Balanced Trees

- COMP1927 Computing 17x1
- Sedgewick Chapters 13

Balanced Binary Search Trees

Binary Search Tree Review…

- data structures designed for *O(logN)* search
- consist of nodes containing item (incl. key) and two links
- can be viewed as recursive data structure (subtrees)
- have overall ordering (values(L) < root < values(R))
- insert new nodes as leaves, delete from anywhere
- have structure determined by insertion order (worst: *O(N)* e.g. when items in sorted order)
- operations: insert, delete, search, rotate, rebalance, …

Balanced Binary Search Trees

Goal: build binary search trees which have

- minimum depth ⇒ minimum worst case search cost (O (log N))

Perfectly balanced tree with *N* nodes has:

- abs(size(LeftSubtree) - size(RightSubtree)) < 2, for every node
- abs(height(leftsubtree) -height(rightSubtree) | < 2 for every node i.e. depth of $log_2 N$ ⇒ worst case search $O(logN)$

Effects of order of insertion on BST shape:

- best case: keys inserted in pre-order
  (median key first, median of lower half, median of upper half, etc.)
- worst case: keys inserted in ascending/descending order
- average case: keys inserted in random order ⇒ $O(log_2 N)$

New Binary Search Tree

```
// Item, Key, Node, Link, Tree types as before
#define key(it) ((it).key)
// operations on keys
#define cmp(k1,k2) ((k1) - (k2))
#define lt(k1,k2) (cmp(k1,k2) < 0)
#define eq(k1,k2) (cmp(k1,k2) == 0)
#define gt(k1,k2) (cmp(k1,k2) > 0)
// standard tree operations
Tree newTree();
Tree insert(Tree, Item);
Tree delete(Tree, Key);
int find(Tree, Key);
void dropTree(Tree);
void showTree(Tree);
int depth(Tree);
int nnodes(Tree); // aka size()
```

New Binary Search Tree ADT (cont)

```
// functions to assist with balancing tree
// internal to ADT
Link rotateR(Link);
Link rotateL(Link);
Tree rebalance(Tree);
Item *get_ith(Tree, int);
Tree partition(Tree, int);
Tree insertAtRoot(Tree, Item);
Tree insertRandom(Tree, Item);
```

Generating Values in Prefix Order

One way of ensuring balance ... insert values in "correct" order.
Write a function that generates prefix order sequence
- generates values in range lo .. hi
- first is mid-point, second is mid of lower-half, ...
- store values in array v[0..N-1]

Function interface:

```
void mkprefix(int *v, int N, int lo, int hi)
```

e.g. lo..hi = 1..7 ⇒ 4 2 1 3 6 5 7

TreeLab

A shell for manipulating binary search trees

- command interpreter (tlab.c) + Tree ADT (Tree.[ch])

Commands:

```
n N Ord Seed = make a new tree
i N = insert N into tree
I N = insert N into tree at root
d N = delete N from tree
f N = search for N in tree
g I = get the i'th element in tree
p I = partition tree around i'th element
R = rotate tree right around root
L = rotate tree left around root
q = quit
```

TREELAB (CONT)

Usage: **./**tlab *#Nodes Order Seed*

Possible orders to supply values for insertion

- A = ascending (10 .. N+9),
- D = descending (N+9 .. 10)
- P = prefix ... builds balanced tree
- R = random ... could do anything ...

Seed = starting value for pseudo-random number generator

Approach 1: Global Rebalancing

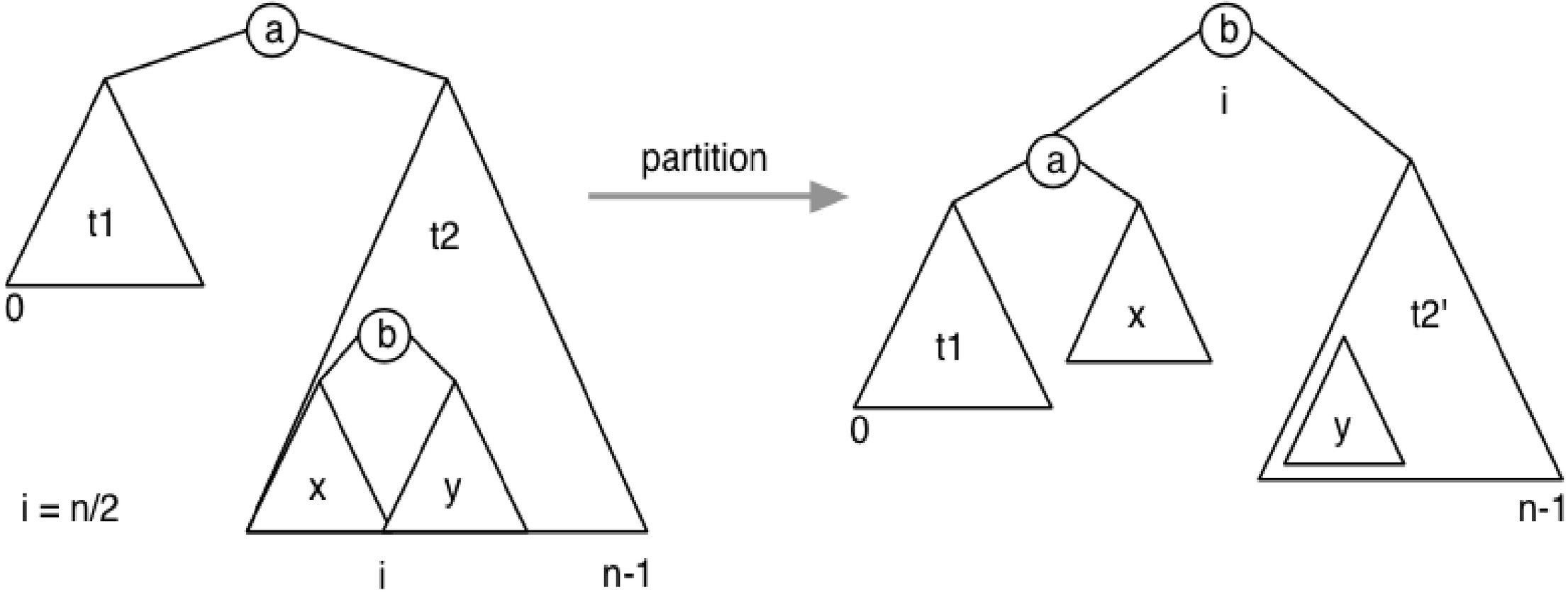
- Insert nodes normally as leaves (as for simple BST)
- Periodically, rebalance the whole tree
  - Question: how frequently/when

```
Tree NewTreeInsert(Tree t, Item it)
{ t = TreeInsert(t,it);
  // e.g. after every 20 insertions
  if (size(t) % 20 == 0)
          t = rebalance(t);
  return t;
}
```

Global Rebalancing

Question: How to rebalance a BST ?

