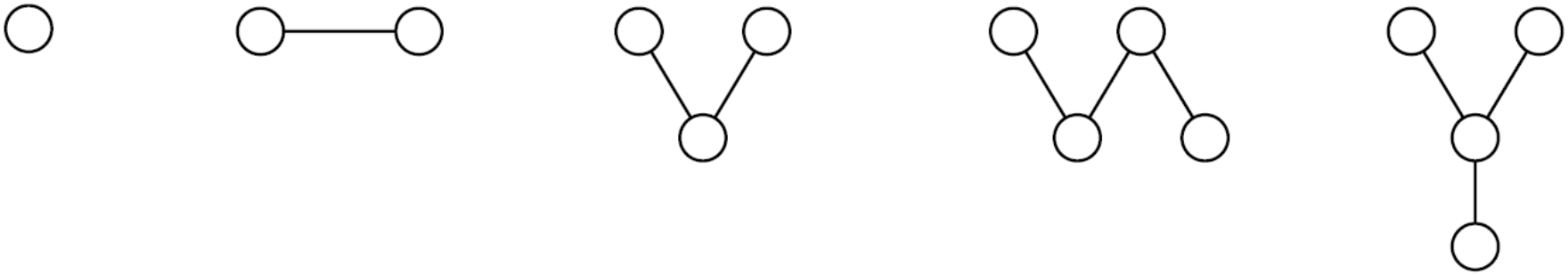
Chapter 11 Multiway Trees

信息科学与技术学院　　黄方军

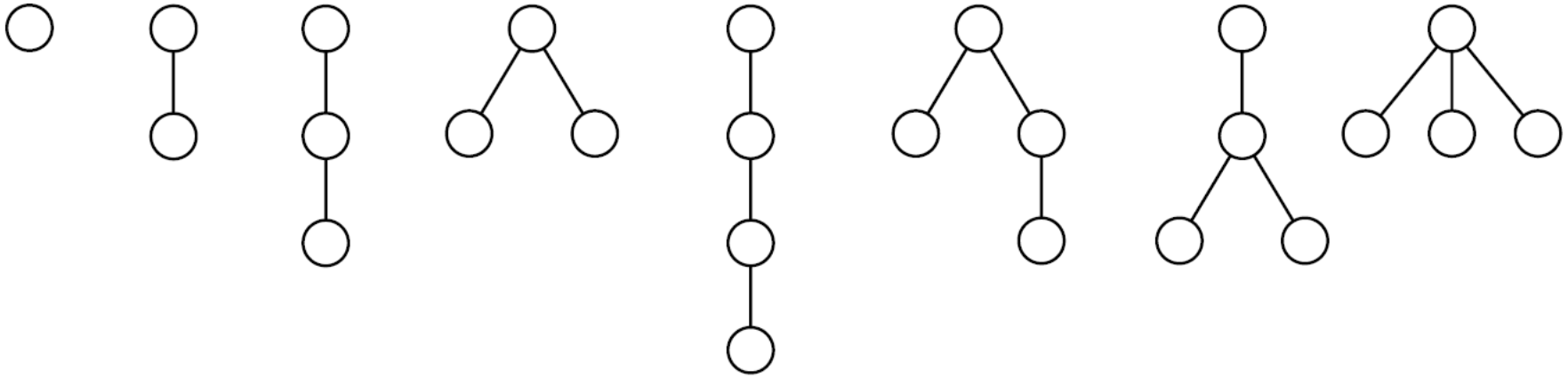
data_structures@163.com

东校区实验中心B502

11.1.1 On the Classification of Species

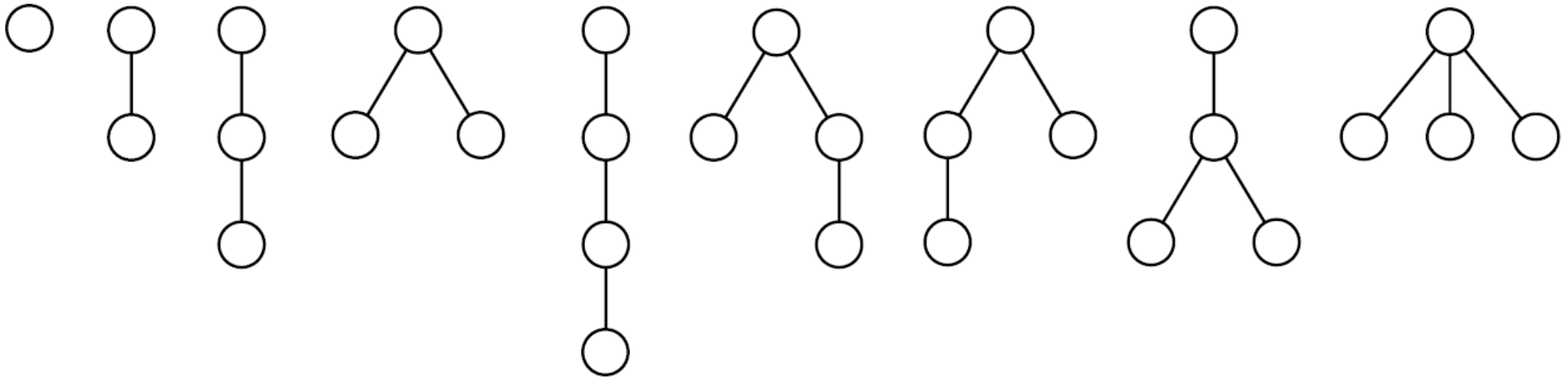


Free trees with four or fewer vertices
(Arrangement of vertices is irrelevant.)



