Chapter 12

- Lexicographic Search Trees: Tries
- Multiway Trees
- B-Tree, B*-Tree, B^+-Tree
- Red-Black Trees (BST and B-Tree)
- 2-d Tree, k-d Tree

Basic Concepts

Definitions:

- A ***(free) tree*** is any set of points (called ***vertices***) and any set of pairs of distinct vertices (called ***edges*** or ***branches***) such that (1) there is a sequence of edges (a ***path***) from any vertex to any other, and (2) there are no ***circuits***, that is, no paths starting from a vertex and returning to the same vertex.

- A ***rooted tree*** is a tree in which one vertex, called the ***root***, is distinguished.

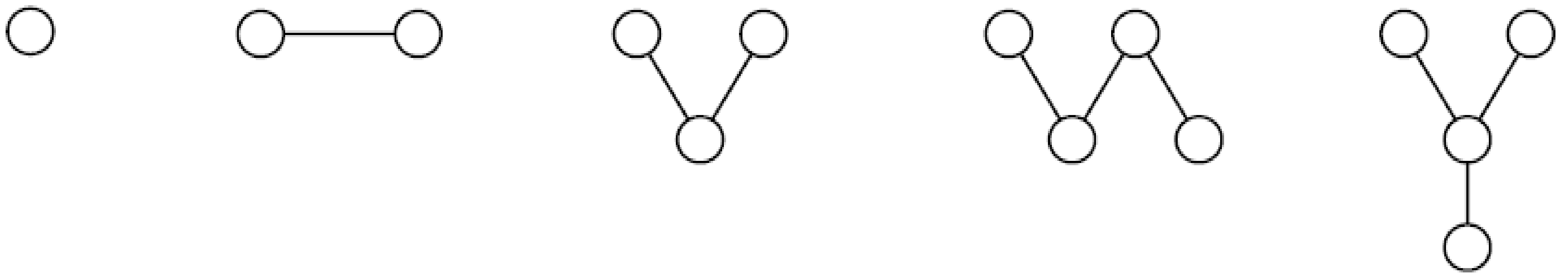
Basic Concepts

- An ***ordered tree*** is a rooted tree in which the children of each vertex are assigned an order.

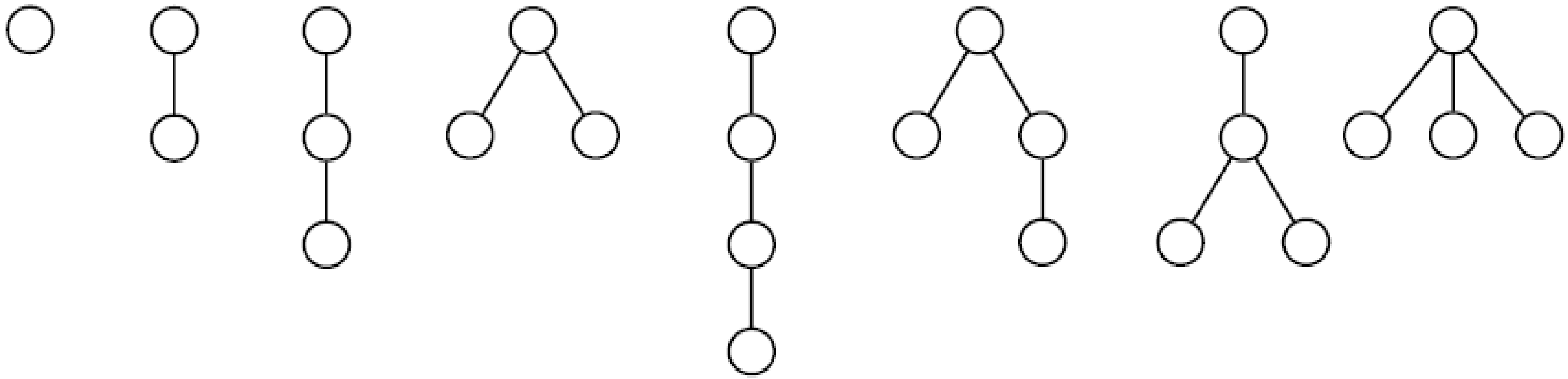
- A ***forest*** is a set of trees. We usually assume that all trees in a forest are rooted.

- An ***orchard*** (also called an ***ordered forest***) is an ordered set of ordered trees.

Trees



Free trees with four or fewer vertices
(Arrangement of vertices is irrelevant.)



Rooted trees with four or fewer vertices
(Root is at the top of tree.)

Recursive Definitions

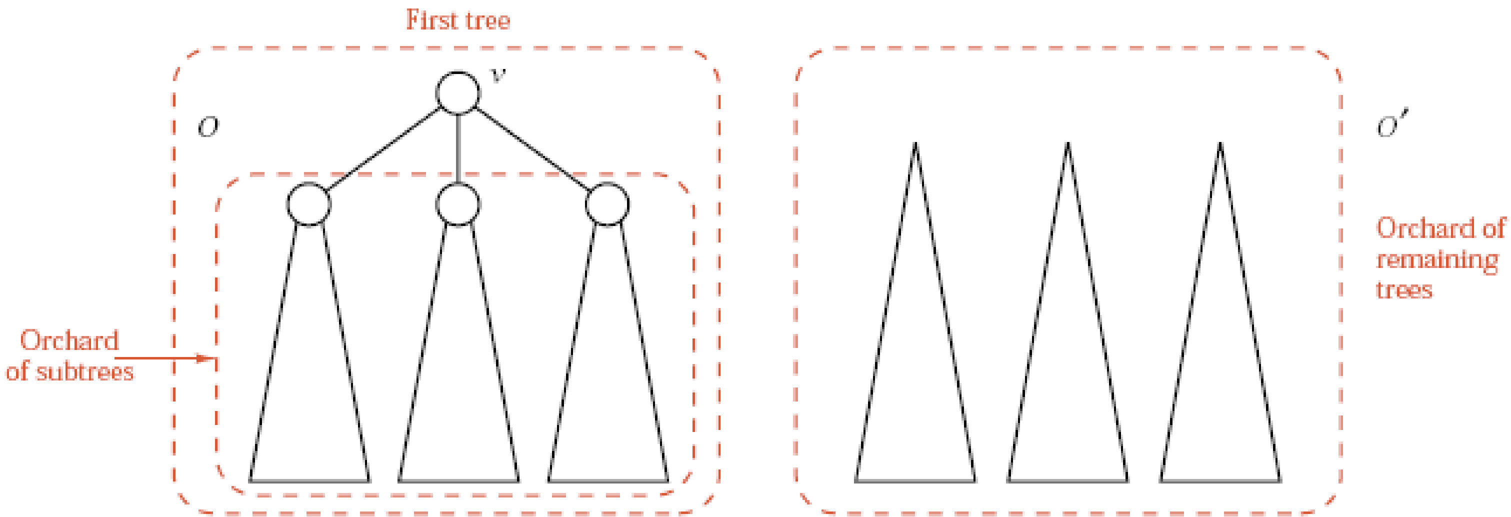
DEFINITION A ***rooted tree*** consists of a single vertex v, called the ***root*** of the tree, together with a forest F, whose trees are called the ***subtrees*** of the root.

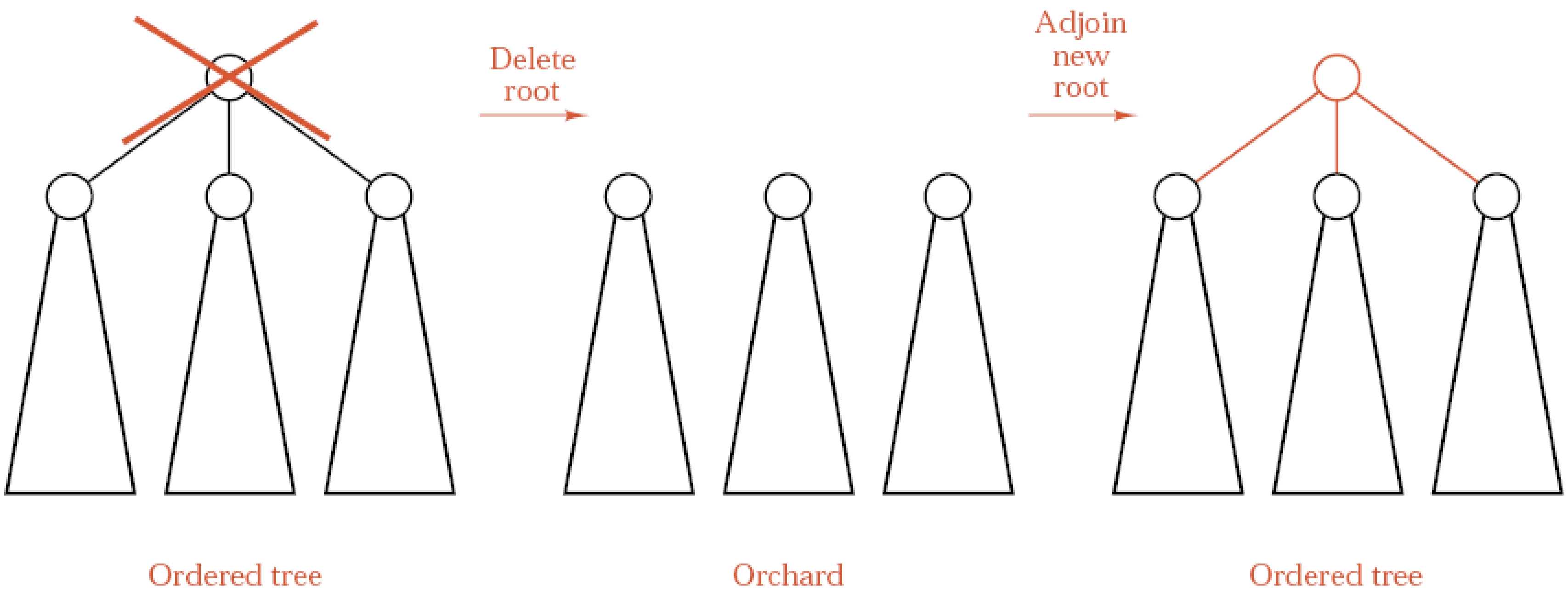
A ***forest*** F is a (possibly empty) set of rooted trees.

