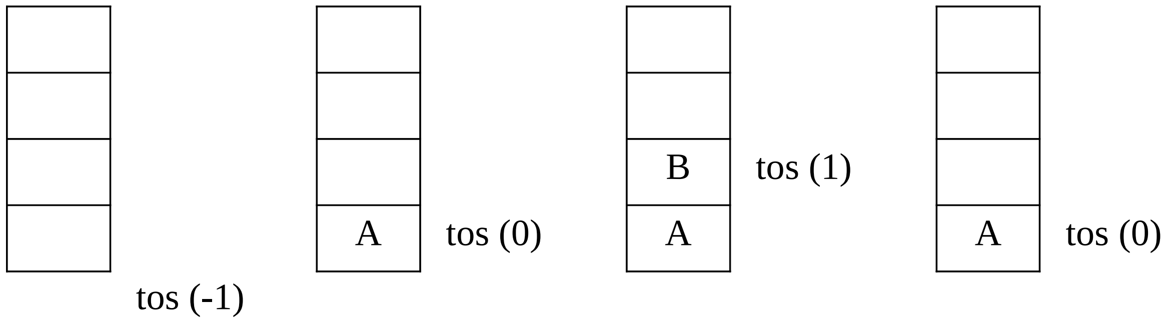
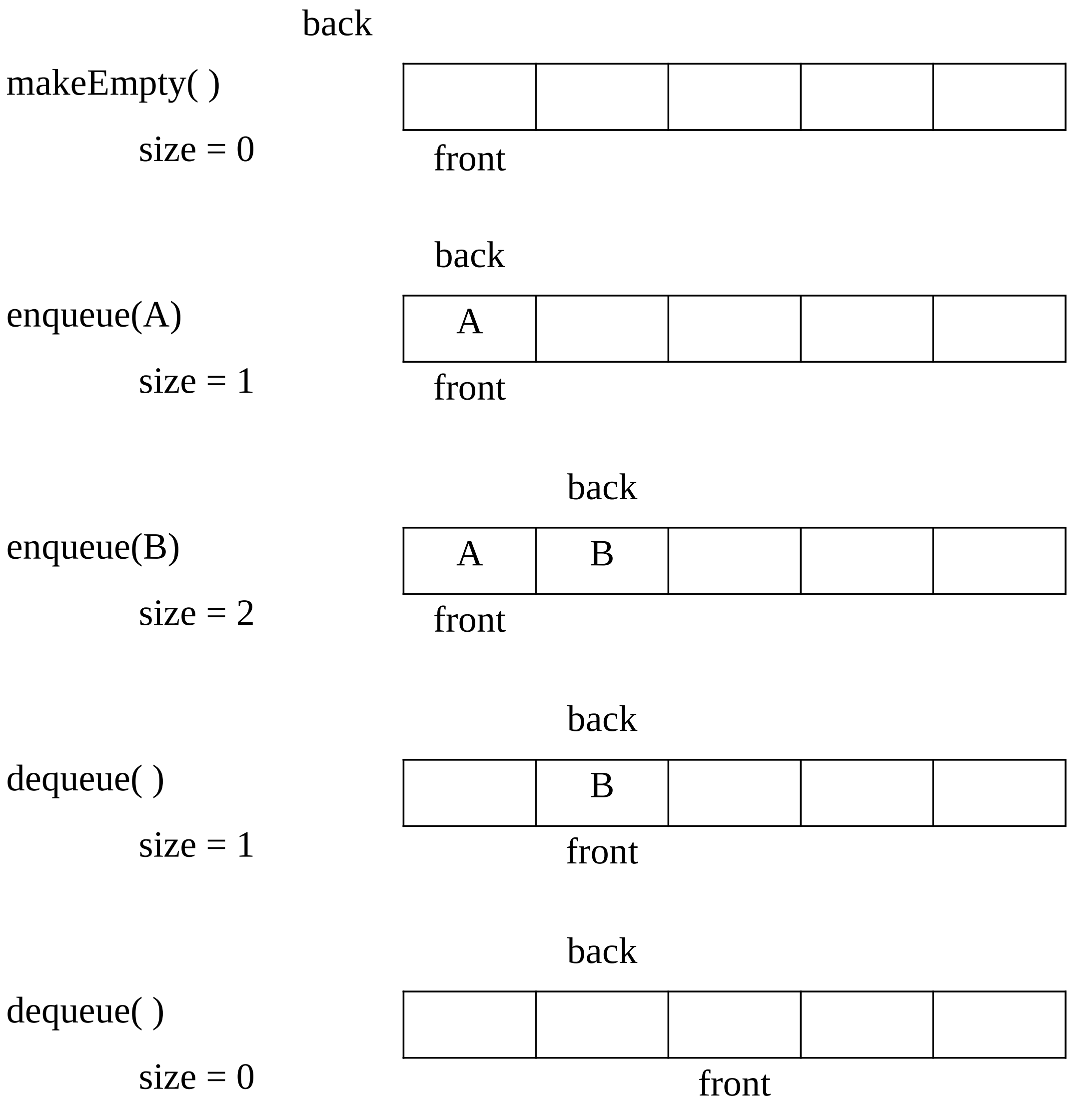
Chapter 15

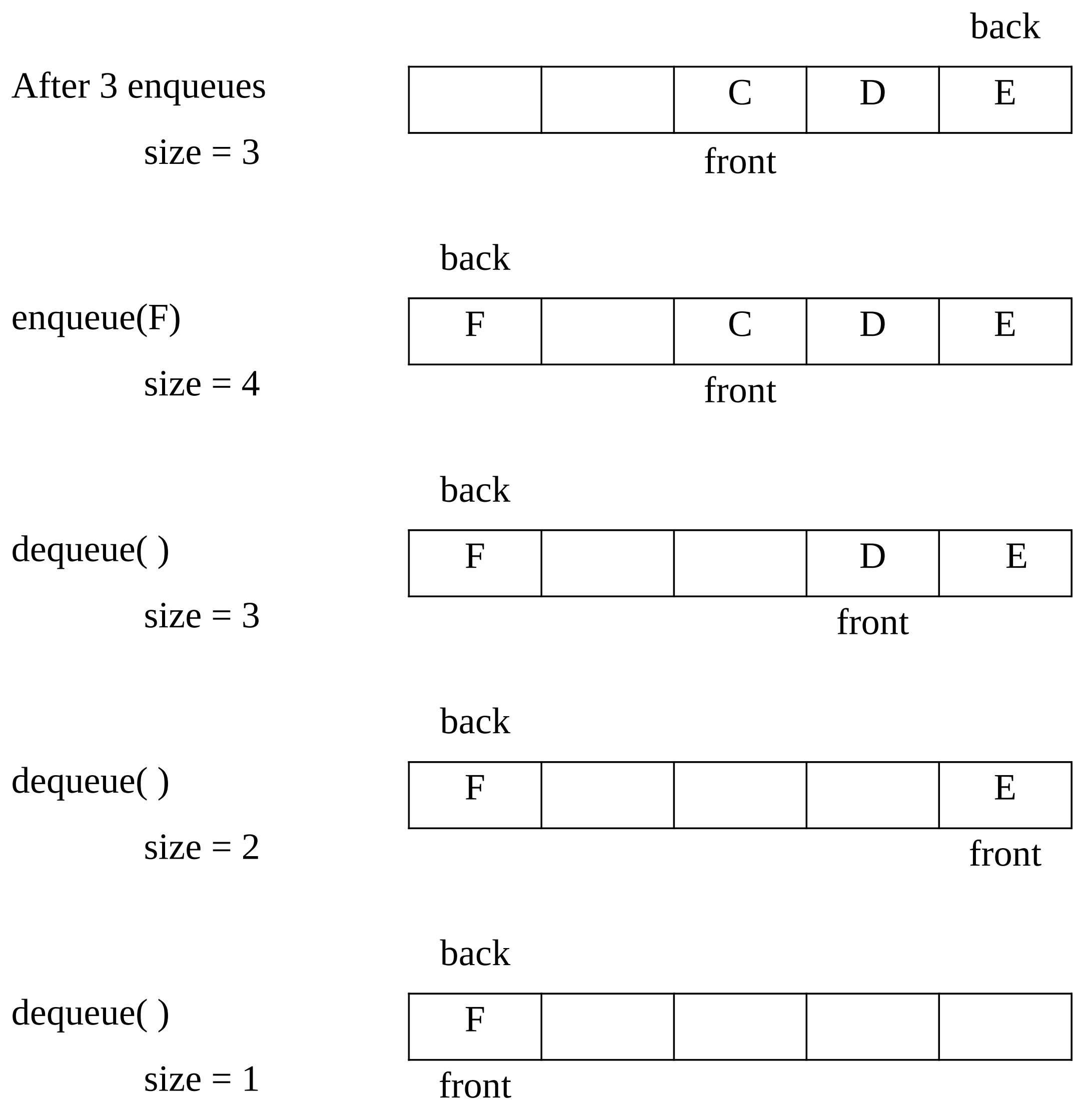
Stacks and Queues



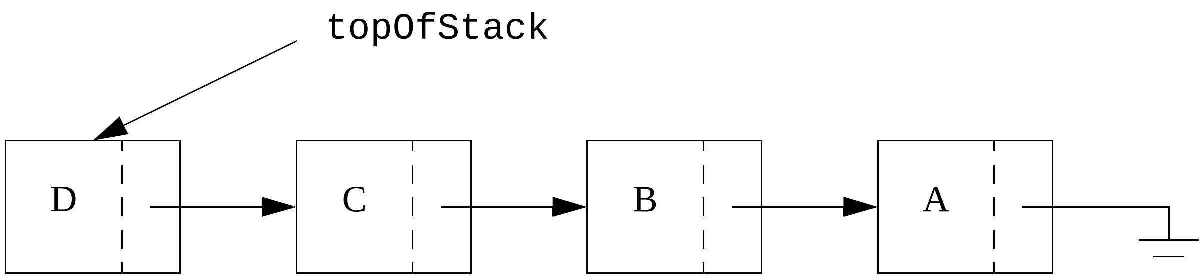
How the stack routines work: empty stack, `push(A)`, `push(B)`, `pop`



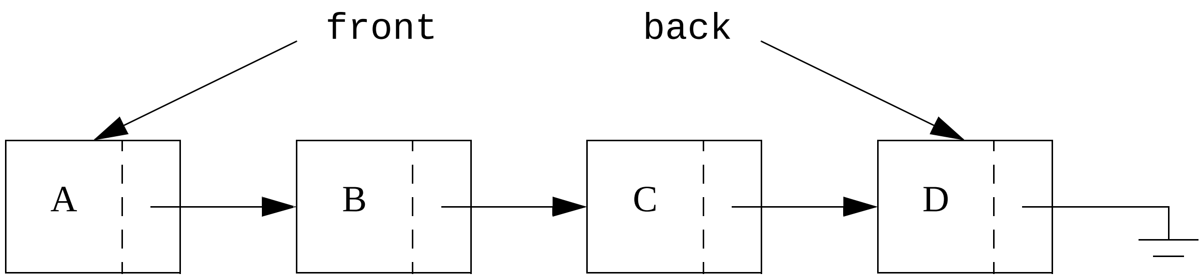
Basic array implementation of the queue



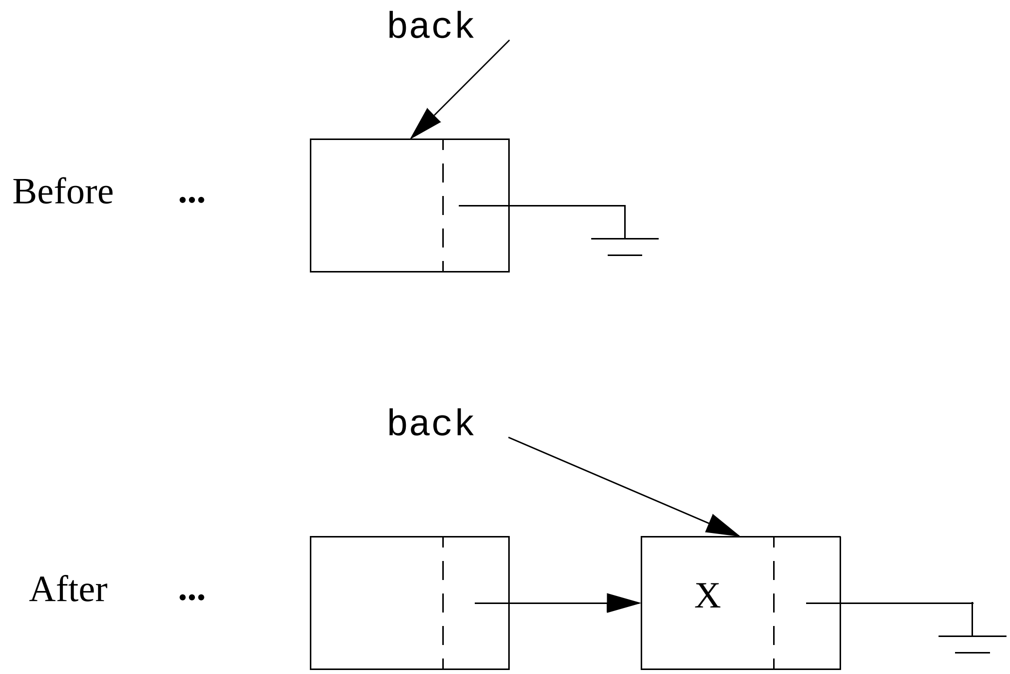
Array implementation of the queue with wraparound