- 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



Global Rebalancing

Implementation of re-balance:

```
Tree rebalance(Tree t) {
   if (t == NULL) return NULL;
   int n = count(t);
   if (n < 3) return t;
// move median node to the root by partitioning on
size/2
   t = partition(t, n/2);
// now rebalance each sub-tree
   t->left = rebalance(t->left);
   t->right = rebalance(t->right);
   return t;
}
```

Global Rebalancing

To do this efficiently requires changes to Tree data structure and operations.

```
typedef struct Node {
   Item value;
   int nnodes;              // #nodes in my tree
  Link left, right;        // subtrees
} Node;
```

```
Link newNode(Item it) {
Link new = malloc(sizeof(Node));
assert(new != NULL);
new->value = it;
new->nnodes = 1;
new->left = new->right = NULL;
return new;
}
```

Global Rebalancing

New functions for determining tree size:

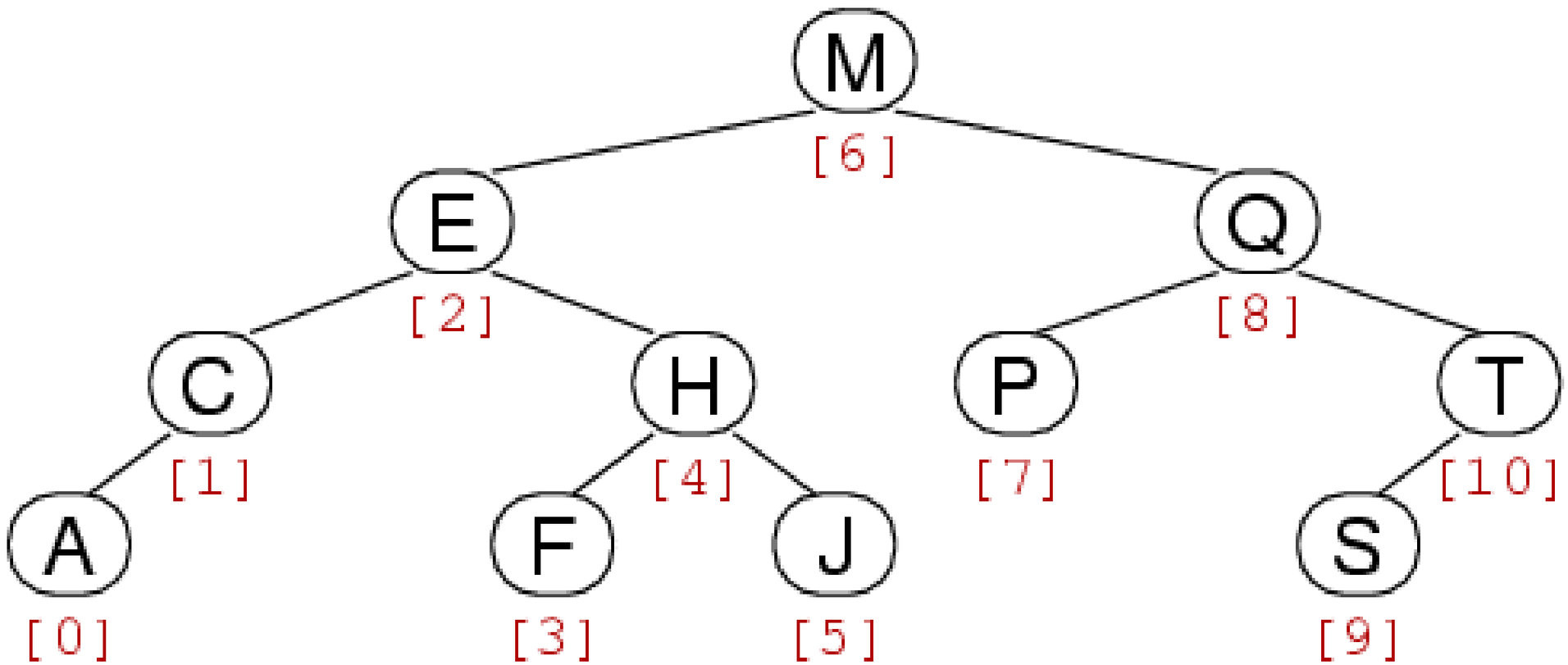
```
// efficient; use outside Tree-changing functions
 int size(Tree t) {
       return (t == NULL) ? 0 : t->nnodes;
}

// inefficient; use while making changes to Tree
int count(Tree t) {
    if (t == NULL) return 0;
    else return 1 + count(t->left) + count(t->right);
}
```

Global Rebalancing

New operations on trees:

- **get_ith()**: select i'th element from inorder sequence of keys
- **partition()**: re-arrange tree so that i'th element becomes root



For tree with *N* nodes, indexes are *0* .. *N-1*

Global Rebalancing

Implementation of selection operation:

```
// select i'th Item in key order
Item *get_ith(Tree t, int i)
{
    if (t == NULL) return NULL;
    assert(0 <= i && i < size(t));
    int n = size(t->left);   // #nodes to left of root
    if (i < n) return get_ith(t->left, i);
    if (i > n) return get_ith(t->right, i-n-1);
    return &(t->item);
}
```

Note: size(t) = n,   size(t->left) = m,   size(t->right) = n-m-1

Note: "-1" in size(t->right) is to exclude root of t

Global Rebalancing

Implementation of partition operation:

```
// move i'th item of t to root
Tree partition(Tree t, int i)
{
    if (t == NULL) return NULL;
    assert(0 <= i && i < size(t));
    int n = size(t->left);
    if (i < n) {
        t->left = partition(t->left, i);
        t = rotateR(t);
    }
    else if (i > n) {
        t->right = partition(t->right, i-n-1);
        t = rotateL(t);
    }
    t->nnodes = count(t); // fix count
return t;
}
```

