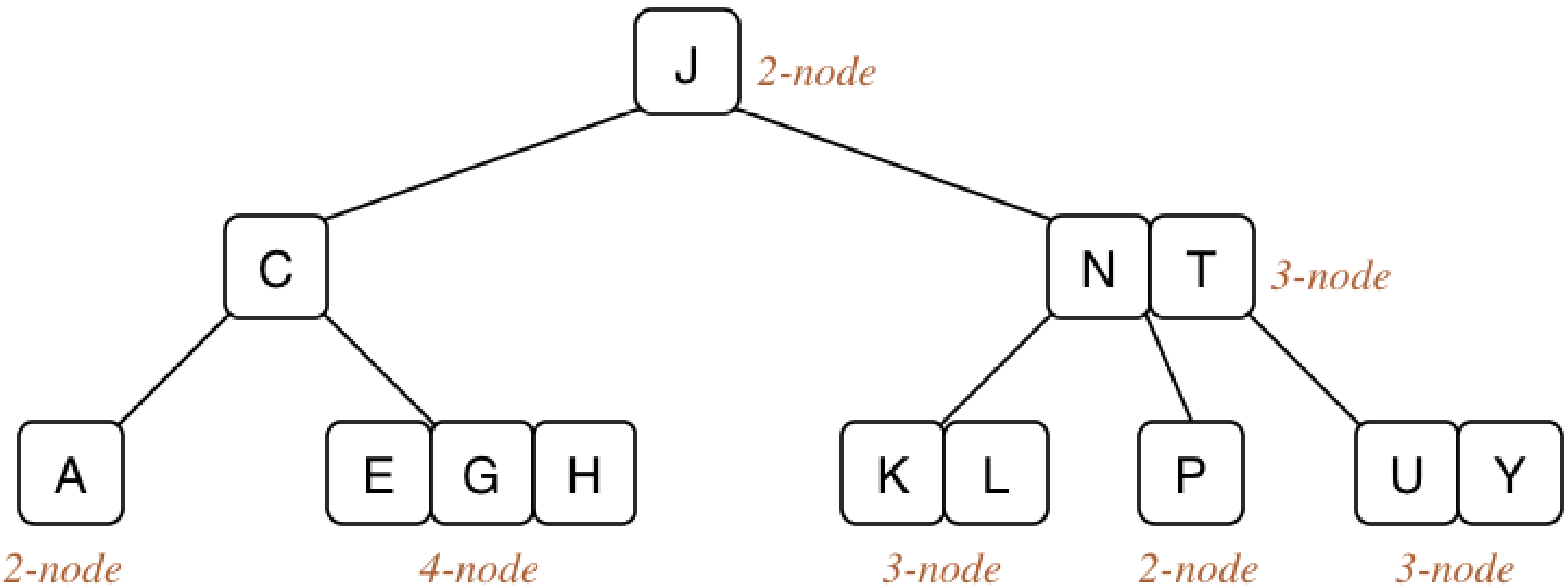
Balanced Trees

- COMP1927 Computing 17x1
- Sedgewick Chapters 13

2-3-4 Trees

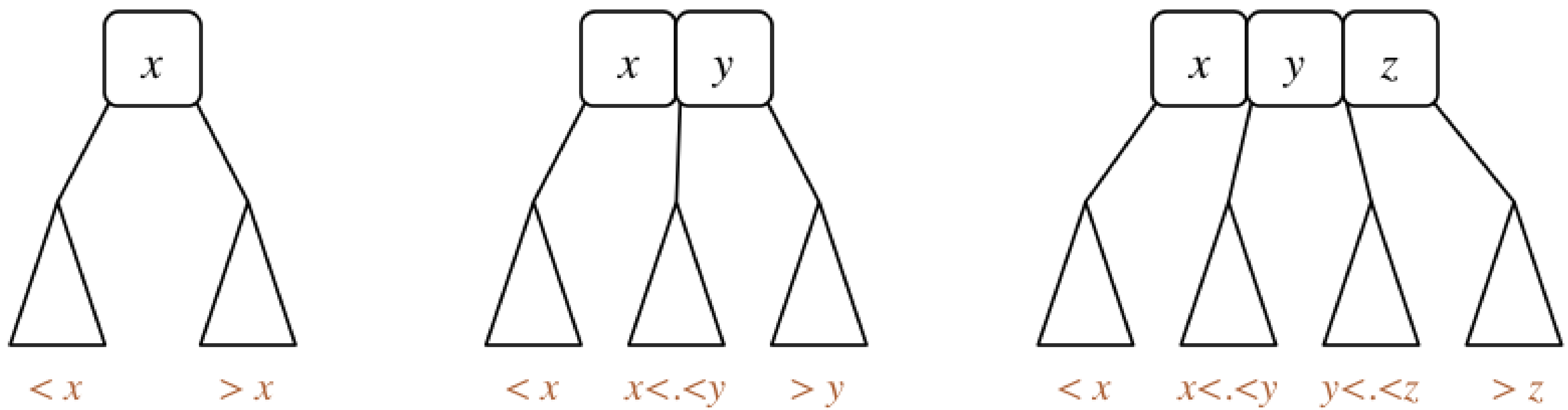
2-3-4 trees allow three kinds of nodes

- 2-nodes , one value and 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 similar to BSTs

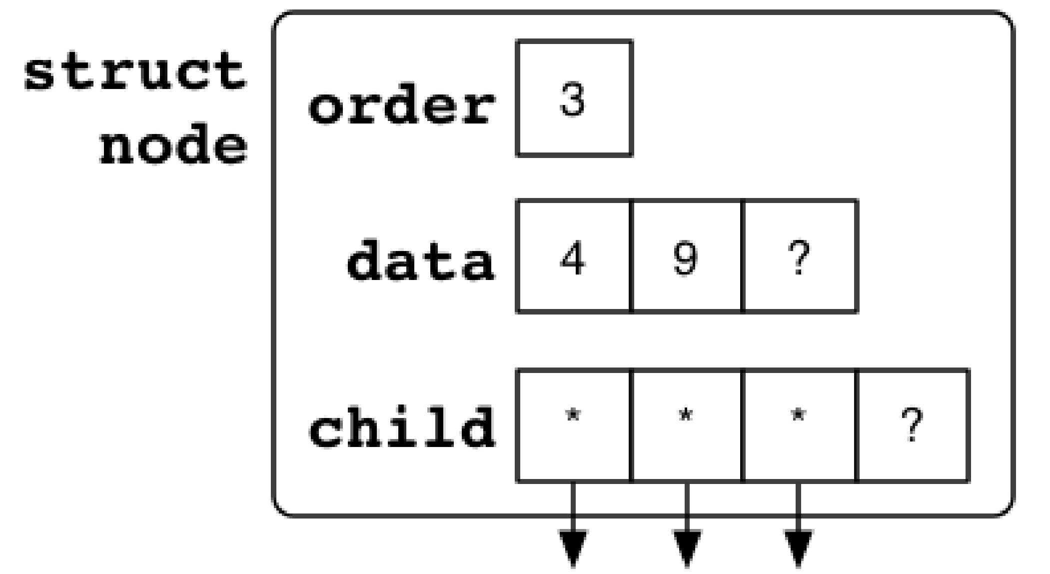


- generalise node to allow multiple keys; keep tree balanced
- each node contains $1 \leq n \leq 3$ Items and $n+1$ subtrees
- new leaves inserted at leaves; in a balanced 2-3-4 tree, all leaves are at same distance from root
- 2-3-4 trees grow "upwards" from the leaves via split-promote

2-3-4 Trees

2-3-4 trees implementation

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



Make a new 2-3-4 node (always order 2):

```
Node *newNode (Item it) {
    Node *new = malloc(sizeof(Node));
    assert(new != NULL); new->order = 2;
    new->data[0] = it;
    new->child[0] = new->child[1] = NULL;
    return new;
};
```

2-3-4 TREES

Searching in 2-3-4 trees:

- compare search key against keys in node
- find interval containing search key
- follow associated line (recursively)

```
Item *search(Tree t, Key k) {
         if (t == NULL) return NULL;
        int i; int diff; int nitems = t->order-1;
                // find relevant slot in items
        for (i = 0; i < nitems; i++) {
            diff = cmp(k, key(t->data[i]));
            if (diff <= 0) break;
         }
          if (diff == 0) {
              // match; return result;
              return &(t-> data[i]);
          else {
               // keep looking in relevant subtree
               return search(t-> child[i], k);
          };
   }
```