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

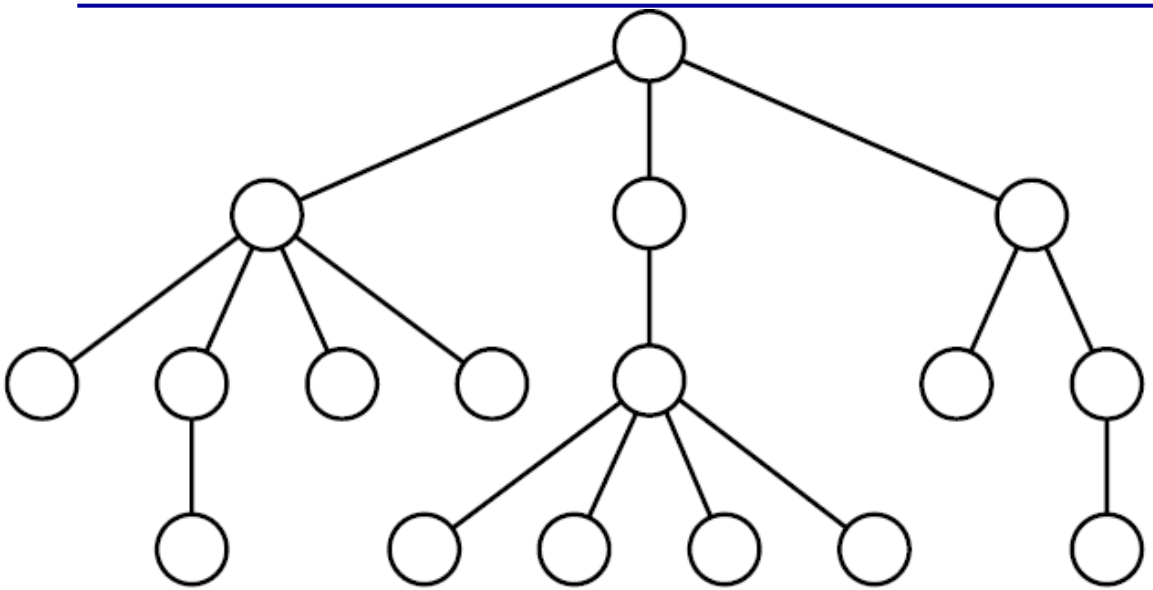
11.1.1 On the Classification of Species



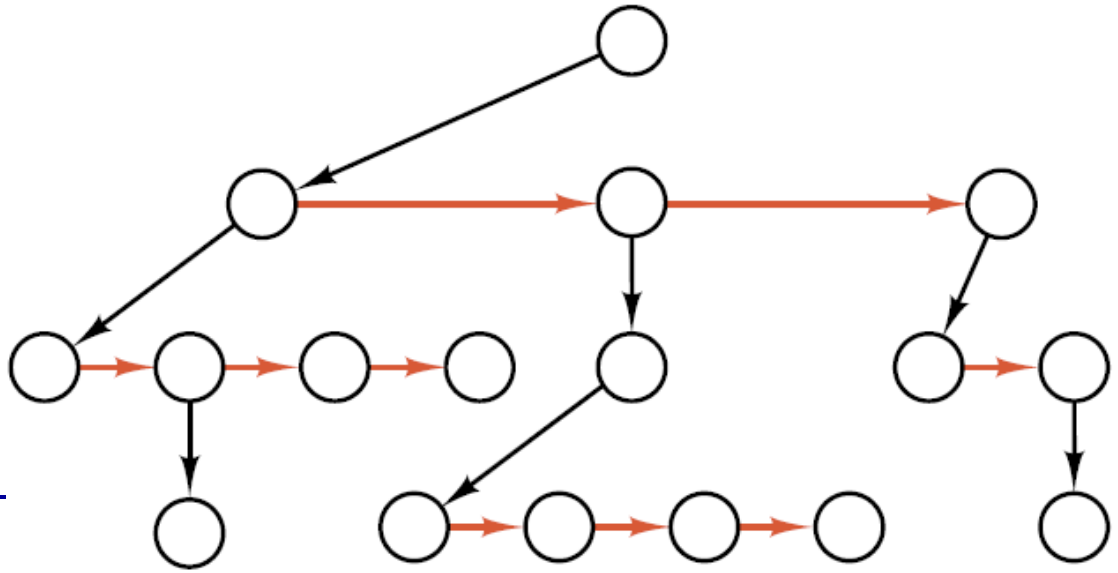
Ordered trees with four or fewer vertices

Figure 11.1. Various kinds of trees

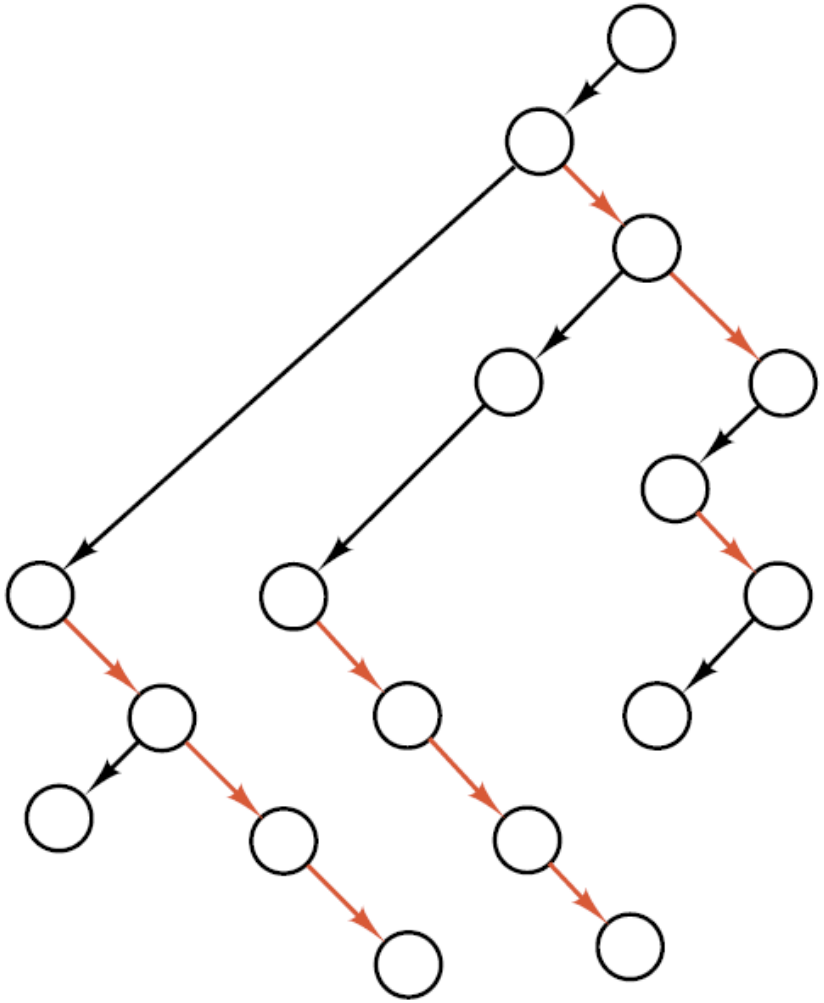
11.1.2 Ordered Trees



first_child: black; next_sibling: color



11.1.2 Ordered Trees



first_child (left) links: black
next_sibling (right) links: color

Figure 11.3. Rotated form of linked implementation

11.1.3 Forests and Orchards

- Forest. A set of rooted trees.
- Orchard. An ordered set of ordered trees.