ROTATIONS

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

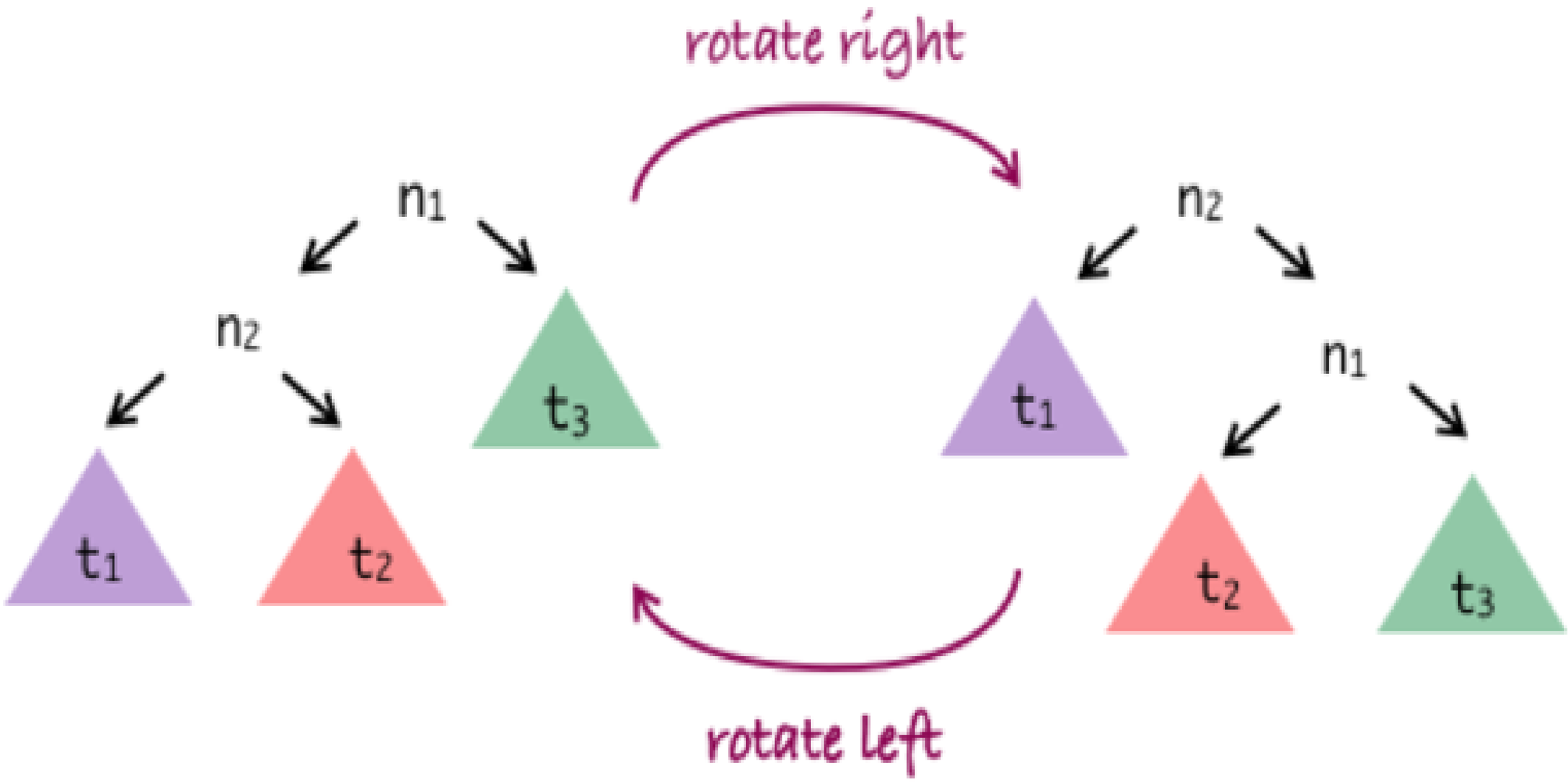
Moving Node To Root Using Rotation

Move node n2 up

- $t1 < n2 < t2 < n1 < t3$

Rotation leaves the relative order of the nodes intact!

We can use it to successively move a node up to the root



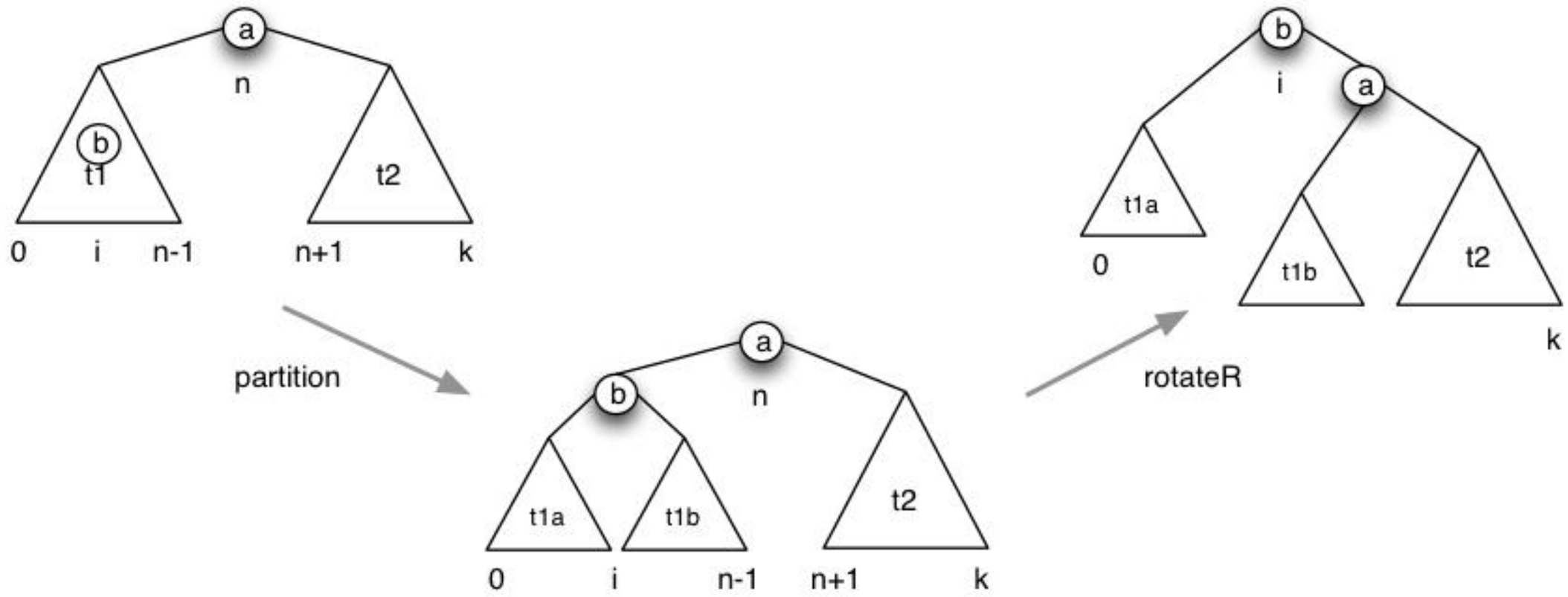
Moving Node to Root Using Rotations

```
Link rotateR(Link n1) {
    if (n1 == NULL) return NULL;
    Link n2 = n1->left;
    if (n2 == NULL) return n1;
    n1->left = n2->right;
    n2->right = n1;
    return n2;
}
```

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

Global Rebalancing

Partition and rotate to move the i^{th} node to the root of the tree



Global Rebalancing

- Analysis of rebalancing: visits every node => O(N) cost implies not feasible to rebalance after each insertion
- When to rebalance? … Some possiblities:
  - after every *k* insertions
  - rebalance whenever "imbalance" exceeds some threshold
  - Rebalance every time – too expensive
- 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 more work at insertion to make search faster

Optimisation

- maintain structural information to be able to provide performance guarantees
- implement all operations with performance bounds

1. Randomised BST Insertion

Tree ADT has no control over order that 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

In the hope that this randomness helps to balance the tree ...

Randomised BST Insertion

Approach: normally do leaf insert, randomly do root insert.

```
Tree insertRandom(Tree t, Item it) {
        if (t == NULL) return newNode(it);
// 25% chance of doing root insert
        if (rand()%100 < 25)
                return insertAtRoot(t,it);
        else
                return insert(t, it);
}
```

Alternatives: if (nnodes(t)%5 == 0), etc.

Insertion At Root

Previous insertion into BSTs

- inserted as leaves.

