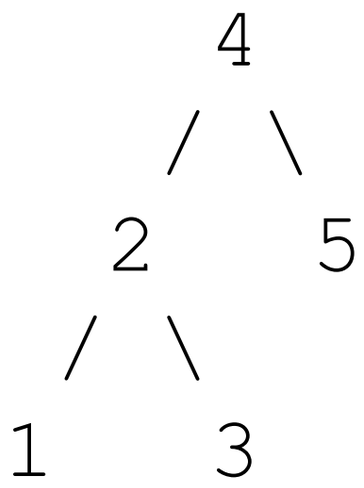
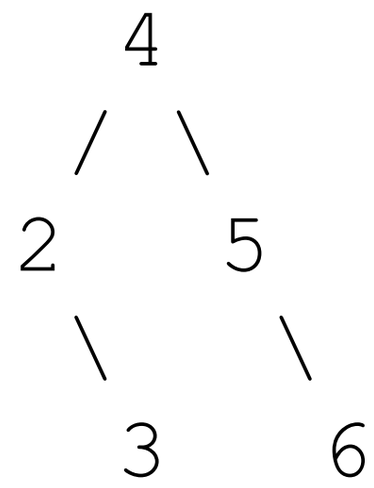
Balanced Trees

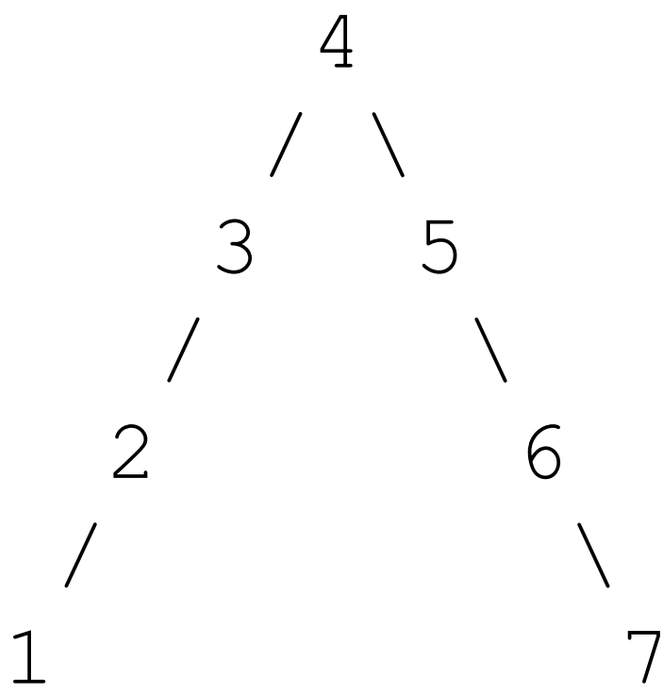
BST Performance Issues

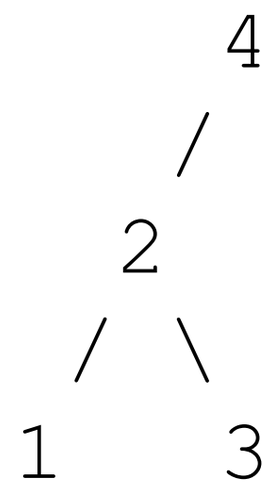
- Performance of insert and search good on average, but poor for some (common) special cases (items inserted in sorted order O(n))
- Goal:
  - build binary search trees with worst case performance of O(log n)
- A perfectly balanced binary tree (weight balanced)
  - height of log n
  - |size(LeftSubtree)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

Are these Trees Balanced?









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

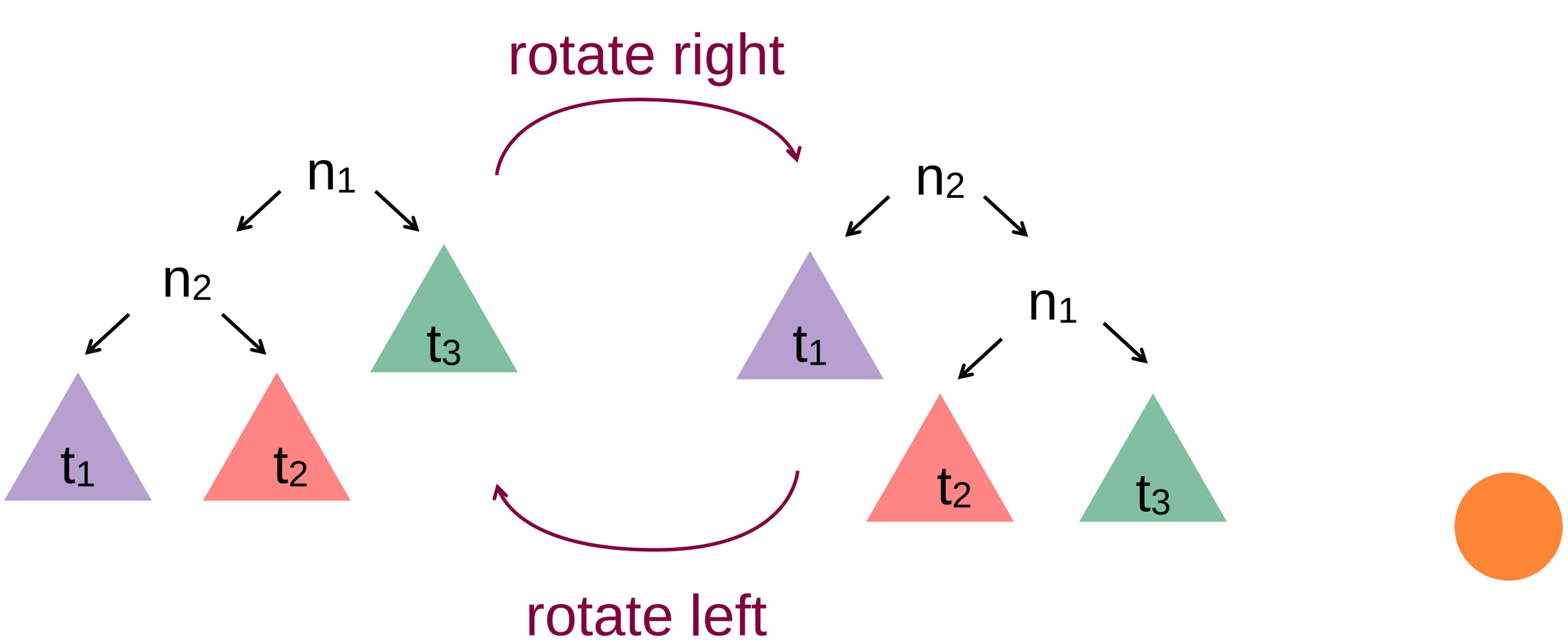
- Move median to the root of the tree
  - Get the median of the left sub-tree and move it to the root of the left sub-tree
  - Get the median of the right sub-tree and move it to the root of the right sub-tree
- How can we even find the median in a size n tree?
  - Median will be the (n/2th) node - similar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

