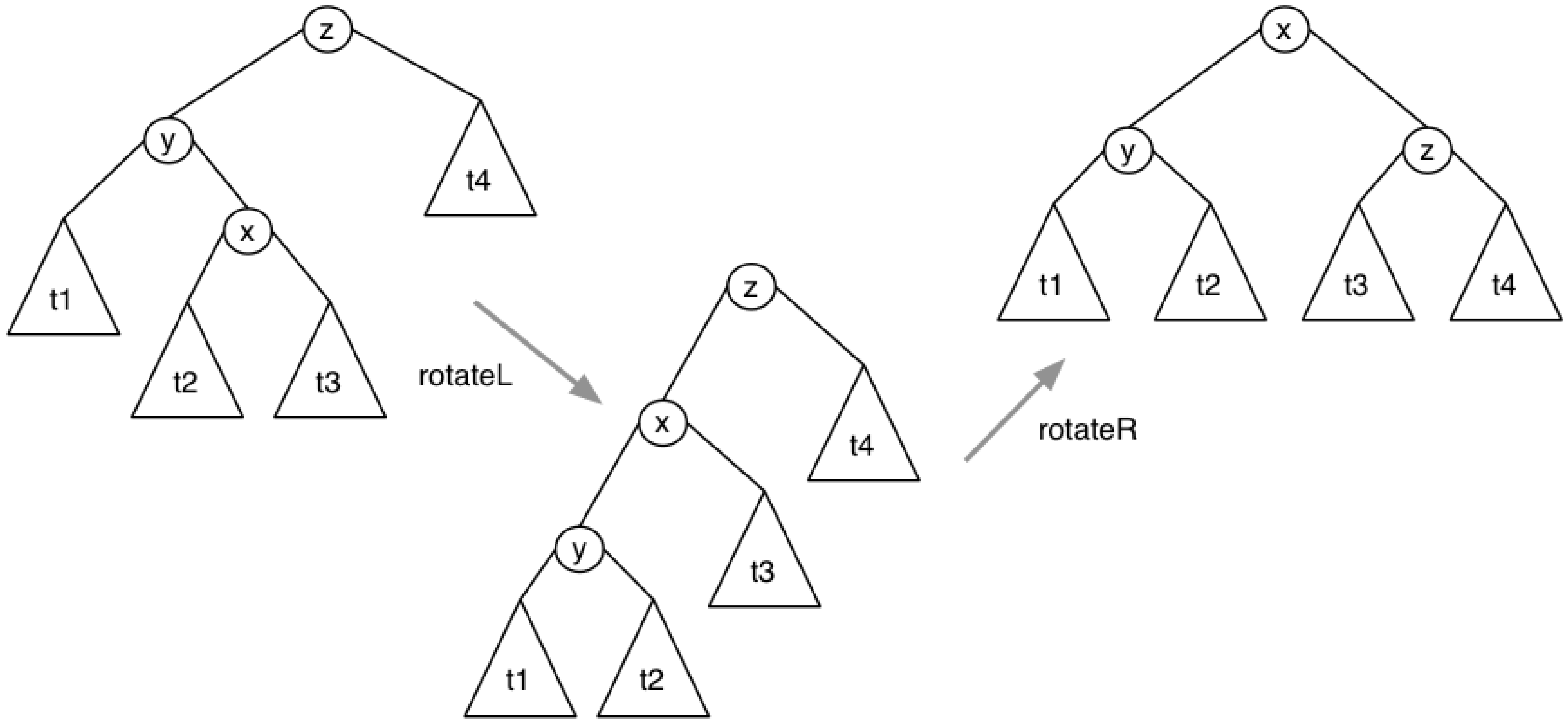
Double Rotation: Left Of Left

- Rotate at y’s grandparent z first
- Then rotate a parent x

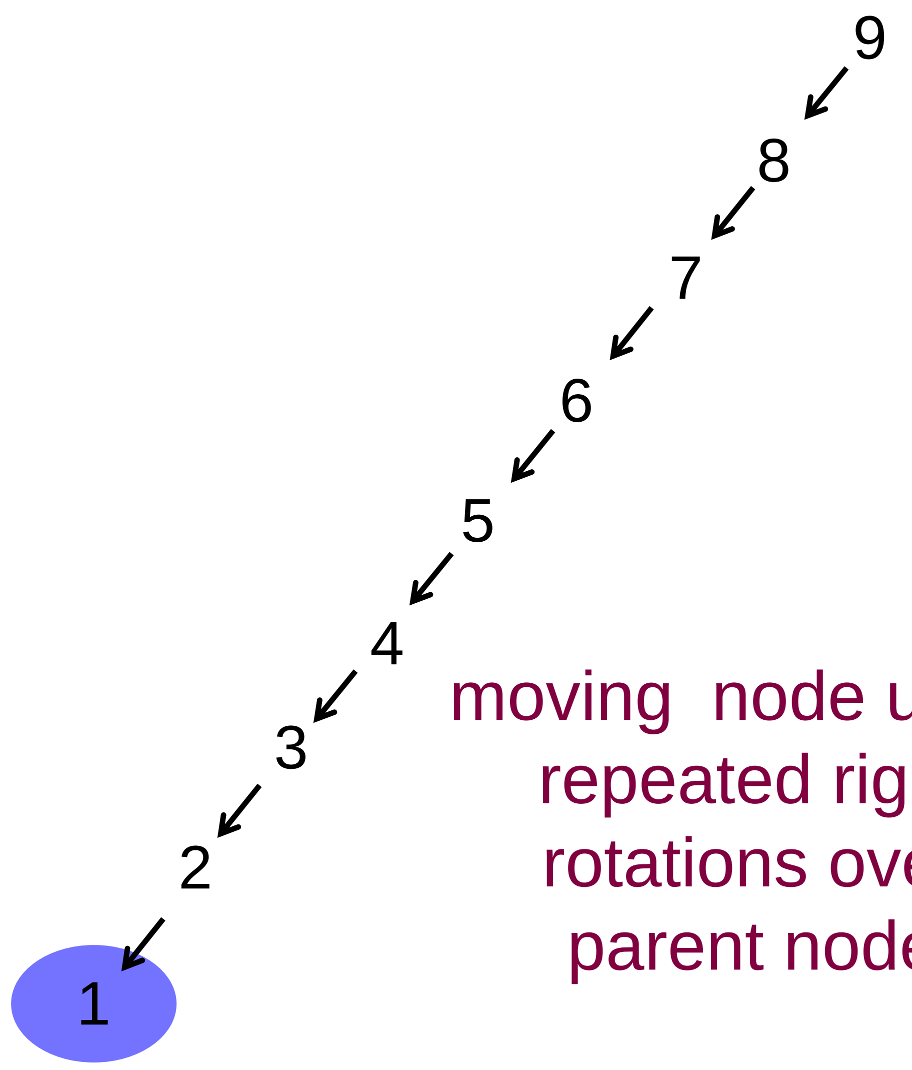


Double Rotation: Right of Left

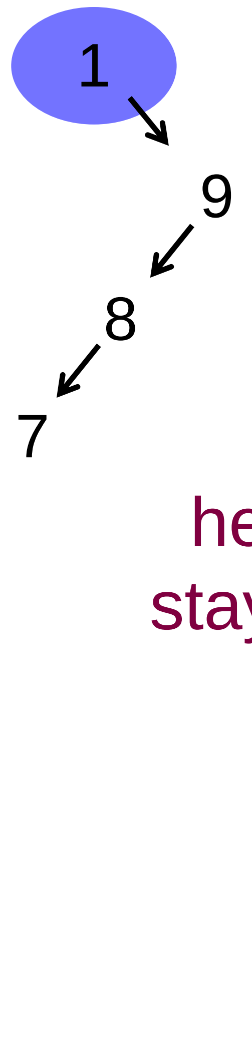
- Just rotate at x's parent, y, then at x's parent z



- Worst case example for normal root insertion: move smallest item up a degenerated tree using normal right rotations from leaf.
-