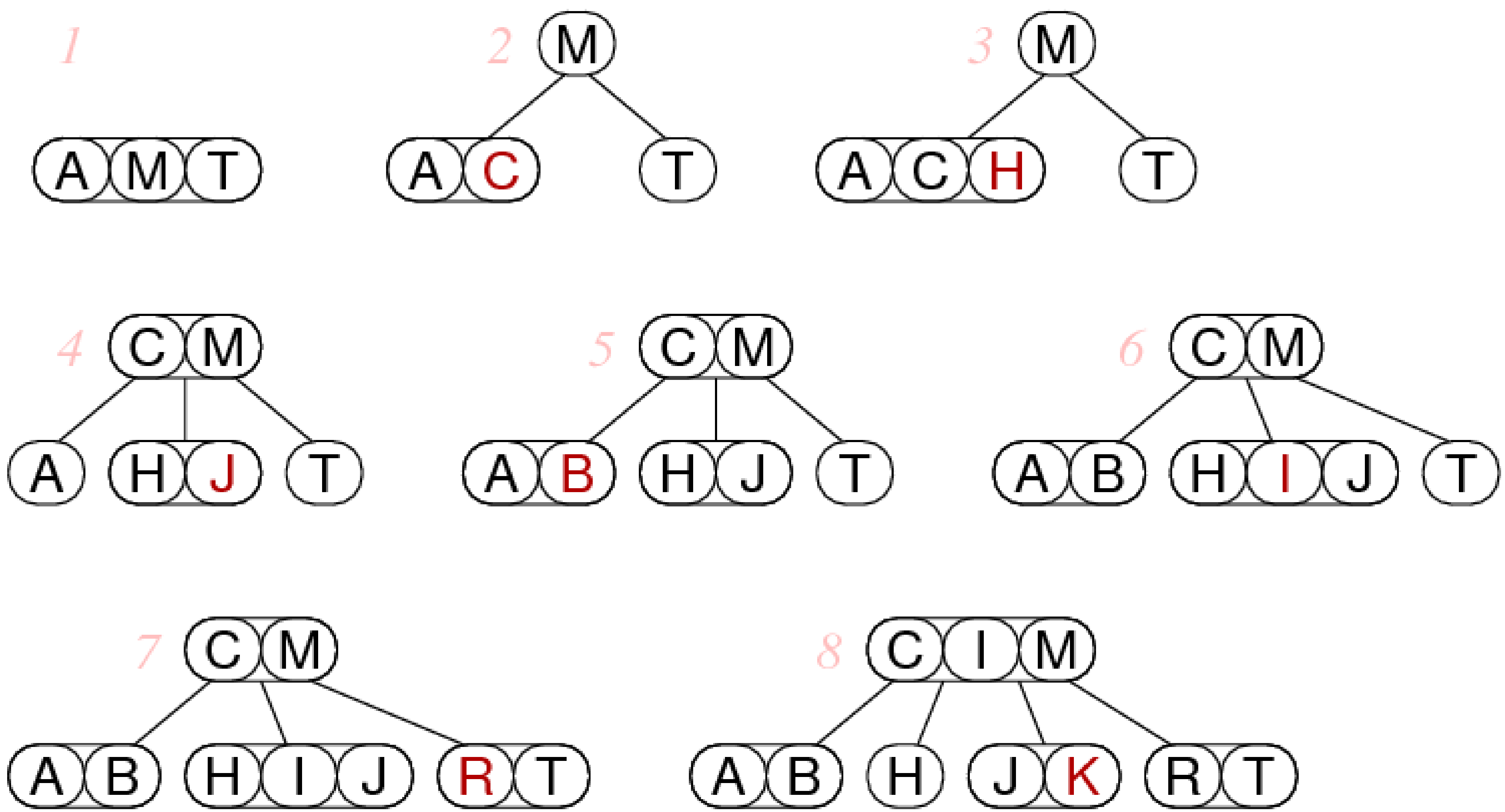
2-3-4 Trees (Cont…)

2-3-4 tree searching cost analysis

- as for other trees, worst case determined by depth d
- 2-3-4 trees are always balanced => depth is $O\ log\ (N)$
- worst case for depth: all nodes are 2-nodes
  same case as for balanced BSTs, i.e. $d \cong log_2 N$
- best case for depth: all nodes are 4-nodes
  balanced tree with branching factor 4, i.e. $d \cong log_4 N$

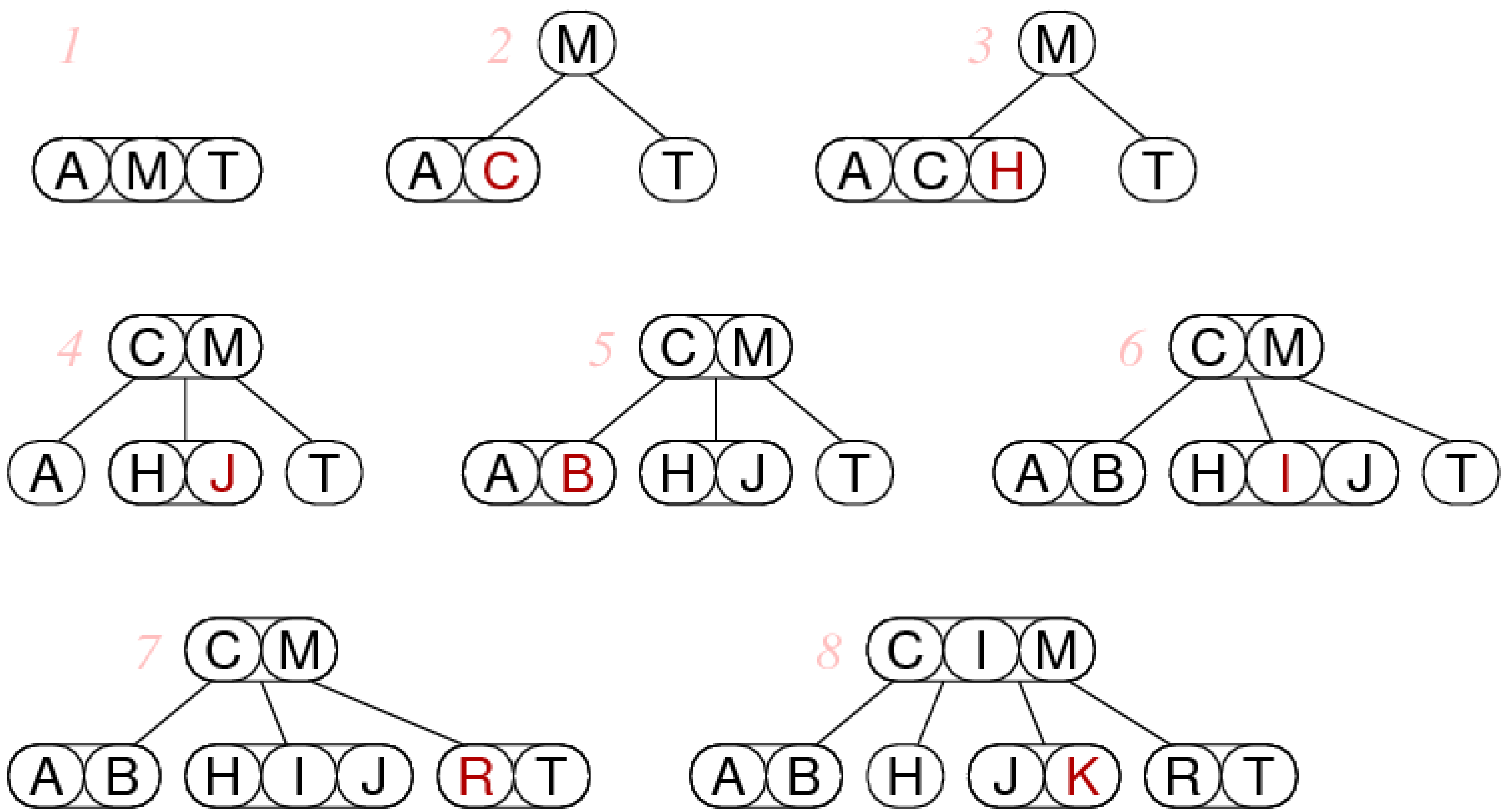
BUILDING A 2-3-4 TREE … 7 INSERTIONS

- To insert, first search for a leaf node in which to put the key
- May need to split a node e.g, insert C
    - when inserting a key into a 4-node, the 4-node splits and a key moves up to the parent node.
    - new key may in turn cause the parent to split, moving a key up to the grandparent, and so on up to the root.