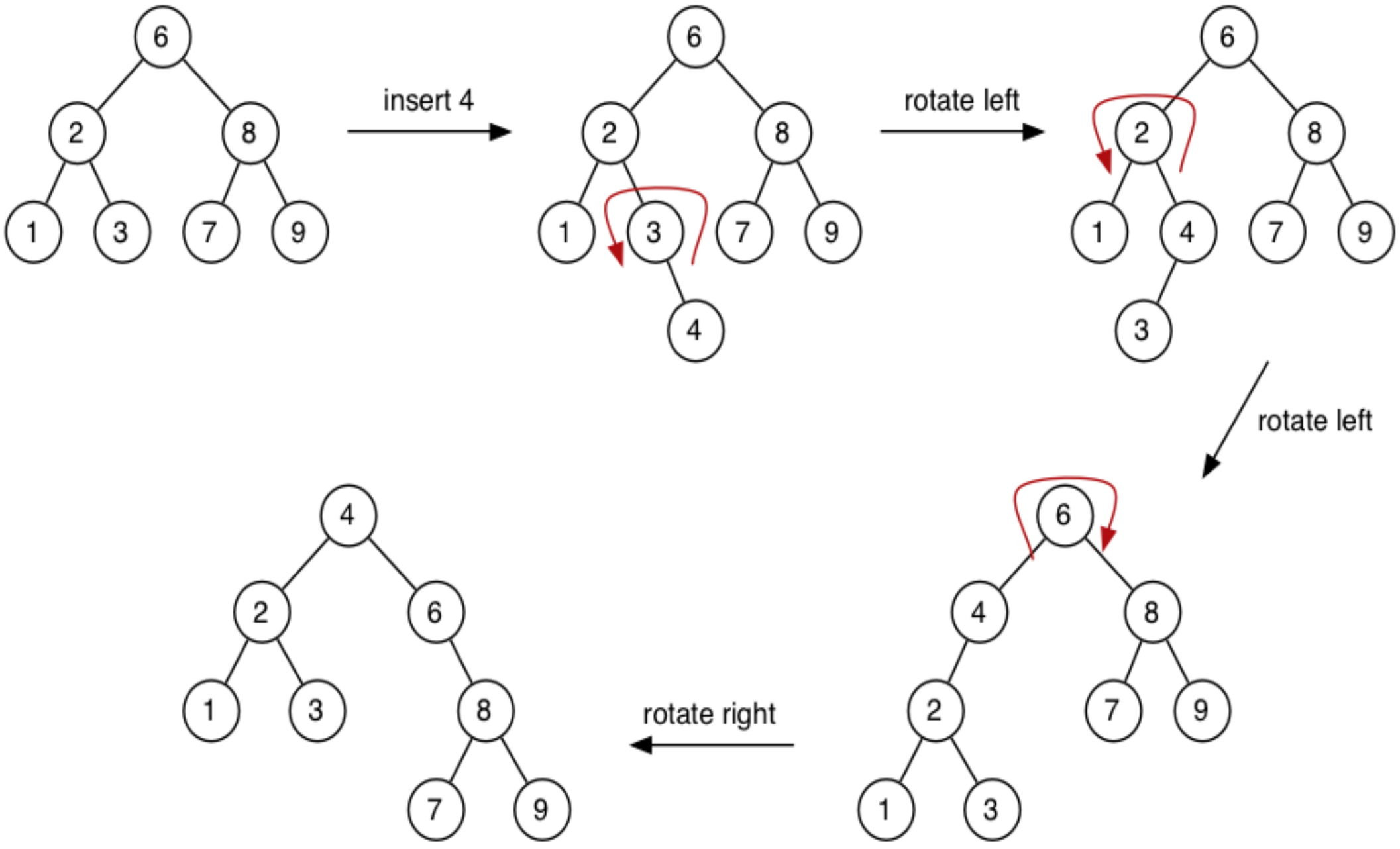
New approach:

insert new value at root.

method for inserting at root (recursive):

- base case:
  - tree is empty; make new node and make it root
- recursive case:
  - insert new node as root of L/R subtree
  - lift new node to root by R/L rotation

Insertion At Root



Insertion At Root (Cont)

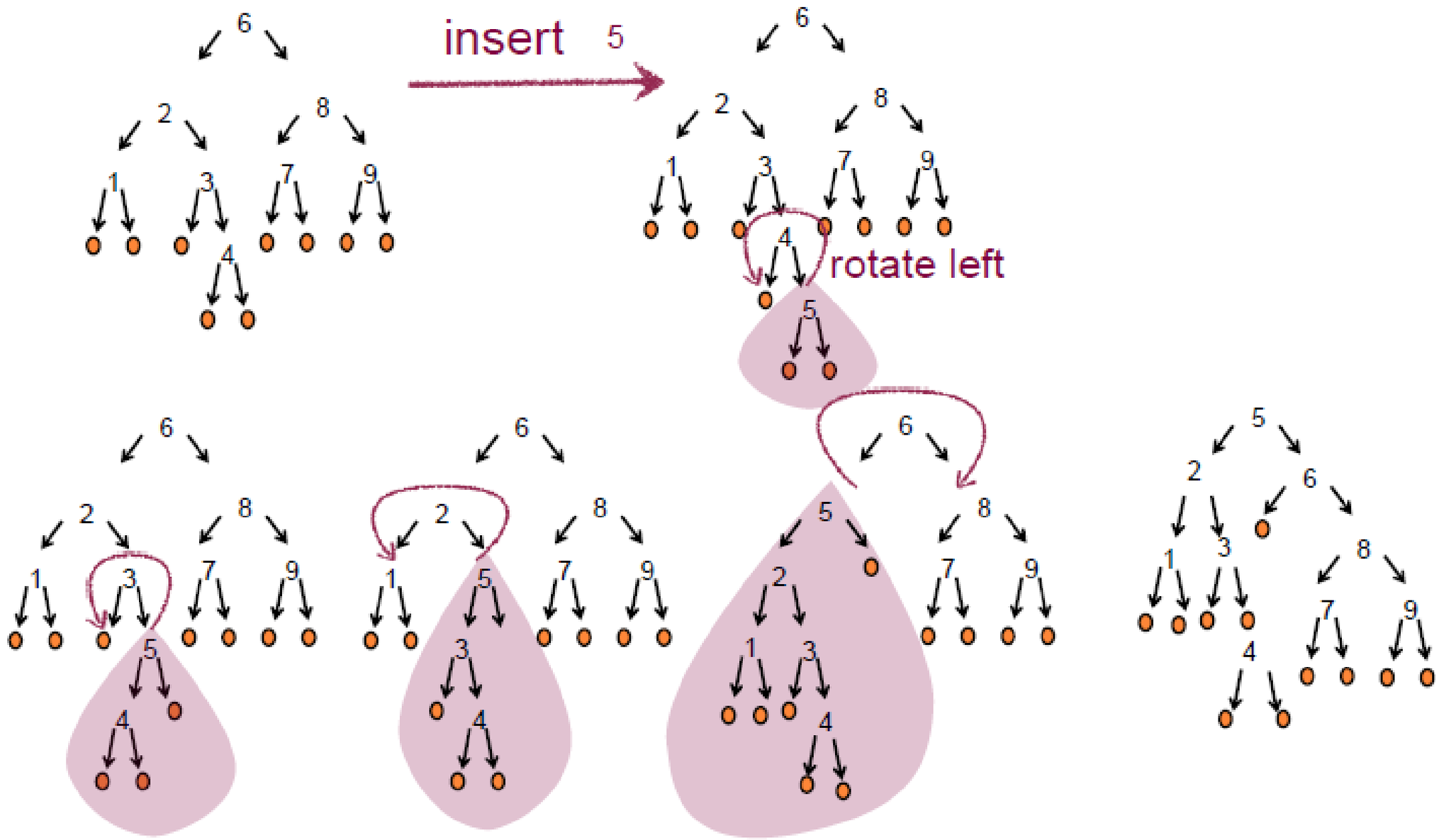
```
Tree insertAtRoot(Tree t, Item it) {
    if (t == NULL) return newNode(item);
    int diff = cmp(key(it), key(t->value));
    if (diff == 0) t->value = it;
    else if (diff < 0)
    {     t->left = insertAtRoot(t->left, it);
          t = rotateR(t);
     }
     else if (diff > 0) {
          t->right = insertAtRoot(t->right, it);
          t = rotateL(t);
     }
return t;
}
```

Insertion At Root –

Base Case: Tree is empty

Recursive Case:

1. Insert it into root of appropriate sub-tree
2. Lift root of sub-tree by rotation



Insertion At Root

What is the work complexity?