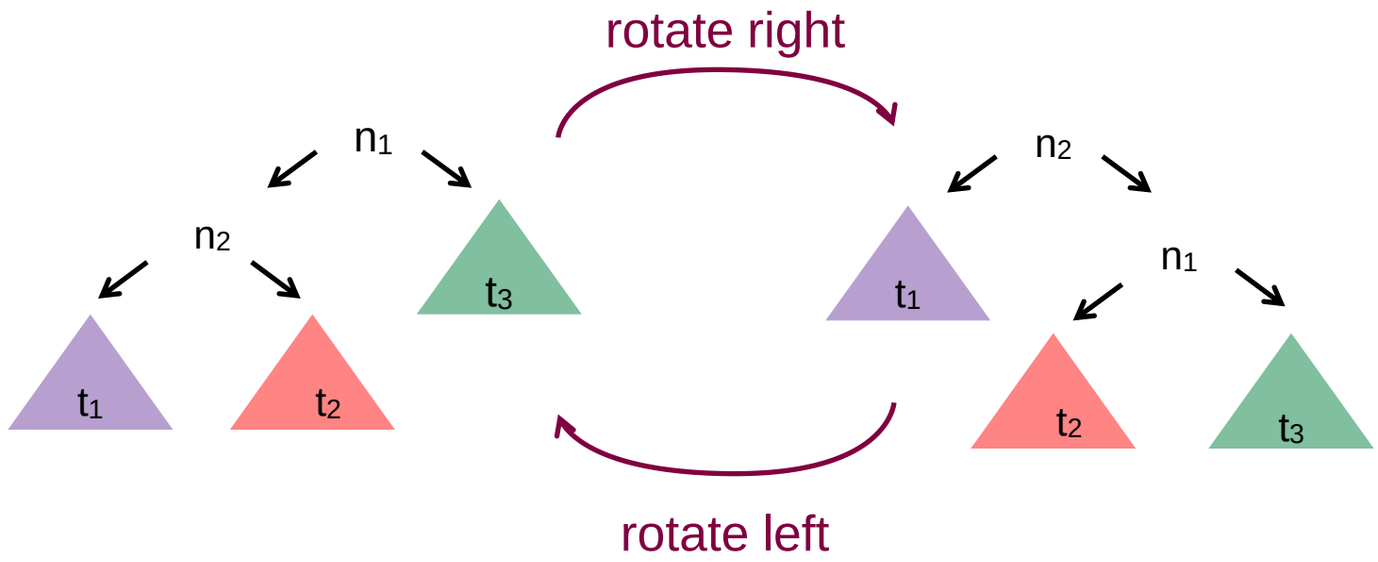
Rotations

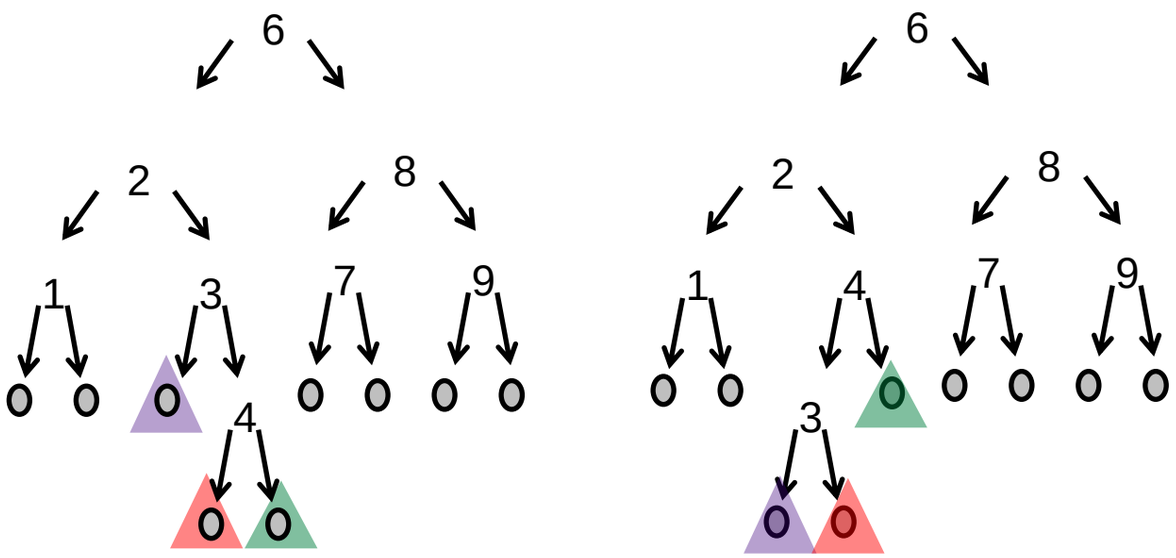
- We can move nodes up to the root using rotations
- Left rotation
  - Makes the original root the LEFT sub-child of the new root
- Right rotation
  - Makes the original root the RIGHT sub-child of the new root

Moving nodes through rotation

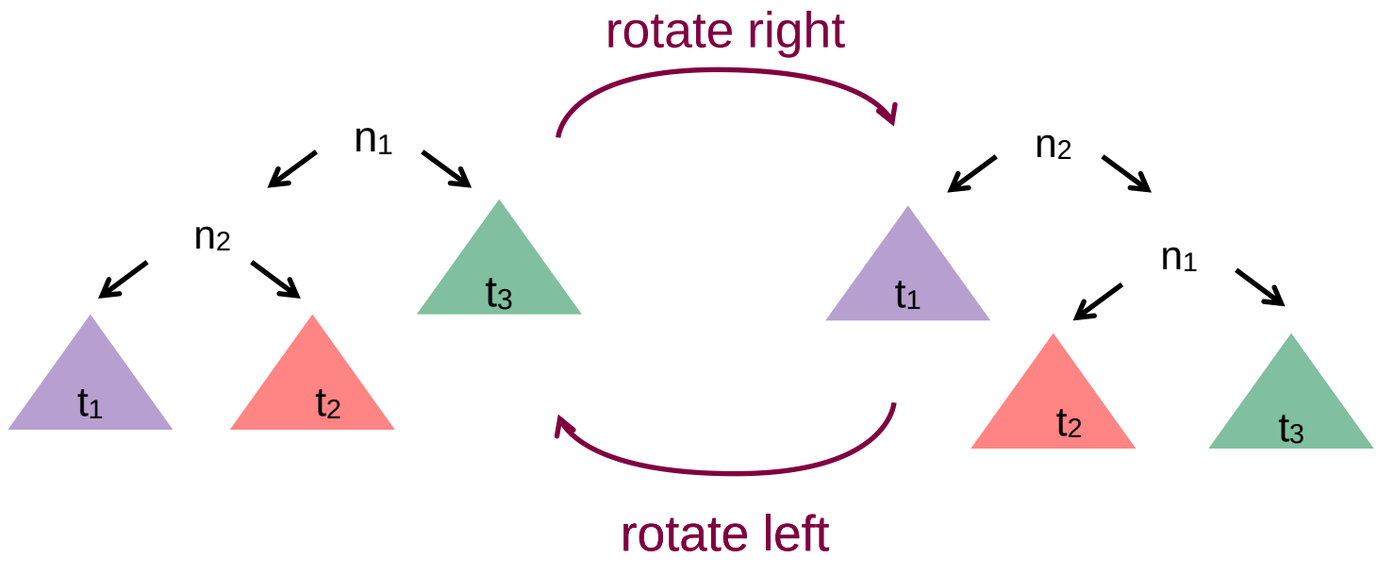
- Move node n_2 up
  - $t_1 < n_2 < t_2 < n_1 < t_3$
- rotation leaves the relative order of the nodes intact!
- we can use it to successively move a node up to the root