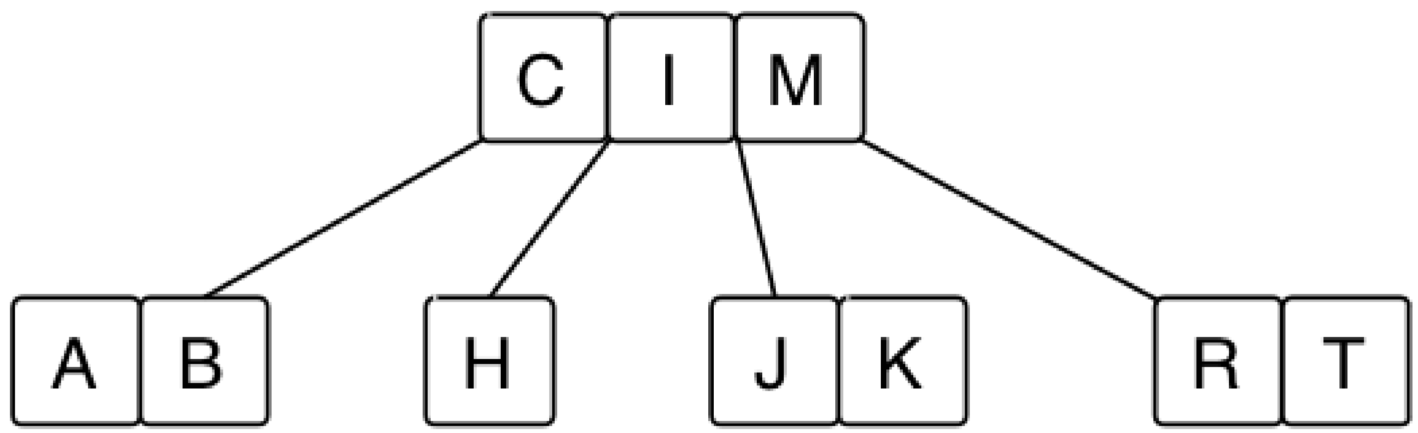
BUILDING A 2-3-4 TREE … 7 INSERTIONS

- To insert, first search for a leaf node in which to put the key
- May need to split a node e.g, insert C
    - when inserting a key into a 4-node, the 4-node splits and a key moves up to the parent node.
    - new key may in turn cause the parent to split, moving a key up to the grandparent, and so on up to the root.



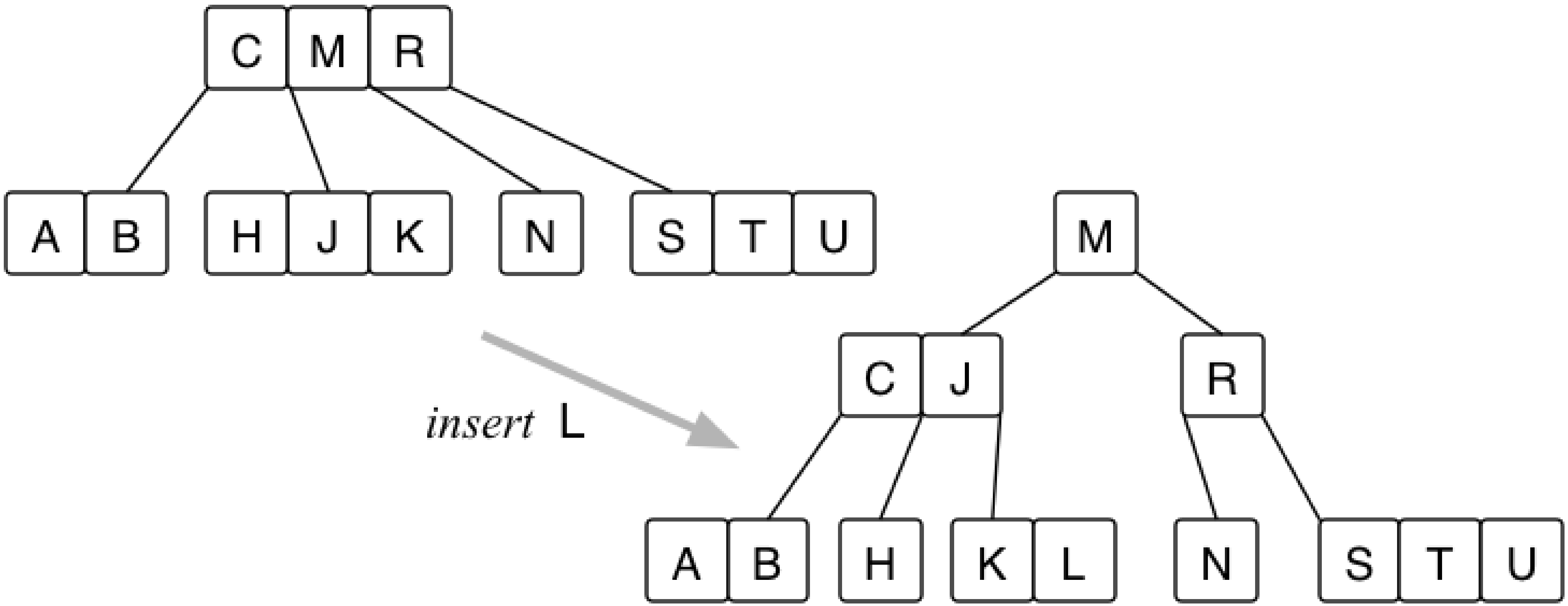
INSERTION INTO A 2-3-4 TREE

- Show what happens when D, S, F, U are inserted into this tree



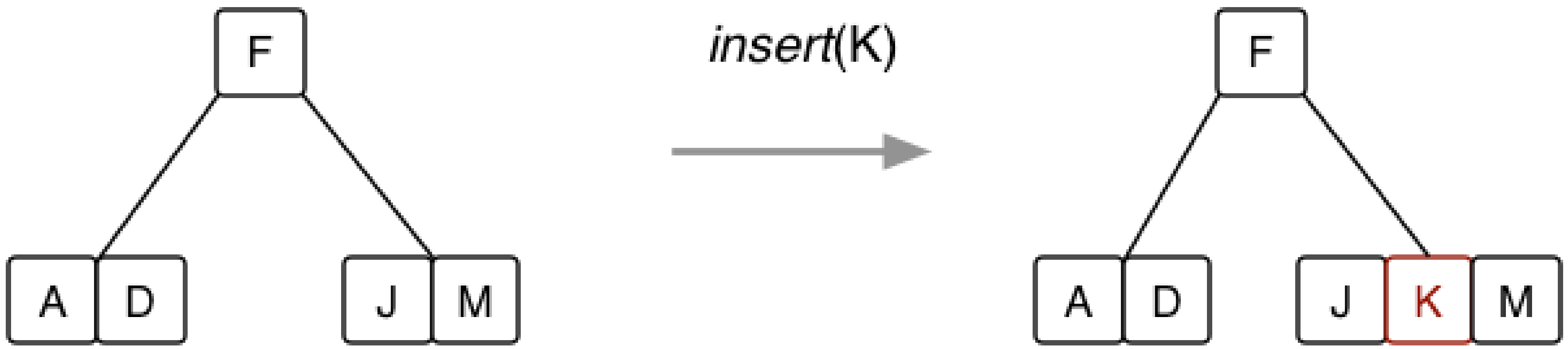
INSERTION INTO A 2-3-4 TREE

- More examples of 2-3-4 insertions:

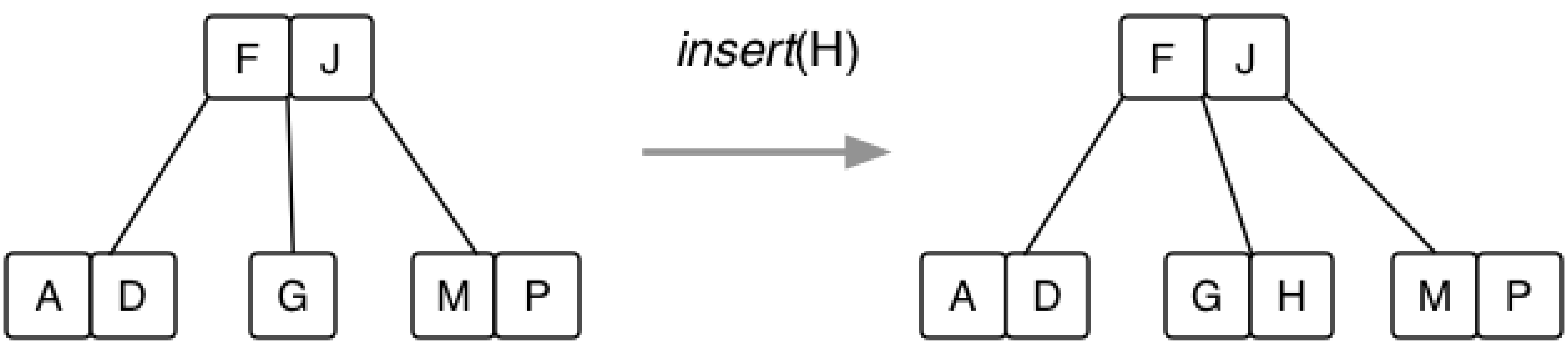


MORE EXAMPLES OF 2-3-4 INSERTIONS

- Insertion into a 2-node:

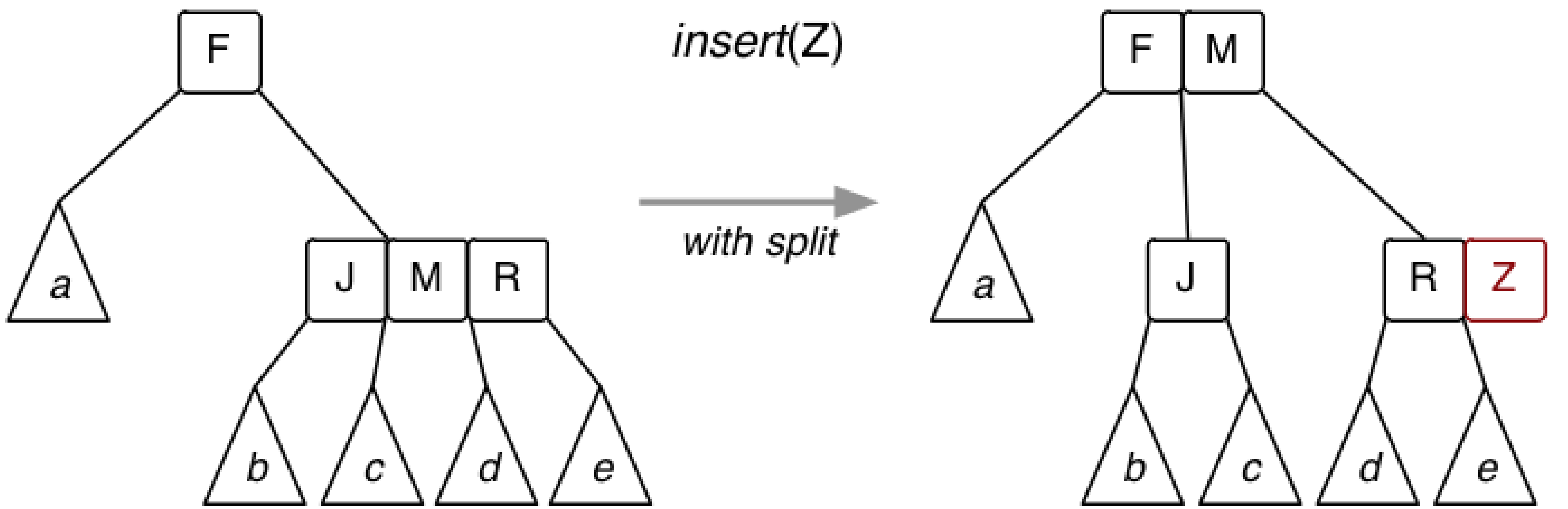


- Insertion into a 3-node:

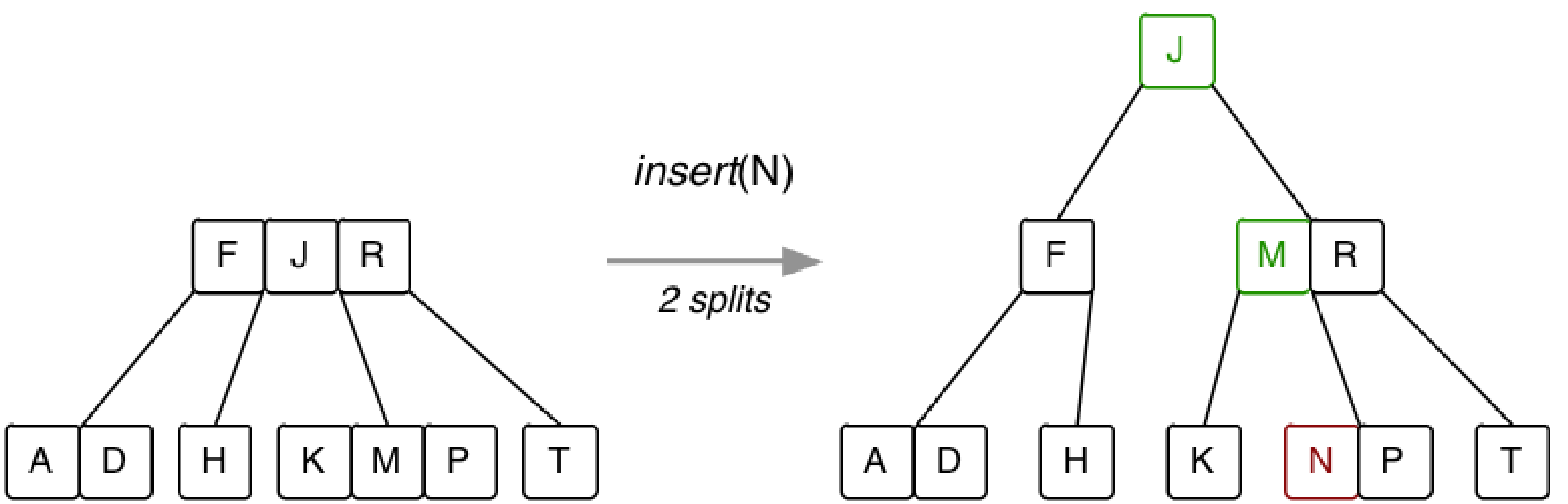


More examples Of 2-3-4 insertions

- Insertion into a 4-node – requires a split



- Splitting the root



2-3-4 Insert

Insertion Algorithm

```
insert(Tree, Item) {
     Node = search(Tree, key(Item)
     Parent = parent of Node
     if (order(Node) < 4)
               insert Item in Node, order++
     else {
               promote = Node.data[1] // middle value
               NodeL = new Node containing data[0]
               NodeR = new Node containing data[2]
                if (key(Item) < key(data[1]))
                    insert Item in NodeL
                else
                    insert Item in NodeR
                 insert promote into Parent
                 while (order(Parent) == 4)
                        continue promote/split upwards
                 if (isRoot(Parent) && order(Parent) == 4)
                               split root, making new root
             }
}
```

2-3-4 Insert

Following a chain of splits up to root

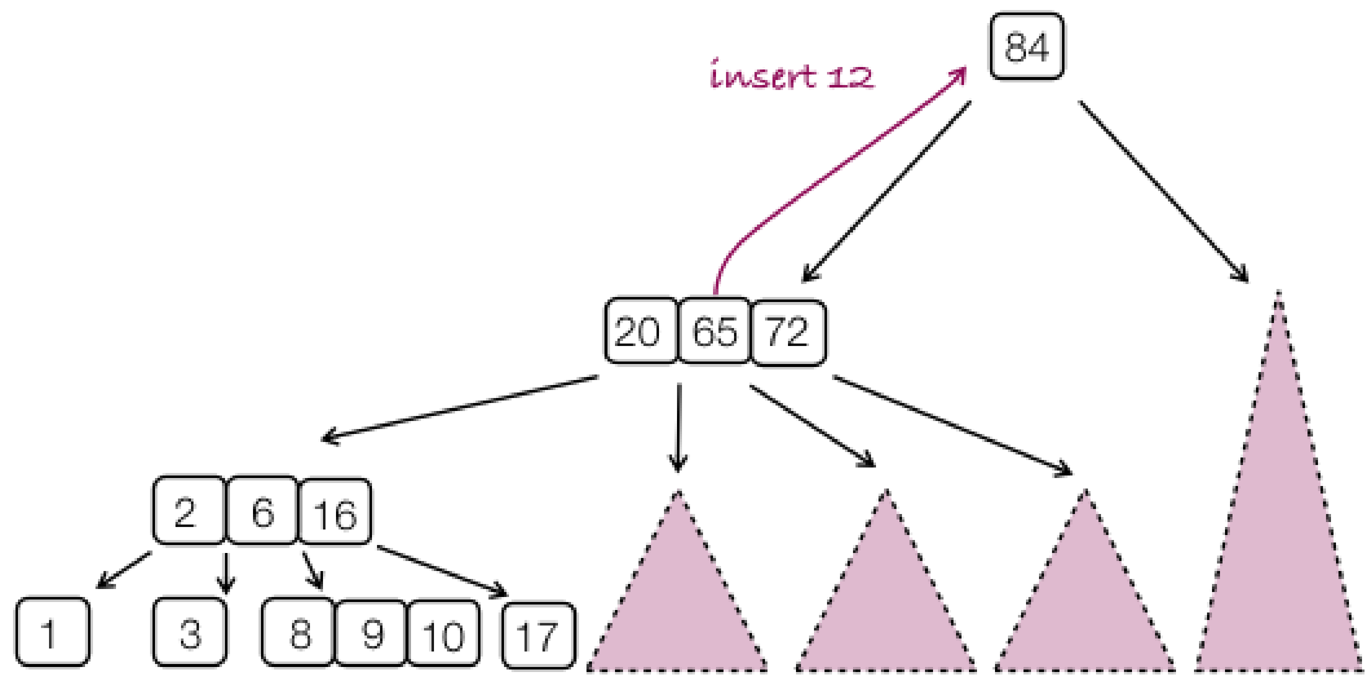
- starting from insertion into a leaf 4-node
- is not necessarily the best approach to insertion