- Same as insertion at leaf, but extra work done for each insertion – O(N)

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 application

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

2. Amortisation: Splay Trees

Idea: 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

Splay-tree insertion modifies insertion-at-root method:

- by considering parent-child-grandchild orientation (three-level-analysis)
- by performing double-rotations based on p-c-g orientation

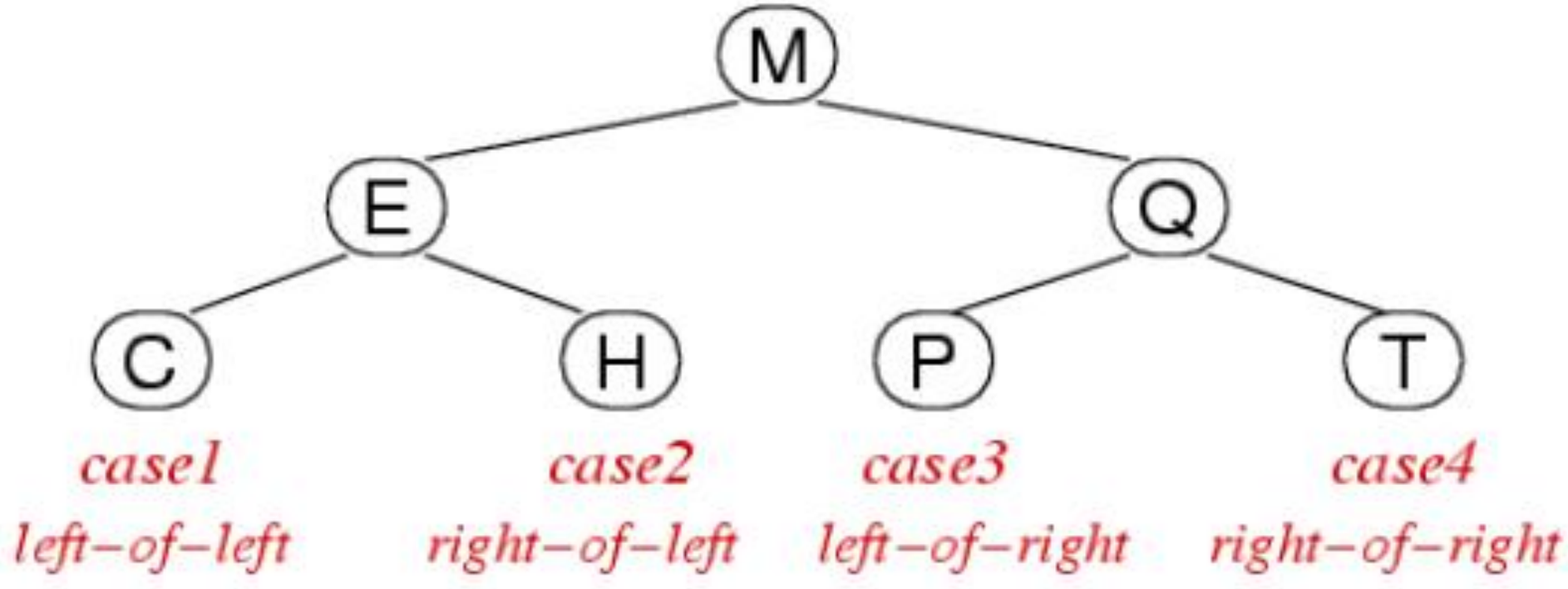
Splay-tree rotations also do rotation-in-search:

- 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 Trees

Cases for splay tree double-rotations:

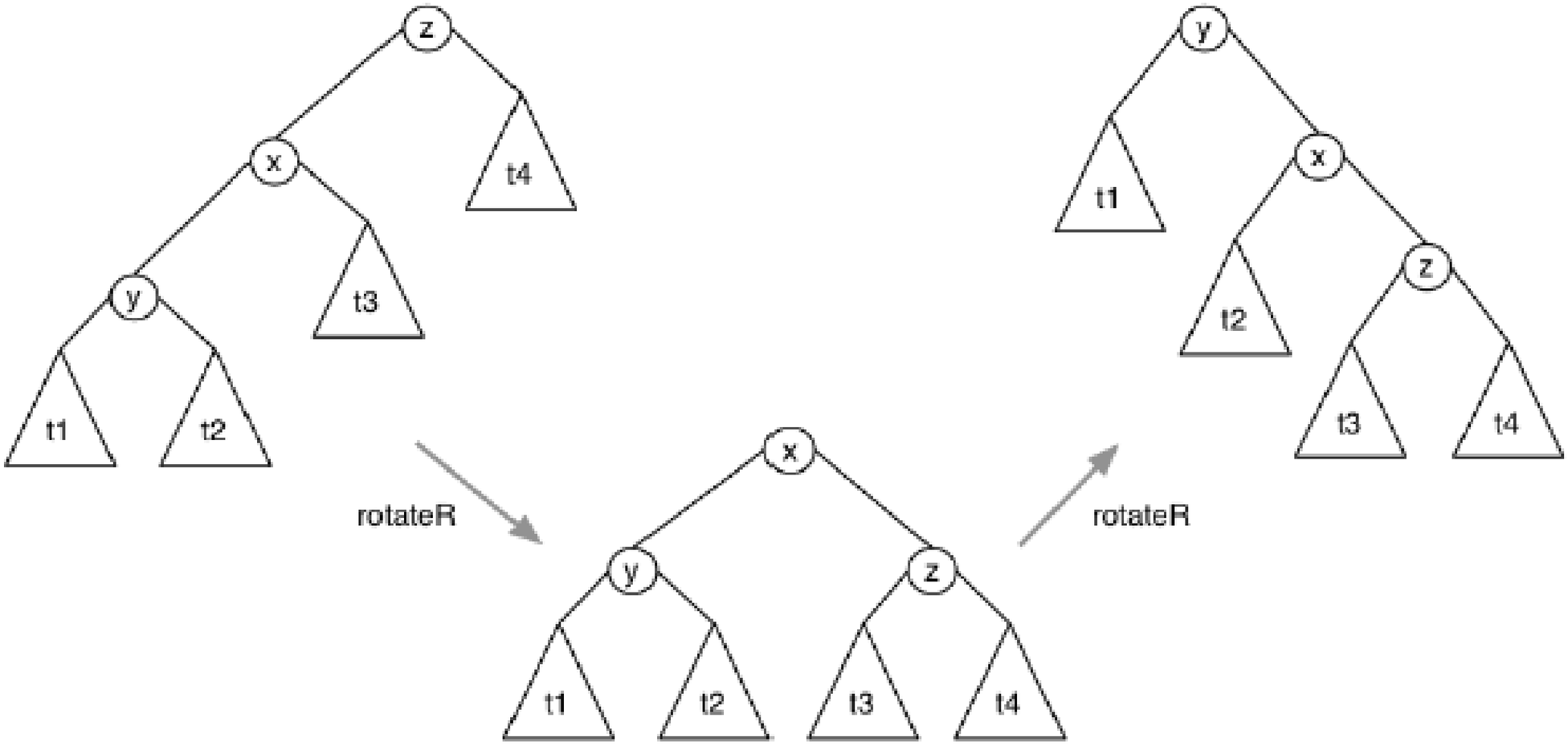
- case 1: grandchild is left-child of left-child
- case 2: grandchild is right-child of left-child
- case 3: grandchild is left-child of right-child
- case 4: grandchild is right-child of right-child