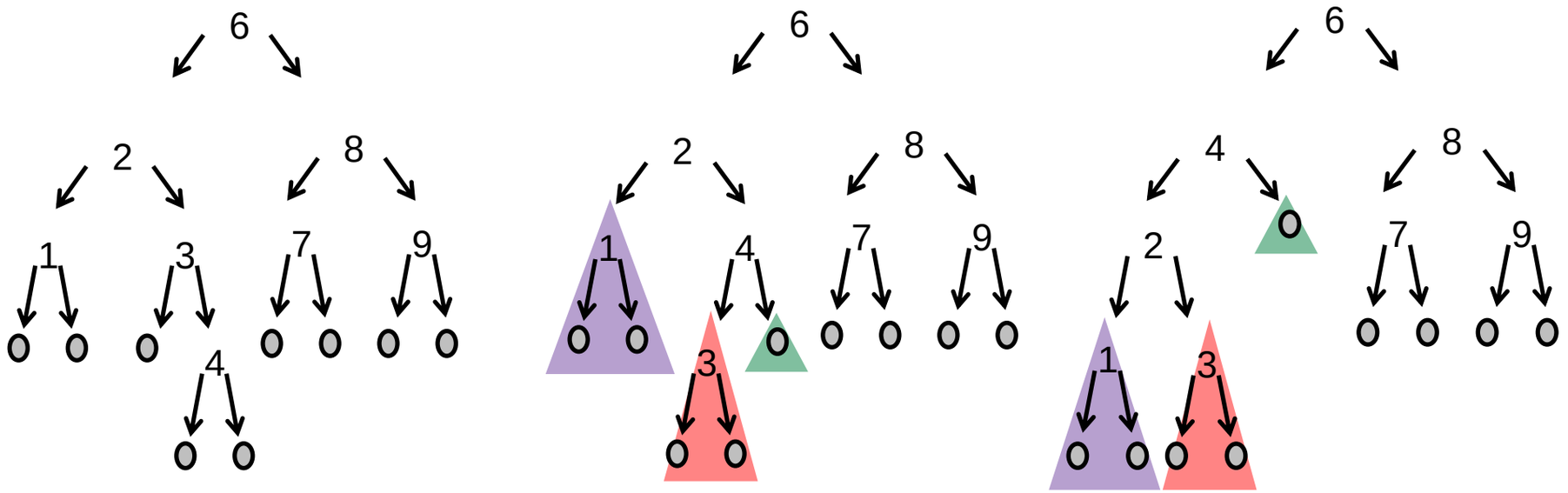



rotate right
n1
n2
t3
t1
t2
n2
n1
t1
t2
t3
rotate left

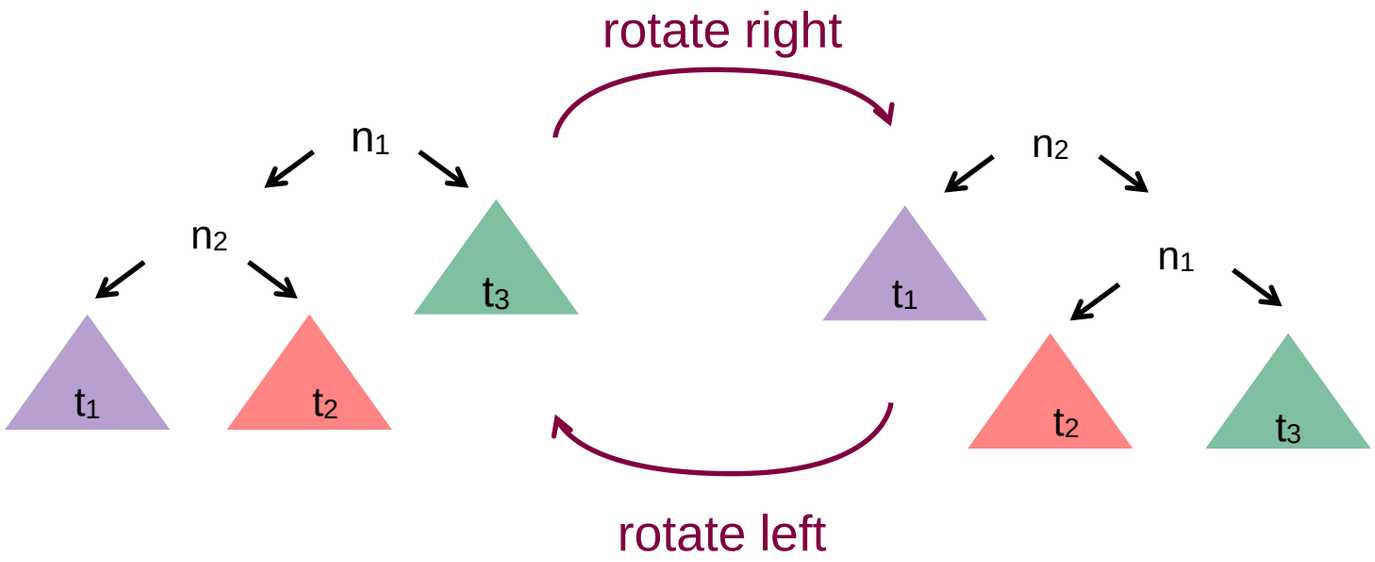

6
2
8
1
3
7
9
4
6
2
8
1
4
7
9
3

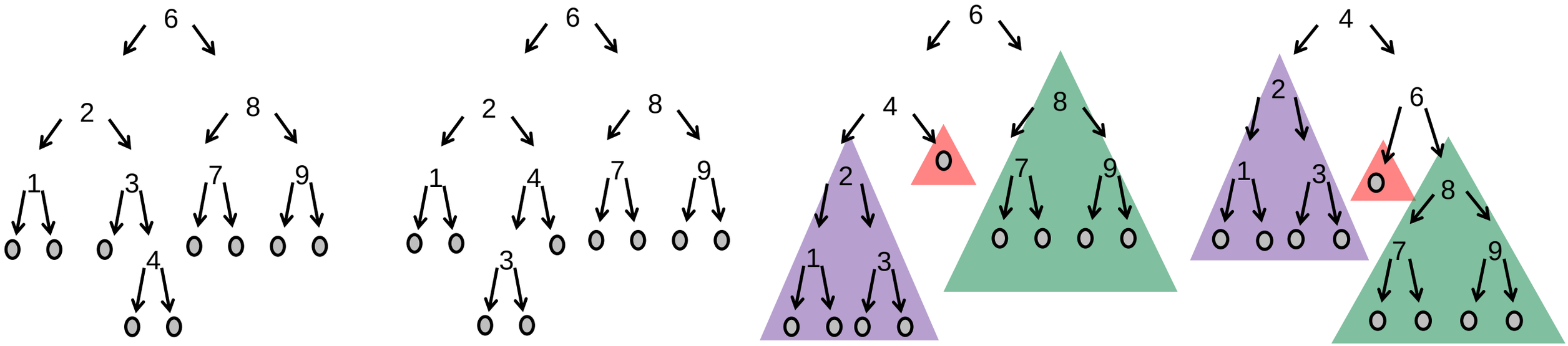
rotate left


rotate right
n1
n2
t3
t1
t2
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
7
9
1
3

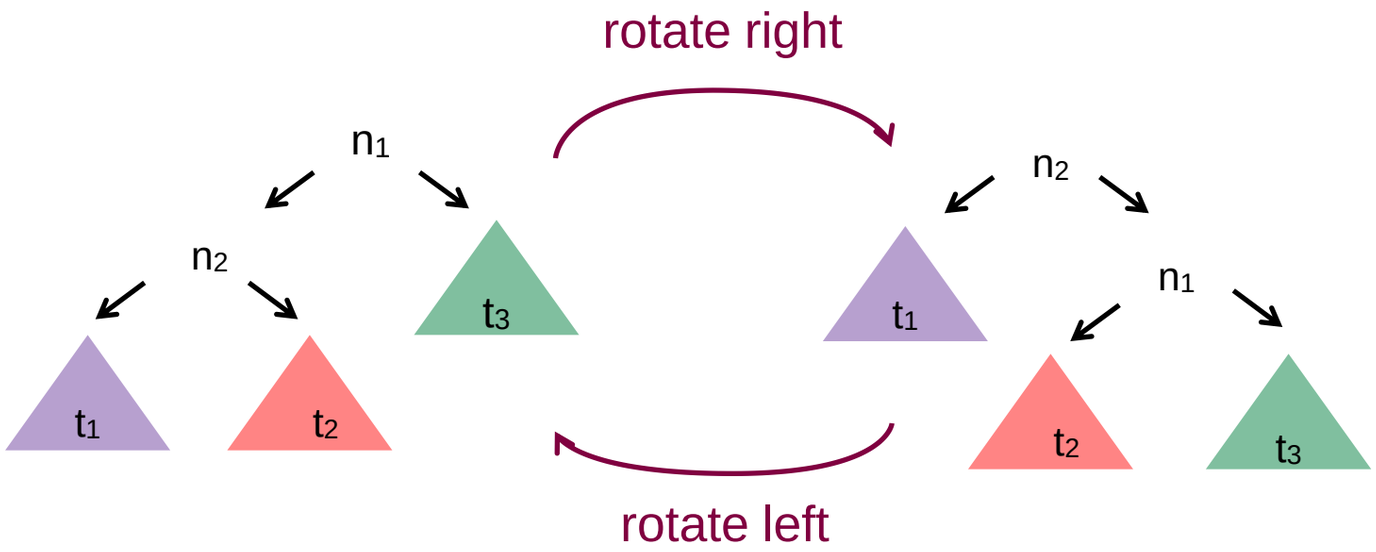
rotate left





rotate right

MOVE NODES THROUGH ROTATION




```
//Does not update the size fields in the nodes
link rotateRight(link n) {
      link n2 = n->left;
      n->left = n2->right;
      n2->right = n;
      return n2;
}
//Does not update the size fields in the nodes
link rotateLeft (link n) {
     link n1 = n->right;
     n->right = n1->left;
     n1->left = n;
     return n1;
 }
```