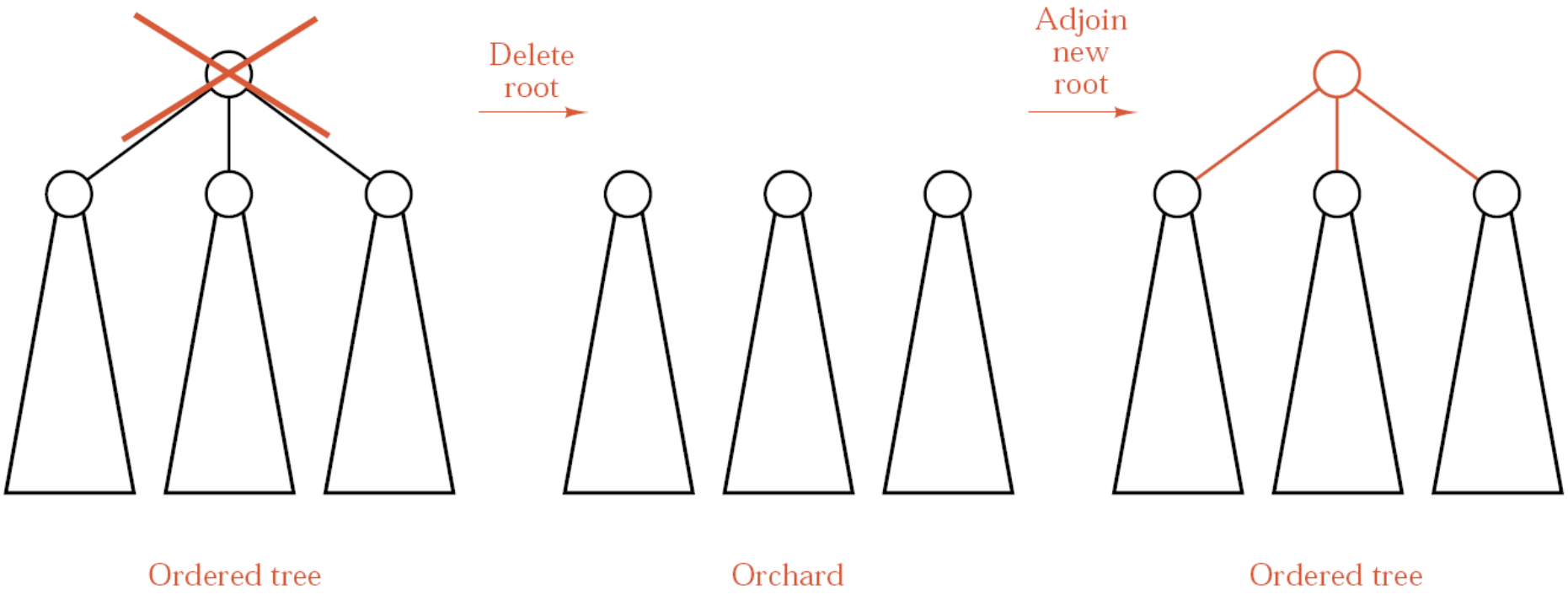
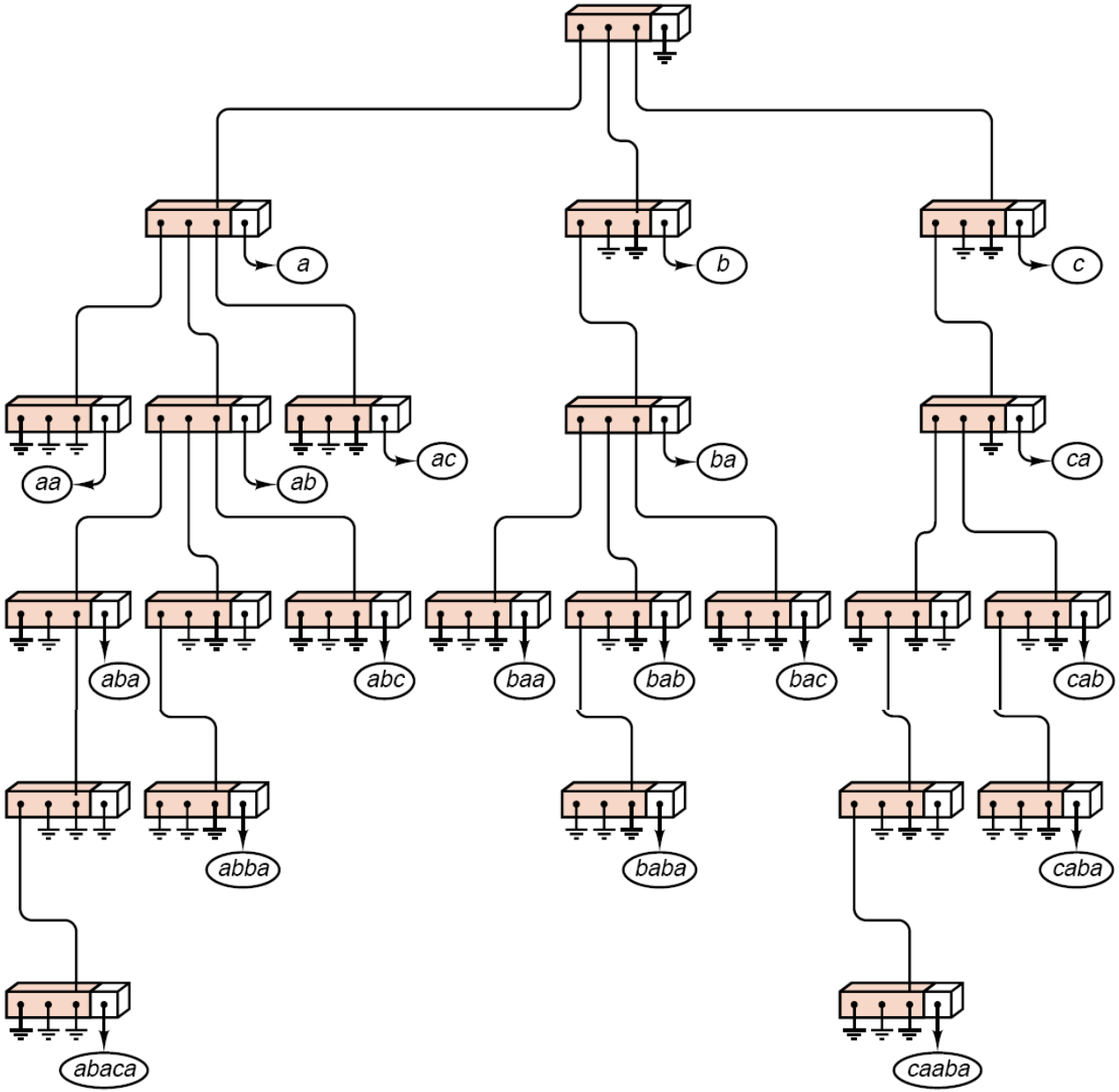