Splay Trees

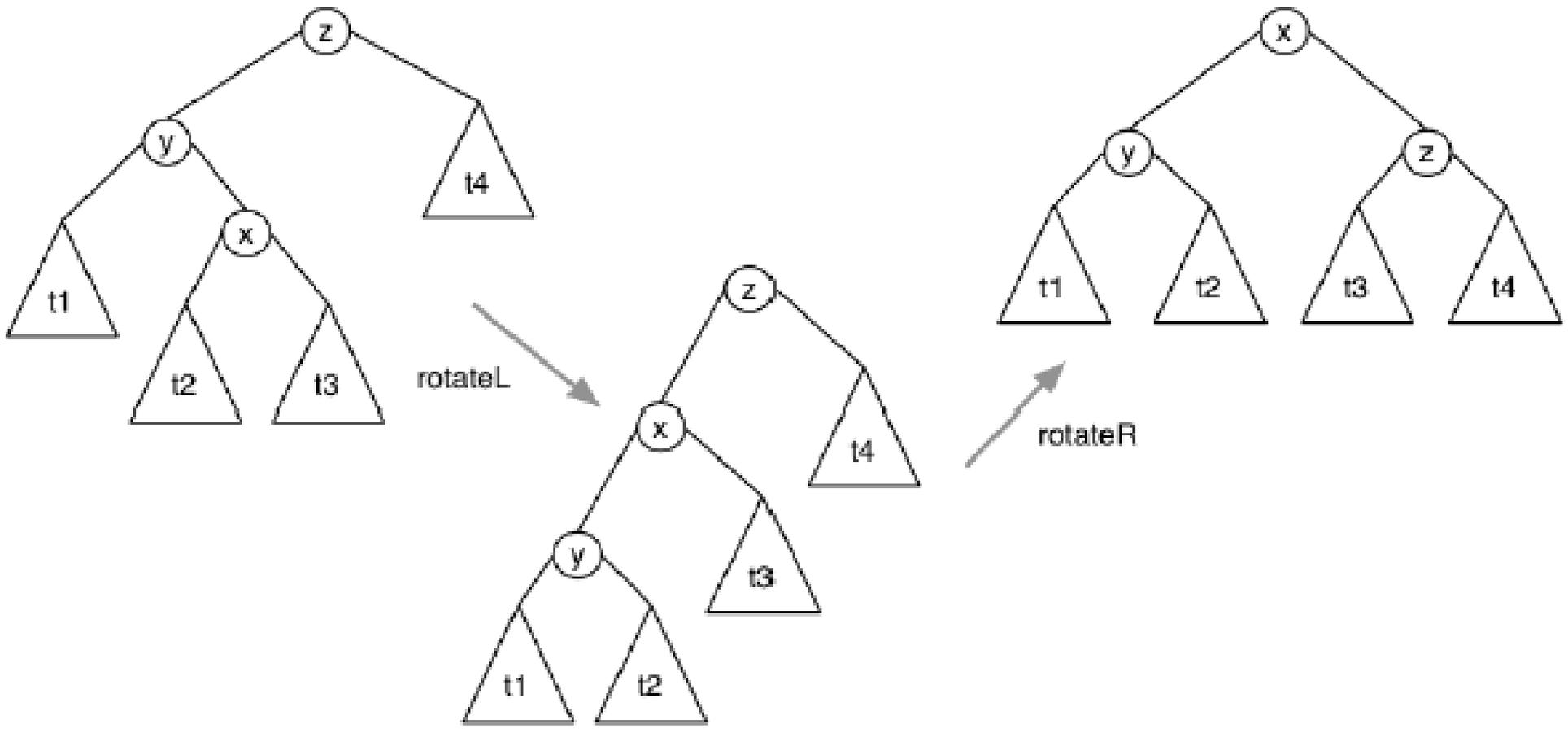
Double Rotation: Left of Left

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



SPLAY TREES

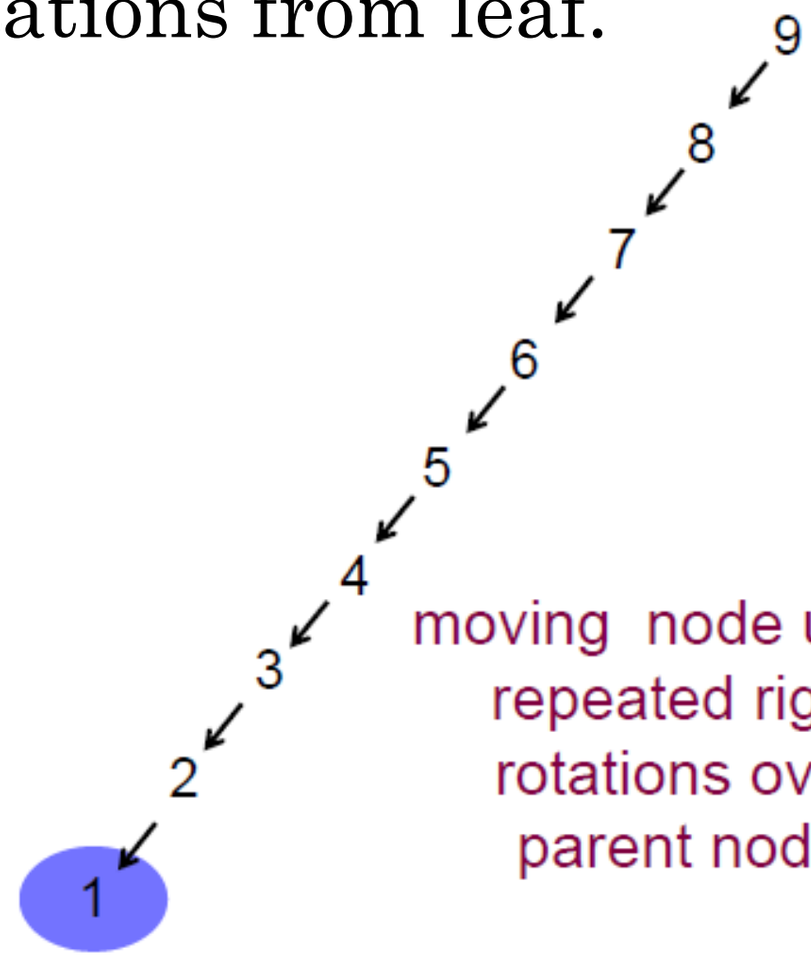
- Double Rotation: Right of Left
- First rotate at x's parent, y, then at x's parent z