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);
}
```

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

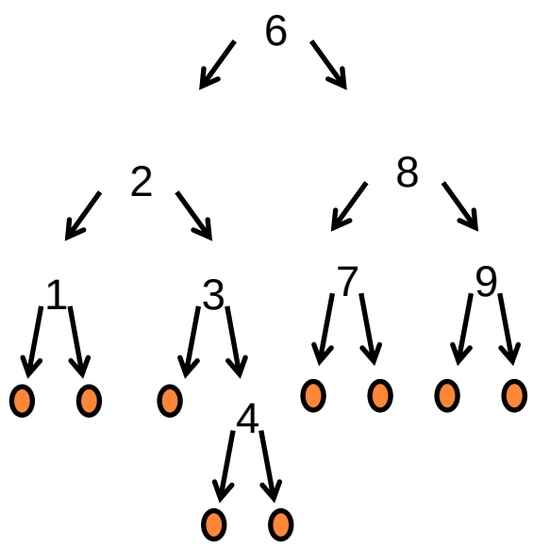
- BST ADT typically has no control over the order keys are supplied.
- To minimise the probability of ending up with a degenerate tree, we make a randomised decision at which level to insert a node.
  - at each level, the probability depends on the size of the remaining tree
    - Do normal leaf insertion 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

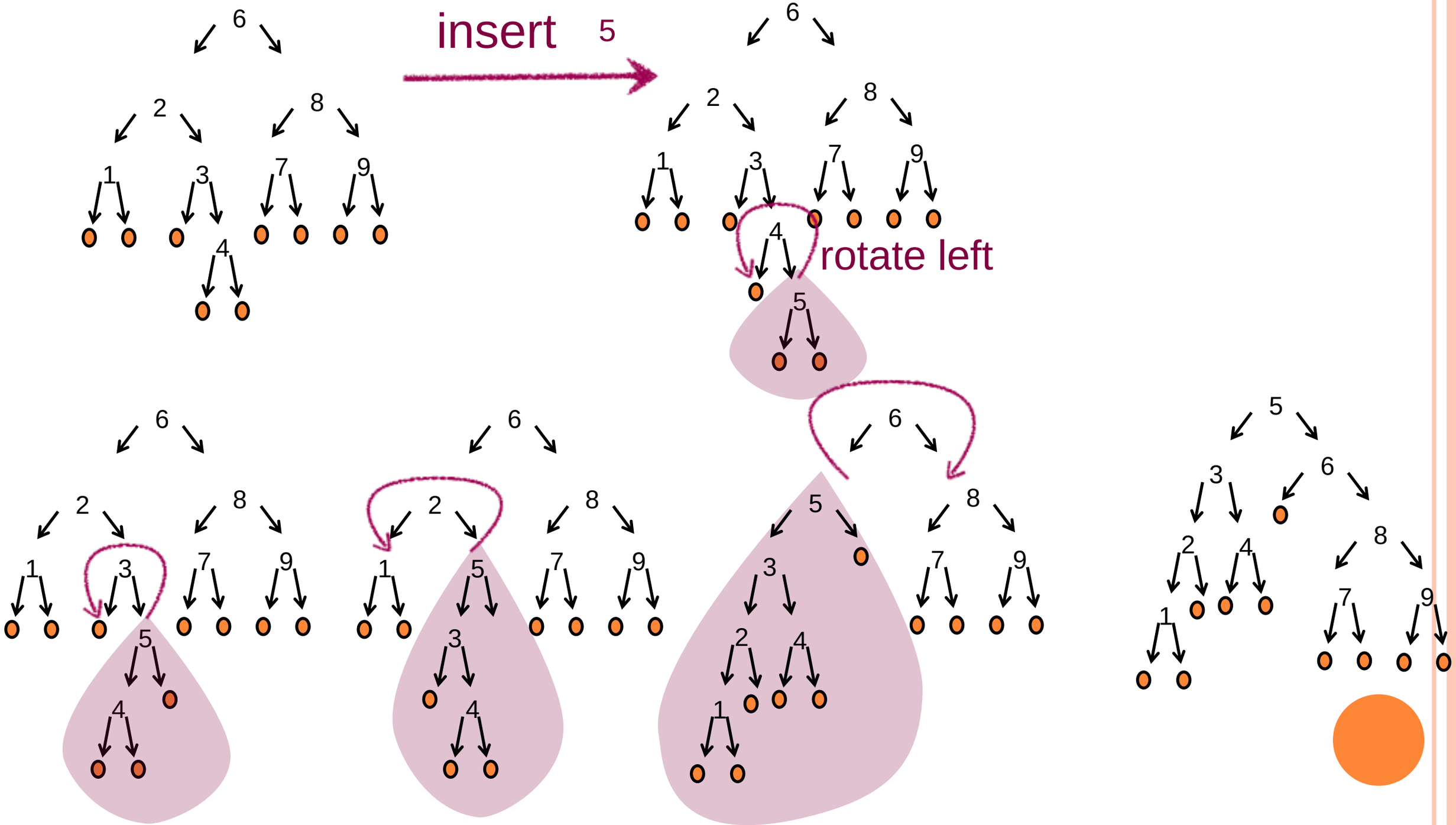
- Let us start with a simpler version of the problem:
  - How can we insert a node at the root instead of the leaves?
  - Problem:
    - this potentially already requires to re-arrange nodes in the whole tree
  - Solution:
    - we insert at the leaf position and move it up the tree without changing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

   currentLink->size++;
   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 overhead of a constant factor for rotation
- 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

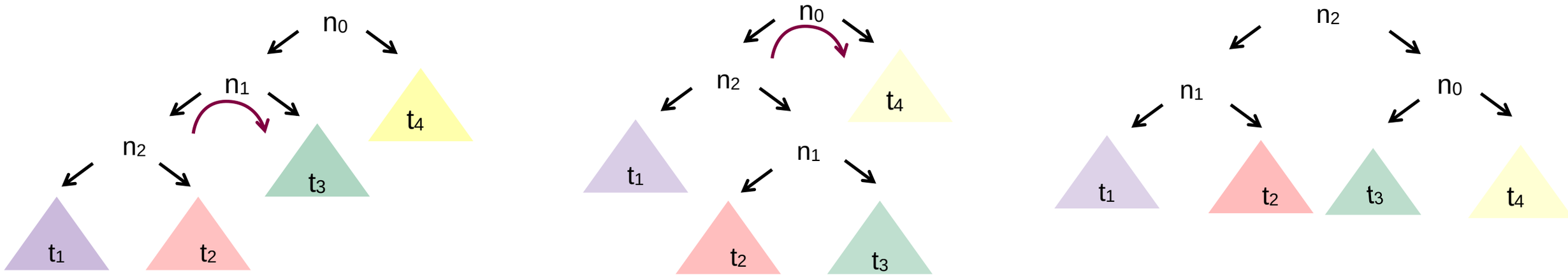
Randomised Trees Deletion