DEFINITION An ***ordered tree*** T consists of a single vertex v, called the ***root*** of the tree, together with an orchard O, whose trees are called the ***subtrees*** of the root v. We may denote the ordered tree with the ordered pair $T = \{v, O\}$.

An ***orchard*** O is either the empty set $\emptyset$, or consists of an ordered tree T, called the ***first tree*** of the orchard, together with another orchard O' (which contains the remaining trees of the orchard). We may denote the orchard with the ordered pair $O = (T, O')$.

Trees and Orchard

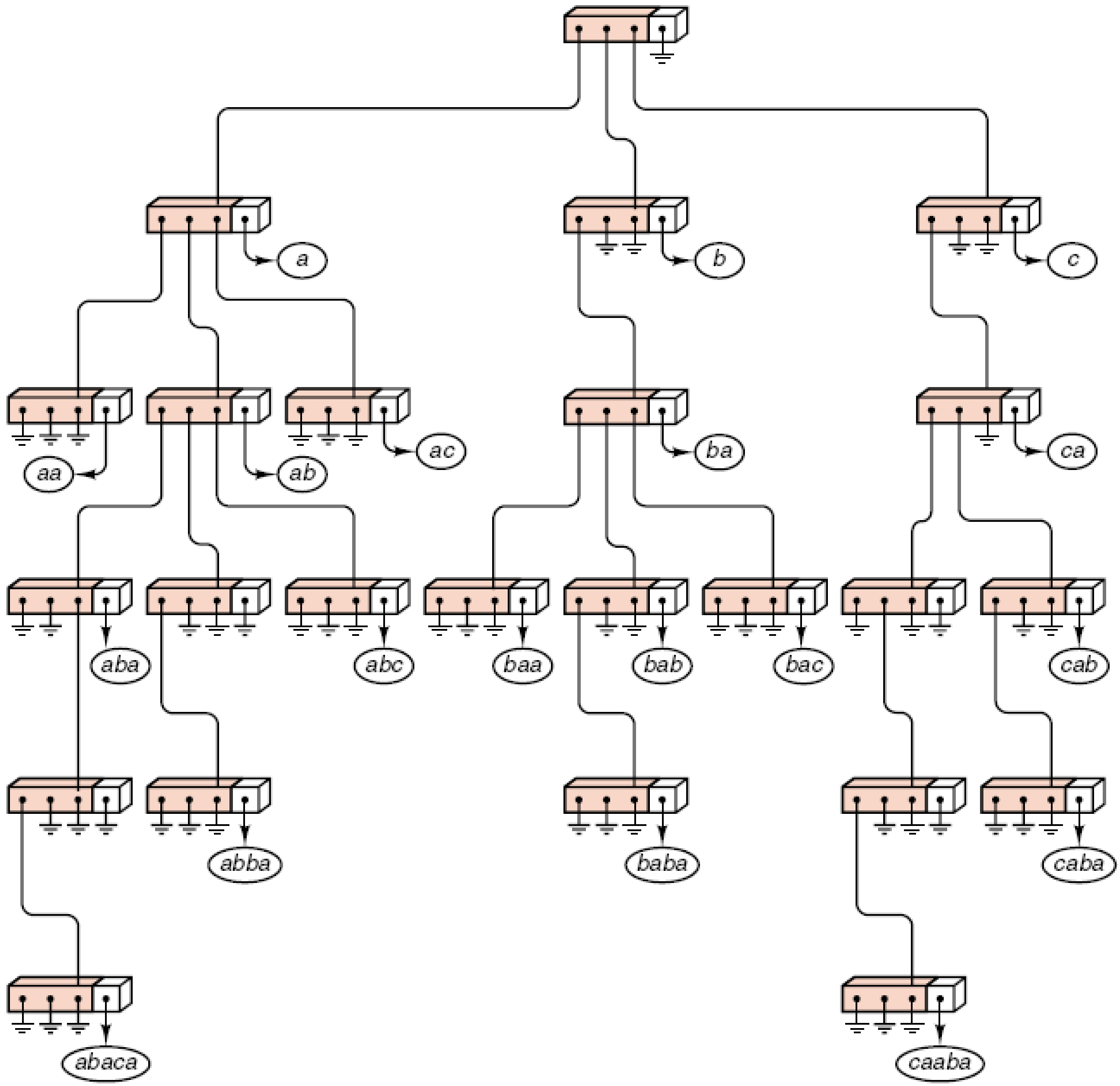




Lexicographic Search Trees: Tries

DEFINITION A ***trie*** of order m is either empty or consists of an ordered sequence of exactly m tries of order m.