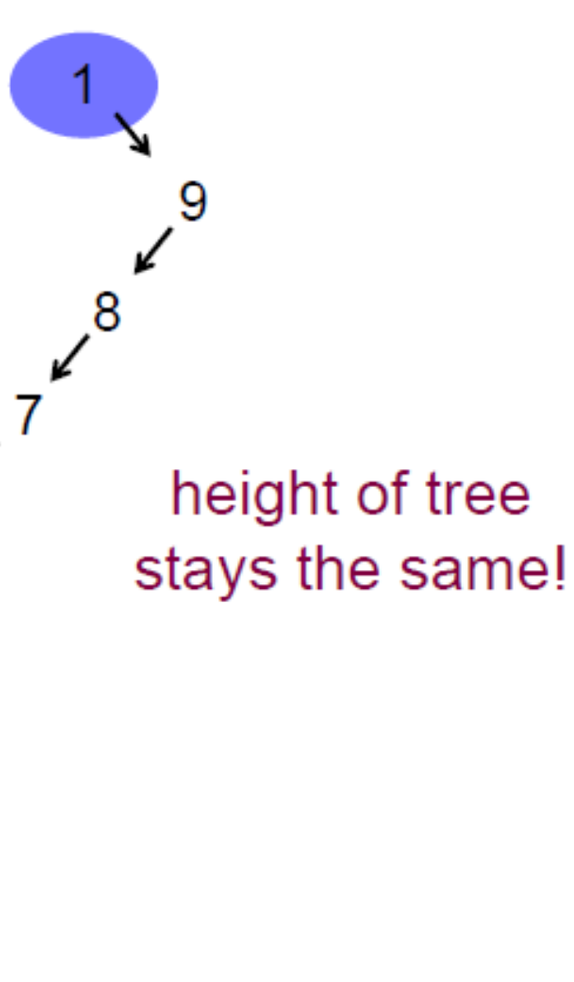
Root Insertion Versus Splay Tree Insertion

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

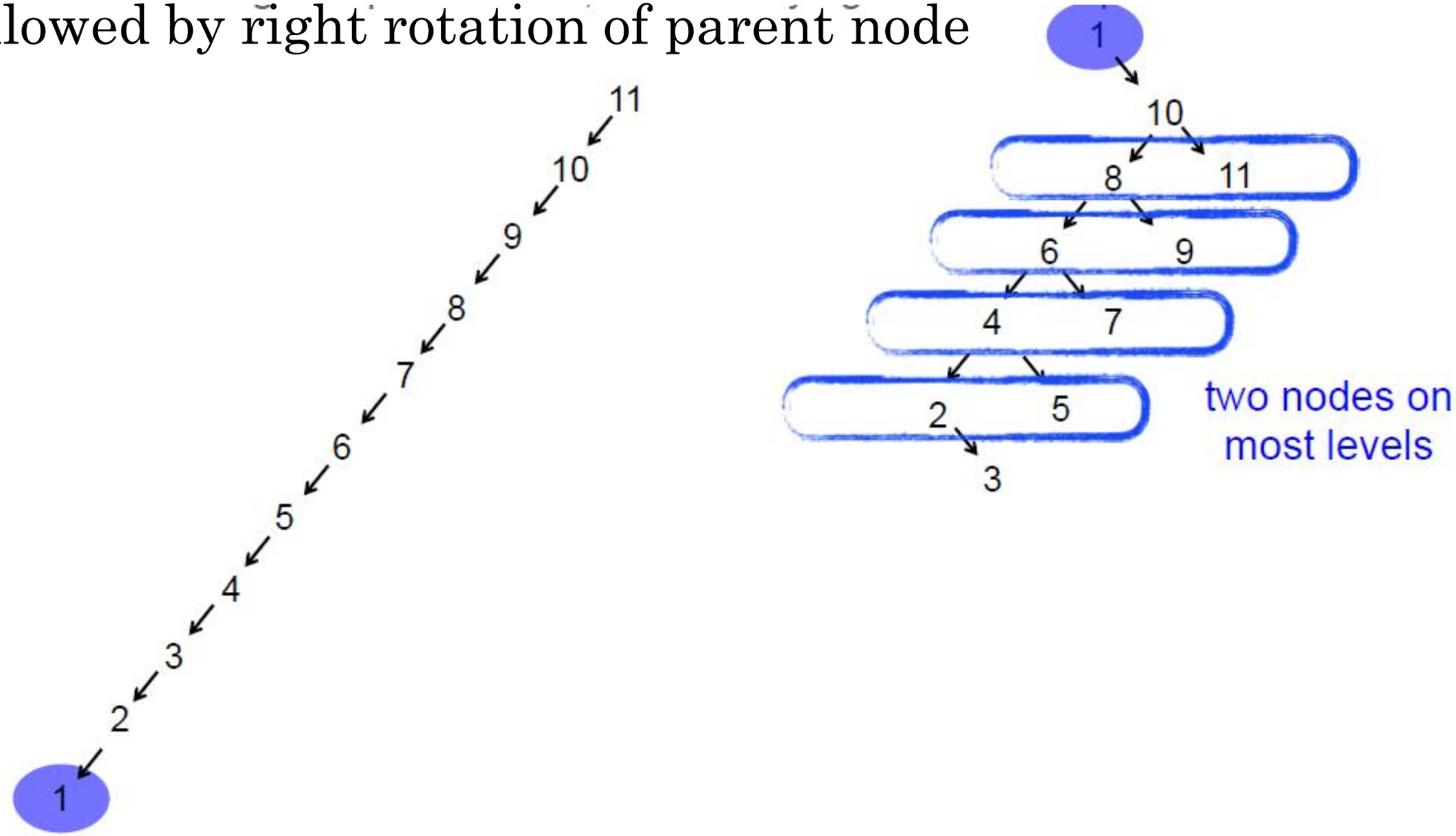


height of tree stays the same!

Root Insertion Versus Splay Tree Insertion

Worst Case for splay insertion:

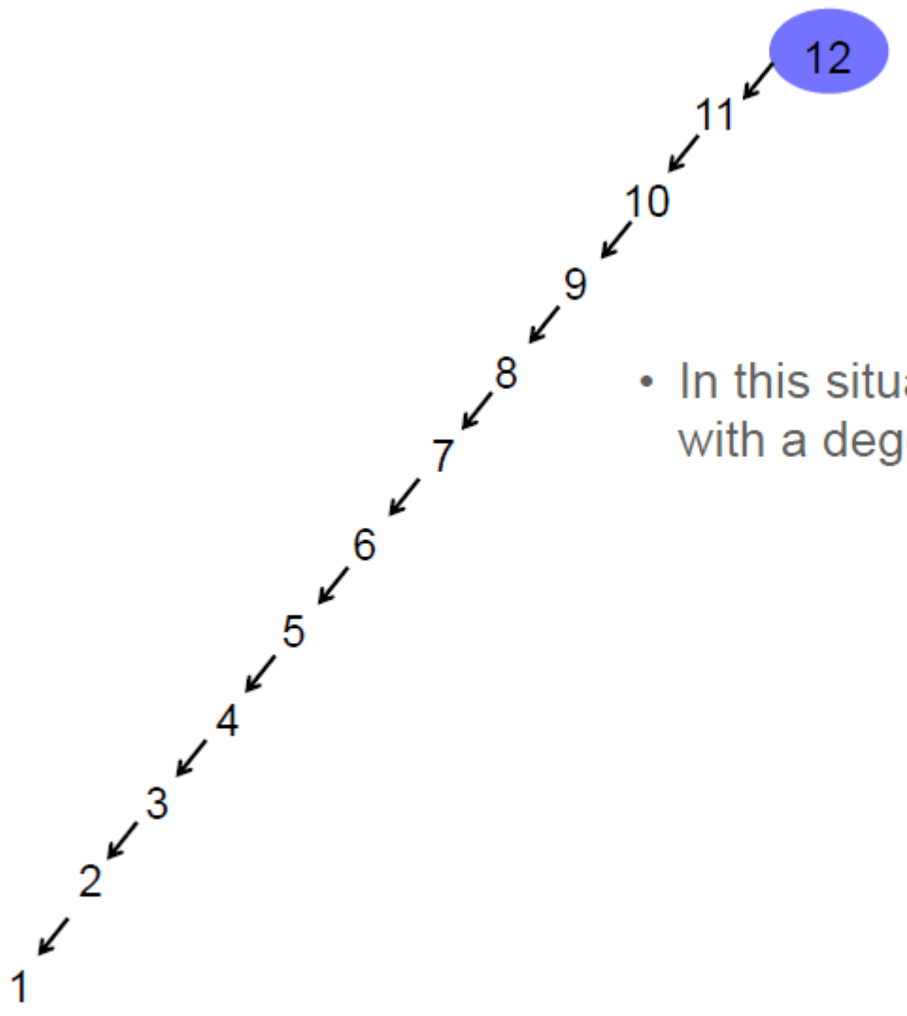
- O(N) steps for insertion : inserting node 1 in this worst case degenerate tree
- Splay insertion improves balance of tree : move smallest item up a degenerated tree using right rotation of grand parent node, followed by right rotation of parent node