- 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

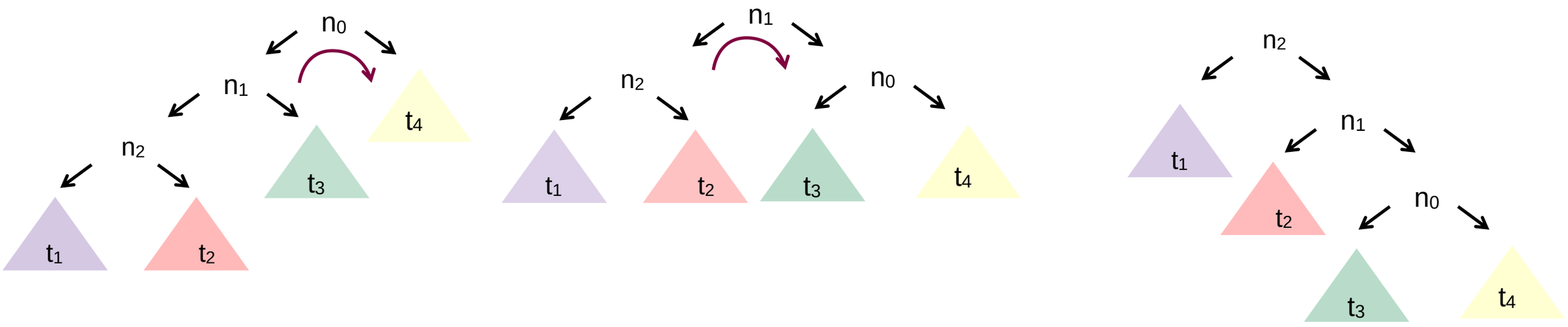
Amortisation: Splay Trees

- 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
- 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

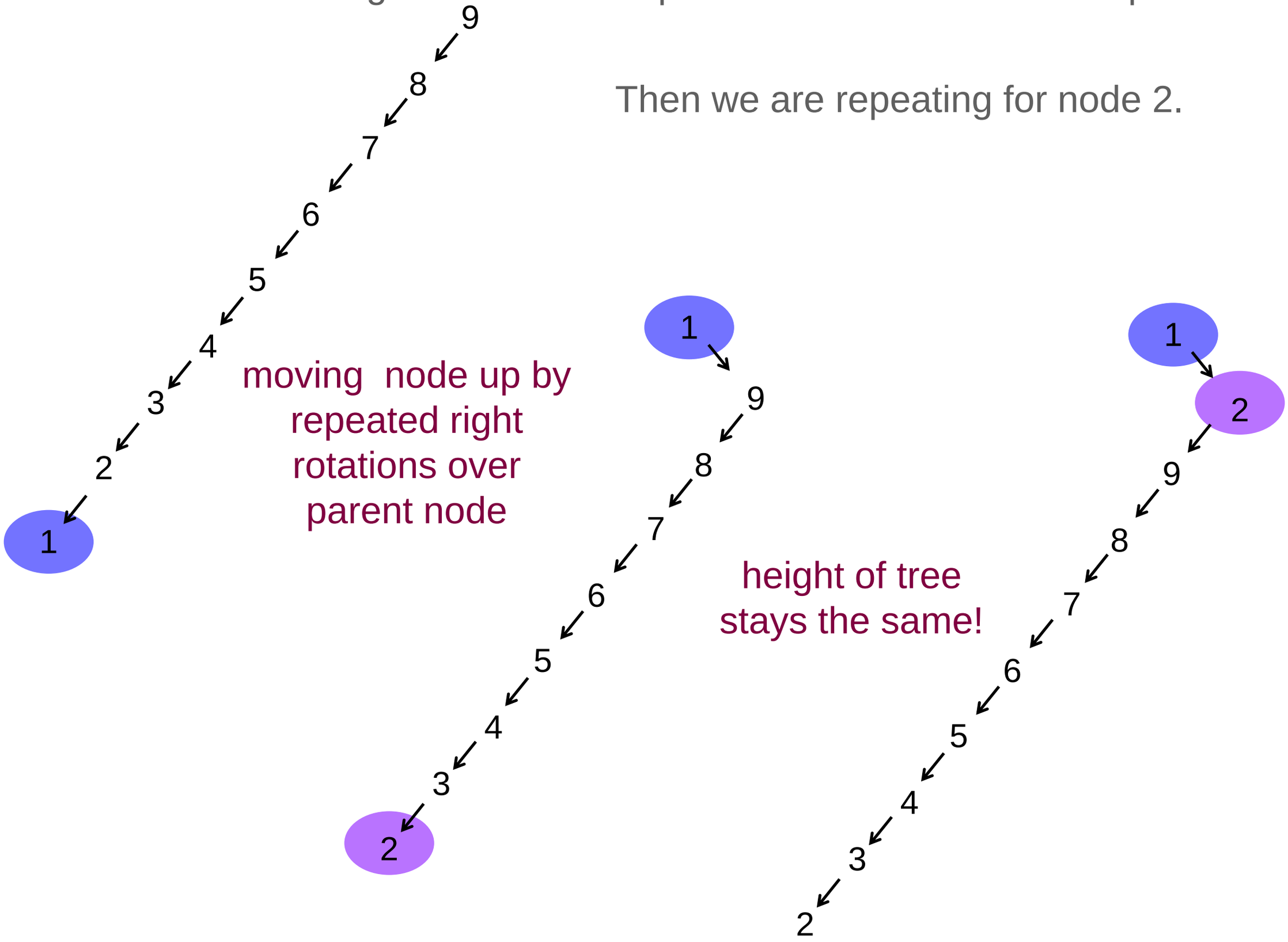
- Move n_2 up by rotating it to the right twice



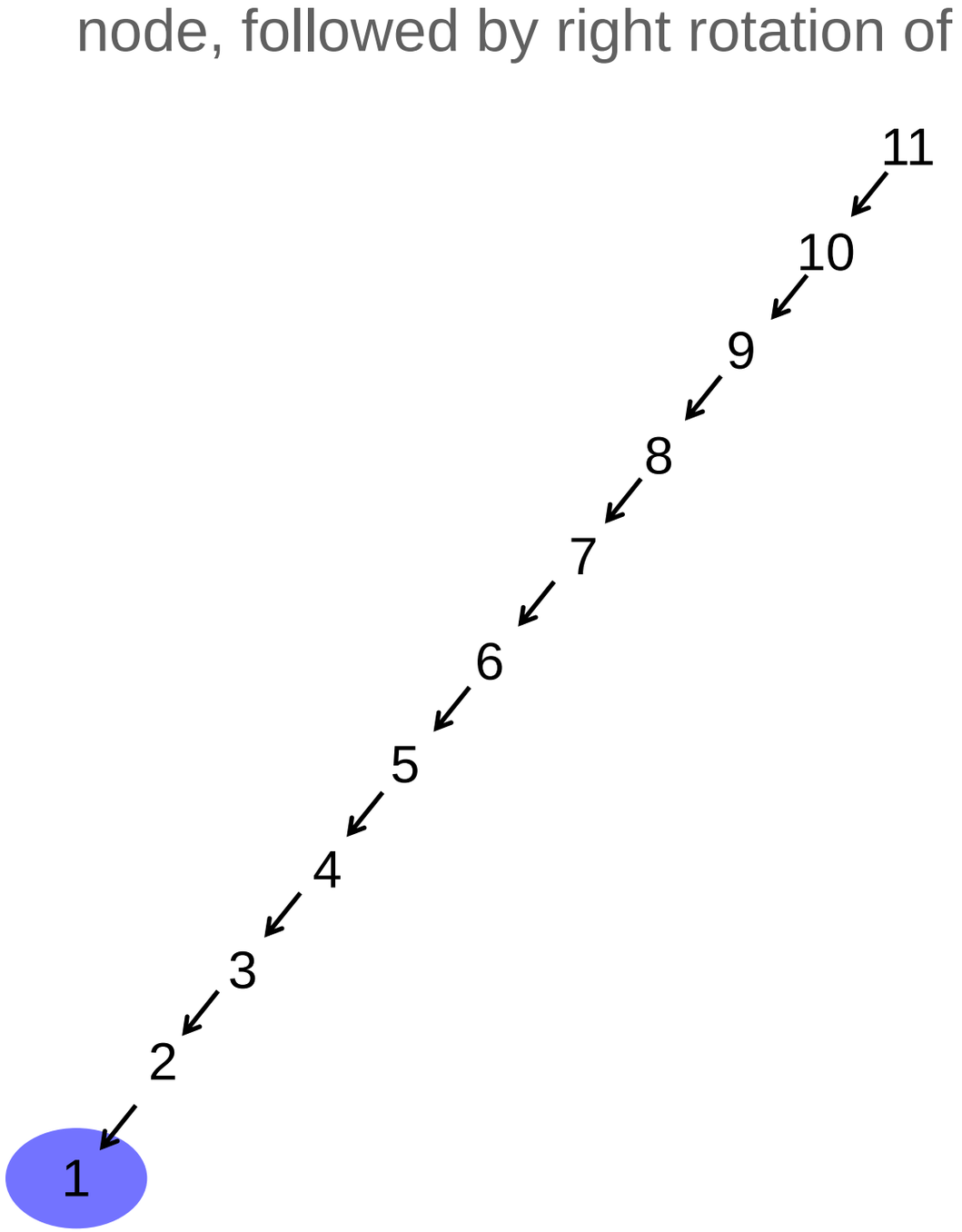
- Splay Tree Double Rotation: Move n_2 up by first rotation its parent, then n_2

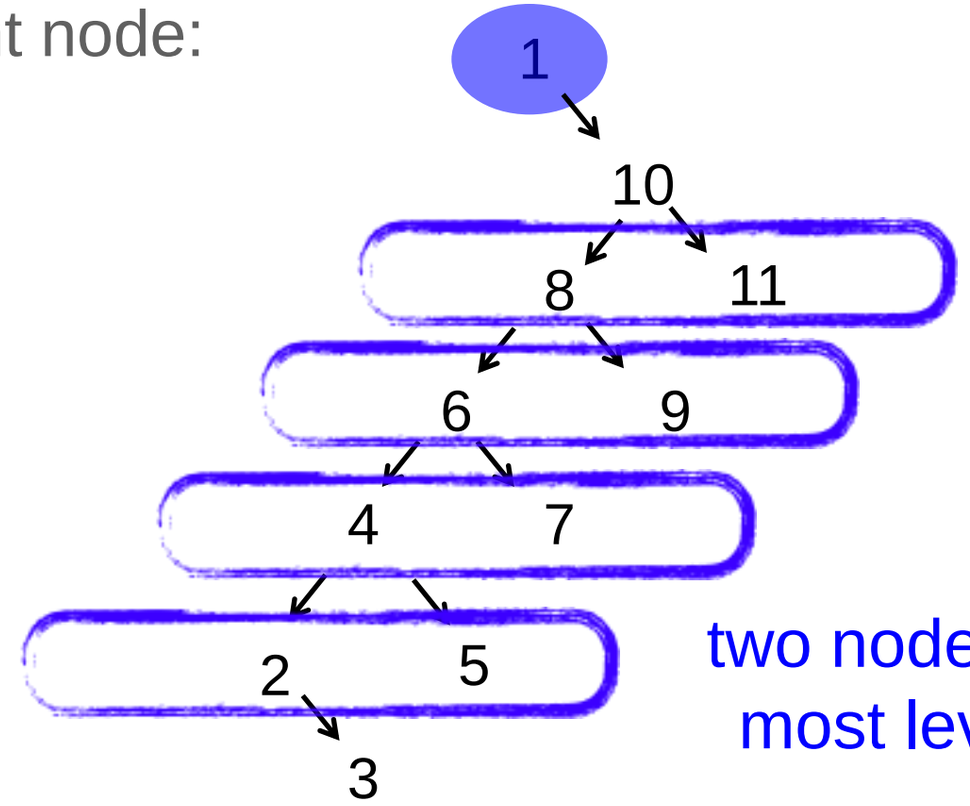


- Example: move smallest item up a degenerated tree using right rotations from leaf. We are doing rotations at the parent of node 1 at each step.
- Then we are repeating for node 2.



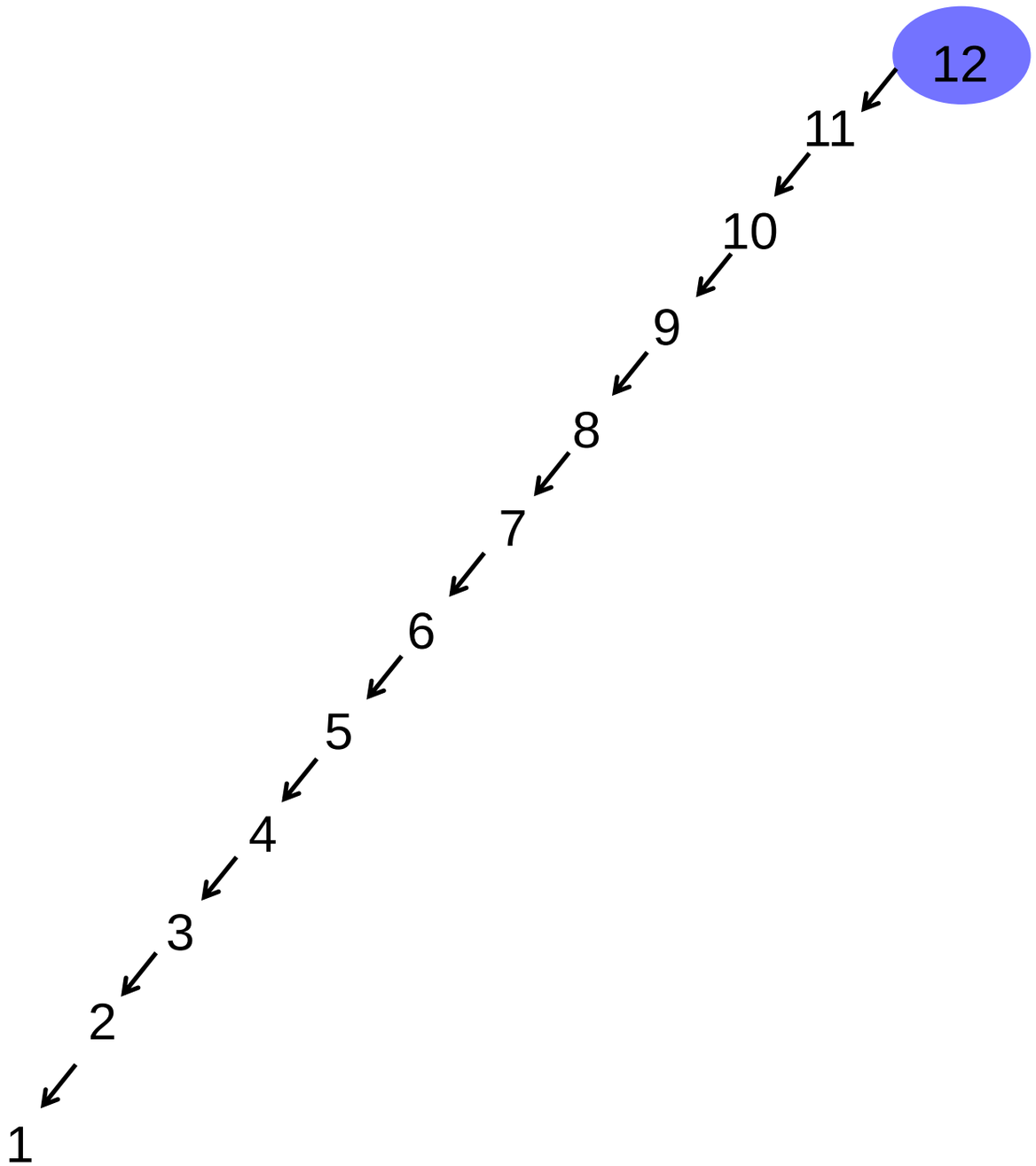
- Example: After inserting 1 in this worst case degenerate tree: move smallest item up a degenerated tree using right rotation of grand parent node, followed by right rotation of parent node:





two nodes on most levels

- Example: 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).



Work Complexity of Splay tree operations

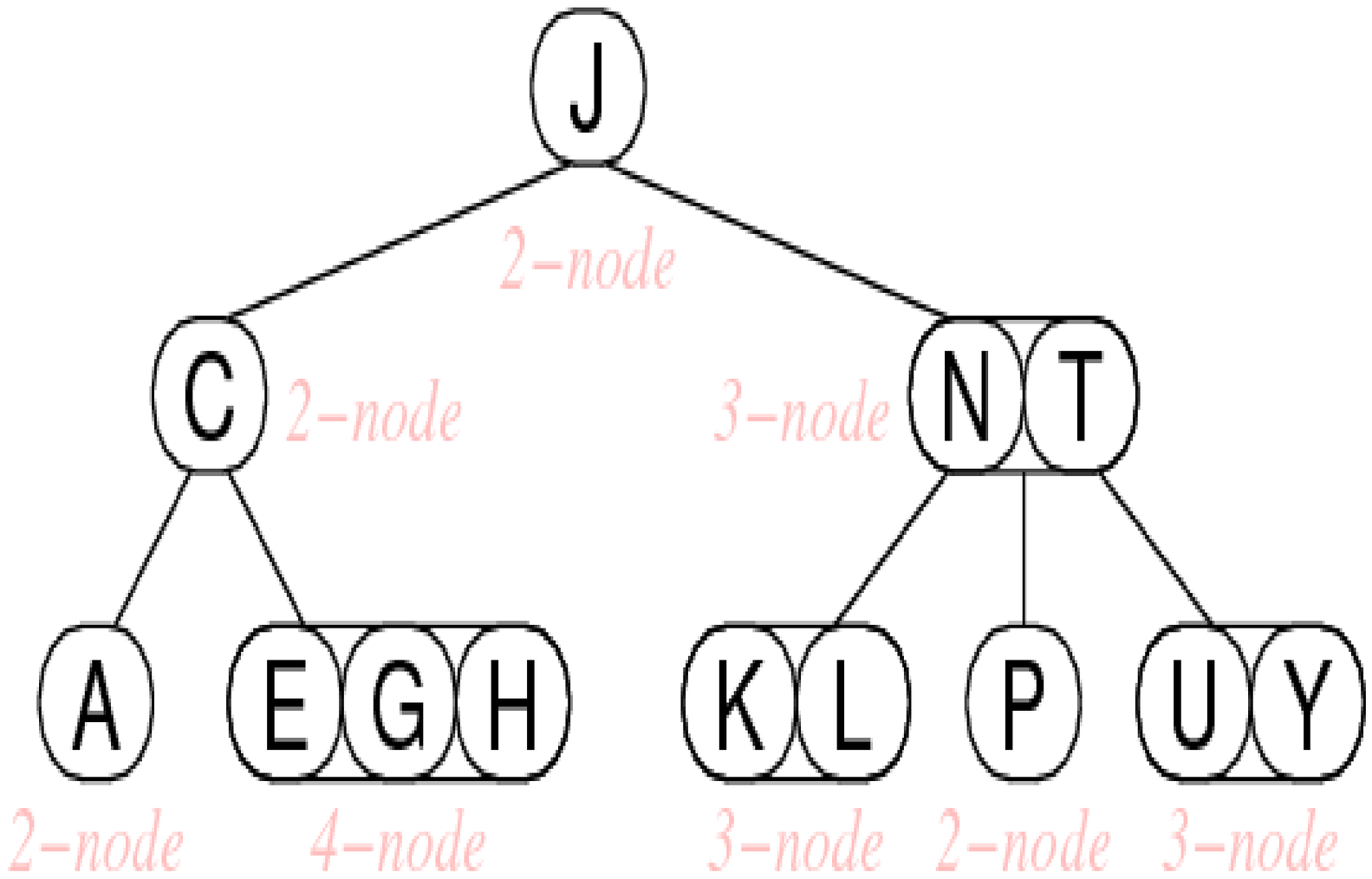
- Insertion