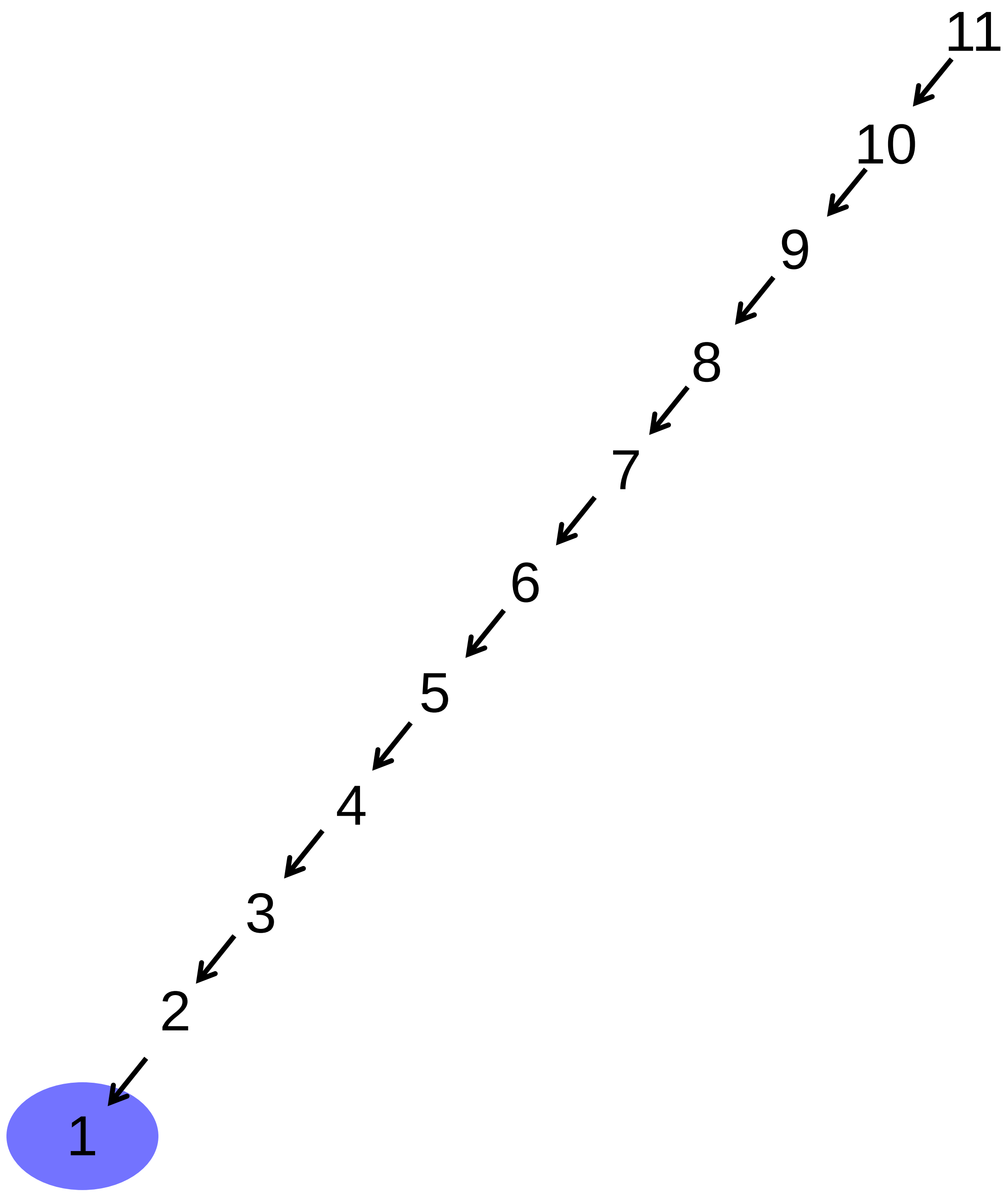
moving node up by repeated right rotations over parent node