Lexicographic Search Tree

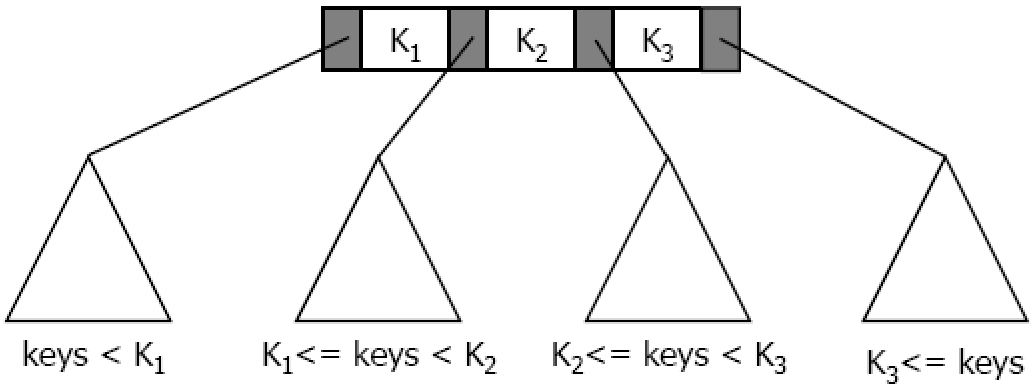


Multiway Trees

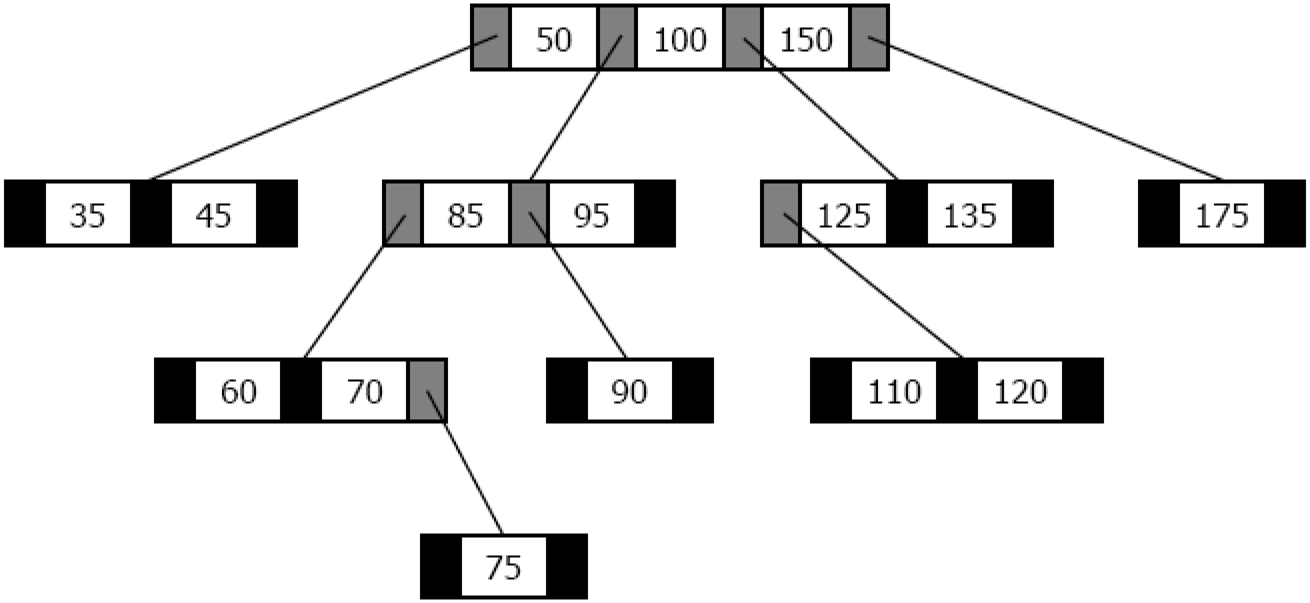
- Tree whose outdegree is not restricted to 2 while retaining the general properties of binary search trees.

M-Way Search Trees

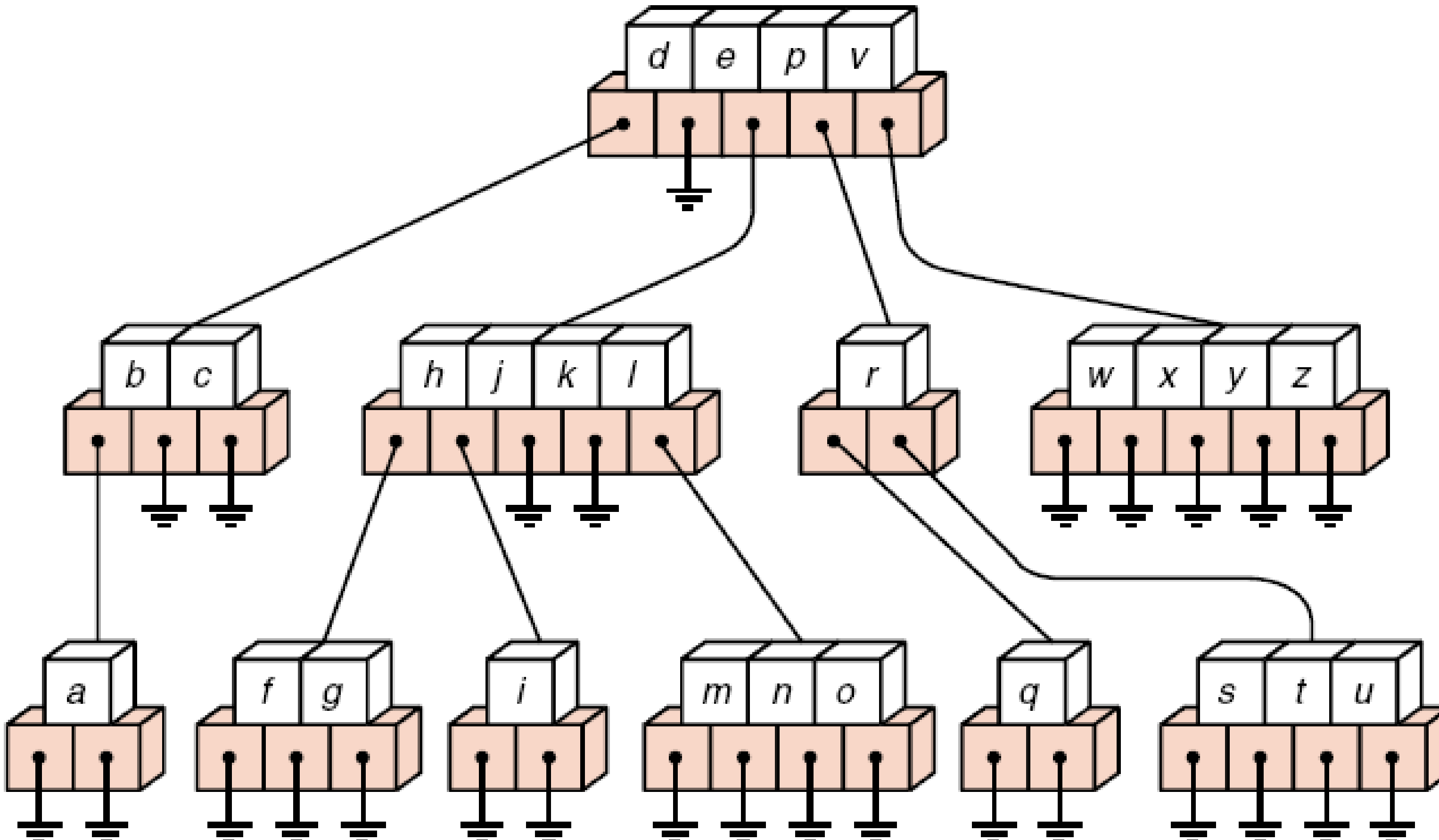
- Each node has m - 1 data entries and m subtree pointers.
- The key values in a subtree such that:
  - >= the key of the left data entry
  - <   the key of the right data entry.



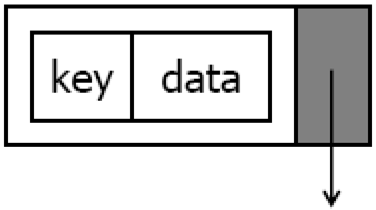
M-Way Search Trees



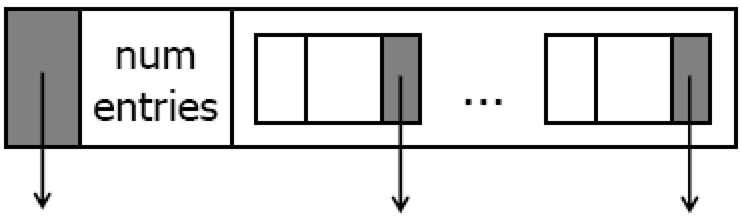
M-Way Search Tree



M-Way Node Structure



entry
    key <key type>
    data <data type>
    rightPtr <pointer>
end entry



node
    firstPtr <pointer>
    numEntries <integer>
    entries <array[1 .. m-1] of entry>
end node

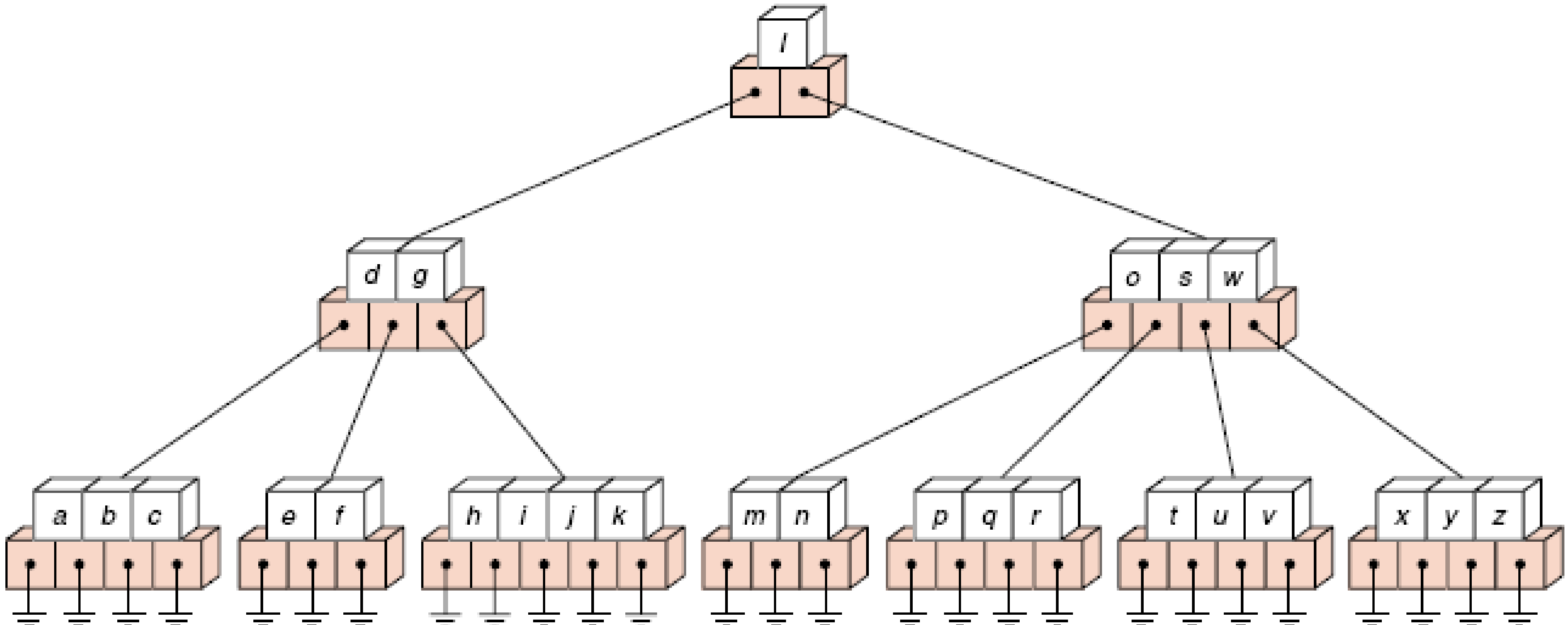
B-Trees

- M-way trees are unbalanced.
- Bayer, R. & McCreight, E. (1970) created B-Trees.