  - worst case (wrt work): item is inserted at the end of a degenerate tree
    - O(n) steps necessary, but tree height reduced by a factor of two
  - worst case (wrt resulting tree): item inserted at the root of a degenerate tree
    - Constant number of steps necessary
  - even in the worst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

Work Complexity of Splay tree operations

- The number of comparisons when inserting M nodes in an N-node splay tree is O($(N+M)\ log\ (N+M)$)
- 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

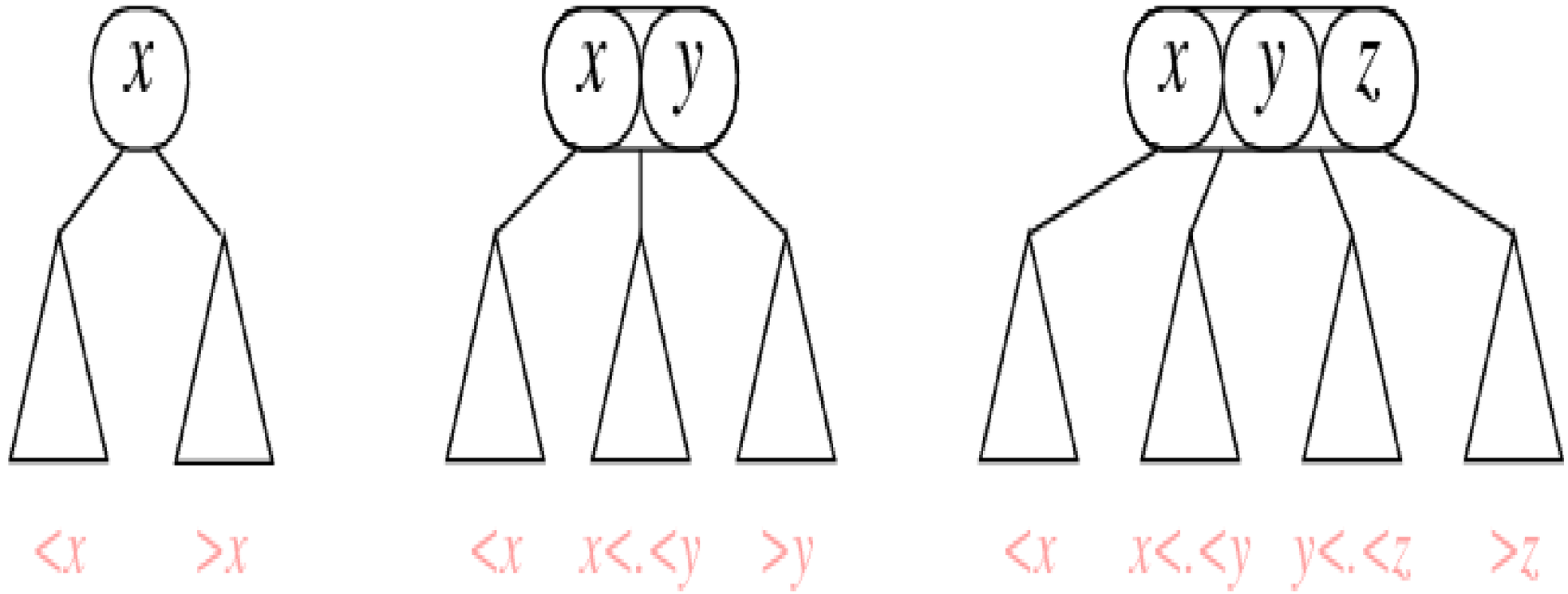
2-3-4 Trees

- 2-3-4 trees have three kinds of nodes 2-nodes, with two children (same as normal BSTs)
- 3-nodes, two values and three children
- 4-nodes, three values and four children



2-3-4 Trees

- 2-3-4 trees are ordered similarly to BSTs



- In a balanced 2-3-4 tree:
  - all leaves are at same distance from the root
  - 2-3-4 trees grow "upwards" from the leaves.

2-3-4 TREE IMPLEMENTATION:

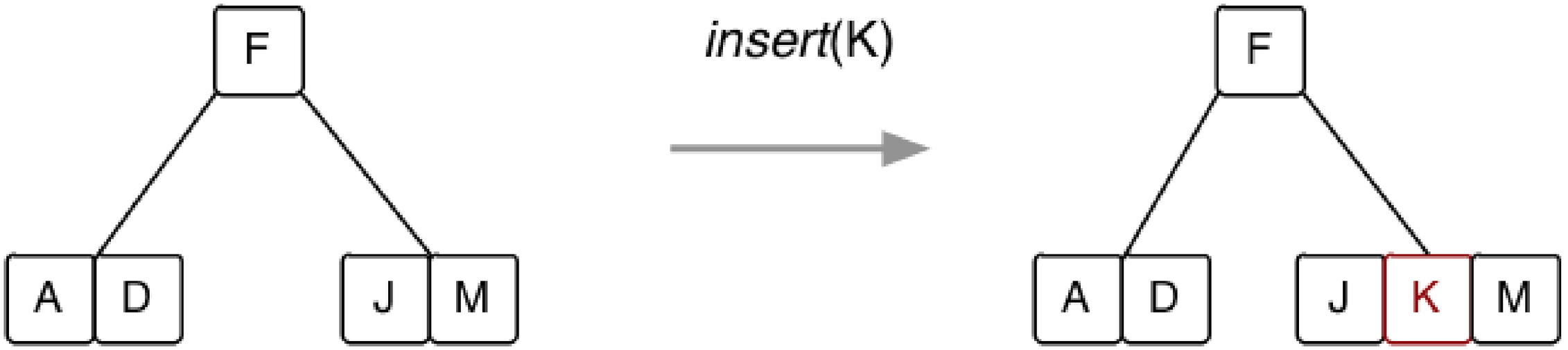
```
typedef struct node *TreeLink;
struct node {
    int order;     // 2, 3 or 4
    Item data[3]; // items in node
    TreeLink child[4]; //links to subtrees