Linked list implementation of the stack



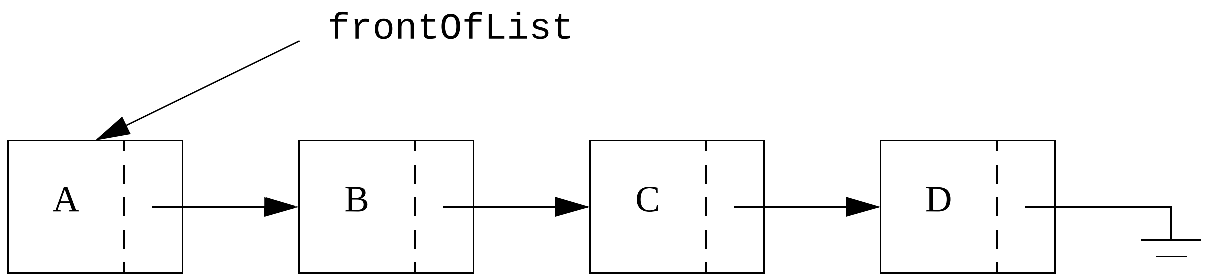
Linked list implementation of the queue



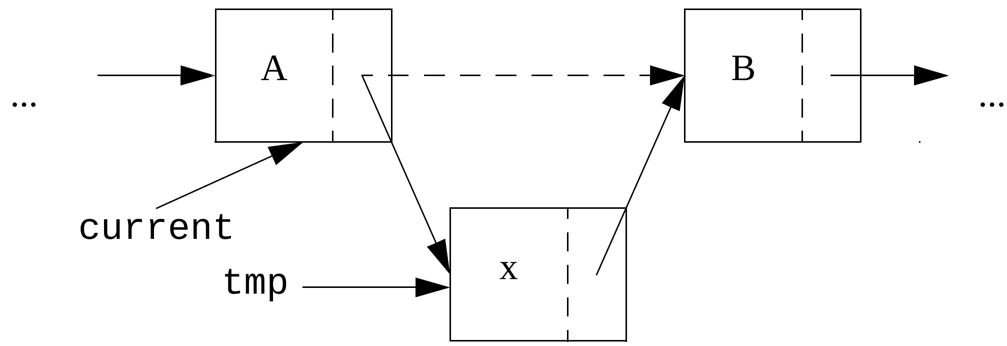
enqueue operation for linked-list-based implementation

Chapter 16

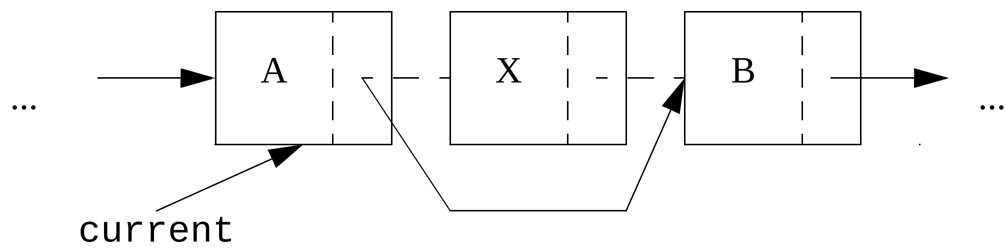
Linked Lists



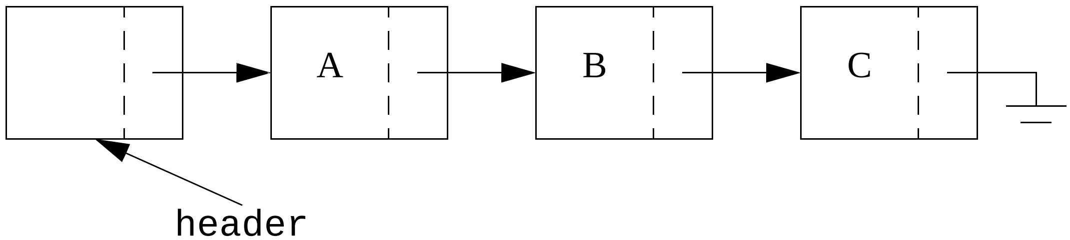
Basic linked list



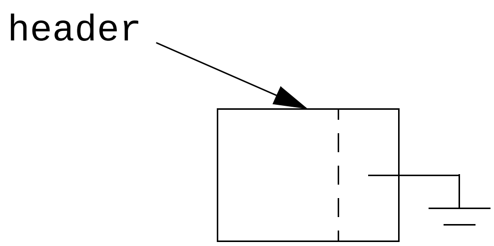
Insertion into a linked list: create new node (`tmp`), copy in `x`, set `tmp`'s `next` reference, set `current`'s `next` reference



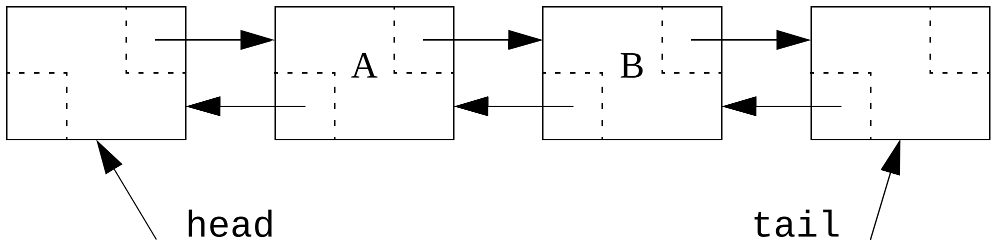
Deletion from a linked list



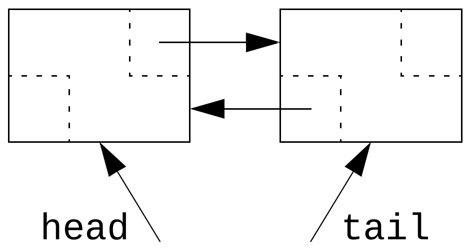
Using a header node for the linked list



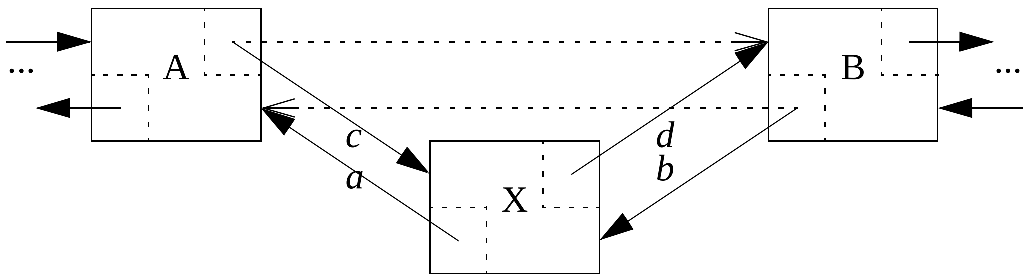
Empty list when header node is used



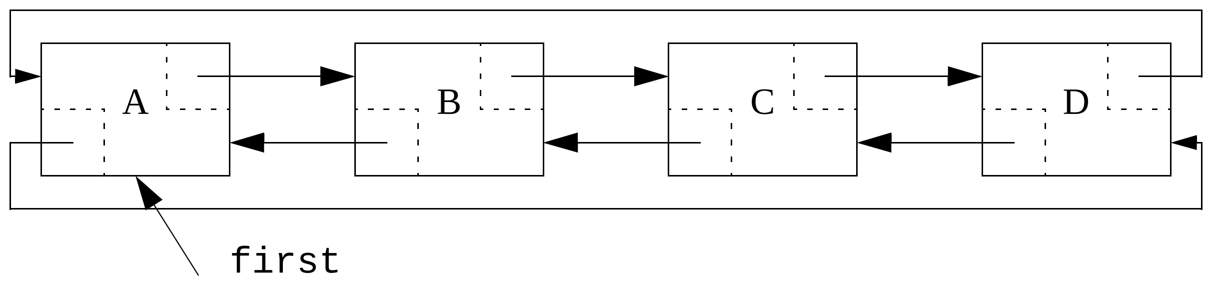
Doubly linked list



Empty doubly linked list



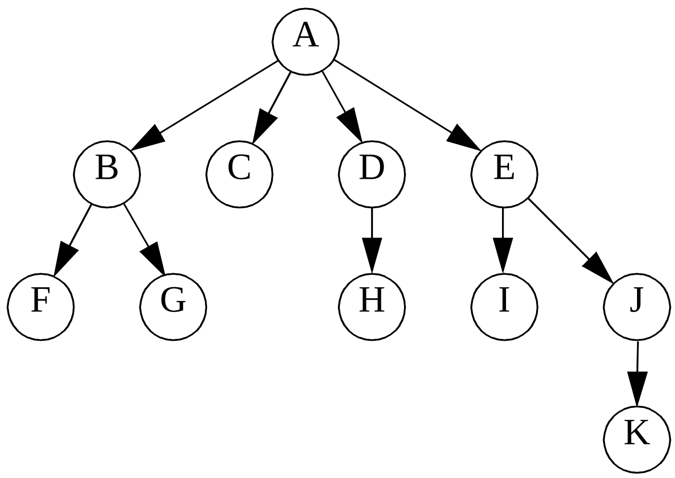
Insertion into a doubly linked list by getting new node and then changing references in order indicated