B-Trees

- A B-tree is an m-way tree with the following additional properties ($m >= 3$):
  - The root is either a leaf or has at least 2 subtrees.
  - All other nodes have at least $\lceil m/2 \rceil - 1$ entries.
  - All leaf nodes are at the same level.

B-Tree

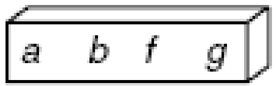


B-Tree Insertion

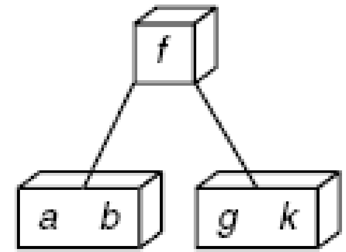
- Insert the new entry into a leaf node.
- If the leaf node is overflow, then split it and insert its median entry into its parent.

B-Tree Insertion

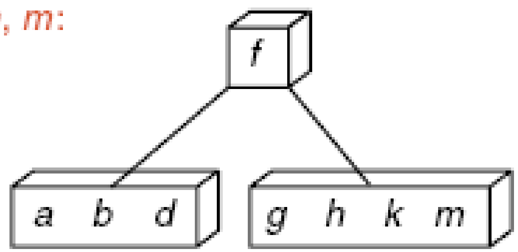
1.
a, g, f, b:



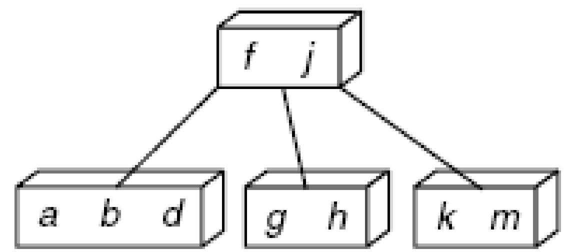
2.
k:



3.
d, h, m:



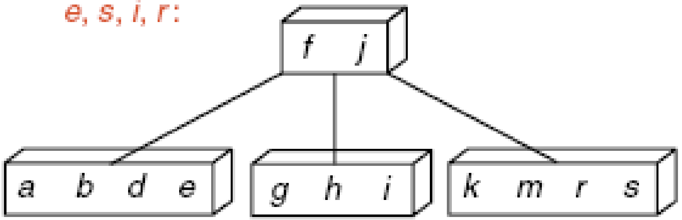
4.
j:



B-Tree Insertion

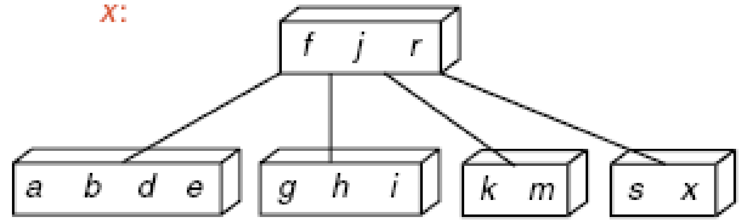
5.

e, s, i, r:



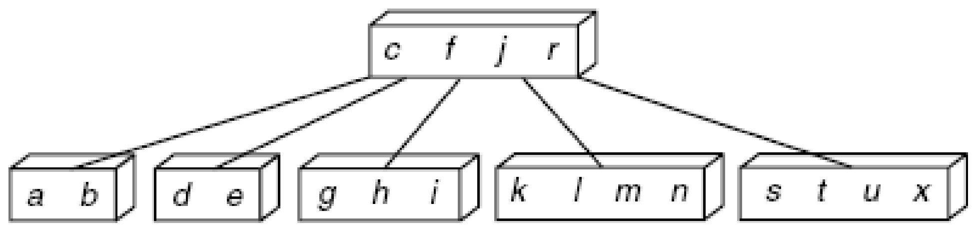
6.

x:



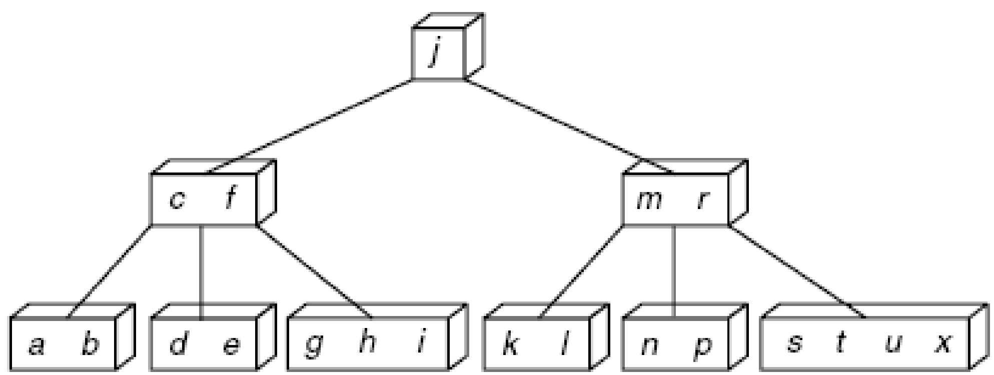
7.

c, l, n, t, u:



8.

p:

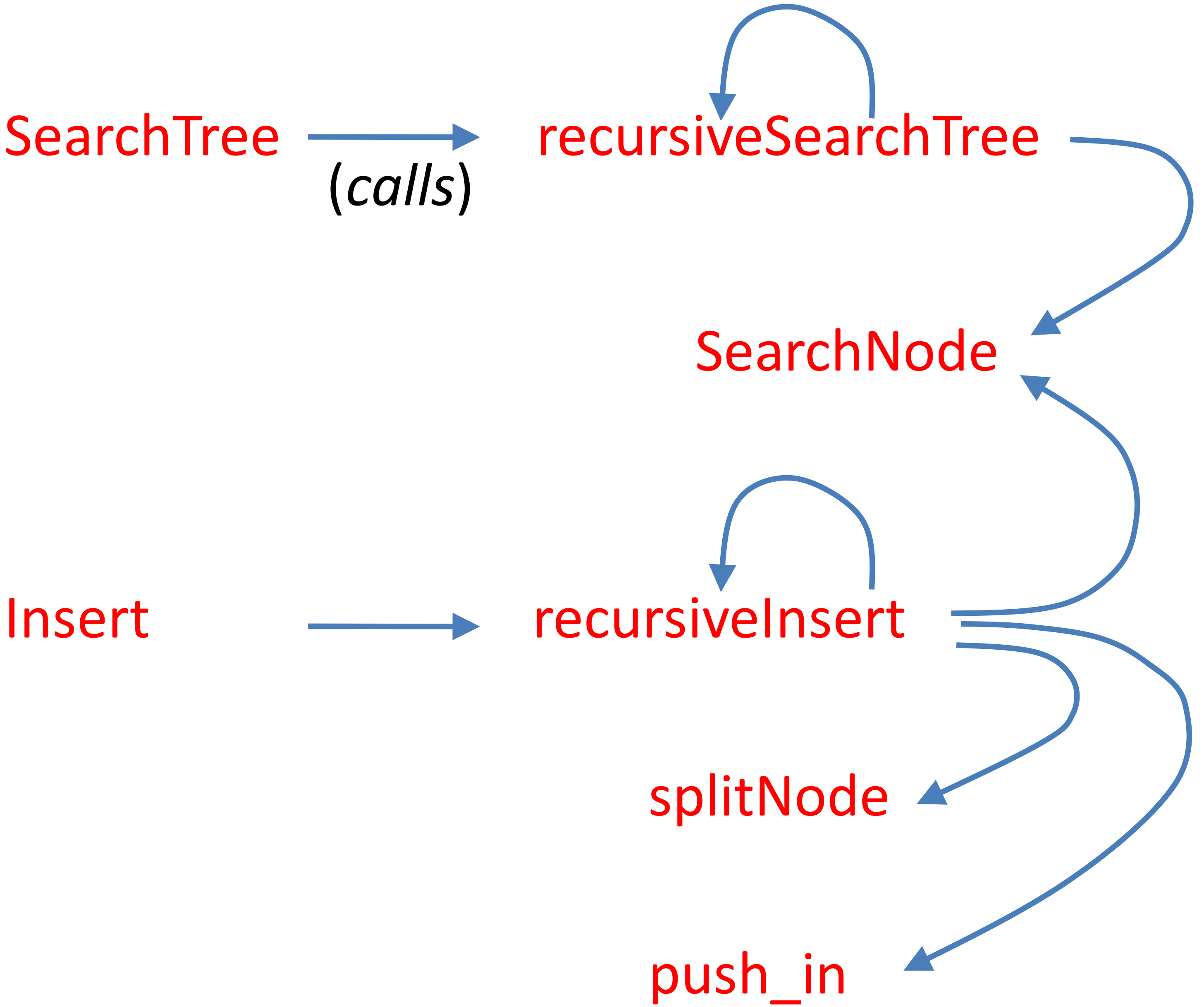


B-Tree

```
B_Node
    count <integer>
    data <array of <DataType>>
    branch <array of <pointer>>
End B_Node
```

```
B_Tree
   root <pointer>
End B_Tree
```

Methods and Functions



B-Tree SeachTree

```
<ErrorCode> SearchTree (ref target <DataType>)


1.    return recursiveSearchTree(root, target)
End SearchTree
```

B-Tree SeachTree

<ErrorCode> **recursiveSearchTree** (val subroot <pointer>,
ref target <DataType>)

1. result = *not_present*
2. **if** (subroot is not NULL)
    1. result = SearhNode (subroot, target, position)
    2. **if** (result = *not_present*)
        1. result = recursiveSearchTree (subroot->$\text{branch}_{\text{position}}$, target)
    3. **else**
        1. target = subroot->$\text{data}_{\text{position}}$
3. return result

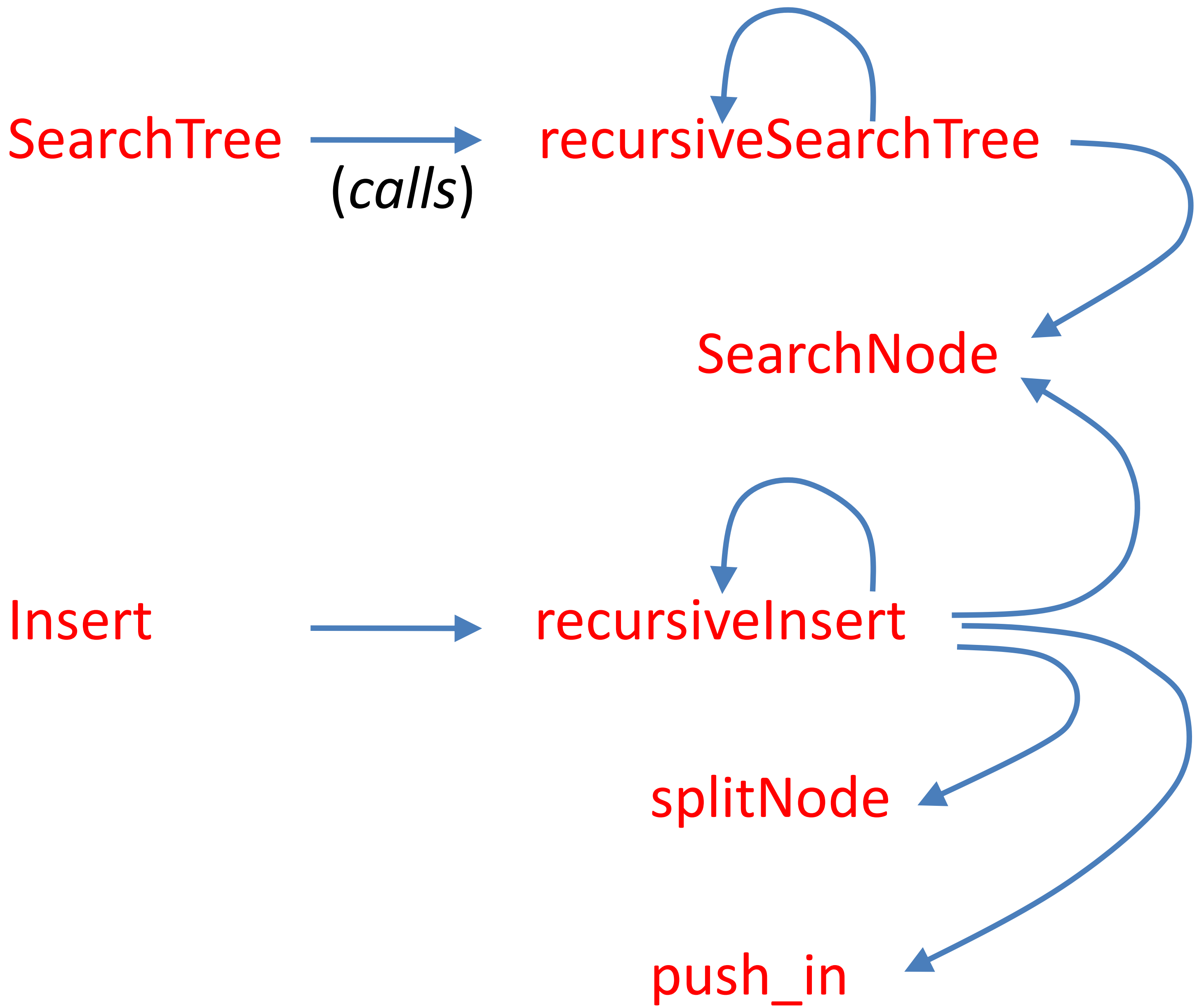
End recursiveSearchTree

B-Tree SeachTree

```
<ErrorCode> SearchNode (val subroot <pointer>,
                         val target <DataType>,
                         ref position <integer>)

1.   position = 0
2.   loop (position <  subroot->count) AND (target>subroot->data[position])
     1.   position = position + 1   // Sequential Search
3.   if (position < subroot->count) AND (target = subroot->data[position])
     1.   return success
4.   else
     1.   return not_present
End SearchNode
```

Methods and Functions



B-Tree Insertion

<ErrorCode> **Insert** (val newData <DataType>)

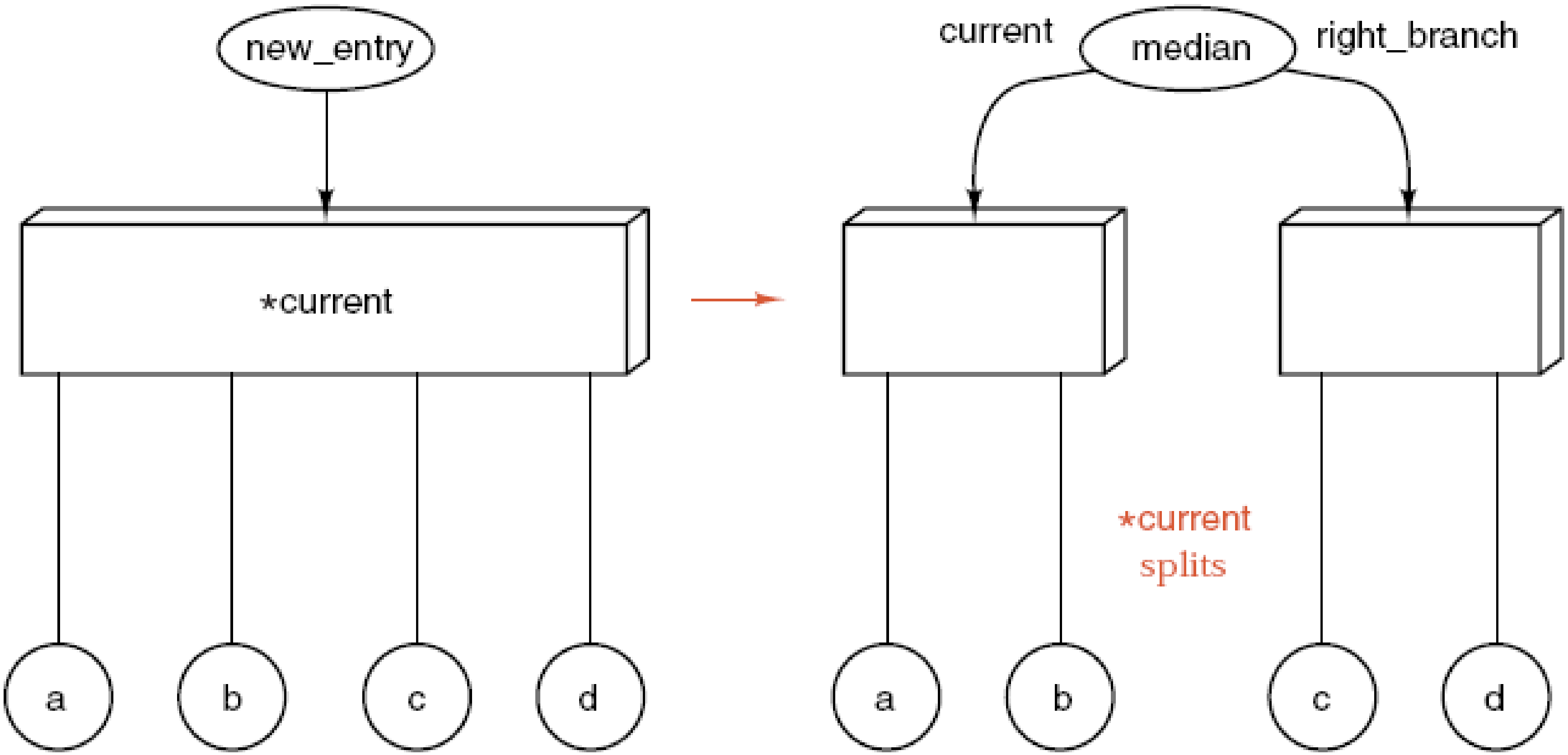
(local variable: **median** <DataType>, **rightBranch** <pointer>,
newroot <pointer>, result <ErrorCode>)