Figure 11.4. Deleting and adjoining a root

有根树和有序树不能为空，二叉树、森林、果园均可为空。

11.2 Lexicographic Search Trees: Tries



11.2 Lexicographic Search Trees: Tries

```
const int num_chars = 28;

struct Trie_node {
//    data members
   Record *data;
   Trie_node *branch[num_chars];
//    constructors
   Trie_node();
};
```

11.2.4 Searching a Trie

```
Error_code Trie::trie_search(const Key &target, Record &x) const
{
   int position = 0;
   char next_char;
   Trie_node *location = root;
   while (location != NULL && (next_char = target.key_letter(position)) != ' ') {
                    //    Terminate search for a NULL location or a blank in the target.
      location = location->branch[alphabetic_order(next_char)];
                                    //    Move down the appropriate branch of the trie.
      position++;
                                    //    Move to the next character of the target.
   }
   if (location != NULL && location->data != NULL) {
      x = *(location->data);
      return success;
   }
   else
      return not_present;
}
```

11.2.5 Insertion into a Trie

```
Error_code Trie::insert(const Record &new_entry)
{
   Error_code result = success;
   if (root == NULL) root = new Trie_node;   //   Create a new empty Trie.
   int position = 0;                         //   indexes letters of new_entry
   char next_char;
   Trie_node *location = root;               //   moves through the Trie
   while (location != NULL &&
          (next_char = new_entry.key_letter(position)) != ' ') {
      int next_position = alphabetic_order(next_char);
      if (location->branch[next_position] == NULL)
         location->branch[next_position] = new Trie_node;
      location = location->branch[next_position];
      position++;
   }
   //   At this point, we have tested for all nonblank characters of new_entry.
   if (location->data != NULL) result = duplicate_error;
   else location->data = new Record(new_entry);
   return result;
}
```

11.3 External Searching: B-Trees

A ***B-tree of order*** m is an m-way search tree in which

1. All leaves are on the same level.
2. All internal nodes except the root have at most m nonempty children, and at least $\lceil m/2 \rceil$ nonempty children.
3. The number of keys in each internal node is one less than the number of its nonempty children, and these keys partition the keys in the children in the fashion of a search tree.
4. The root has at most m children, but may have as few as 2 if it is not a leaf, or none if the tree consists of the root alone.