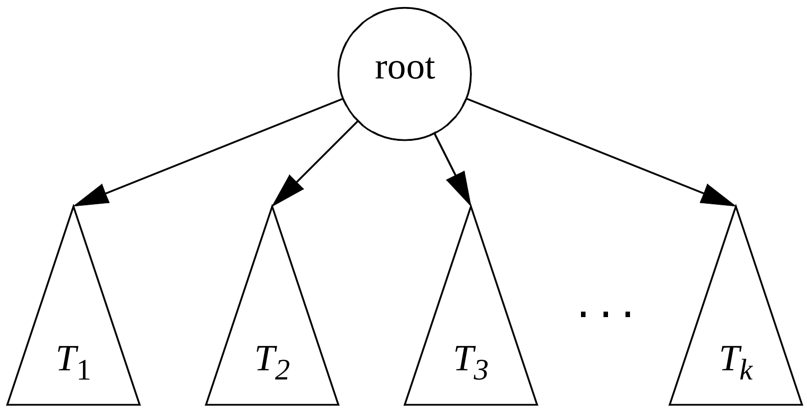
Circular doubly linked list

Chapter 17

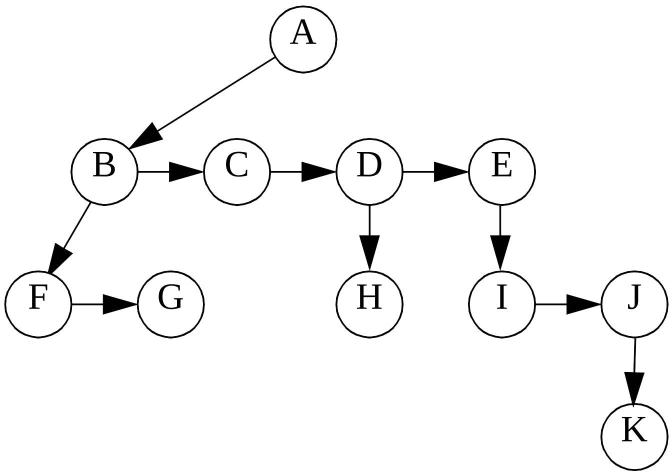
Trees



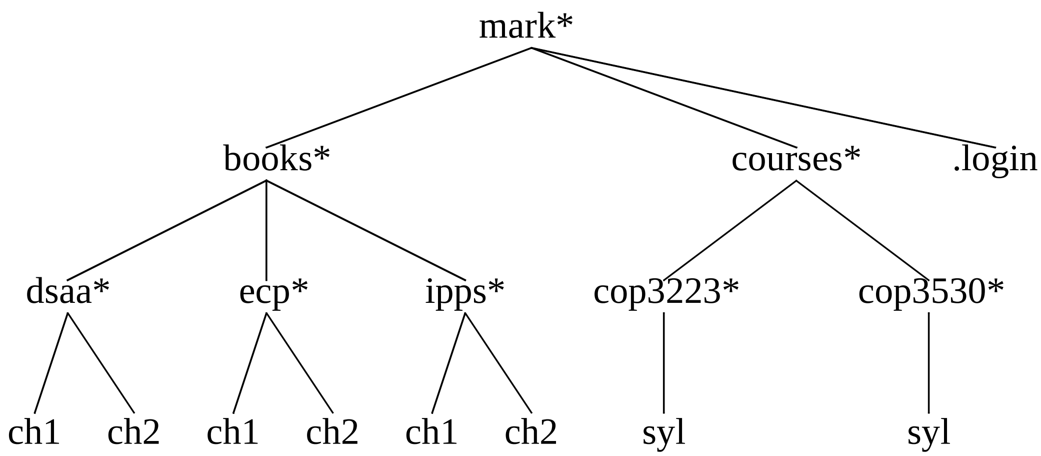
A tree



Tree viewed recursively



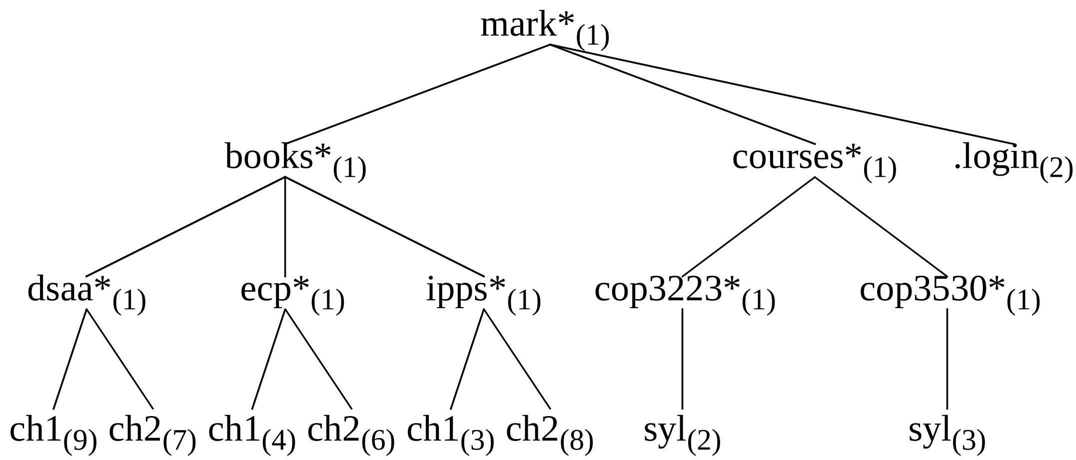
First child/next sibling representation of tree in Figure 17.1



Unix directory

```
mark
        books
                dsaa
                        ch1
                        ch2
                ecp
                        ch1
                        ch2
                ipps
                        ch1
                        ch2
        courses
                cop3223
                        syl
                cop3530
                        syl
        .login
```

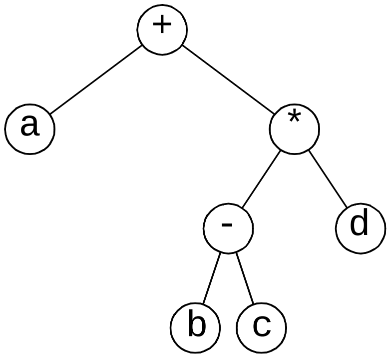
The directory listing for tree in Figure 17.4

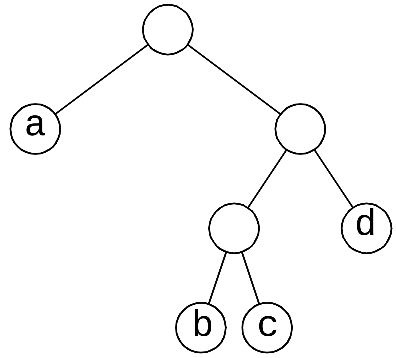


Unix directory with file sizes

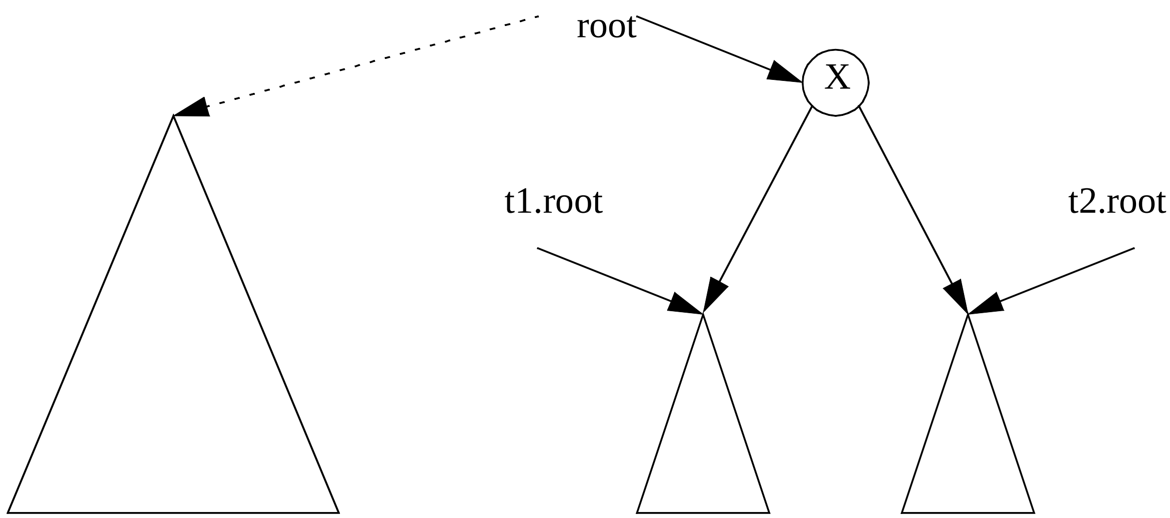
```
                        ch1                         9
                        ch2                         7
                dsaa                               17
                        ch1                         4
                        ch2                         6
                ecp                                11
                        ch1                         3
                        ch2                         8
                ipps                               12
        books                                      41
                        syl                         2
                cop3223                             3
                        syl                         3
                cop3530                             4
        courses                                     8
        .login                                      2
mark                                               52
```

Trace of the `size` method

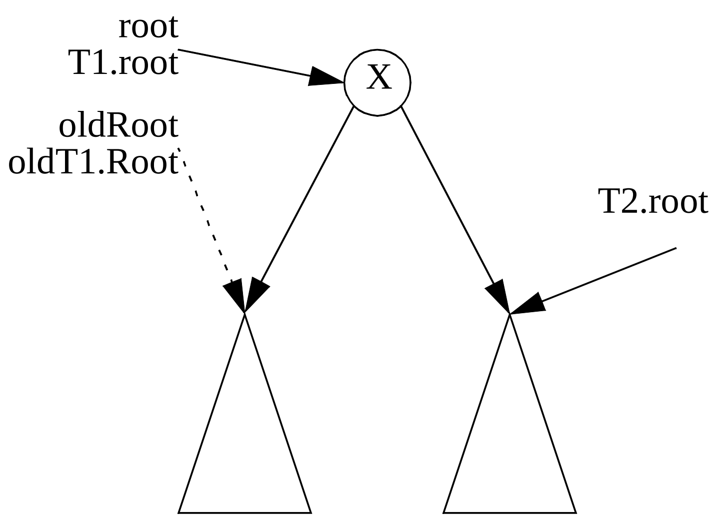




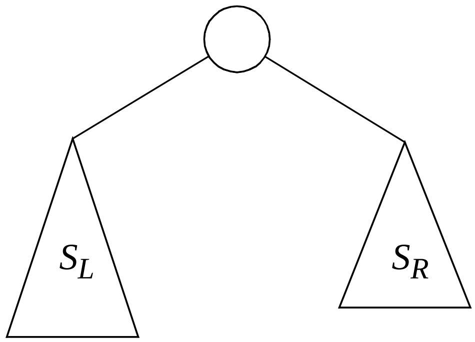
Uses of binary trees: left is an expression tree and right is a Huffman coding tree



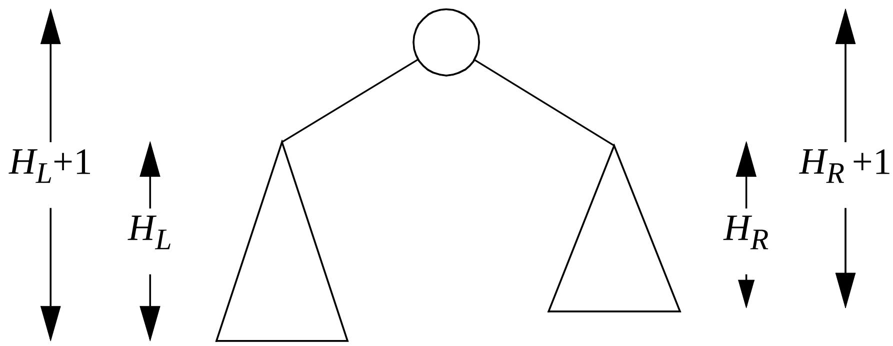
Result of a naive `merge` operation



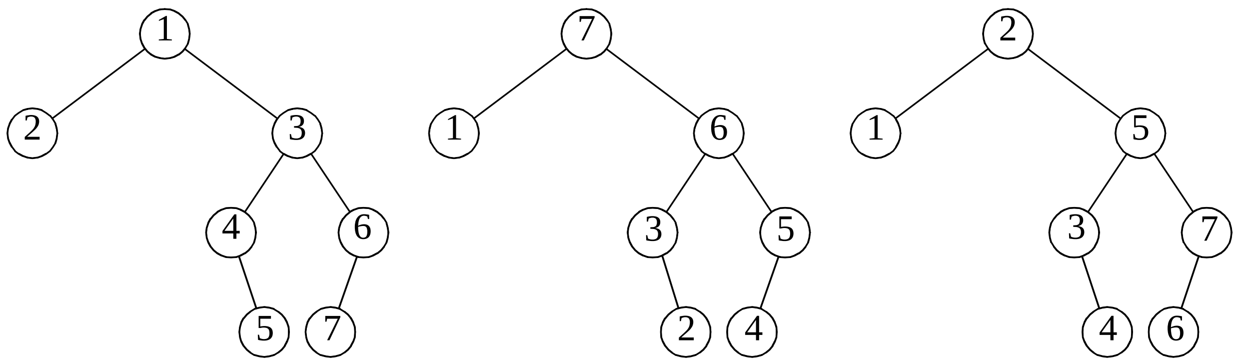
Aliasing problems in the `merge` operation; `T1` is also the current object



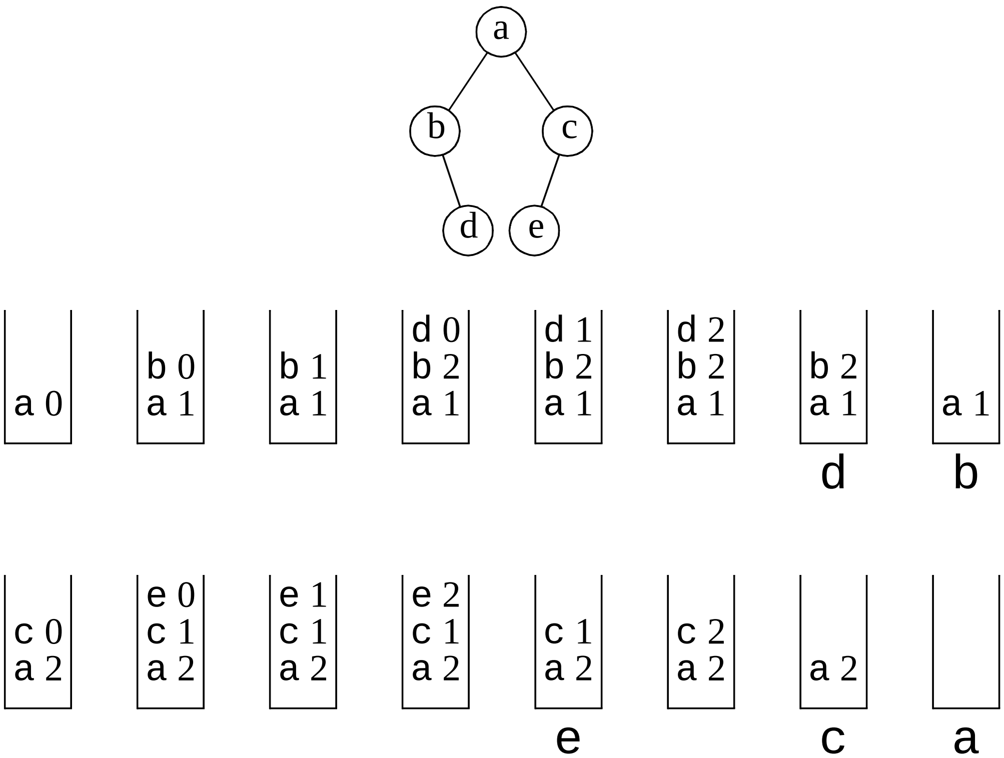
Recursive view used to calculate the size of a tree:
$S_T = S_L + S_R + 1$



Recursive view of node height calculation:
H_T = max(H_L+1, H_R +1)