};
```

2-3-4 Insertion Algorithm

- Find leaf node where Item belongs (via search)
  - if not full (i.e. order < 4)
    - insert Item in this node, order++
  - if node is full (i.e. contains 3 Items)
    - split into two 2-nodes as leaves
    - promote middle element to parent
    - insert item into appropriate leaf 2-node
    - if parent is a 4-node
      - continue split/promote upwards
    - if promote to root, and root is a 4-node
      - split root node and add new root

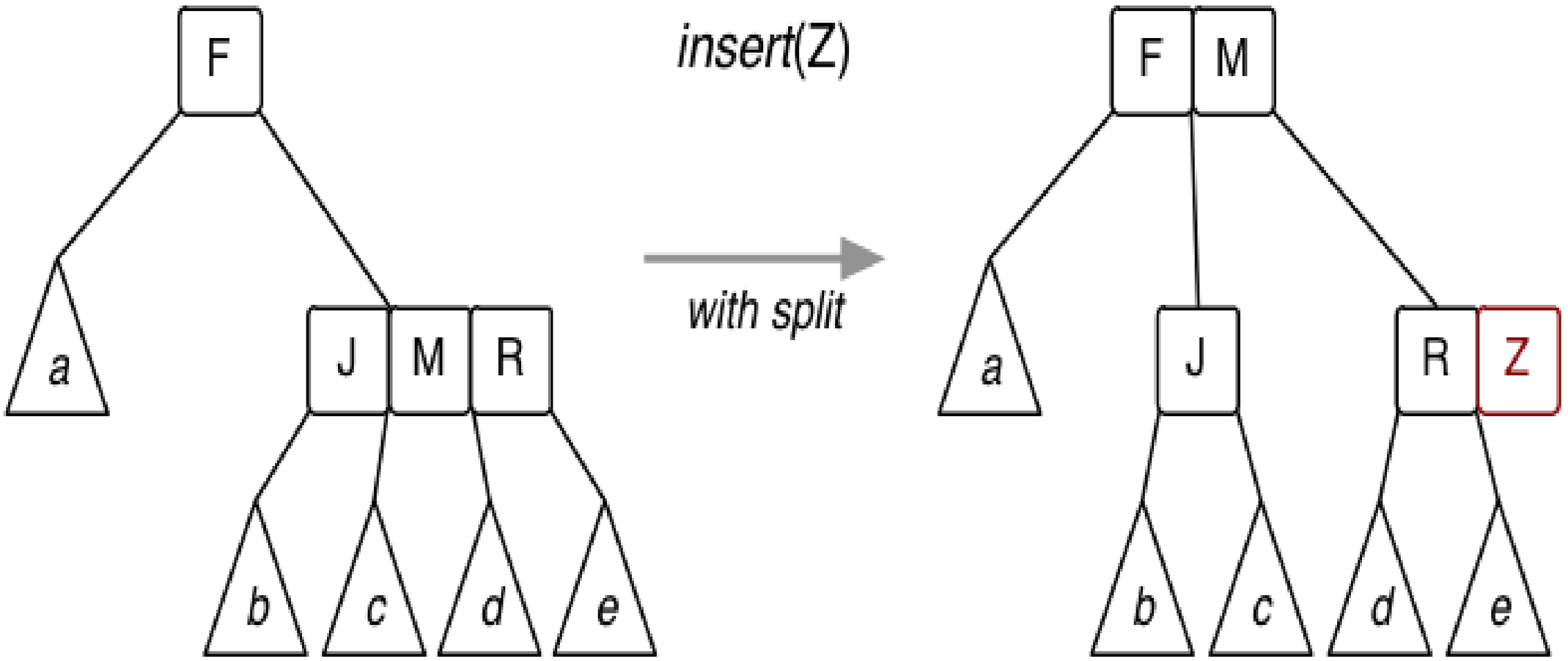
2-3-4 INSERTION

- Insertion into a 2-node or 3-node:

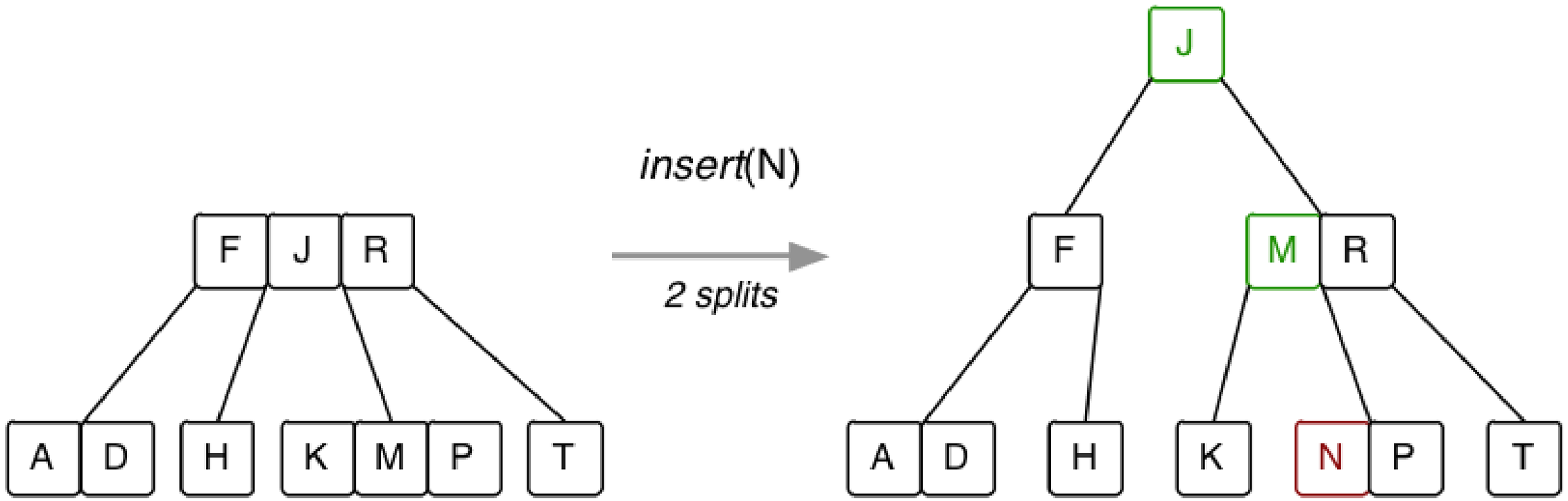


Splitting a node in a 2-3-4 tree:

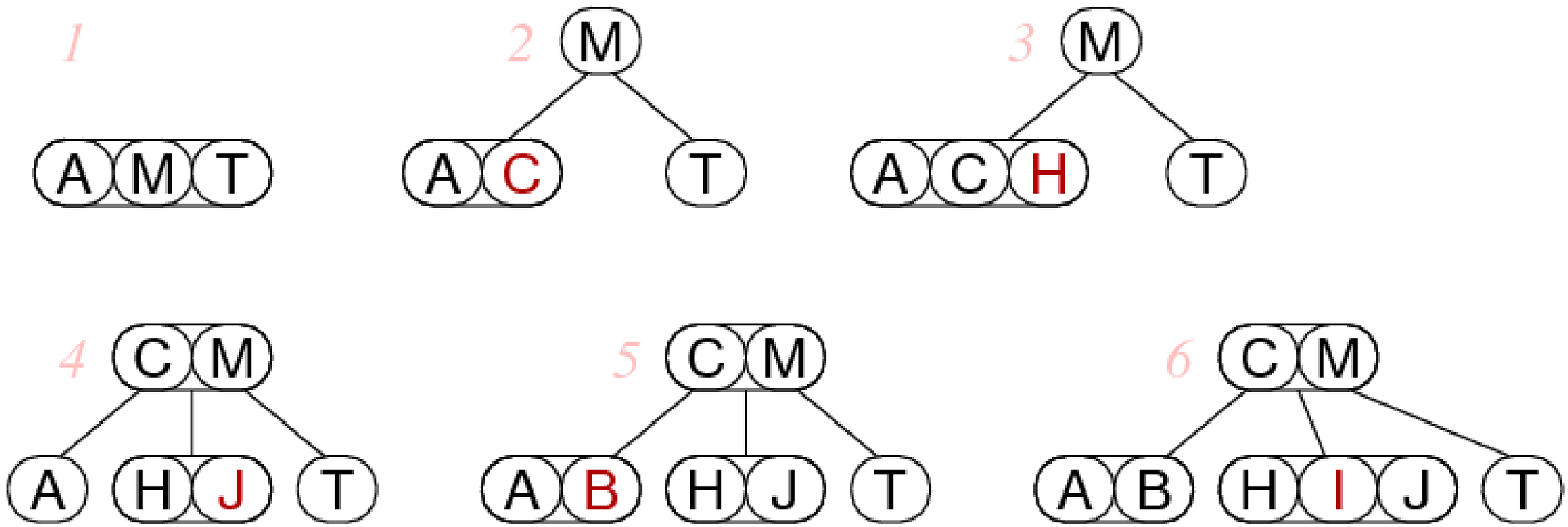
Insertion into a 4-node
(requires a split):



SPLITTING THE ROOT:



Building a 2-3-4 tree ... 7 insertions:



Show what happens when S then U are inserted into this tree:

2-3-4 TREE SEARCHING COST ANALYSIS:

- Worst case determined by height h
  - 2-3-4 trees are always balanced ⇒ height is *O(logN)*
  - worst case for height: all nodes are 2-nodes same case as for balanced BSTs, i.e. $h \cong log_2 N$
  - best case for depth: all nodes are 4-nodes balanced tree with branching factor 4, i.e. $h \cong log_4 N$

Some Notes on 2-3-4 Trees

- Splitting the root is the only way depth increases.
- One variation: why stop at 4?
  - Allow nodes to hold up between *M/2* and *M* items.
  - If each node is a disk-page, then we have a B-tree (databases).
- Implement similar ideas using just BST nodes → red-black trees.

Red-Black Trees

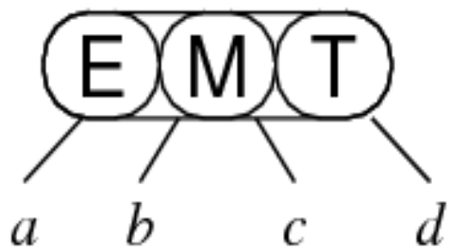
- Red-black trees are a representation of 2-3-4 trees using BST nodes.
  - each node needs one extra value to encode link type
  - but we no longer have to deal with different kinds of nodes
- Link types:
  - red links ... combine nodes to represent 3- and 4-nodes