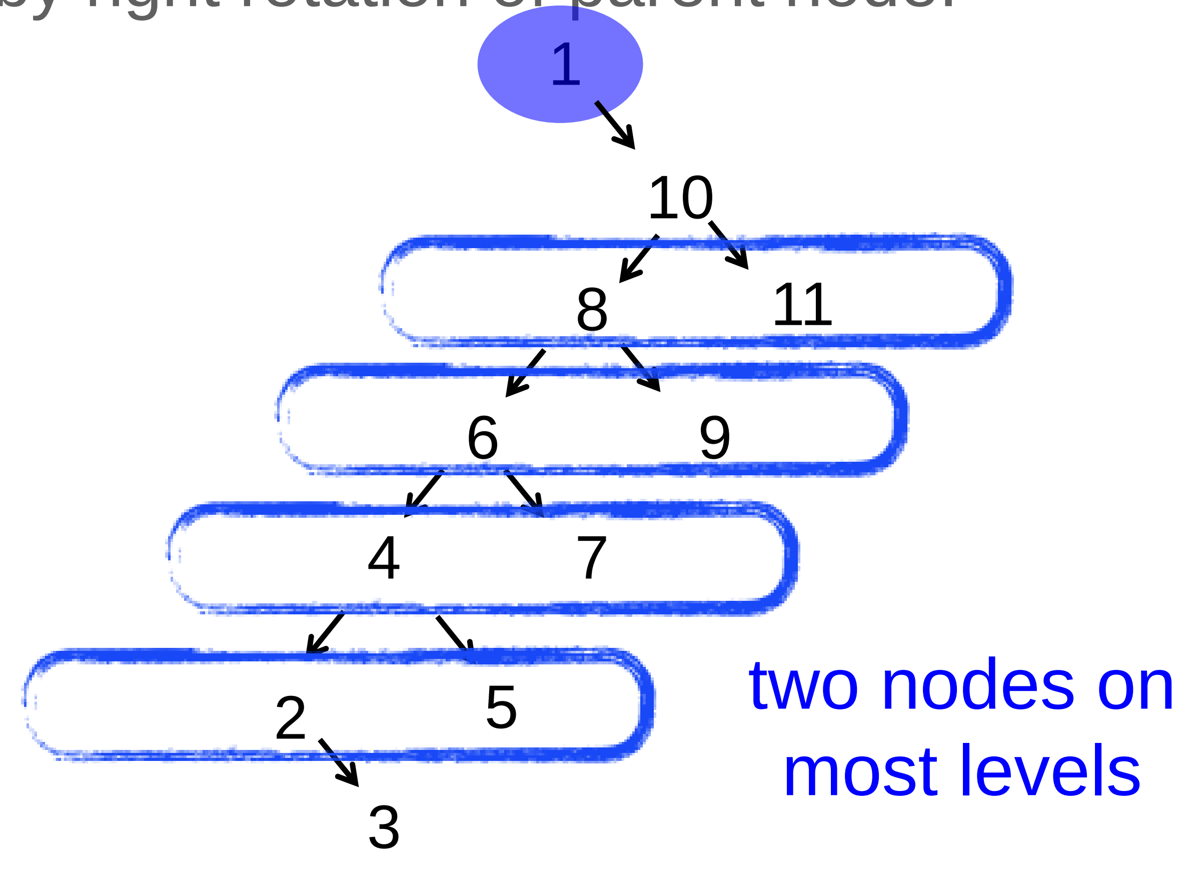


height of tree stays the same!