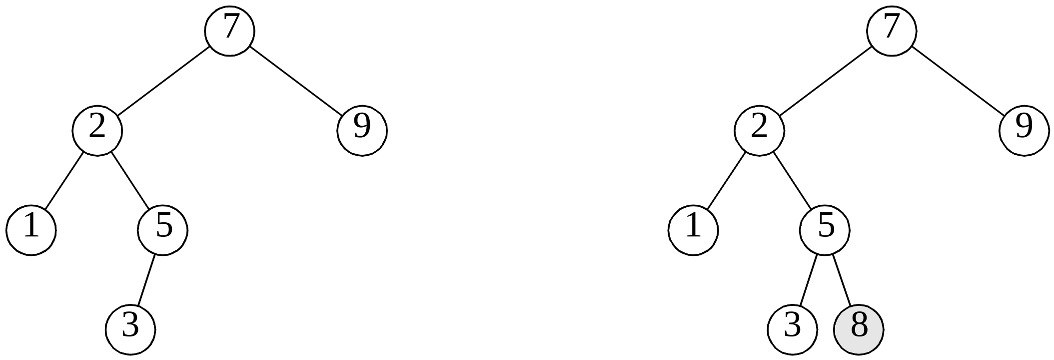
Preorder, postorder, and inorder visitation routes



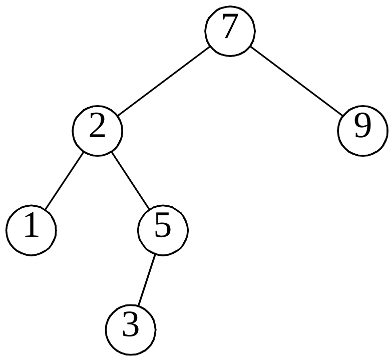
Stack states during postorder traversal

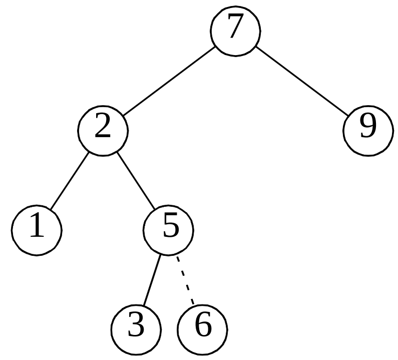
Chapter 18

Binary Search Trees

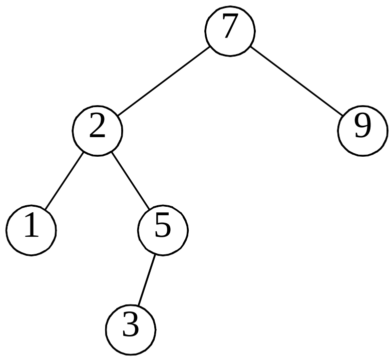


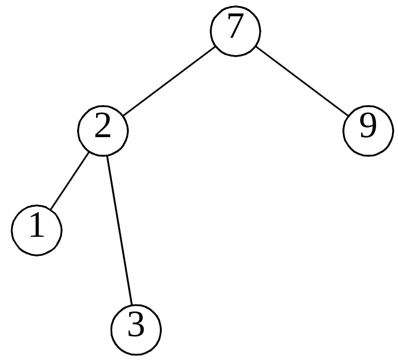
Two binary trees (only the left tree is a search tree)



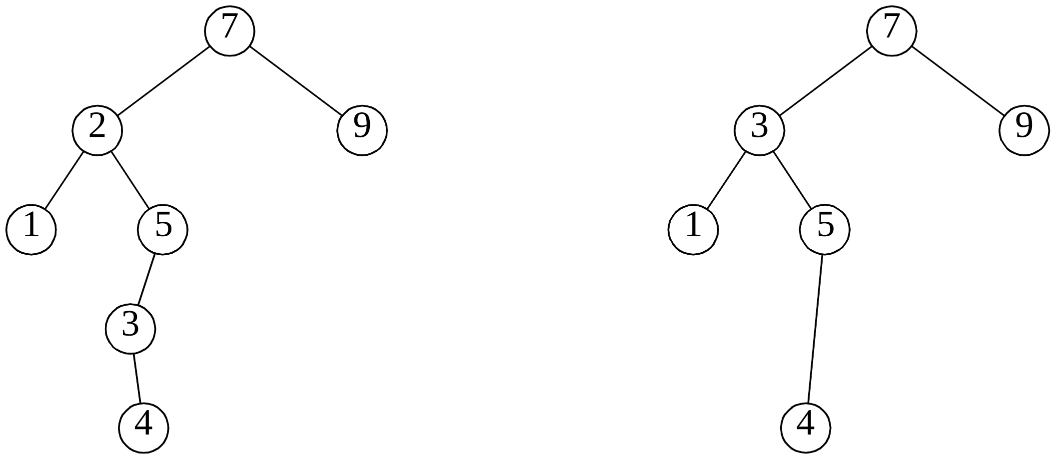


Binary search trees before and after inserting 6

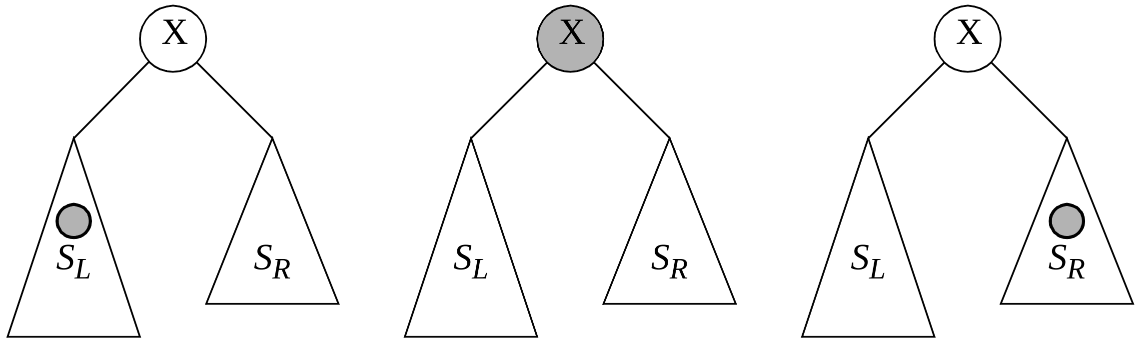




Deletion of node 5 with one child, before and after

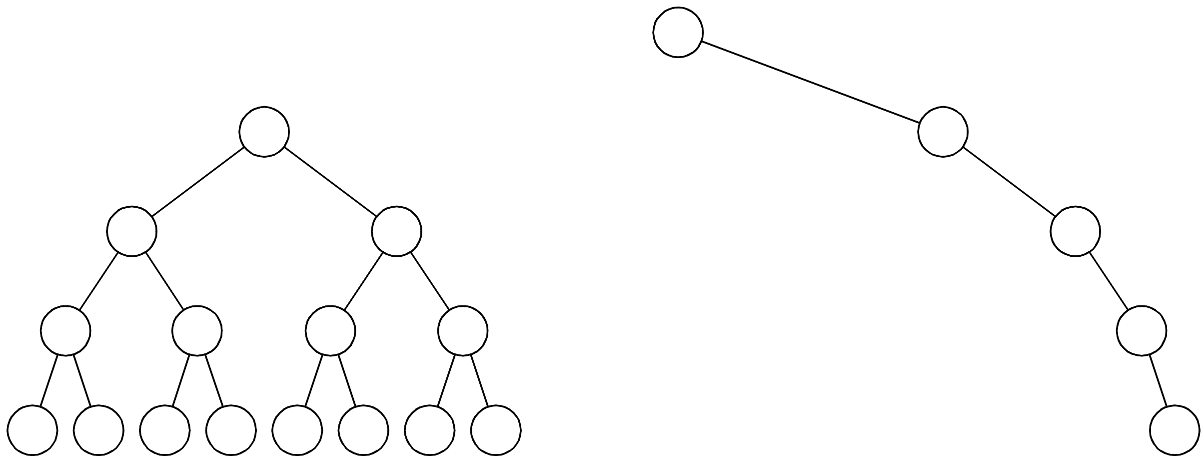


Deletion of node 2 with two children, before and after

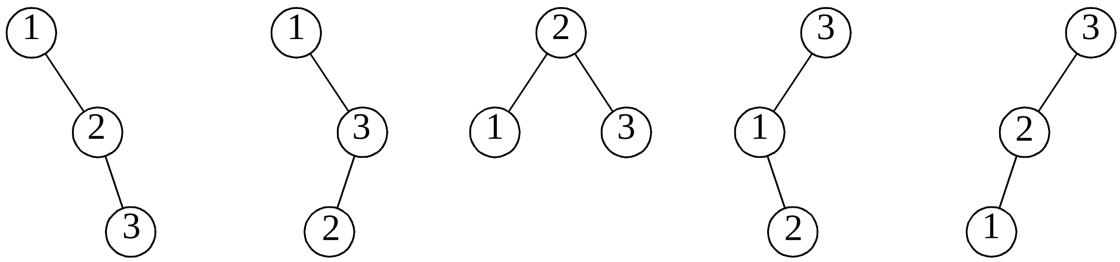


$K < S_L + 1$ $K == S_L + 1$ $K > S_L + 1$

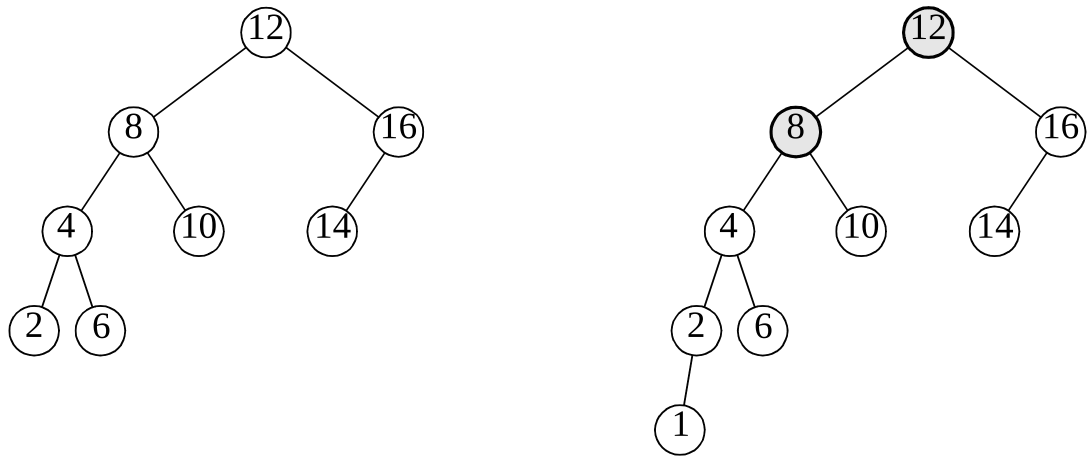
Using the `size` data field to implement `findKth`



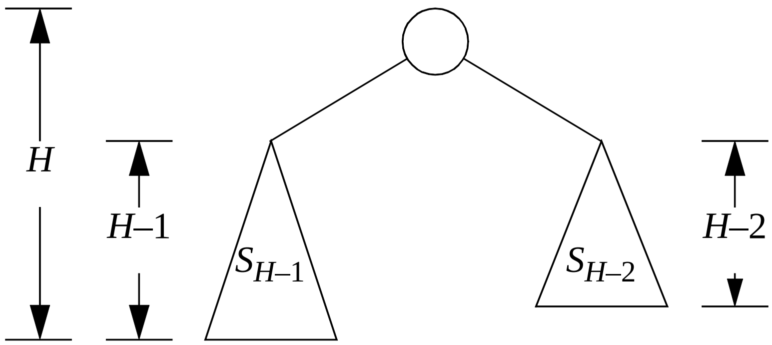
Balanced tree on the left has a depth of log *N*; unbalanced tree on the right has a depth of *N*–1



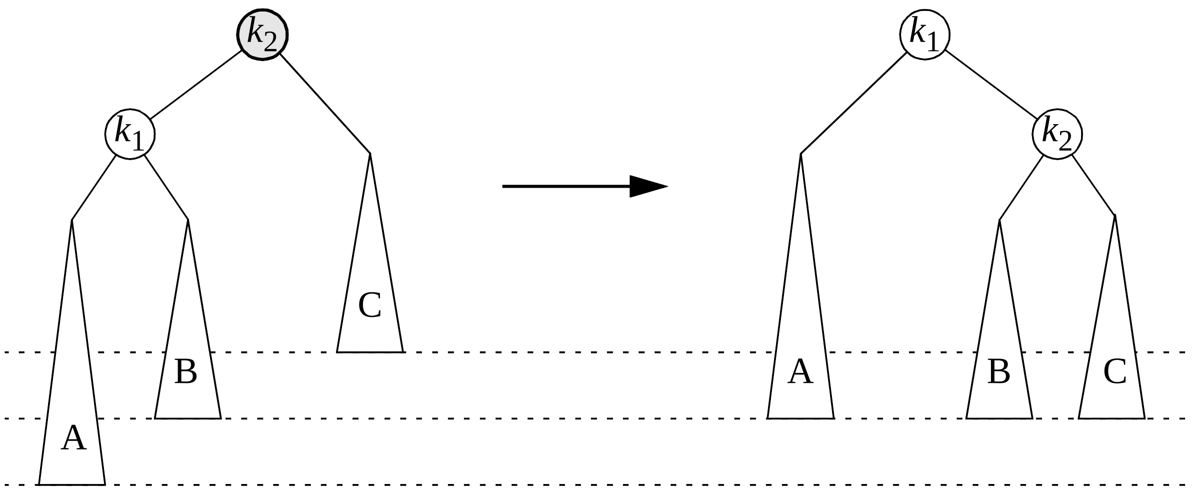
Binary search trees that can result from inserting a permutation 1, 2, and 3; the balanced tree in the middle is twice as likely as any other