Return *duplicate_error, success*

1. result = recursiveInsert (root, newData, **median, rightBranch**)
2. if (result = *overflow*)
    1. Allocate newroot
    2. newroot->count = 1
    3. newroot->$data_0$ = **median**
    4. newroot->$branch_0$ = root
    5. newroot->$branch_1$ = **rightBranch**
    6. root = newroot
    7. result = *success*
3. return result

End Insert

Split Node


new_entry
*current
a
b
c
d
current
median
right_branch
*current
splits
a
b
c
d

B-Tree Insertion

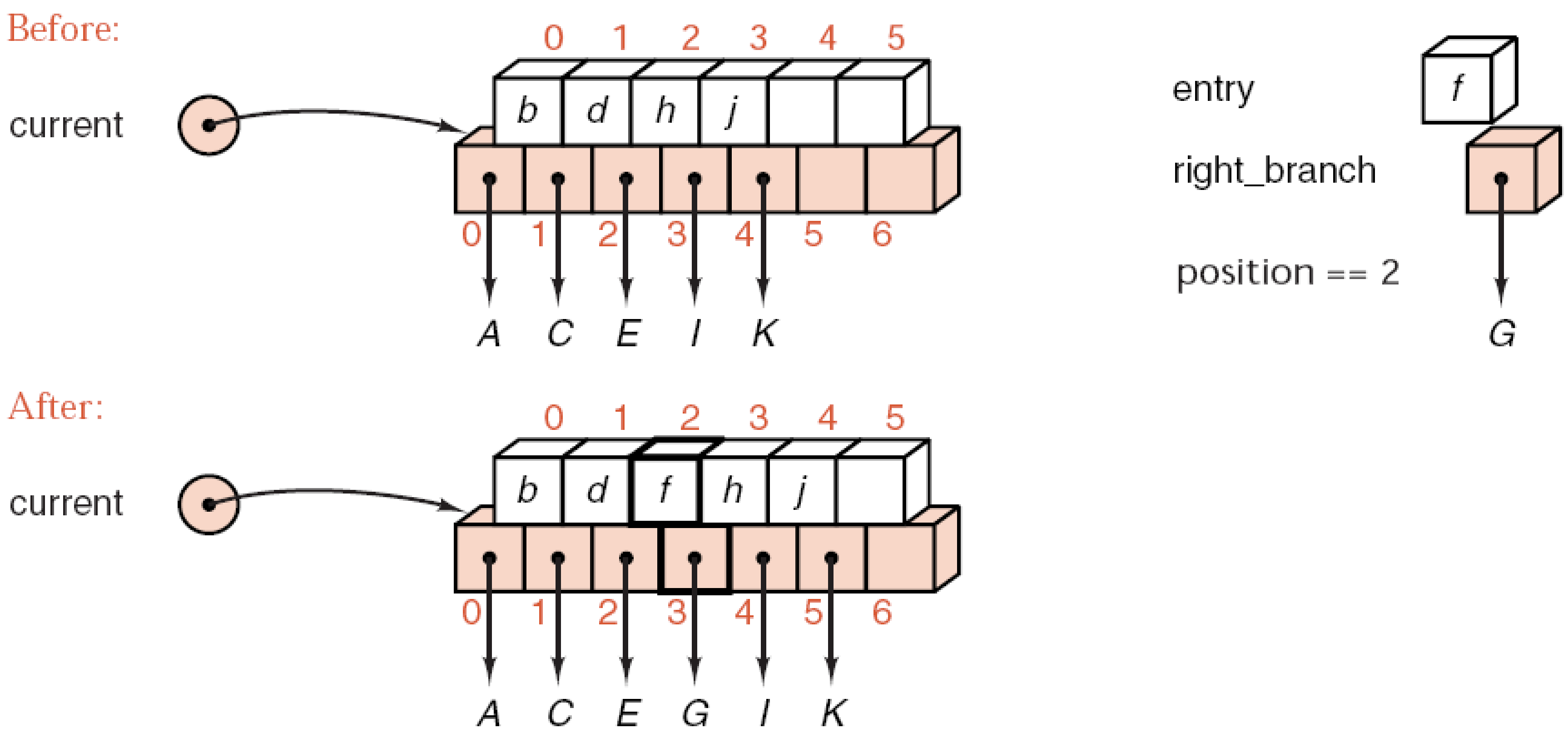
```
<ErrorCode> recursiveInsert (val subroot <pointer>,
                             val newData <DataType>,
                             ref median <DataType>,
                             ref rightBranch <pointer>)

Return overflow, duplicate_error, success

1.  if (subroot = NULL)
    1.  median = newData
    2.  rightbranch = NULL
    3.  result = overflow
2.  else
```

```
<ErrorCode> recursiveInsert (val subroot <pointer>,
                             val newData <DataType>,
                             ref median <DataType>,
                             ref rightBranch <pointer>)                (cont.)
2.  // else, local variables: extraEntry, extraBranch
    1.  if (SearchNode (subroot, newData, position) = success)
        1.  result = duplicate_error
    2.  else
        1.  result = recursiveInsert (subroot->branch_position, newData,
                                      extraEntry, extraBranch)
        2.  if (result = overflow)
            1.  if (subroot->count < order-1)
                1.  result = success
                2.  push_in (subroot, extraEntry, extraBranch, position)
            2.  else
                1.  splitNode (subroot, extraEntry, extraBranch, position,
                               rightBrach, median)
3.  return result
End recursiveInsert
```

Push In



Before:
current
0 1 2 3 4 5
b d h j
0 1 2 3 4 5 6
A C E I K
entry
f
right_branch
position == 2
G
After:
current
0 1 2 3 4 5
b d f h j
0 1 2 3 4 5 6
A C E G I K

B-Tree

<void> **push_in** (val subroot <pointer>,
val entry <DataType>,
val rightBranch <pointer>,
val position <integer>)

1. i = subroot->count
2. **loop** (i > position)
   1. subroot->$data_i$ = subroot->$data_{i-1}$
   2. subroot->$branch_{i+1}$ = subroot->$branch_i$
   3. i = i + 1
3. subroot->$data_{position}$= entry
4. subroot->$branch_{position+1}$= rightBranch
5. subroot->count = -subroot->count + 1

End push_in

B-Tree

```
<void> splitNode (val subroot <pointer>,
                  val extraEntry <DataType>,
                  val extraBranch <pointer>,
                  val position <integer>,
                  ref rightHalf <pointer>,
                  ref median <DataType>)
```

B-Tree Insertion

In contrast to binary search trees, B-trees are not allowed to grow at their leaves; instead, they are forced to grow at the root. General insertion method:

1. Search the tree for the new key. This search (if the key is truly new) will terminate in failure at a leaf.

2. Insert the new key into to the leaf node. If the node was not previously full, then the insertion is finished.

B-Tree Insertion

3. When a key is added to a full node, then the node splits into two nodes, side by side on the same level, except that the median key is not put into either of the two new nodes.