Alternative approach:

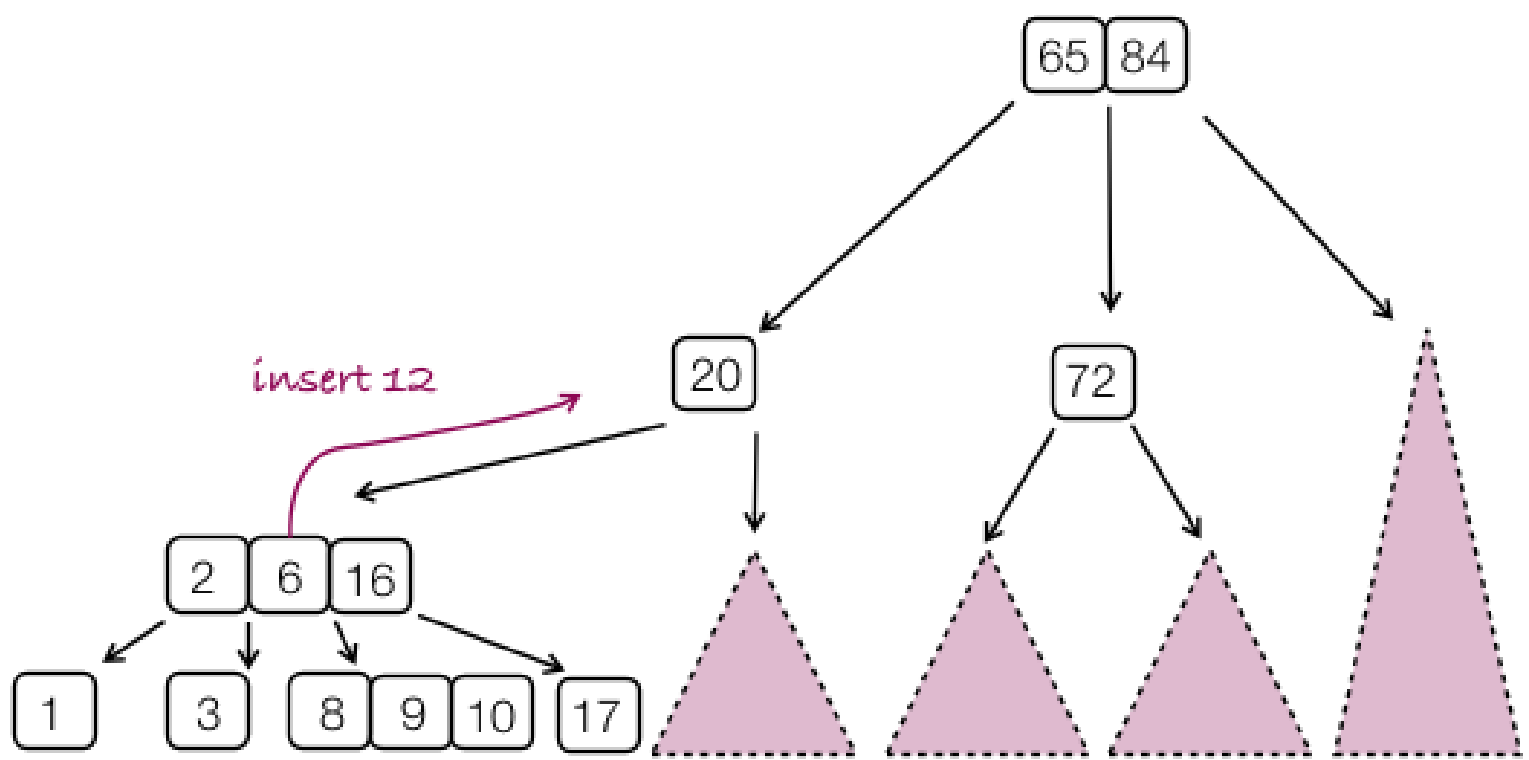
- split 4-nodes attached to 2- or 3-nodes while we descend tree to leaf node to insert
- guaranteed that split of leaf propagates up only 1 level

2-3-4 Insert

Top-Down Splitting strategy (part 1):

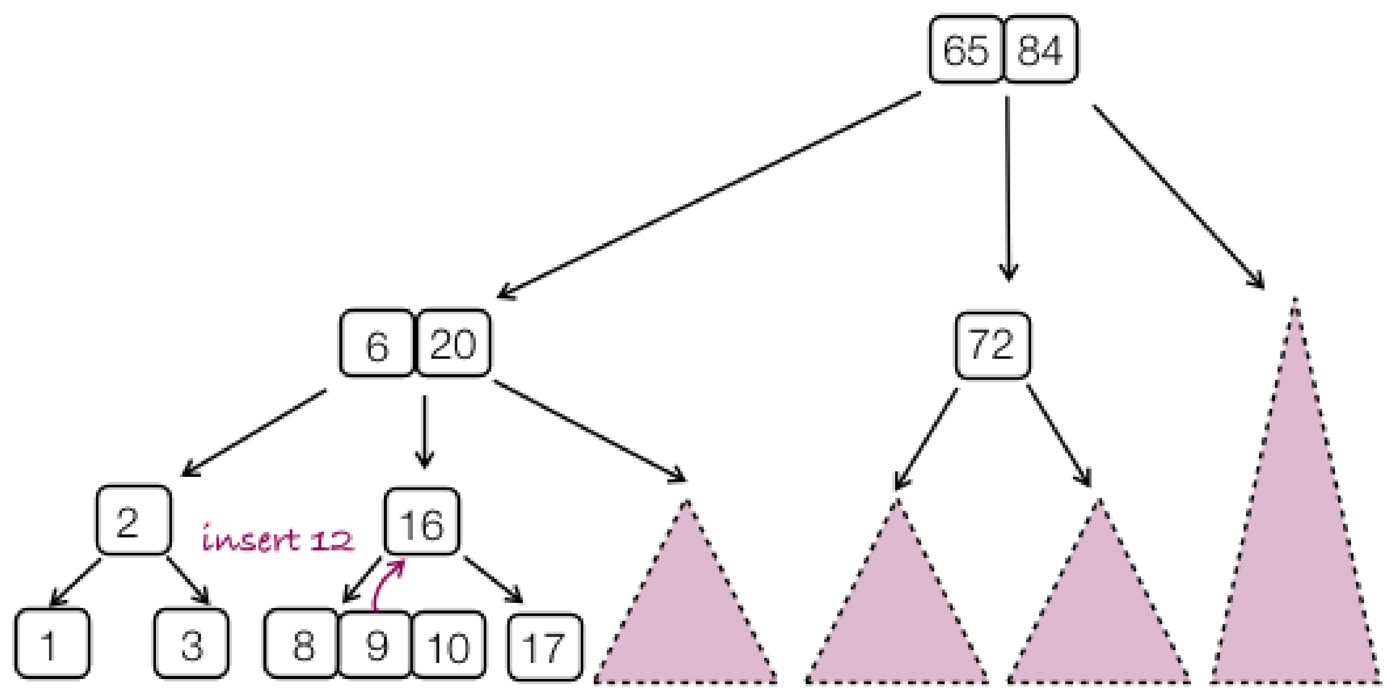


Top-Down Splitting strategy (part 2):