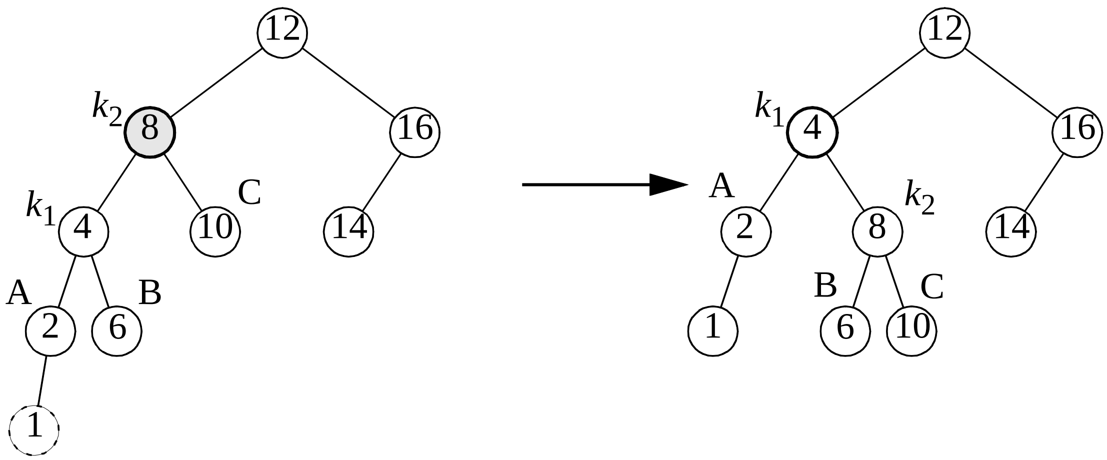
Two binary search trees: the left tree is an AVL tree, but the right tree is not (unbalanced nodes are darkened)



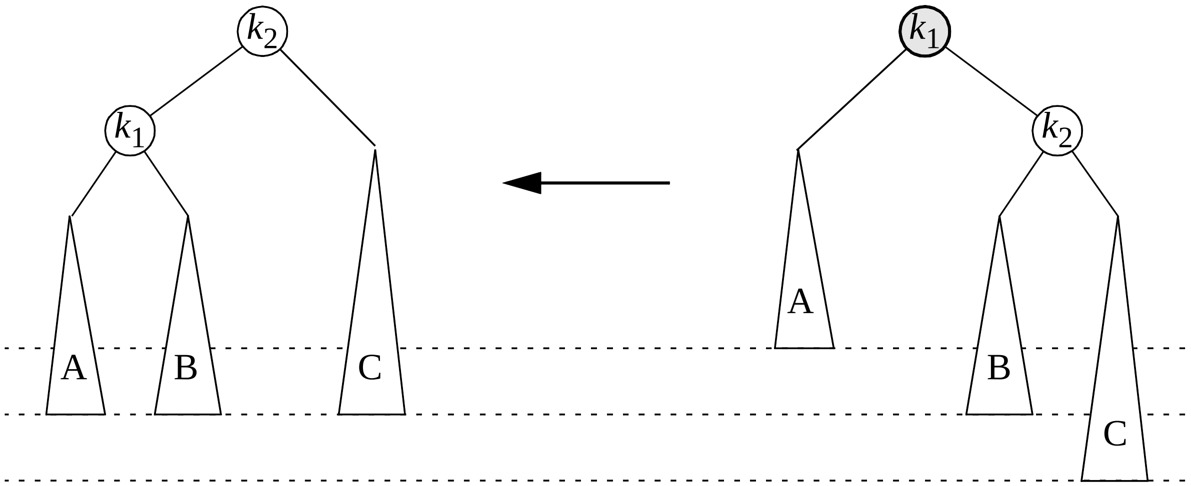
Minimum tree of height *H*



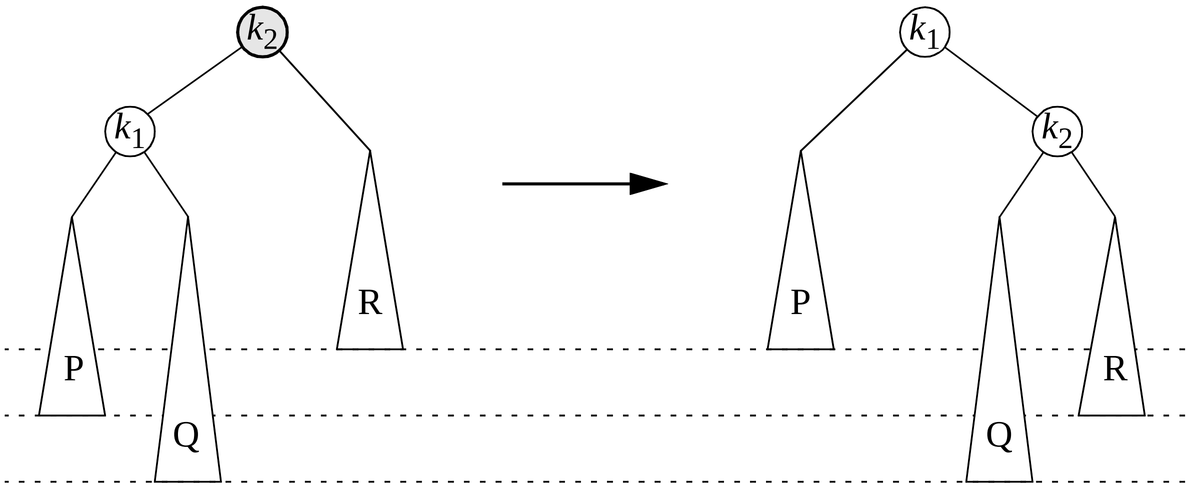
Single rotation to fix case 1



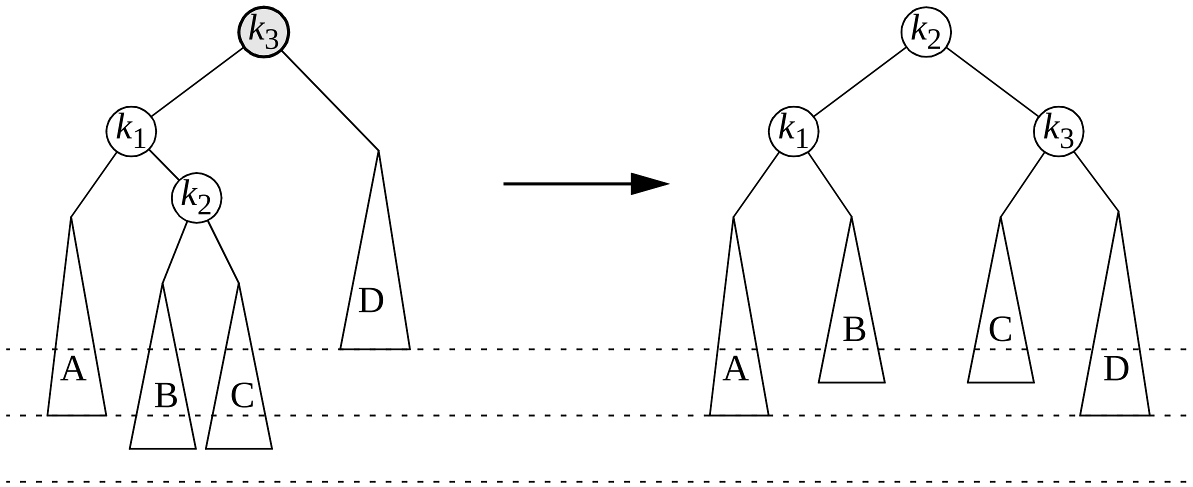
Single rotation fixes AVL tree after insertion of 1



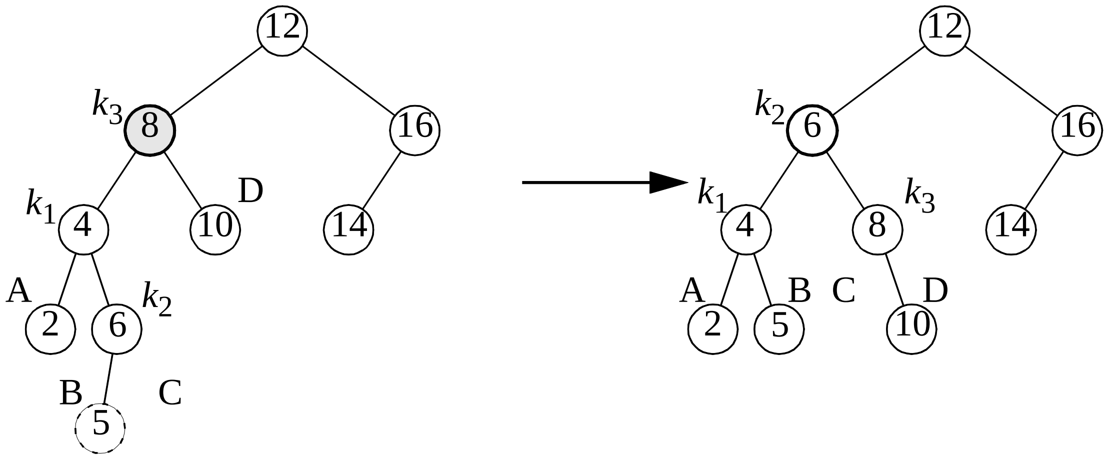
Symmetric single rotation to fix case 4



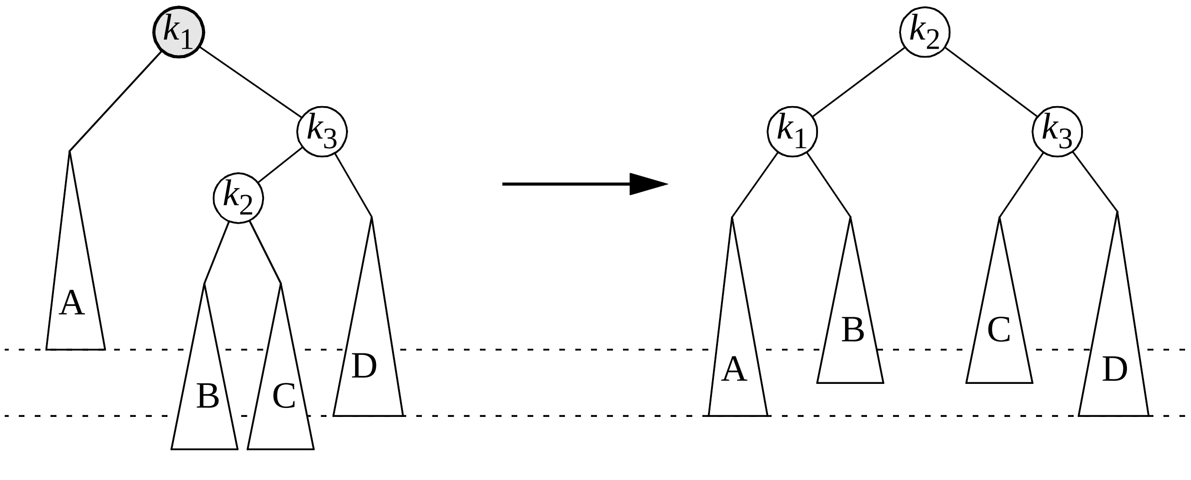
Single rotation does not fix case 2



Left-right double rotation to fix case 2



Double rotation fixes AVL tree after insertion of 5

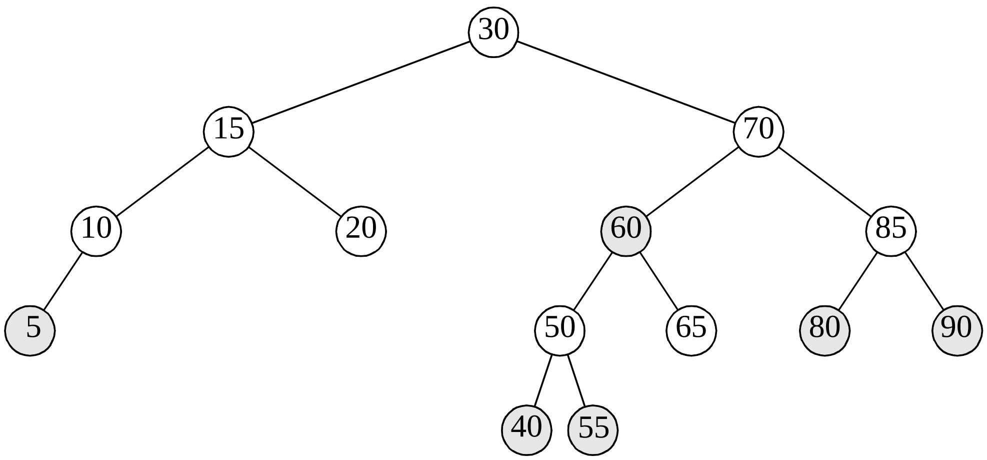


Left-right double rotation to fix case 3

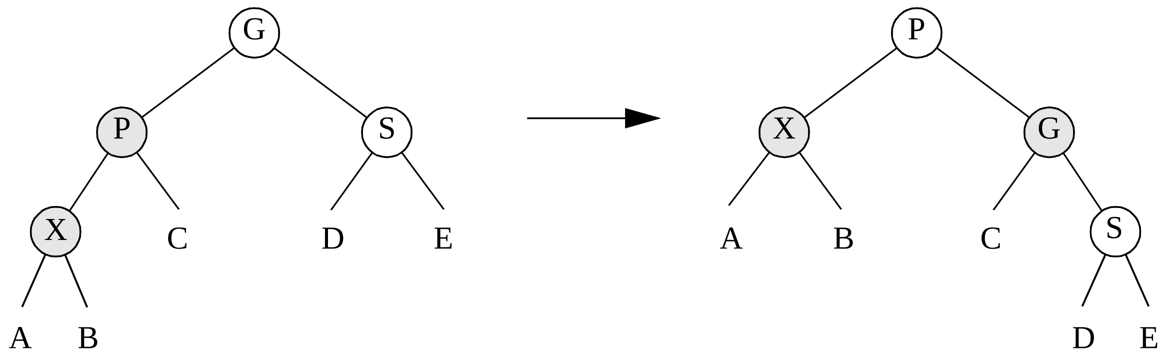
A red-black tree is a binary search tree with the following ordering properties:

1. Every node is colored either red or black.
2. The root is black.
3. If a node is red, its children must be black.
4. Every path from a node to a `null` reference must contain the same number of black nodes.

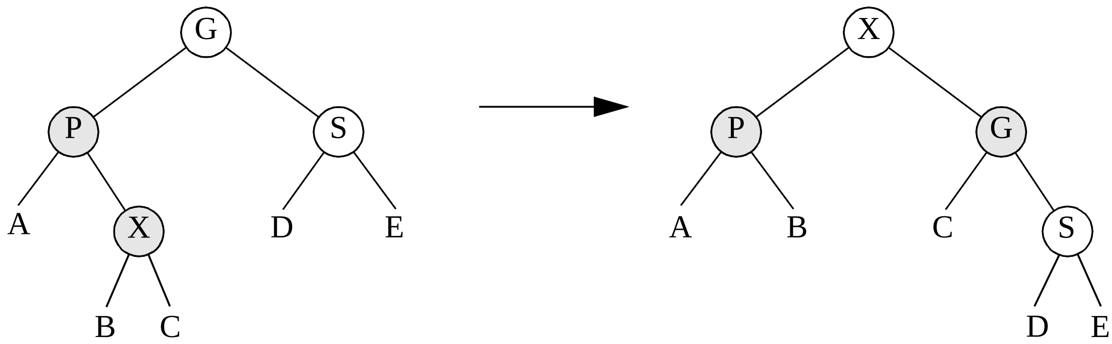
Red-black tree properties



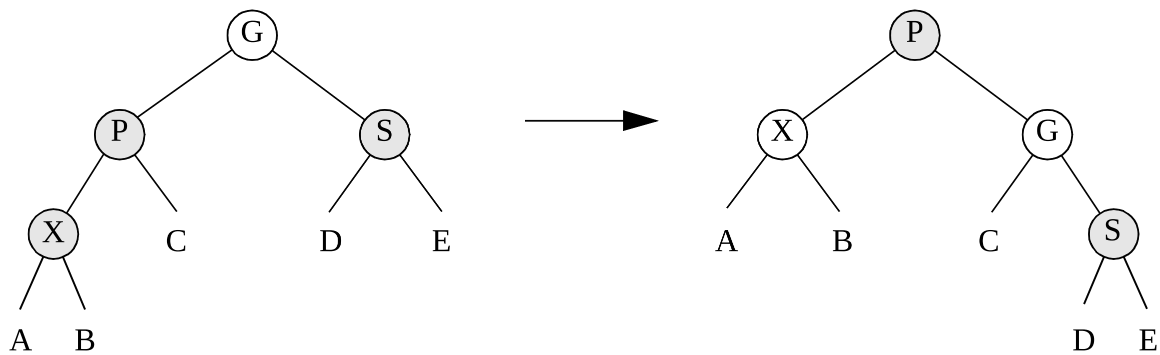
Example of a red-black tree; insertion sequence is 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55)



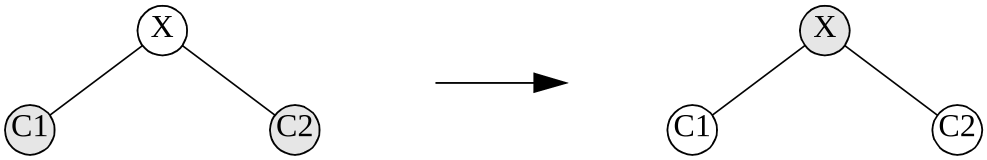
If *S* is black, then a single rotation between the parent and grandparent, with appropriate color changes, restores property 3 if *X* is an outside grandchild



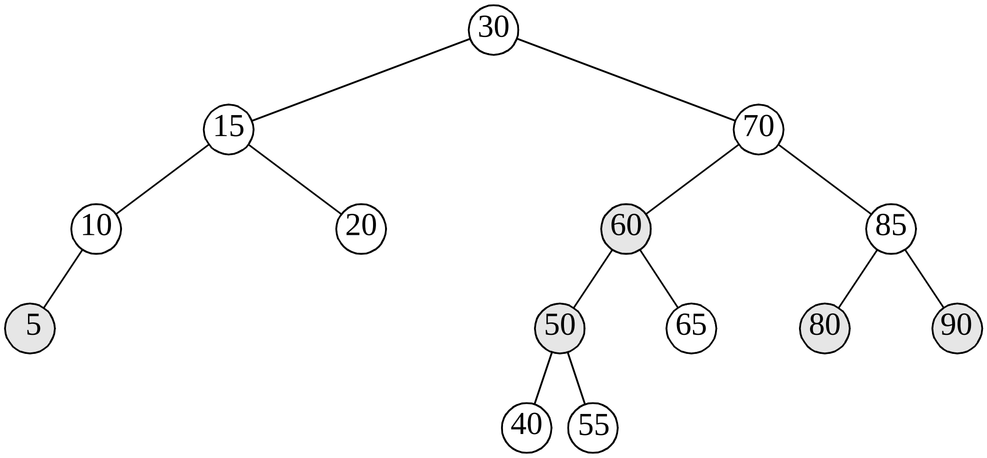
If *S* is black, then a double rotation involving *X*, the parent, and the grandparent, with appropriate color changes, restores property 3 if *X* is an inside grandchild



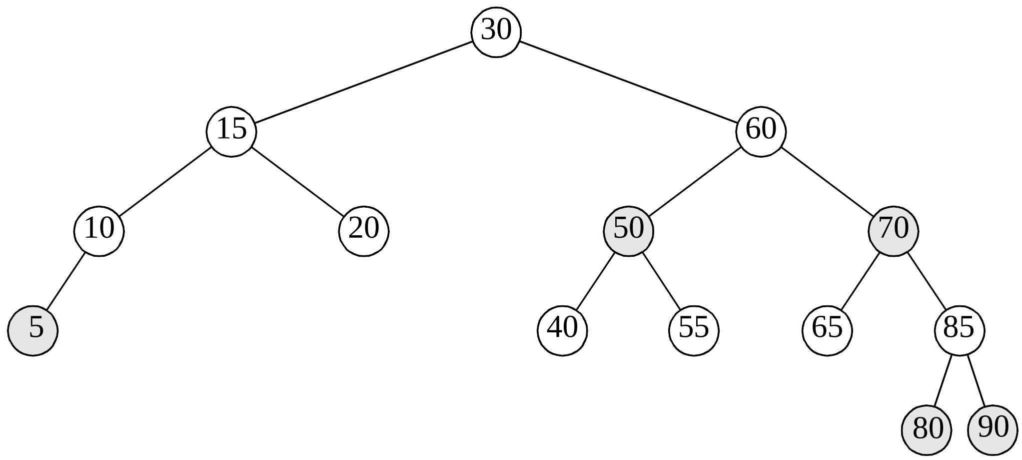
If S is red, then a single rotation between the parent and grandparent, with appropriate color changes, restores property 3 between X and P



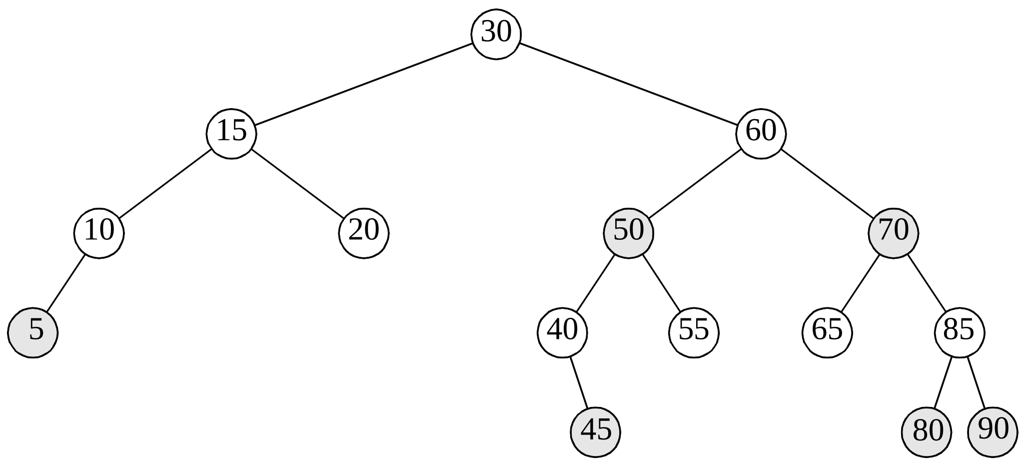
Color flip; only if *X*'s parent is red do we continue with a rotation



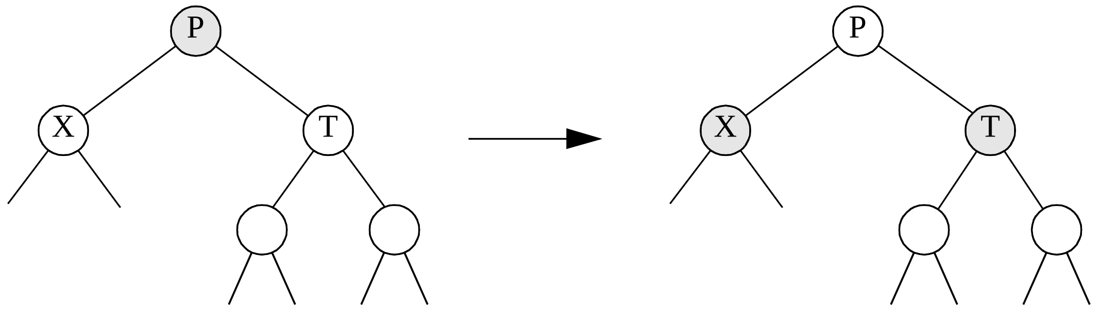
Color flip at 50 induces a violation; because it is outside, a single rotation fixes it



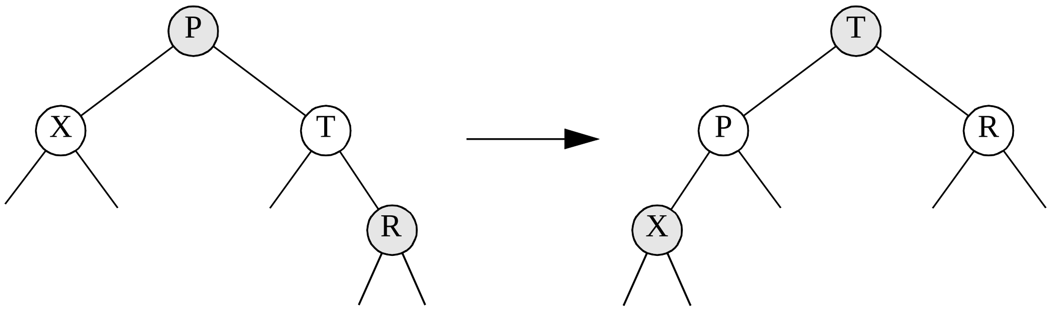
Result of single rotation that fixes violation at node 50