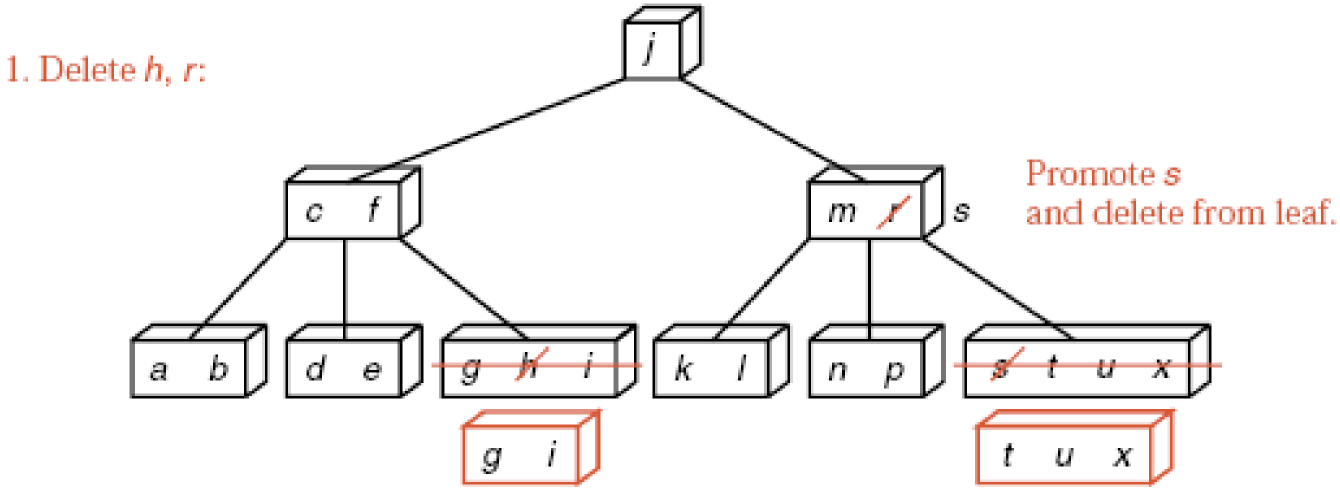
4. When a node splits, move up one level, insert the median key into this parent node, and repeat the splitting process if necessary.

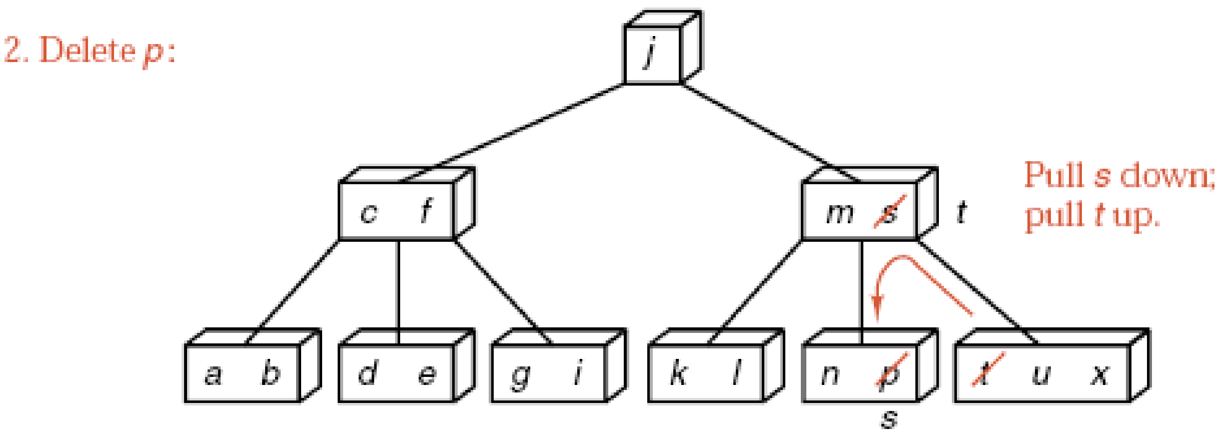
5. When a key is added to a full root, then the root splits in two and the median key sent upward becomes a new root. This is the only time when the B-tree grows in height.

B-Tree Deletion

- It must take place at a leaf node.
- If the data to be deleted are not in a leaf node, then replace that entry by the largest entry on its left subtree.