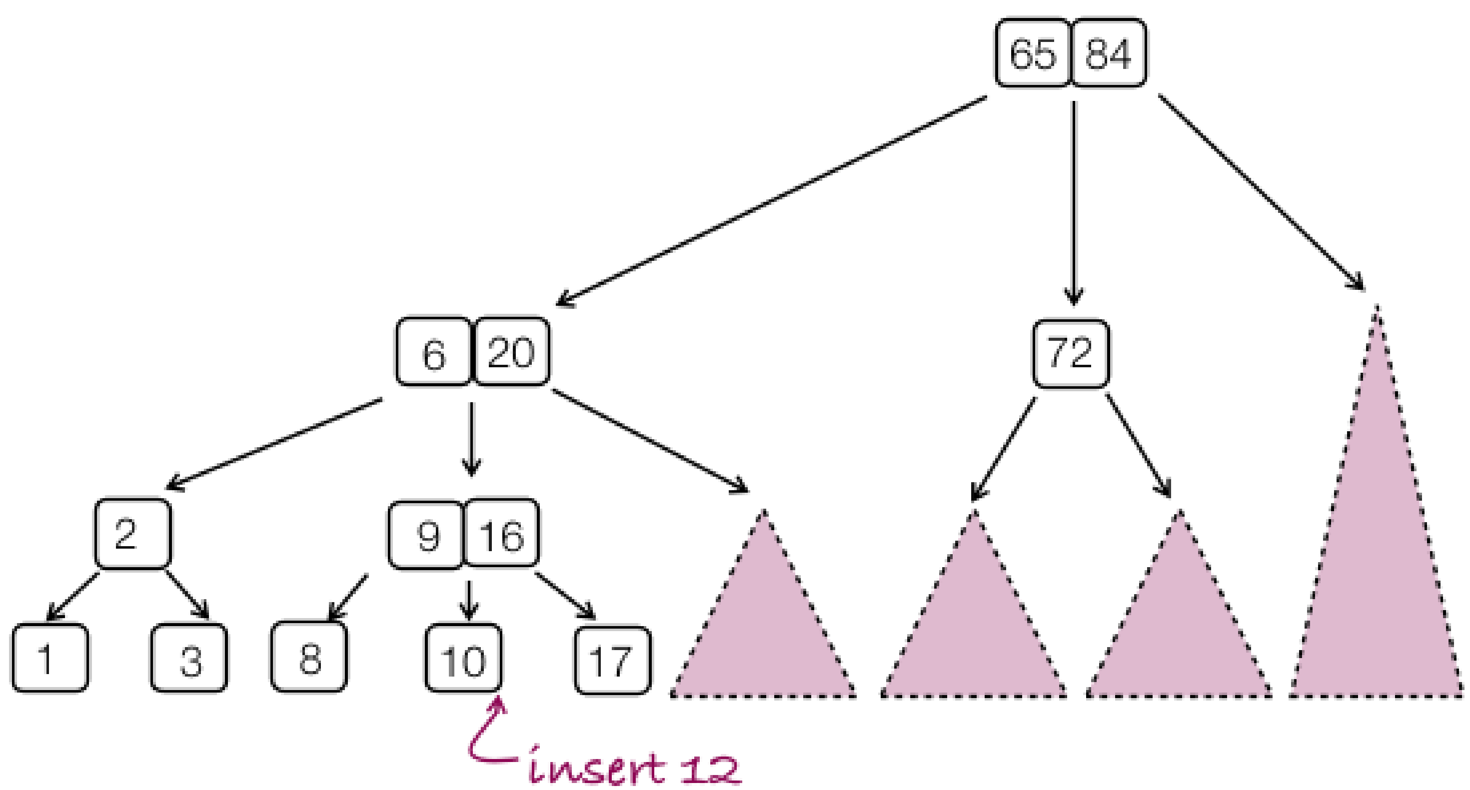


2-3-4 Insert

Top-Down Splitting strategy (part 3):



Top-Down Splitting strategy (part 4):



2-3-4 Tree performance

Insertion (into tree of depth d) = $O(d)$ comparisons

- multiple comparisons in each of d 2-3-4 nodes
- along with occasional splitting to shift values between nodes

Search (in tree of depth d) = $O(d)$ comparisons

- multiple comparisons in each of d 2-3-4 nodes

Depth of 2-3-4 tree with N nodes = $log_4 N < d < log_2 N$

Note that all paths in a 2-3-4 tree have same length d

2-3-4 Tree Variations

Variation #1: why stop at 4? why not 2-3-4-5 trees? or *M*-way trees?

- allow nodes to hold up to *M-1* items, and at least *M/2*
- if each node is a disk-page, then we have a B-tree (databases)
- for B-trees, depending on Item size, *M > 100/200/400*

Variation #2: Variation #2: don't have "variable-sized" nodes

- use standard BST nodes, augmented with one extra piece of data
- implement similar strategy as 2-3-4 trees → red-black trees.

Red-Black Trees

Red-Black trees are a representation of 2-3-4 trees using BST nodes

A red-black tree is defined as:

- a BST in which each node is marked red or black
- no two red nodes appear consecutively on any path
- a red node corresponds to a 2-3-4 sibling of its parent
- a black node corresponds to a 2-3-4 child of its parent

Insertion algorithm:

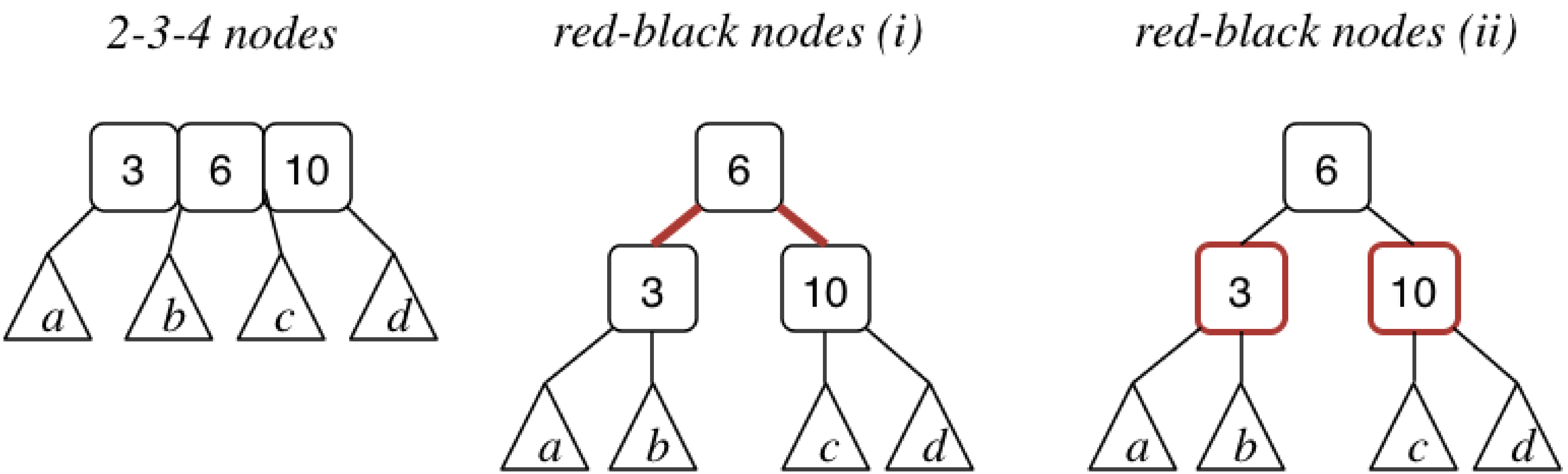
- avoids worst case $O(n)$ behaviour

Search algorithm:

- standard BST search

Red-Black Trees

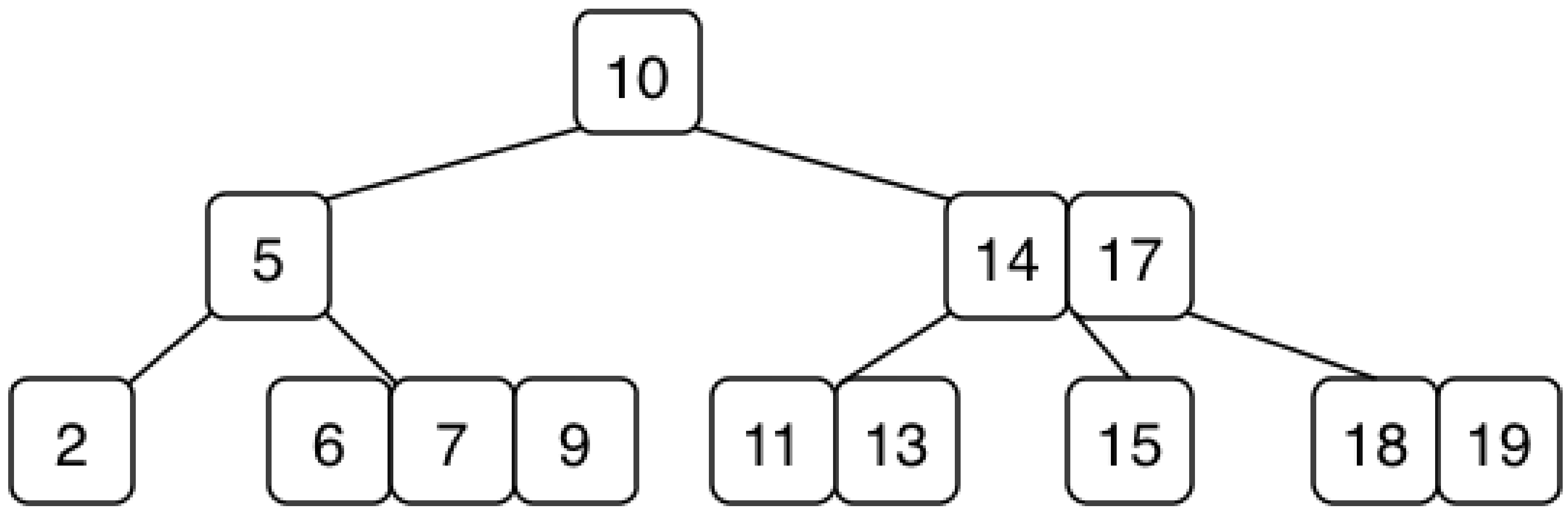
Representing 4-nodes in red-black trees:

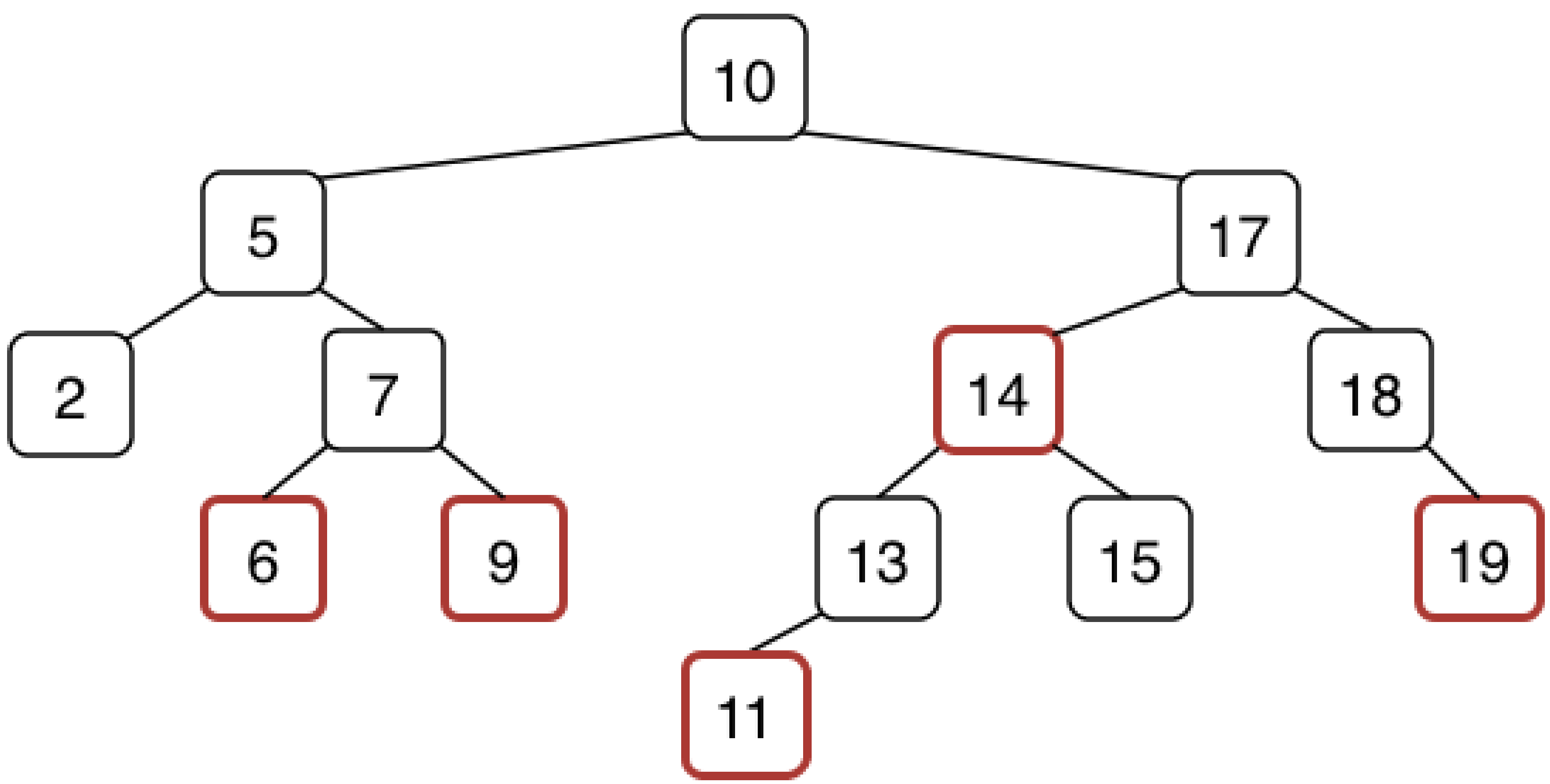


Note: some texts colour the links rather than the nodes

Red-Black Trees

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





Red-Black Trees

Red-black tree implementation:

```
typedef enum {RED,BLACK} Colr;
typedef struct Node *Link;
typedef struct Node *Tree;
typedef struct Node {
        Item data; // actual data
        Colr colour; // relationship to parent
        Link left; // left subtree
        Link right; // right subtree
} Node;
```

`RED` = node is part of the same 2-3-4 node as its parent (sibling)
`BLACK` = node is a child of the 2-3-4 node containing the parent

Red-Black Trees

Making new nodes requires a colour:

```
Node *newNode(Item it, Colr c) {
	Node *new = malloc(sizeof(Node));
	assert(new != NULL);
	new->data = it;
	new->colour = c;
	new->left = new->right = NULL;
	return new;
}
```

RED = node is part of the same 2-3-4 node as its parent (sibling)
BLACK = node is a child of the 2-3-4 node containing the parent

Red-Black Trees

Searching method is standard BST search:

```
Item *search(Tree t, Key k) {
    if (t == NULL) return NULL;
    int diff = cmp(k, key(t->data));
    if (diff < 0)
        return search(t->left, k);
    else if (diff > 0)
        return search(t->right, k);
    else // matches
        return &(t->data);
}
```

Red-Black Tree Insertion

Insertion is more complex than for standard BSTs

- need to recall direction of last branch (L or R)
- need to recall whether parent link is red or black
- splitting/promoting implemented by rotateL/rotateR
- several cases to consider depending on colour/direction combinations

We first consider some of the components of this algorithm.

```
#define L(t) (t)->left
#define R(t) (t)->right
#define red(t) ((t) != NULL && (t)->colour == RED)
#define blk(t) ((t) != NULL && (t)->colour == BLACK)
```