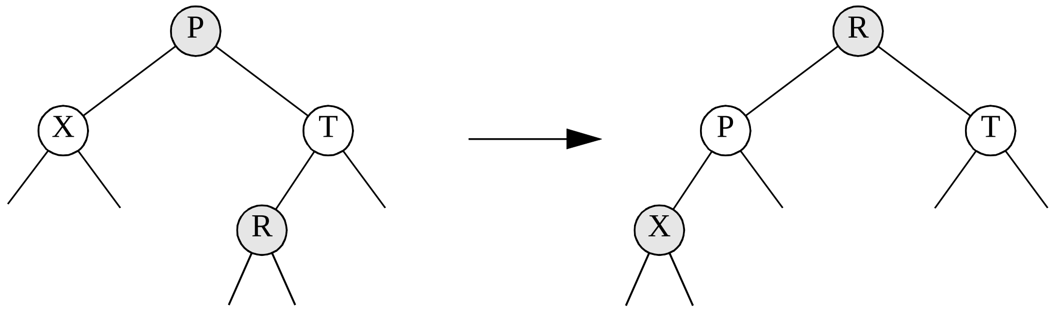
Insertion of 45 as a red node



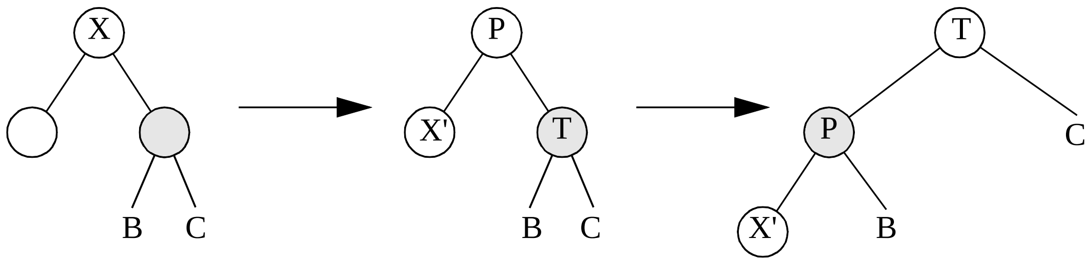
Deletion: *X* has two black children, and both of its sibling’s children are black; do a color flip



Deletion: *X* has two black children, and the outer child of its sibling is red; do a single rotation



Deletion: *X* has two black children, and the inner child of its sibling is red; do a double rotation



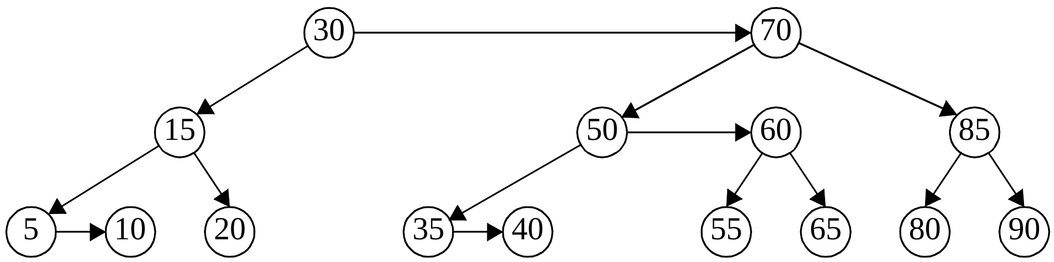
X is black and at least one child is red; if we fall through to next level and land on a red child, everything is good; if not, we rotate a sibling and parent

The level of a node is

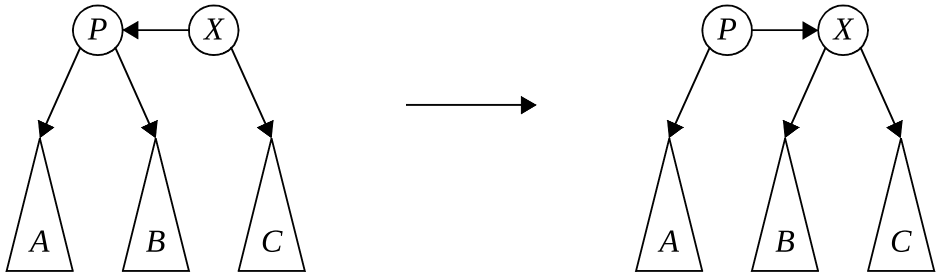
- One if the node is a leaf
- The level of its parent, if the node is red
- One less than the level of its parent, if the node is black

1. Horizontal links are right pointers (because only right children may be red).
2. There may not be two consecutive horizontal links (because there cannot be consecutive red nodes).
3. Nodes at level 2 or higher must have two children.
4. If a node does not have a right horizontal link, then its two children are at the same level.

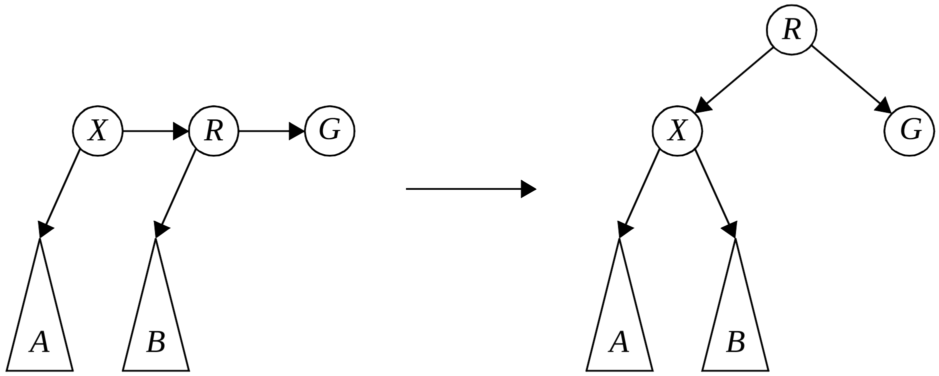
AA-tree properties



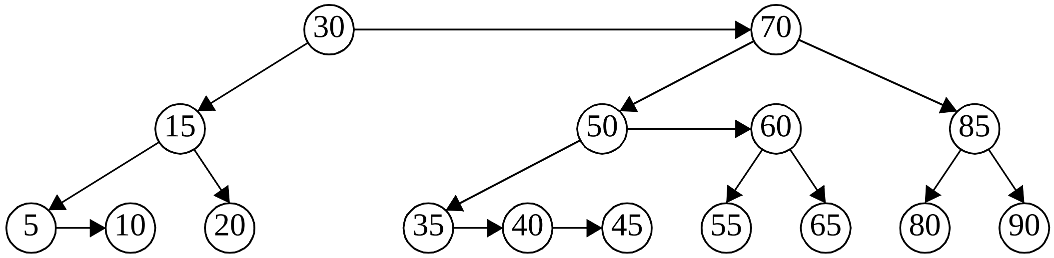
AA-tree resulting from insertion of 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55, 35



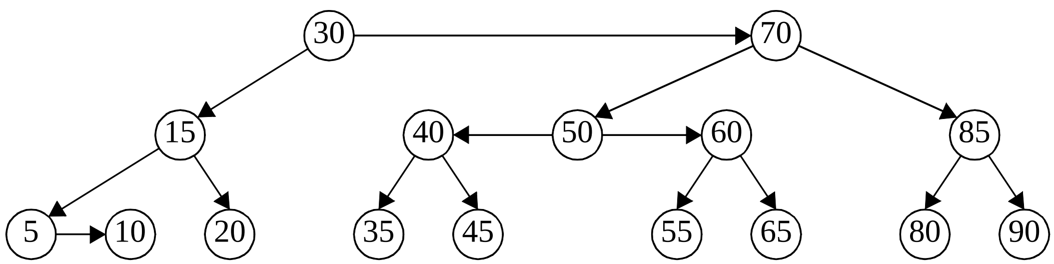
`skew` is a simple rotation between *X* and *P*



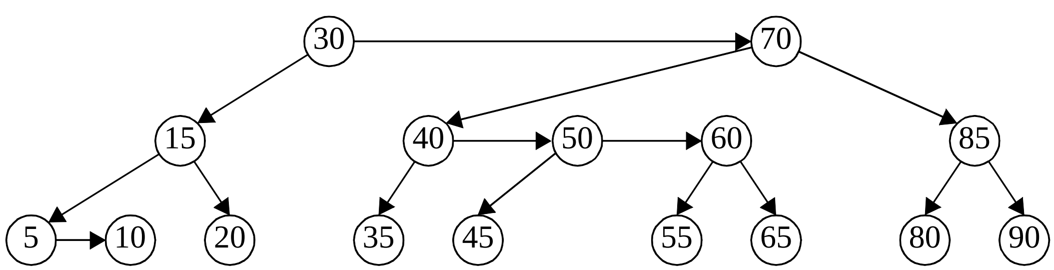
`split` is a simple rotation between *X* and *R*; note that *R*'s level increases



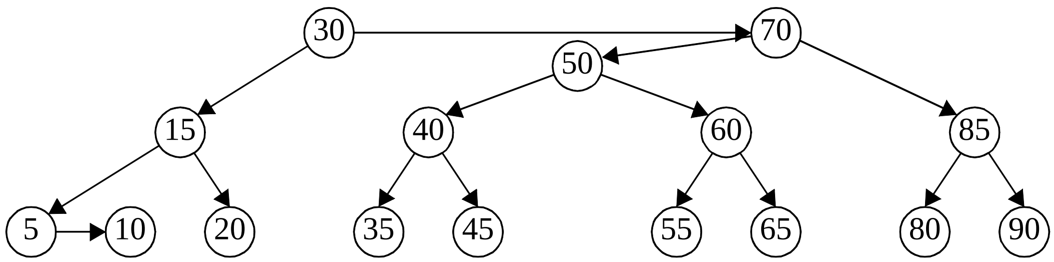
After inserting 45 into sample tree; consecutive horizontal links are introduced starting at 35



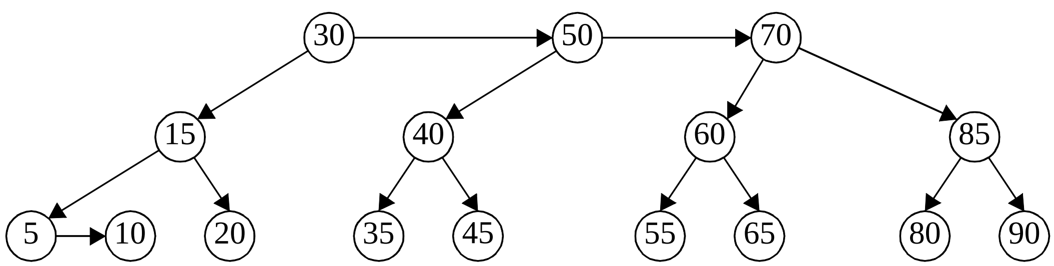
After `split` at 35; introduces a left horizontal link at 50



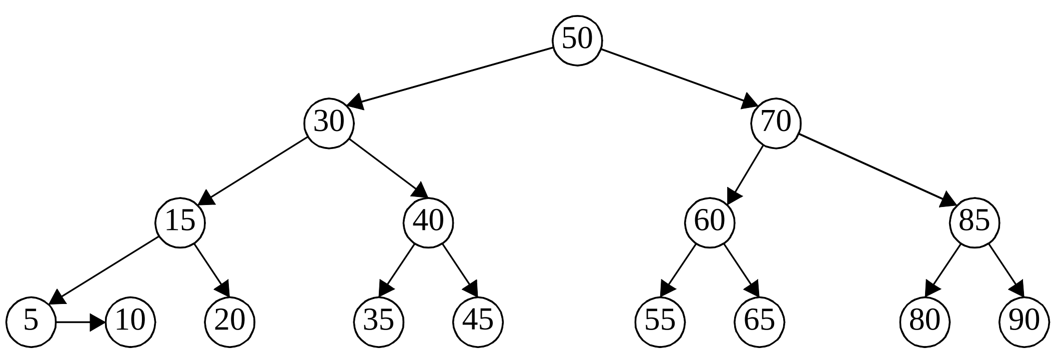
After `skew` at 50; introduces consecutive horizontal nodes starting at 40



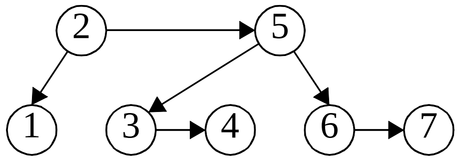
After `split` at 40; 50 is now on the same level as 70, thus inducing an illegal left horizontal link



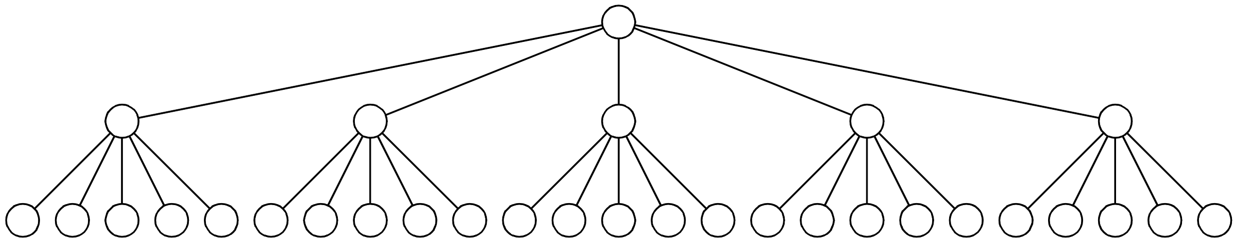
After `skew` at 70; this introduces consecutive horizontal links at 30