B-Tree Deletion



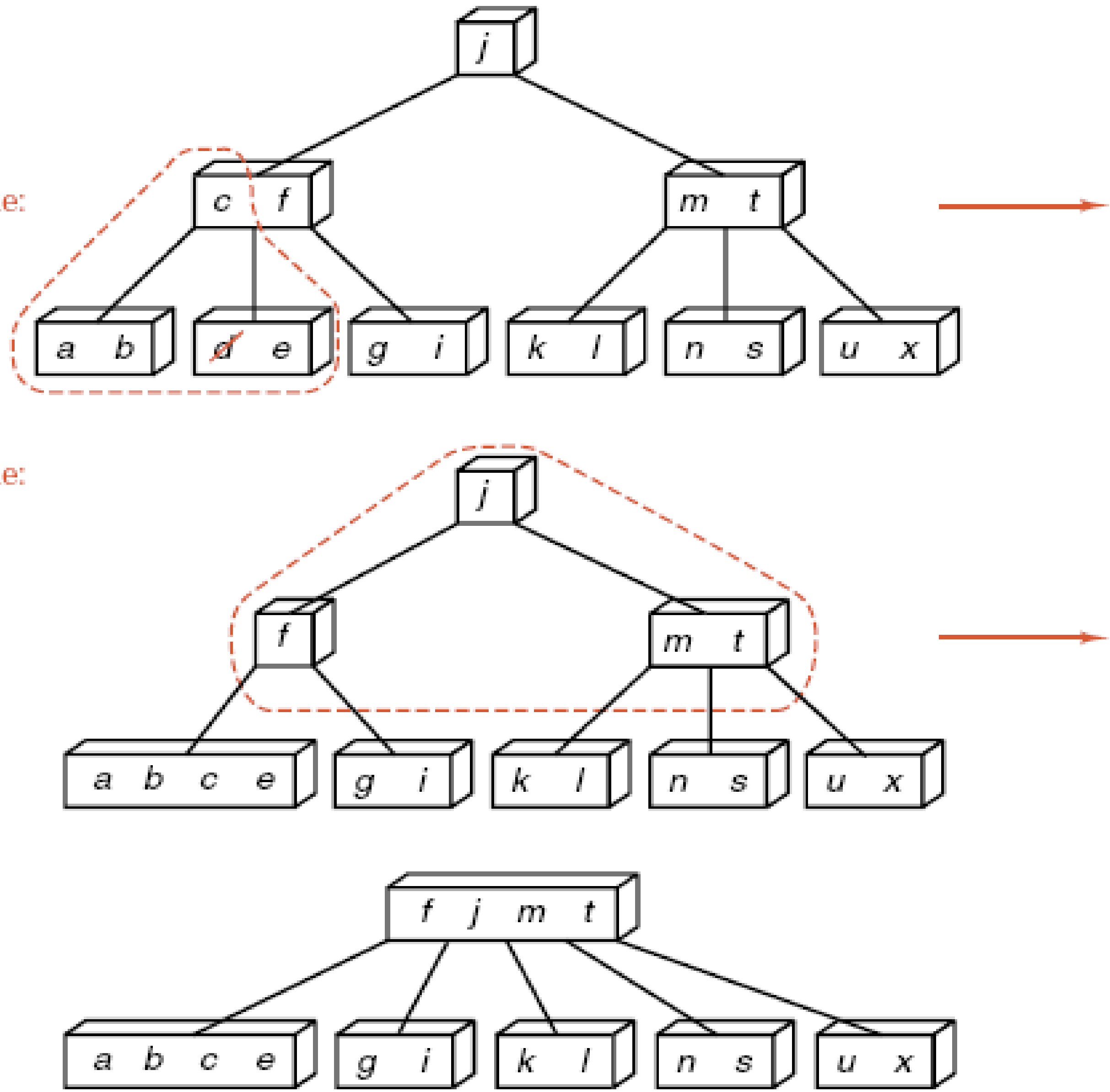


B-Tree Deletion

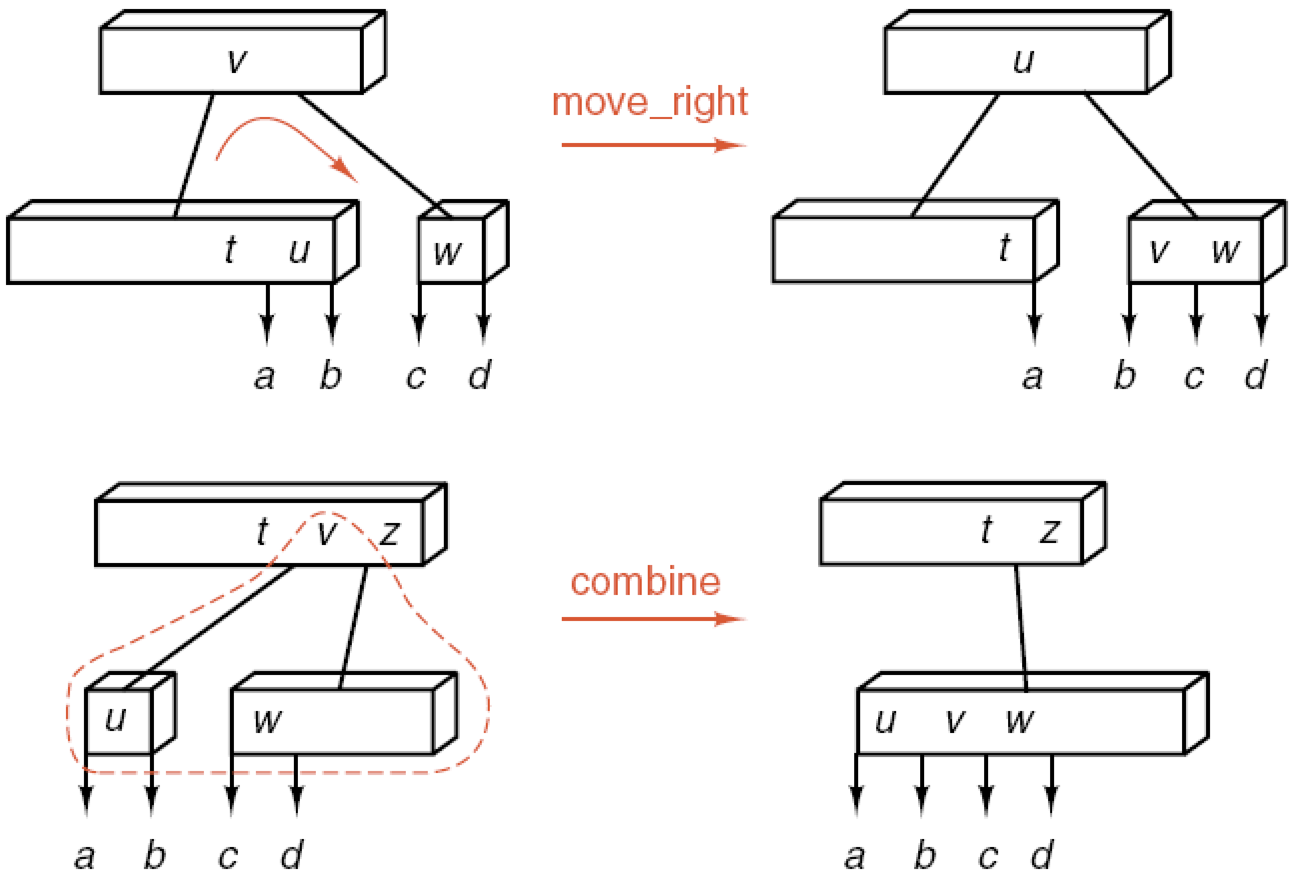
3. Delete *d*:

Combine:

Combine:



B-Tree Deletion



Reflow

- For each node to have sufficient number of entries:
  - Balance: shift data among nodes.
  - Combine: join data from nodes.

Balance

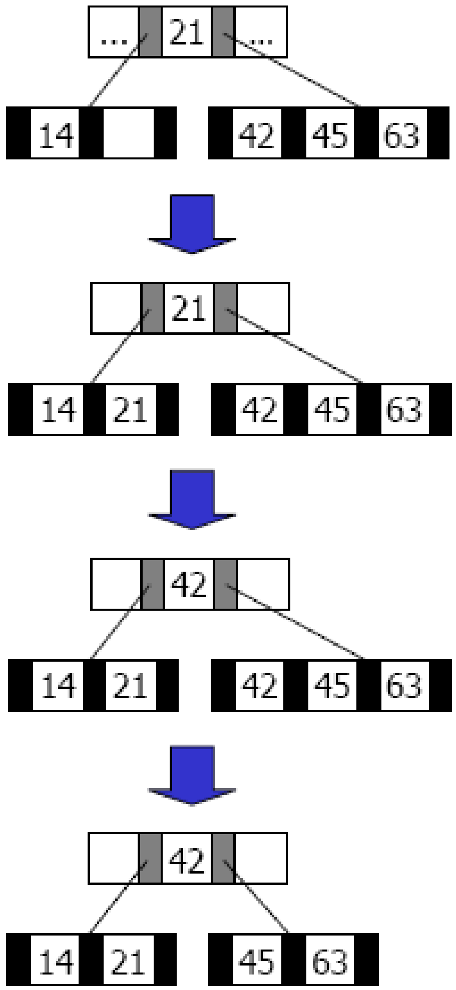
Borrow from right

Original node

Rotate parent data down

Rotate data to parent

Shift entries left



Balance

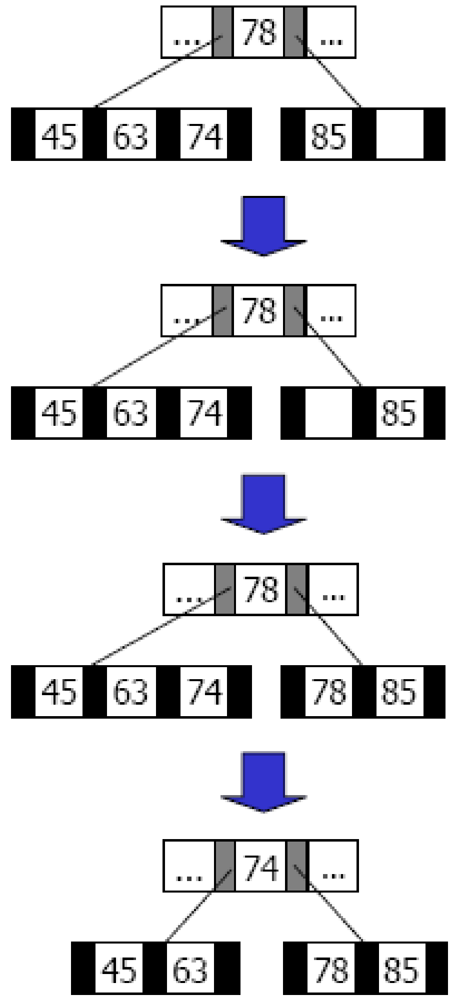
Borrow from left

Original node

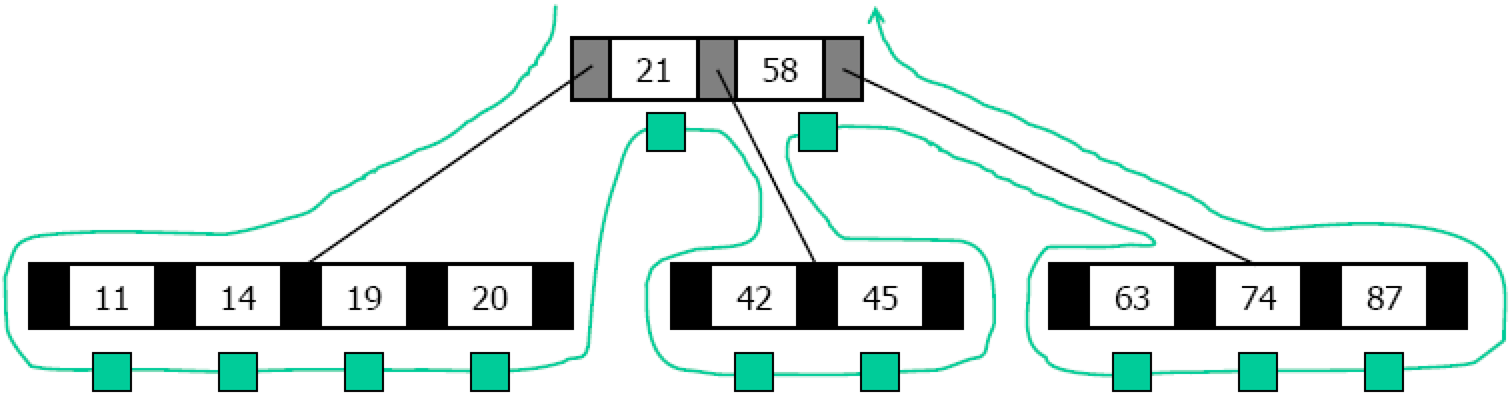
Shift entries right

Rotate parent data down

Rotate data up



B-Tree Traversal



B-Tree Variations

- B*Tree: the minimum number of (used) entries is two thirds.
- B+Tree:
  - Each data entry must be represented at the leaf level.
  - Each leaf node has one additional pointer to move to the next leaf node.

k-d Trees

- 2-d Tree
- k-d Tree