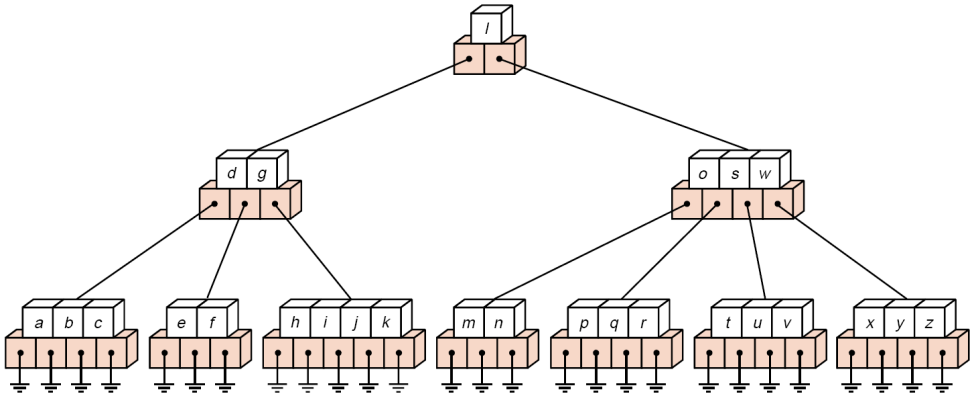


Figure 11.9. A B-tree of order 5

11.3 External Searching: B-Trees

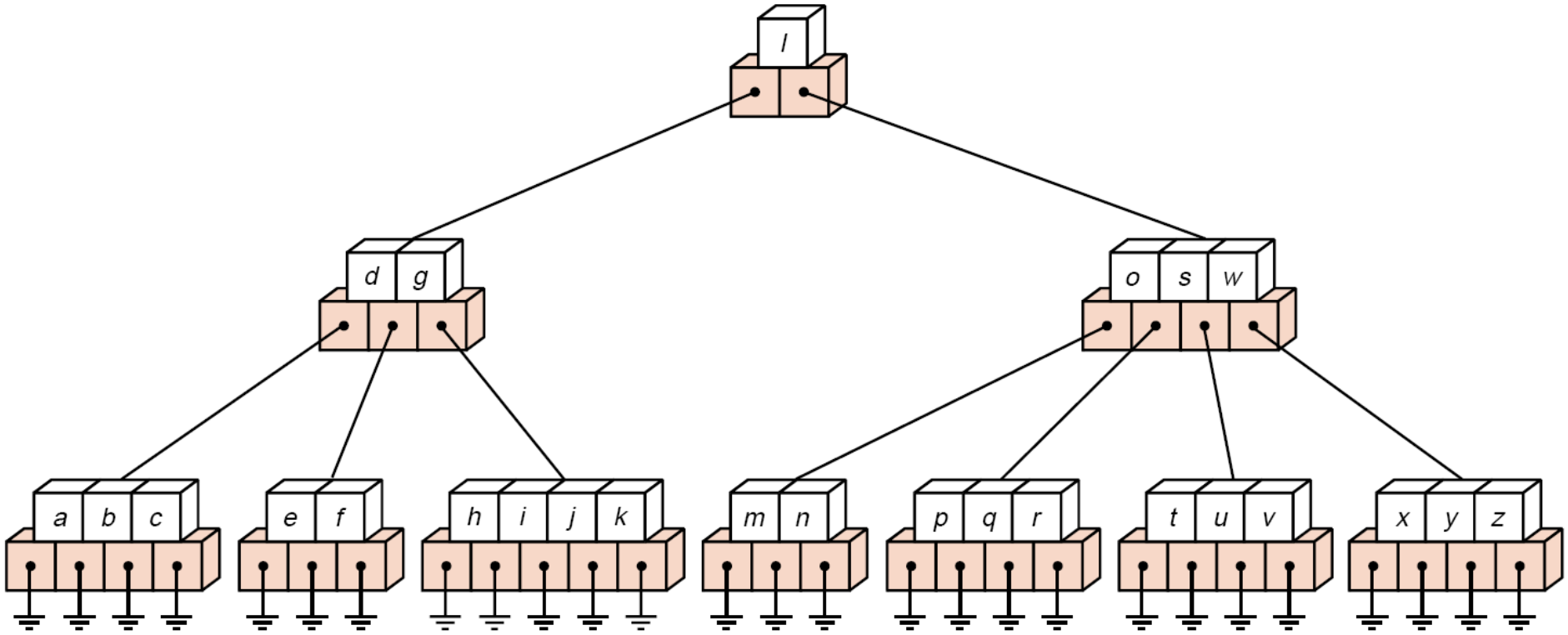


Figure 11.9. A B-tree of order 5

11.3 External Searching: B-Trees

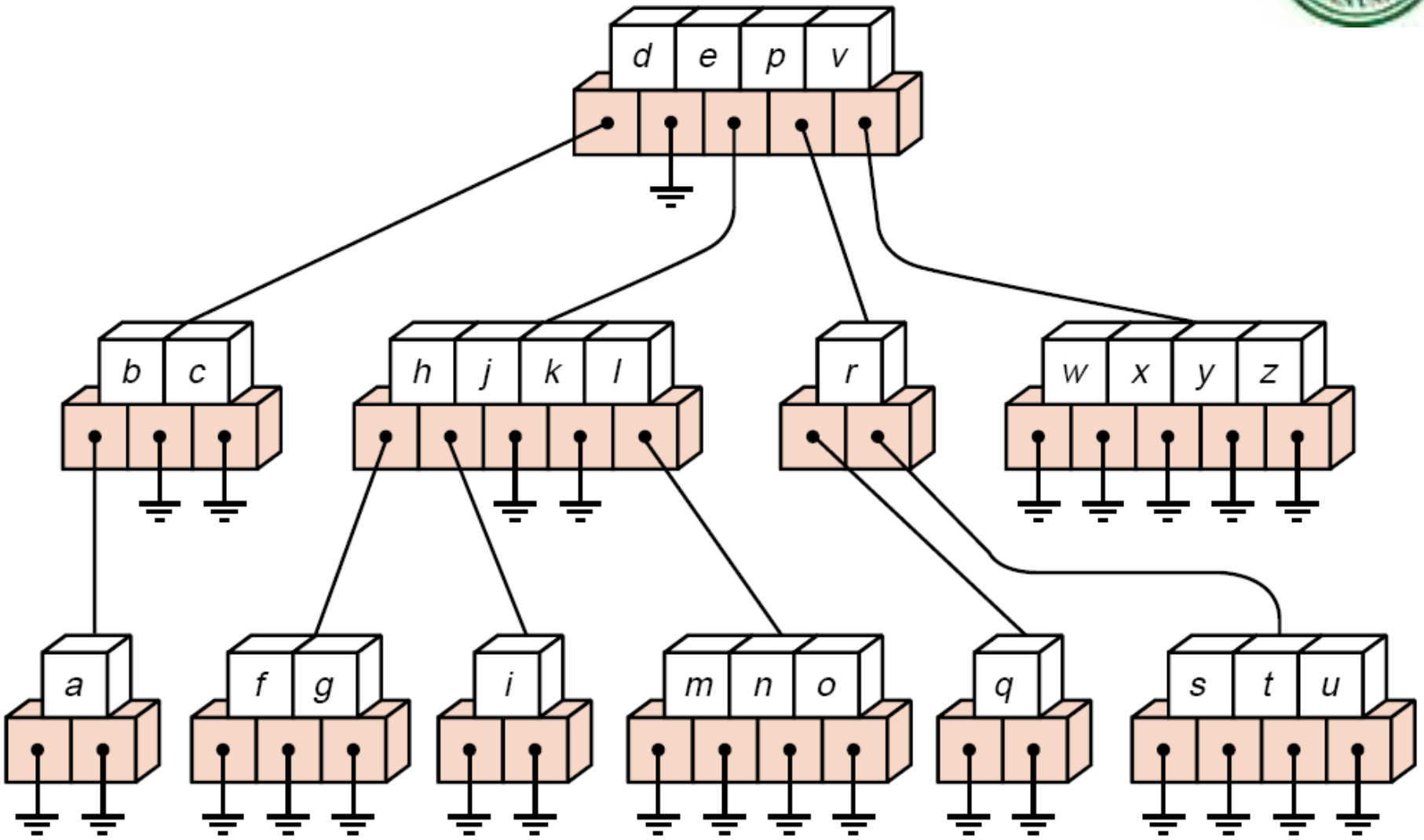


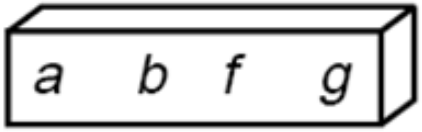
Figure 11.8. A 5-way search tree (not a B-tree)

11.3.4 Insertion into a B-Tree

a g f b k d h m j e s i r x c l n t u p

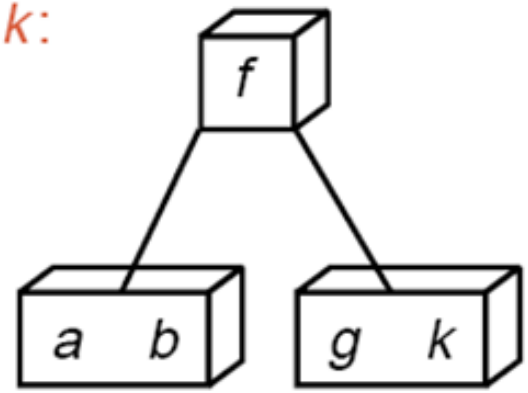
1.

a, g, f, b:



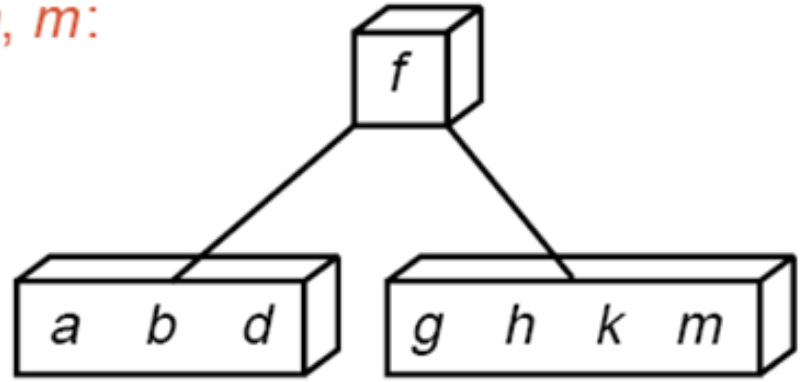
2.

k:



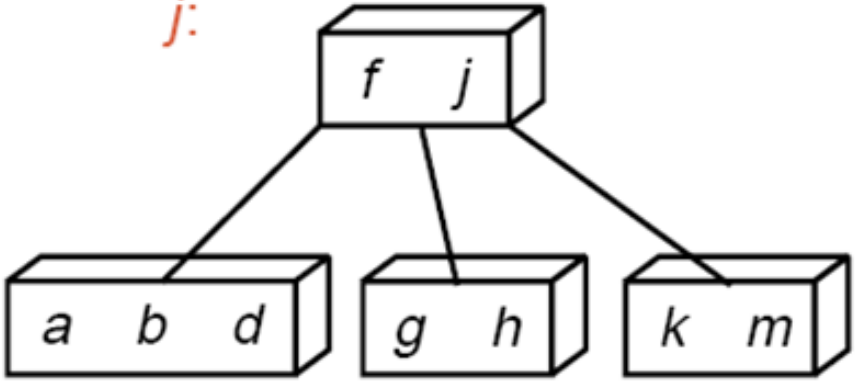
3.

d, h, m:



4.

j:

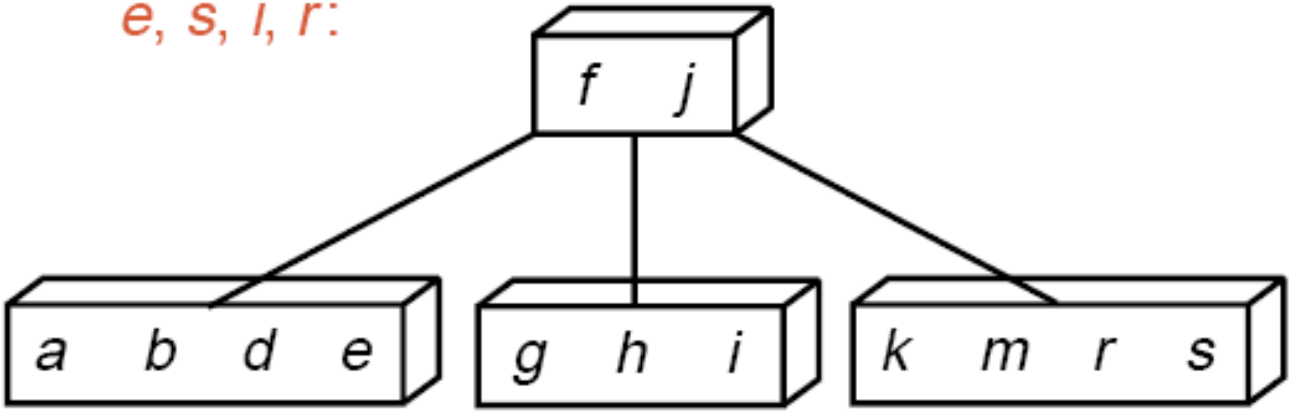


11.3.4 Insertion into a B-Tree

a g f b k d h m j e s i r x c l n t u p

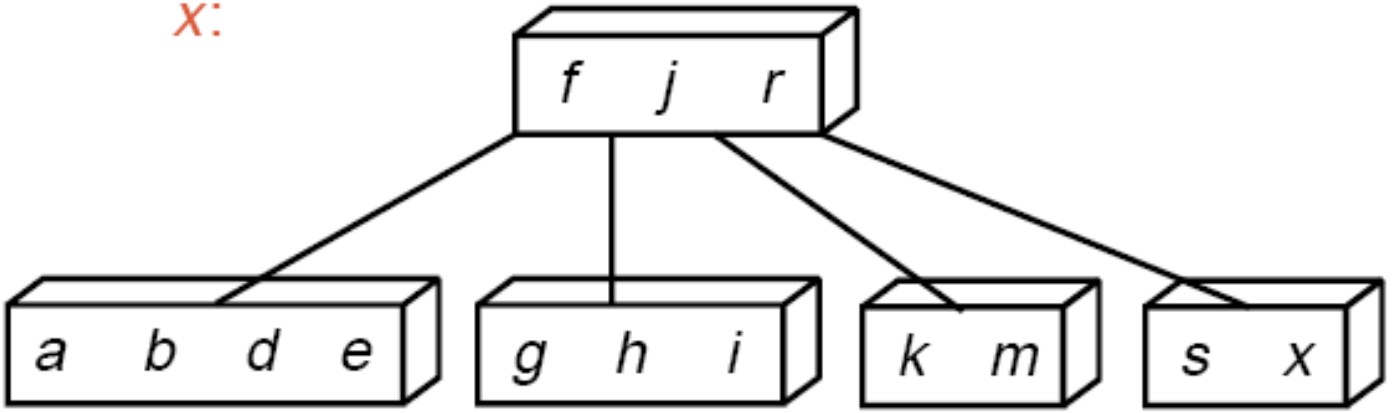
5.

e, *s*, *i*, *r*:



6.

x:

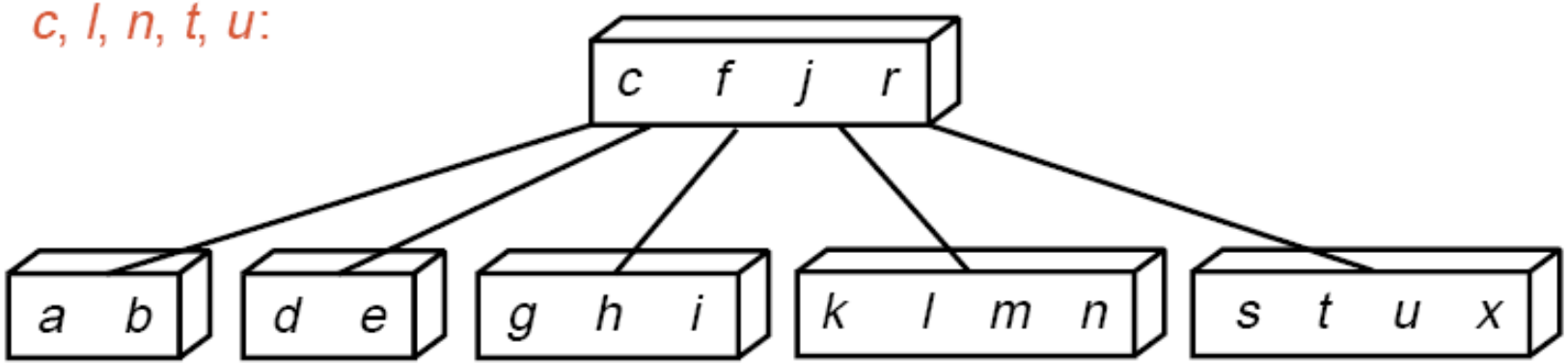


11.3.4 Insertion into a B-Tree

a g f b k d h m j e s i r x c l n t u p

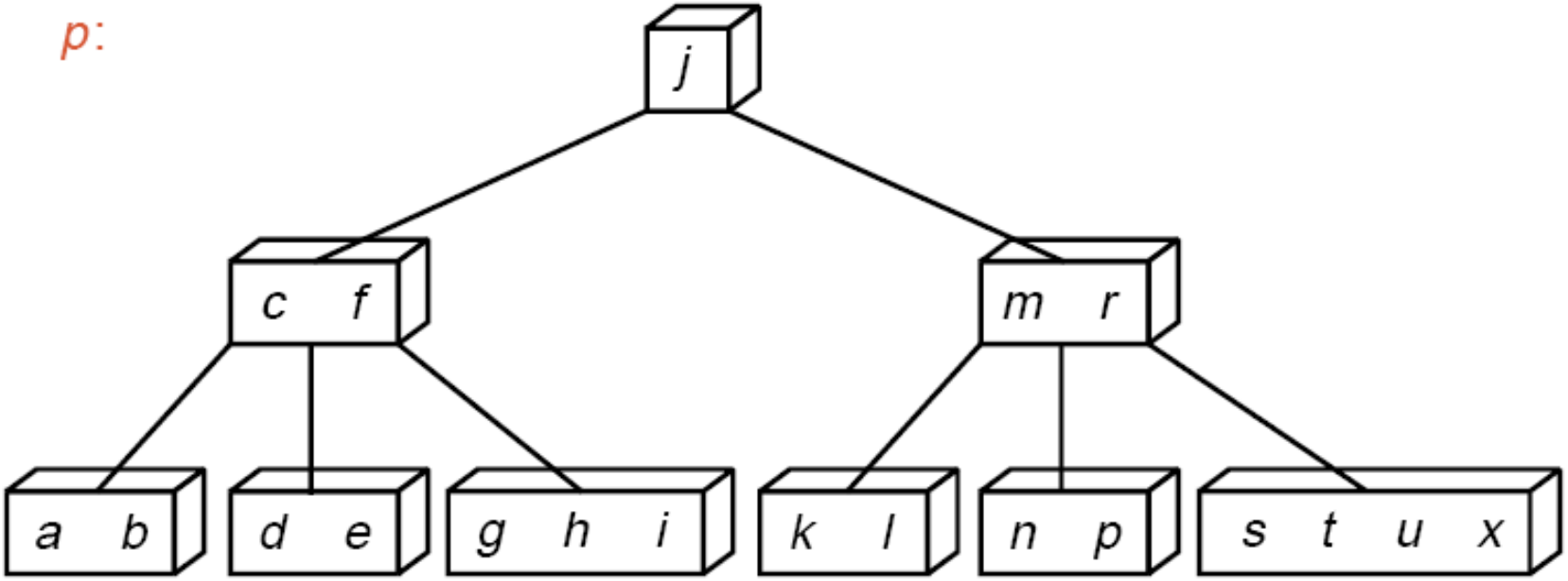
7.

c, l, n, t, u:

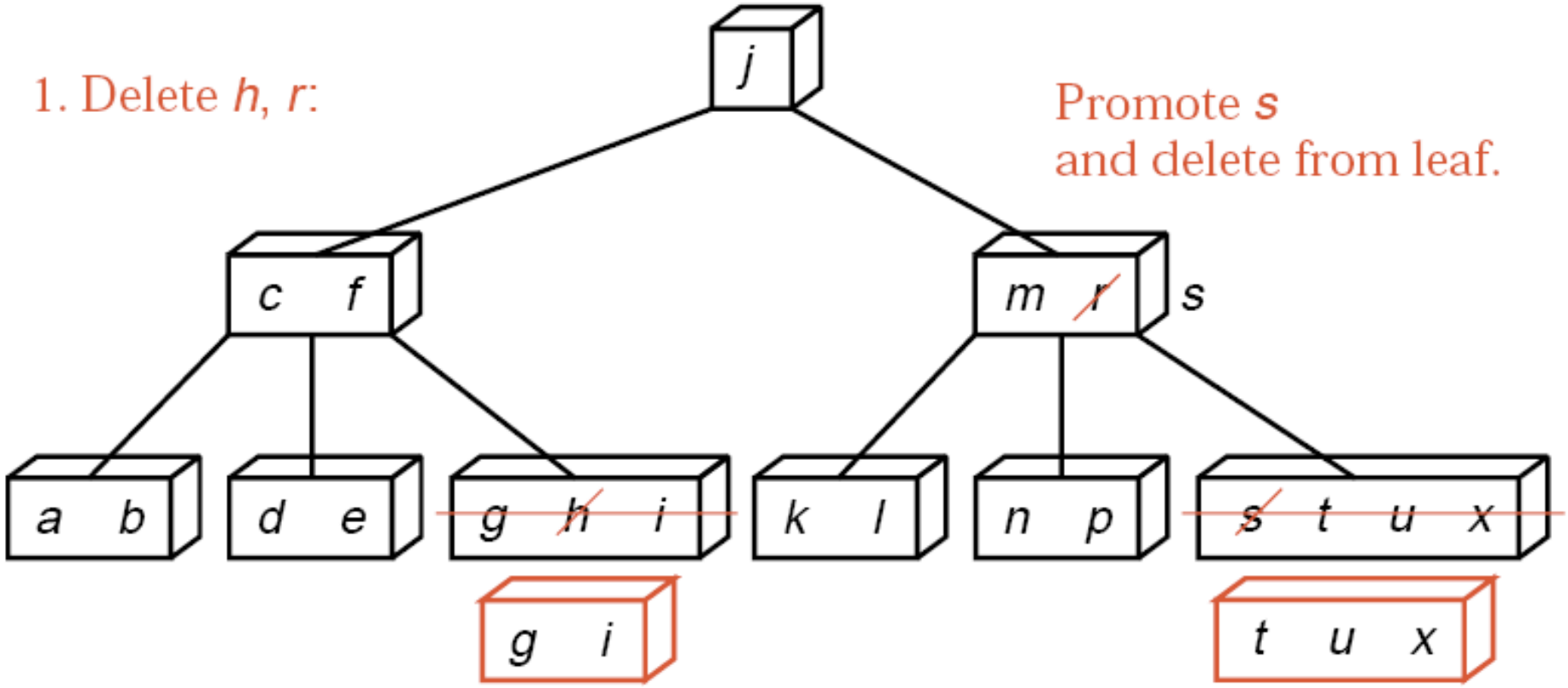


8.