  - black links ... analogous to "ordinary" BST links (child links)
- Advantages:
  - standard BST search procedure works unmodified
  - get benefits of 2-3-4 tree self-balancing (although deeper)

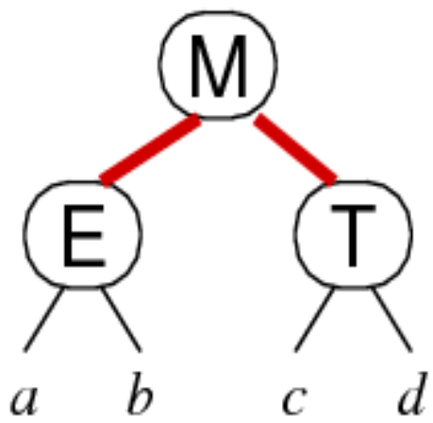
Red Black Trees

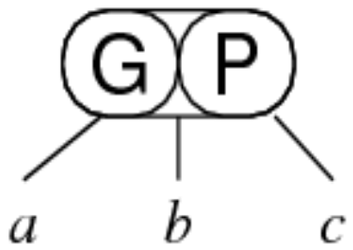
- Definition of a red-black tree:
  - a BST in which each node is marked red or black
  - no two red nodes appear consecutively on any path
- Balanced red-black tree:
  - all paths from root to leaf have same number of black nodes
- Insertion algorithm below ensures that balanced trees remain balanced

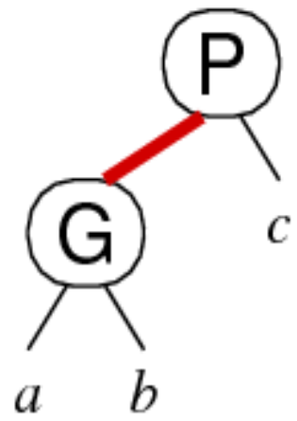
Representing 3- and 4-nodes in red-black trees:

2–3–4 nodes | *red–black nodes*

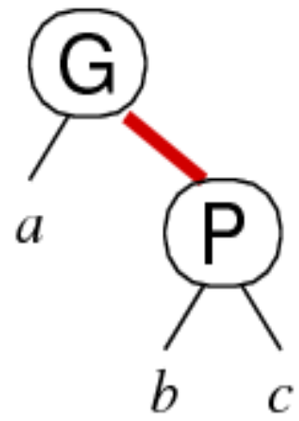




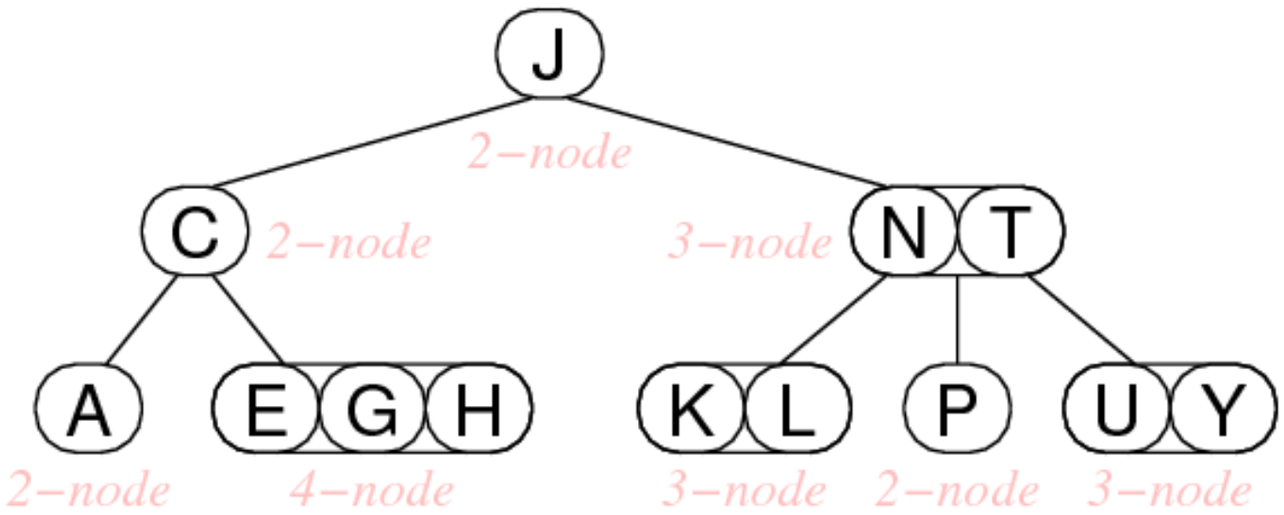


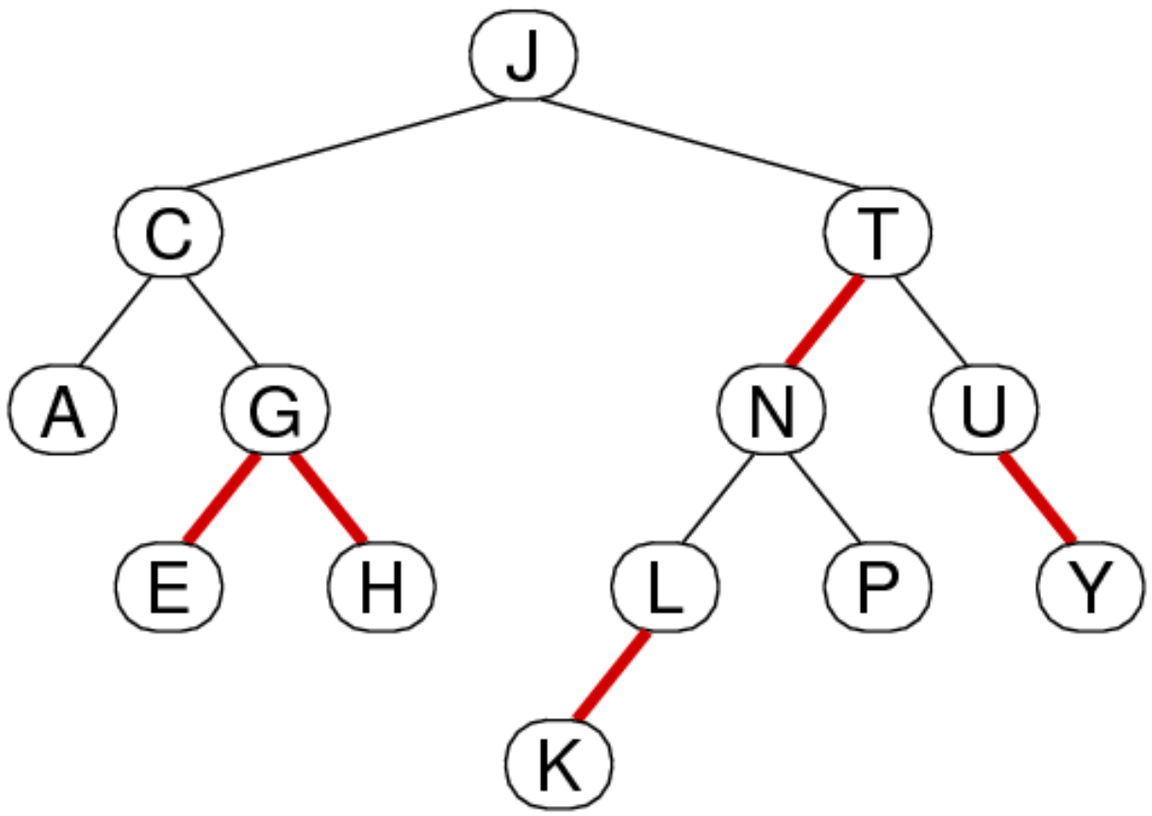


or



Equivalent trees (one 2-3-4, one red-black):



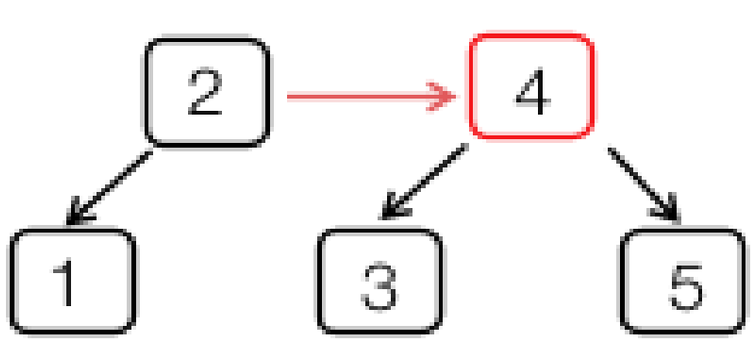


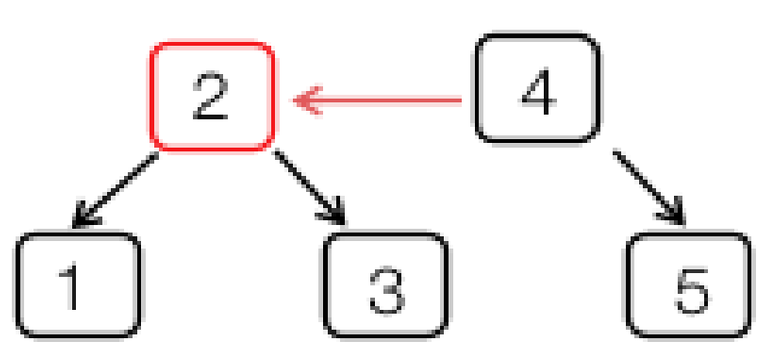
RED-BLACK TREE IMPLEMENTATION:

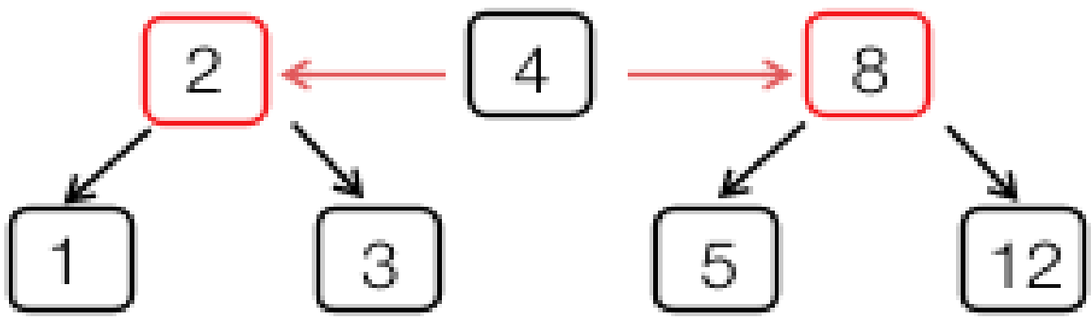
```
typedef enum {RED,BLACK} Colr;
typedef struct Node *TreeLink;
struct Node {
    Item data;    // actual data
    Colr colour; // colour of link to
                 //parent
    TreeLink left; // left subtree
    TreeLink right; // right subtree
} ;
```

Red Black Links

- Colr allows us to distinguish links
  - black = parent node is a "real"parent
  - red = parent node is a 2-3-4 neighbour







Red-Black Tree Insertion

- Search is same as standard BST
- Insertion is more complex than for standard BSTs
  - need to recall which direction we're going down tree
  - need to recall whether parent link is red or black
  - splitting/promoting implemented by rotateL/rotateR
  - several cases depending on colour/direction combinations