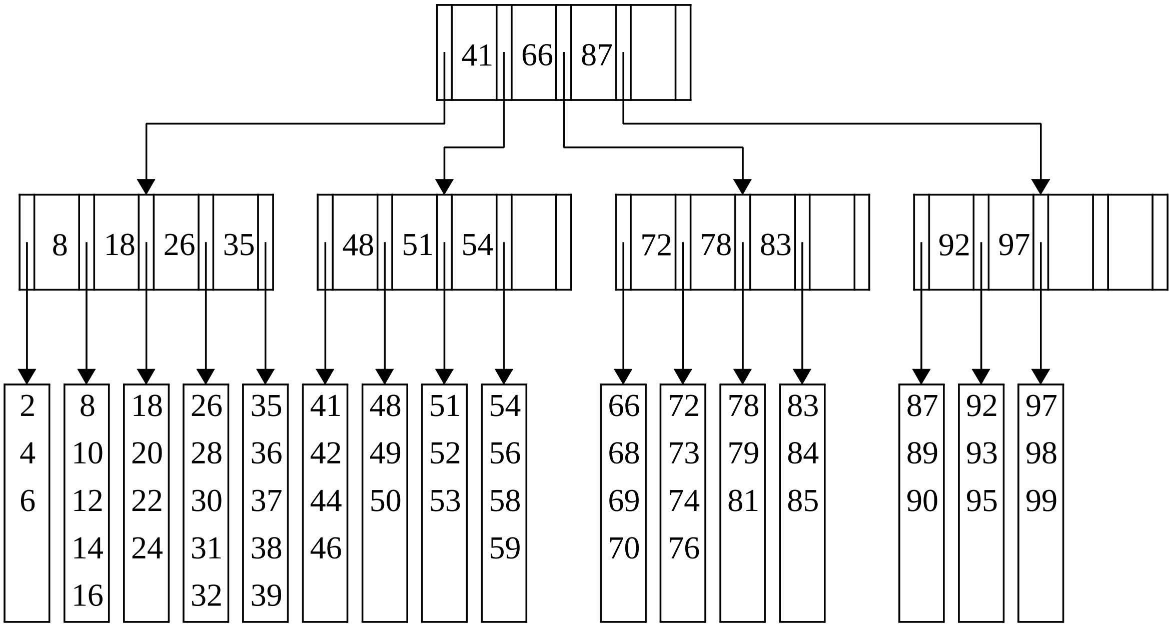
After `split` at 30; insertion is complete



When 1 is deleted, all nodes become level 1, introducing horizontal left links



Five-ary tree of 31 nodes has only three levels

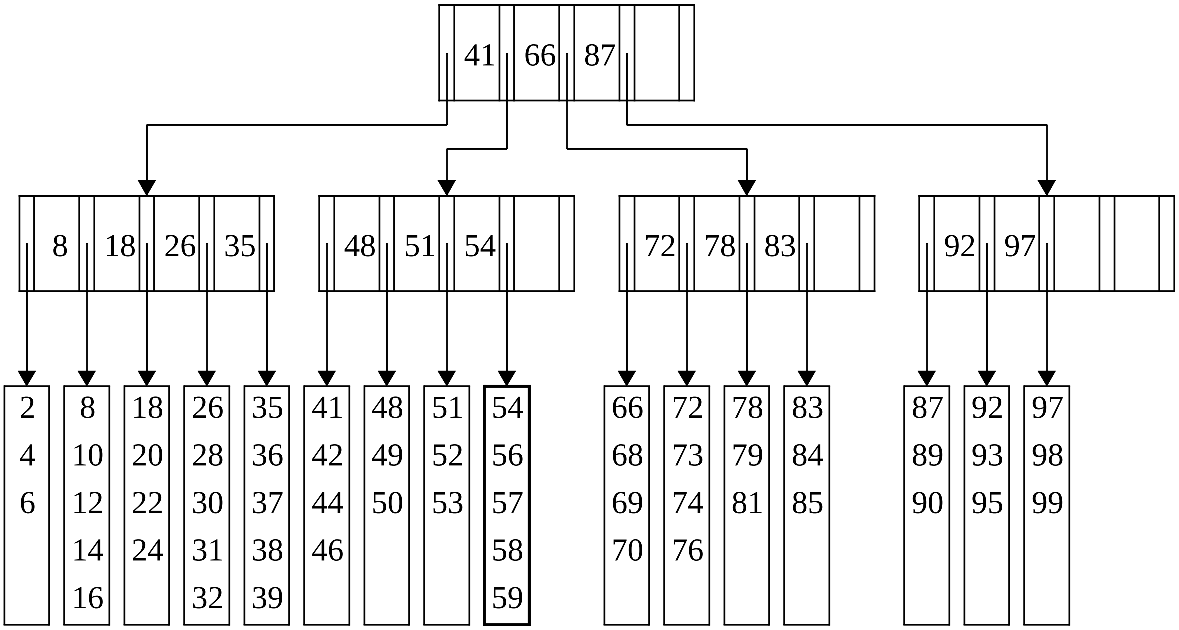


B-tree of order 5

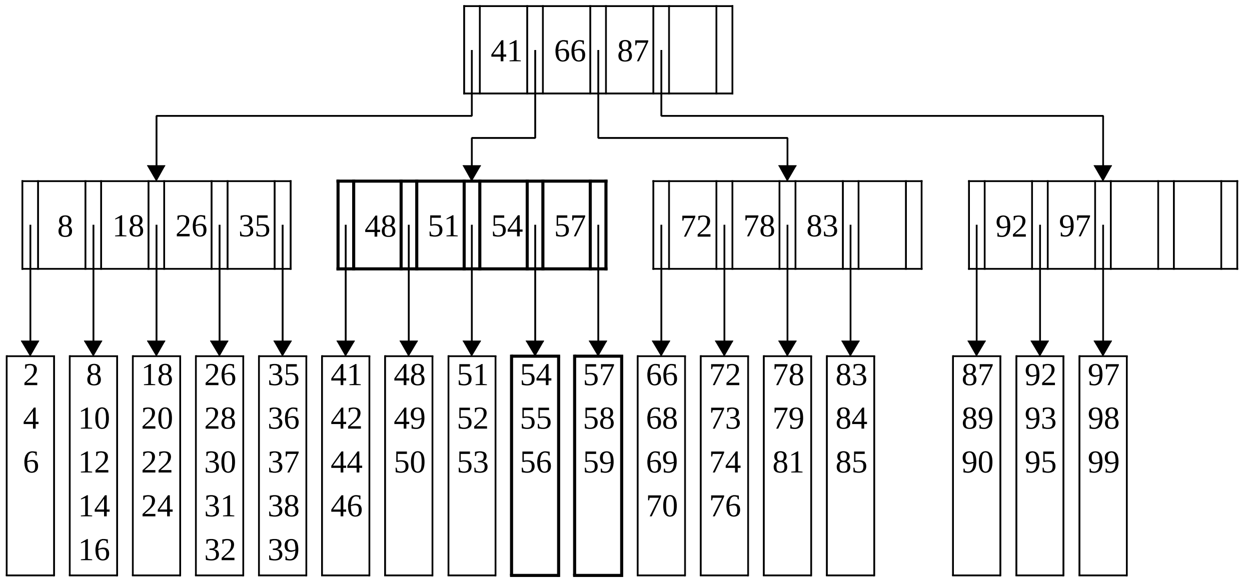
A B-tree of order M is an M-ary tree with the following properties:

1. The data items are stored at leaves.
2. The nonleaf nodes store up to $M - 1$ keys to guide the searching; key i represents the smallest key in subtree $i + 1$.
3. The root is either a leaf or has between 2 and M children.
4. All nonleaf nodes (except the root) have between $\lceil M/2 \rceil$ and M children.
5. All leaves are at the same depth and have between $\lceil L/2 \rceil$ and L children, for some L.

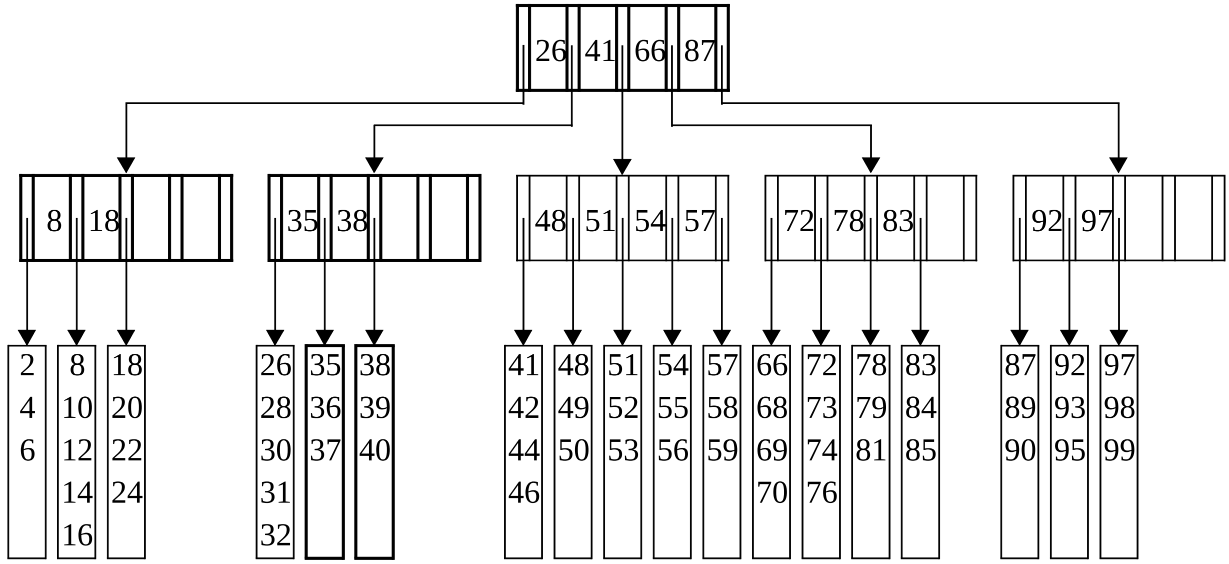
B-tree properties



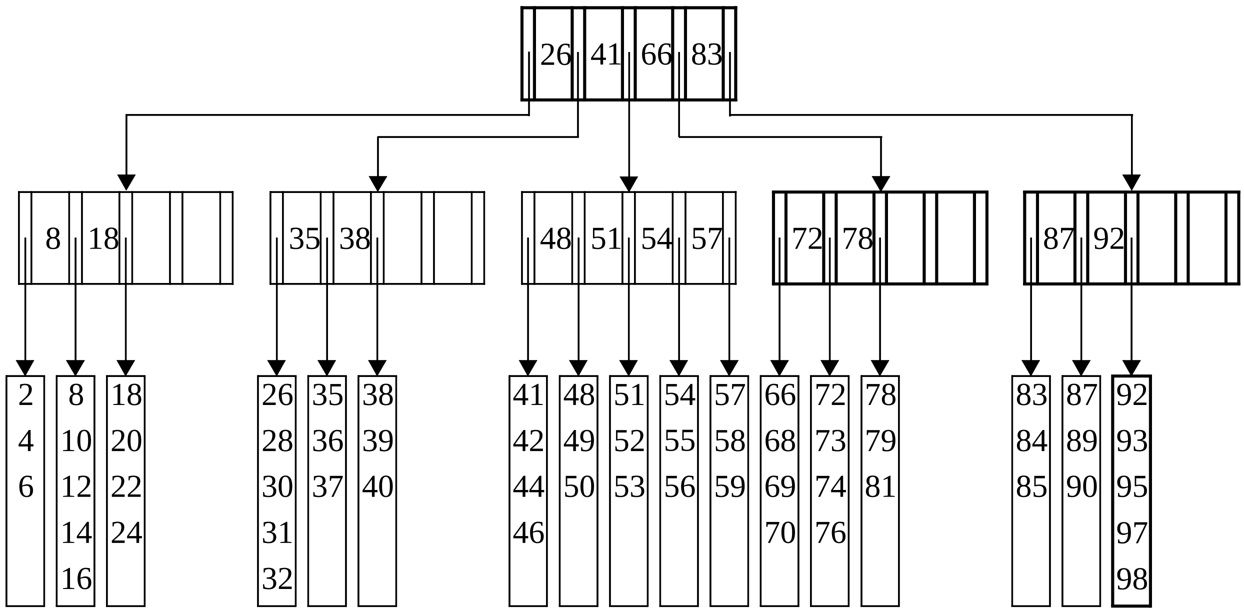
B-tree after insertion of 57 into tree in Figure 18.70



Insertion of 55 in B-tree in Figure 18.71 causes a split into two leaves



Insertion of 40 in B-tree in Figure 18.72 causes a split into two leaves and then a split of the parent node



B-tree after deletion of 99 from Figure 18.73