Root Insertion Versus Splay Tree Insertion

Another worst case scenario for splay insertion :

- 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 insertion 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 (O(1) in last example)

Overall Work Complexity

- assuming we splay for both insert and search
  - assume N initial inserts, then M searches
    N log N insert cost, M log N 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

AVL Trees

Approach

- insertion (at leaves) may cause imbalance
- repair balance as soon as we notice imbalance
- repairs done locally, not by overall tree restructure

A tree is unbalanced when: abs(depth(left)-depth(right)) > 1
This can be repaired by a single rotation:

- if left subtree too deep, rotate right
- if right subtree too deep, rotate left

Problem: determining height/depth of subtrees may be expensive.

AVL Trees

Implementation of AVL Insertion

```
Tree insertAVL(Tree t, Item it)
    { if (t == NULL) return newNode(it);
      int diff = cmp(key(it), key(t->value));
      if (diff == 0) t->value = it;
      else if (diff < 0) t->left = insertAVL(t->left, it);
else if (diff > 0) t->right = insertAVL(t->right, it);
 int dL = depth(t->left);
 int dR = depth(t->right);
 if ((dL - dR) > 1) t = rotateR(t);
 else if ((dR - dL) > 1)
            t = rotateL(t);
 return t; }
```

AVL Trees

Function **insertAVL()** in TreeLab

- use option **A** in command-line to perform AVL insertion
- inefficient function

Why is it inefficient?

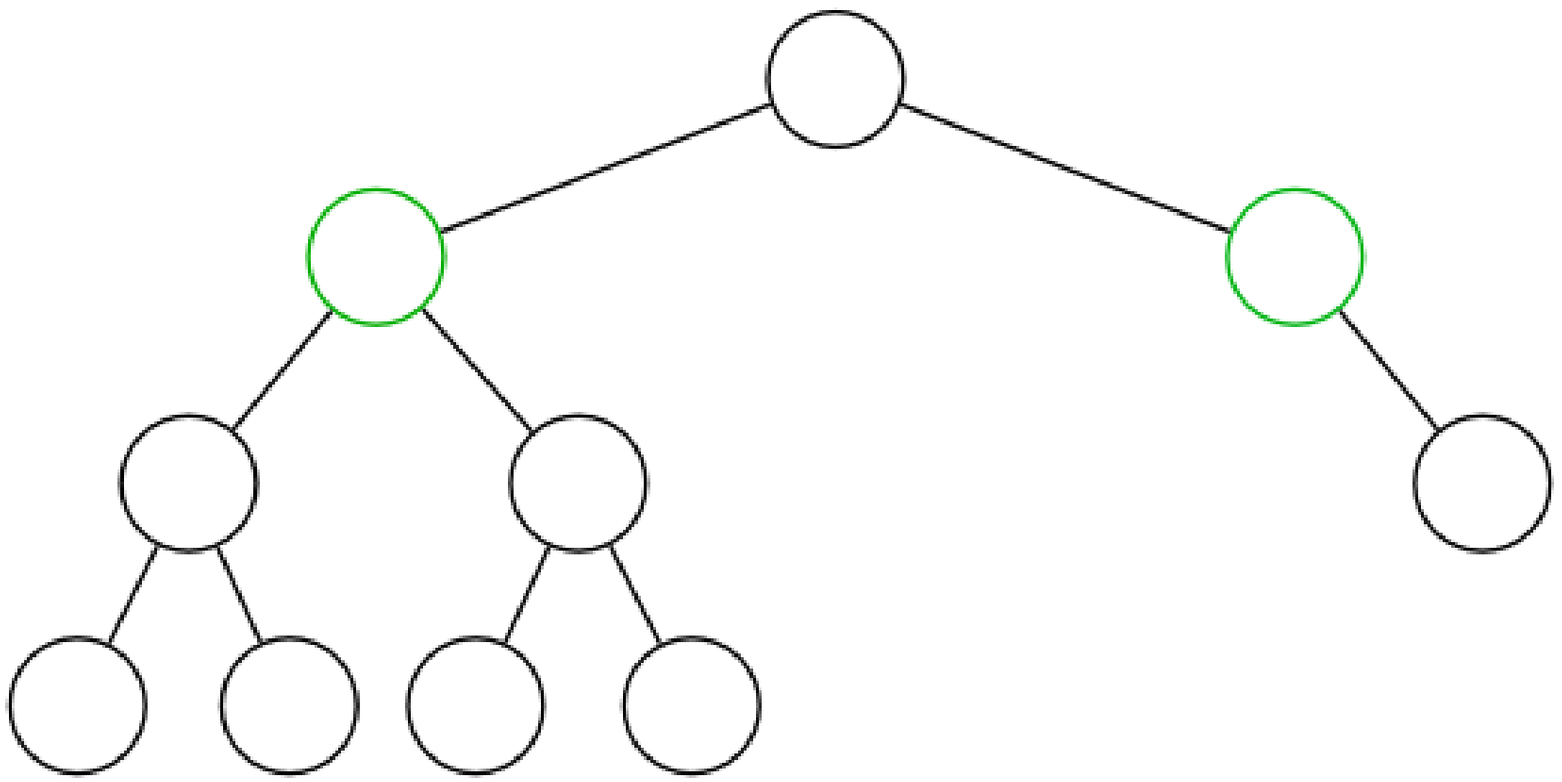
- computes **depth()** as part of recursion
- computes **depth()** multiple times on same branch
- depth() itself requires multiple recursion

Could assist by storing height of subtree in each node

AVL Trees

Analysis of AVL Trees

- trees are height-balanced; sub-tree depths differ by +/-1
- average/worst-case search performance of *O(logN)*
- *require* extra data to be stored in each node (efficiency)
- may not be weight-balanced; sub-tree sizes may differ



2-3-4 Trees

Next, 2-3-4 Trees…