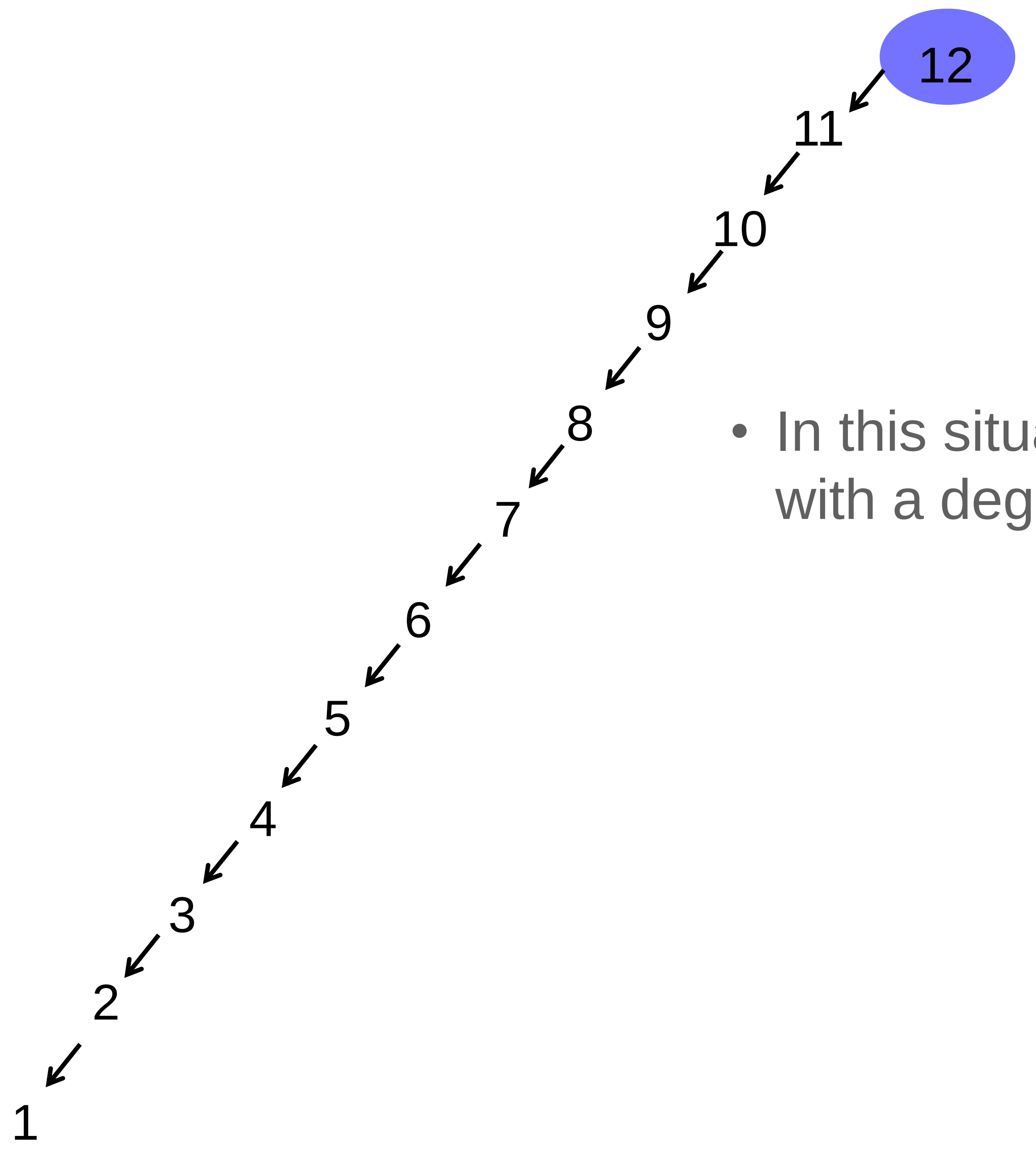
2

- Example worst case splay insertion: After inserting 1 in this worst case degenerate tree: move smallest item up a degenerated tree using right rotation of grand parent node, followed by right rotation of parent node:





- Example Other worst case : Inserting 12 into this degenerate tree. There would be no grandparent, relationship between 11 and 12 as 11 has no right child. So we just insert 12 as the parent and make 11 the left child. (The same as if we inserted 12 then did a rotate left at 12’s parent).



- In this situation insertion is O(1) but we are left with a degenerate tree.

WORK COMPLEXITY OF SPLAY TREE OPERATIONS

- Insertion
  - worst case (wrt work): item is inserted at the end of a degenerate tree
    - O(n) steps necessary, but tree height reduced by a factor of two
  - worst case (wrt resulting tree): item inserted at the root of a degenerate tree
    - Constant number of steps necessary
  - even in the worst case, it's not possible to repeatedly have O(n) steps for insertion

Work Complexity of Splay tree operations

- Assuming we splay for both insert and search
- Assume N initial inserts, then M searches
  - NlogN insert cost
  - MlogN search cost
- Gives good (amortized) cost overall. But no guarantee for any individual operation;
  - worst-case behaviour may still be O(N)
- It is based on the idea that if you recently used something you'll likely need it again soon
  - keeps the most commonly used data near the top

Exercise

- Insert the following keys into a splay tree:

  10 5 12 11 3

- Final Solution

```
   3
    \
     10
    /  \
   5    11
          \
           12
```