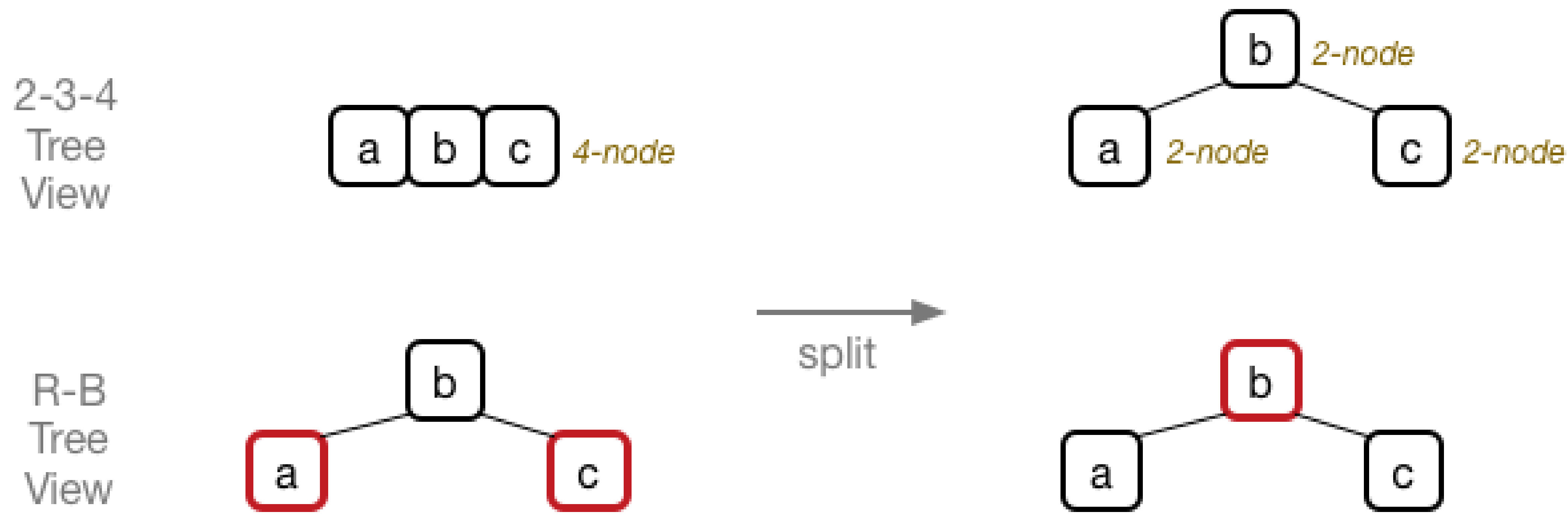
Red-Black Trees

Insertion function top-level:

```
void insertRedBlack(Tree t, Item it)
{
   t->root = insertRB(t->root, it, 0);
   t->root->colour = BLACK;
}
Link insertRB(Link t, Item it, int inRight)
{
   if (t == NULL) return newNode(it,RED);
   if (red(L(t)) && red(R(t))) {
      // split 4-node and promote middle value
      // performed as we descend tree
   }
   // recursive insert cases (cf. regular bst)
   // then re-arrange links/colours after insert
   return t';
}
```

Red-Black Trees

Splitting a 4-node, in a red-black tree:



Code:

```
if (red(L(t)) && red(R(t)) {
   t->colour = RED;
   t->left->colour = BLACK;
   t->right->colour = BLACK;
}
```

Red-Black Trees

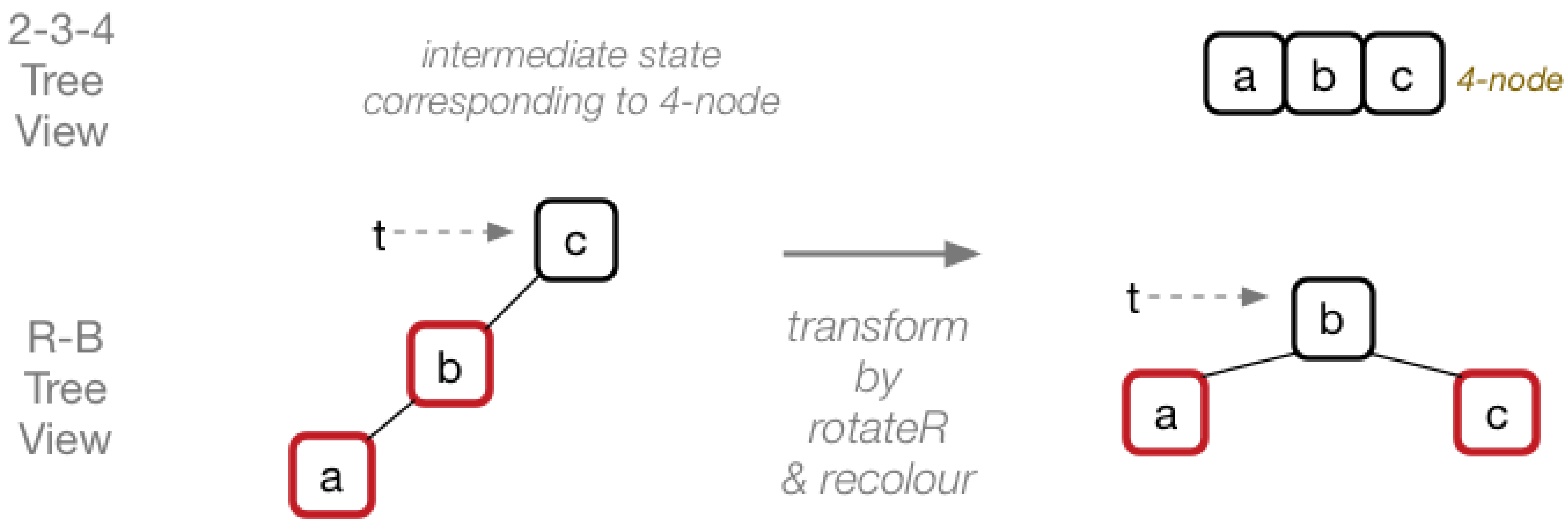
Recursive insert part (cf. bst insert):

Code:

```
if (less(key(it), key(t->item))) {
    t->left = insertRB(t->left, it, 0);
    ...
}
else { key(it) larger than key in root
    t->right = insertRB(t->right, it, 1);
    ...
}
```

Red-Black Trees

Check after insert: two successive red links = newly-created 4-node

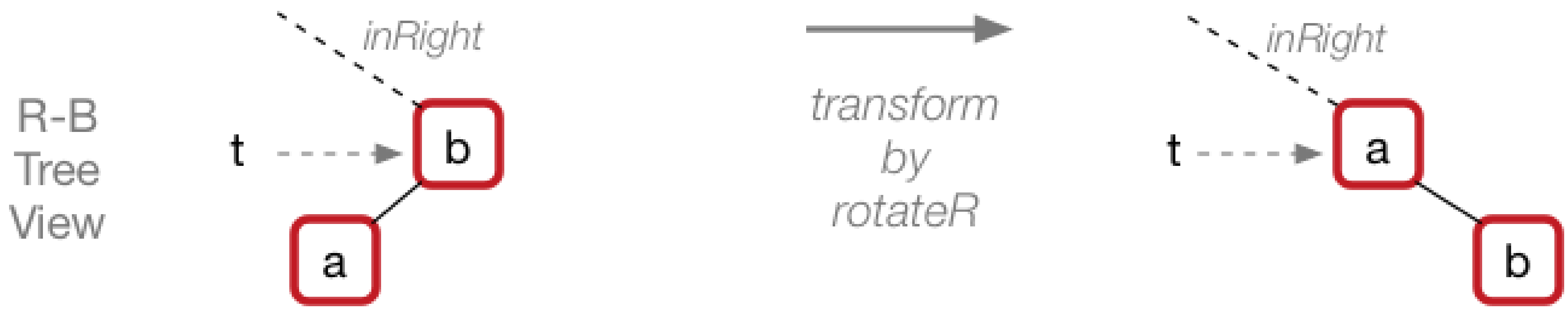


Code:

```
if (red(L(t)) && red(L(L(t)))) {
   t = rotateR(t);
   t->colour = BLACK;
   t-right->colour = RED;
}
```

Red-Black Trees

Check after insert: "normalise" direction of successive red links



Code:

```
if (red(t) && red(L(t)) && inRight) {
    t = rotateR(t);
}
```

Red-Black Trees

Full code for handling insertion into left subtree ..

Code:

```
if (less(key(it), key(t->item))) {
   L(t) = insertRB(L(t), it, 0);
   if (red(t) && red(L(t)) && inRight)
      t = rotateR(t);
   if (red(L(t)) && red(L(L(t)))
      t = rotateR(t);
      t->colour = BLACK;
      R(t)->colour = RED;
   }
}
```

Similar "mirror-image" code if inserted into right subtree

Red-Black Trees

Exercise 1: 2-3-4 vs Red-Black Insertion

Show the 2-3-4 tree resulting from the insertion of:

10 5 9 6 2 4 20 15 18 19 17 12 13 14

Compare this to the red-black tree with the same values.

Use this Algorithm Visualiser to build the red-black tree

Red-Black Trees

Add red-black trees to TreeLab

- Modify Node to include colour
- Implement insertRedBlack() and insert RB()

Compare against the Algorithm Visualiser to build the red-black tree

Red-Black Trees

- Cost analysis for red-black trees:

- tree is well-balanced; worst case search is O(log2N)
- insertion affects nodes down one path; max rotations is 2d (where d is the depth of the tree)

- Only disadvantage is complexity of insertion/deletion code.

- Note: red-black trees were popularised by Sedgewick.