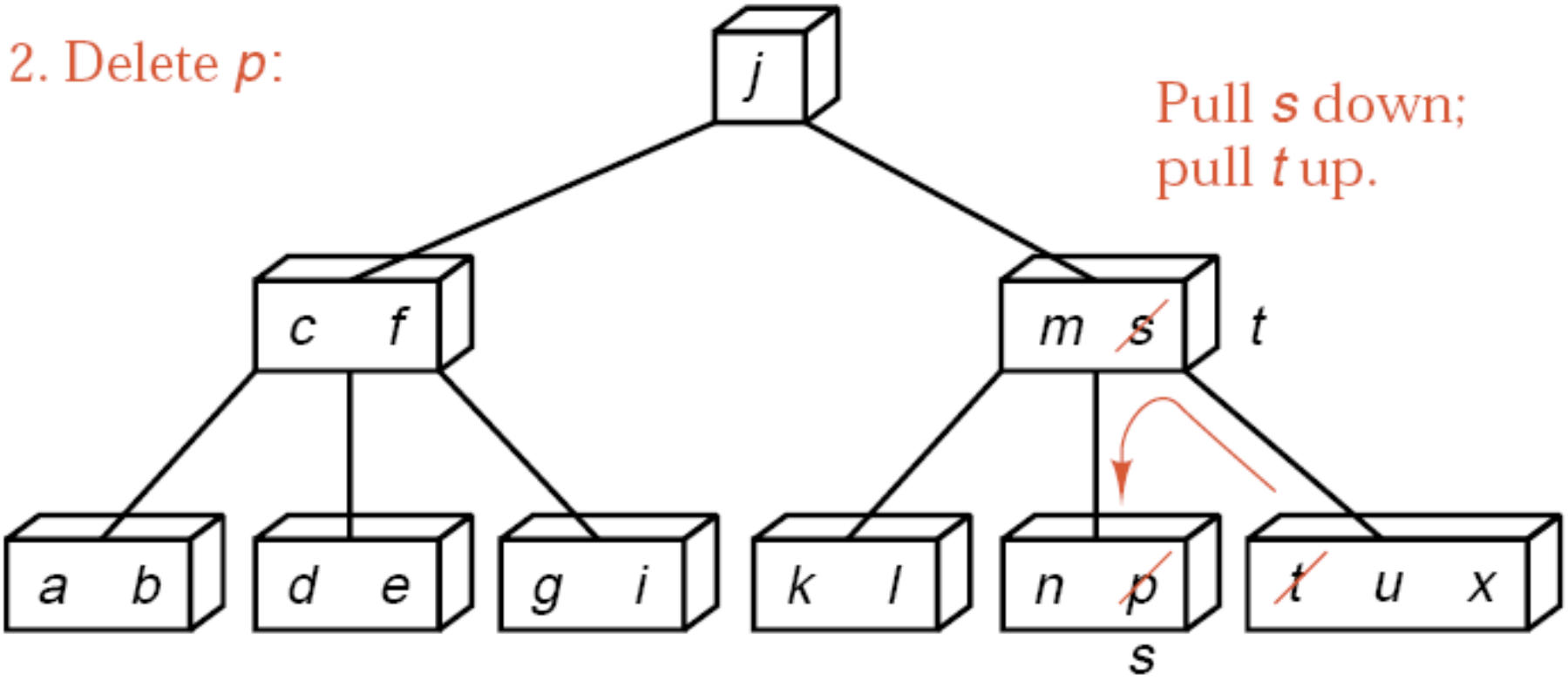
p:



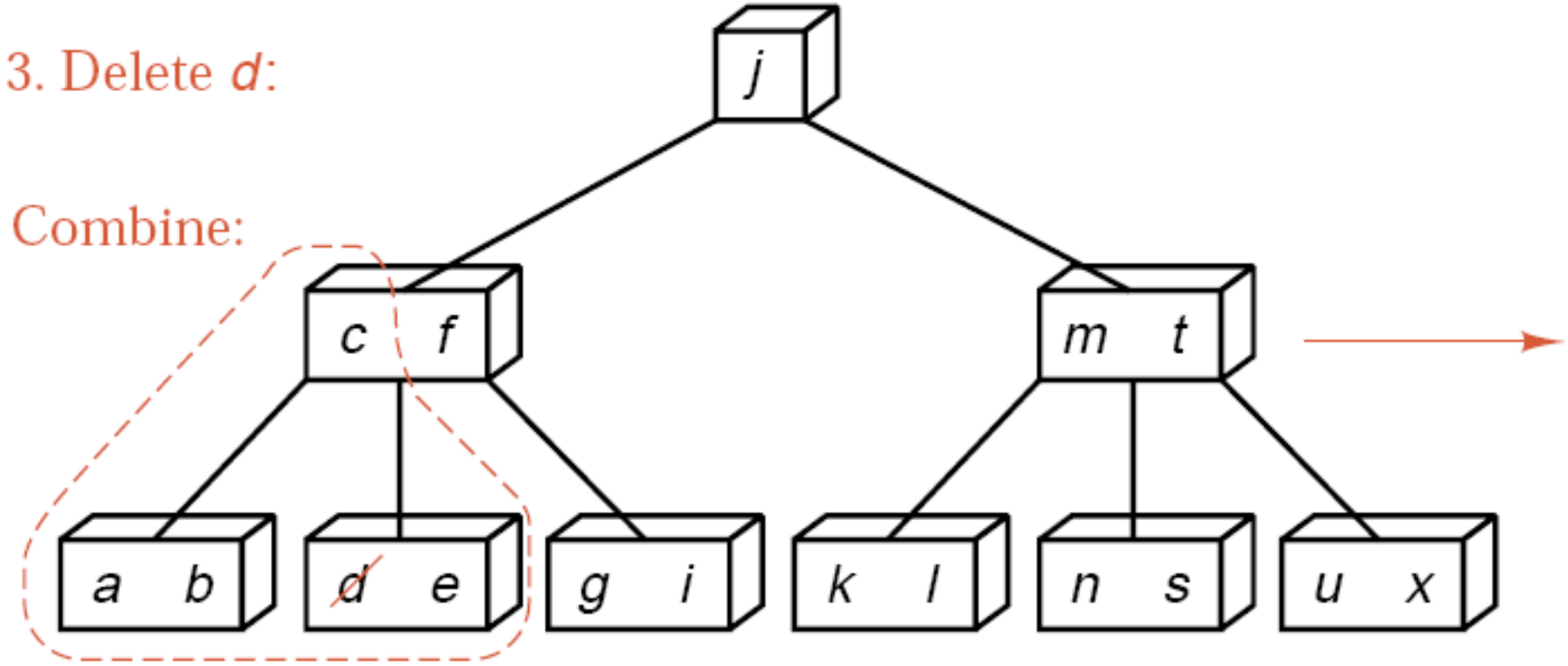
11.3.6 Deletion from a B-Tree



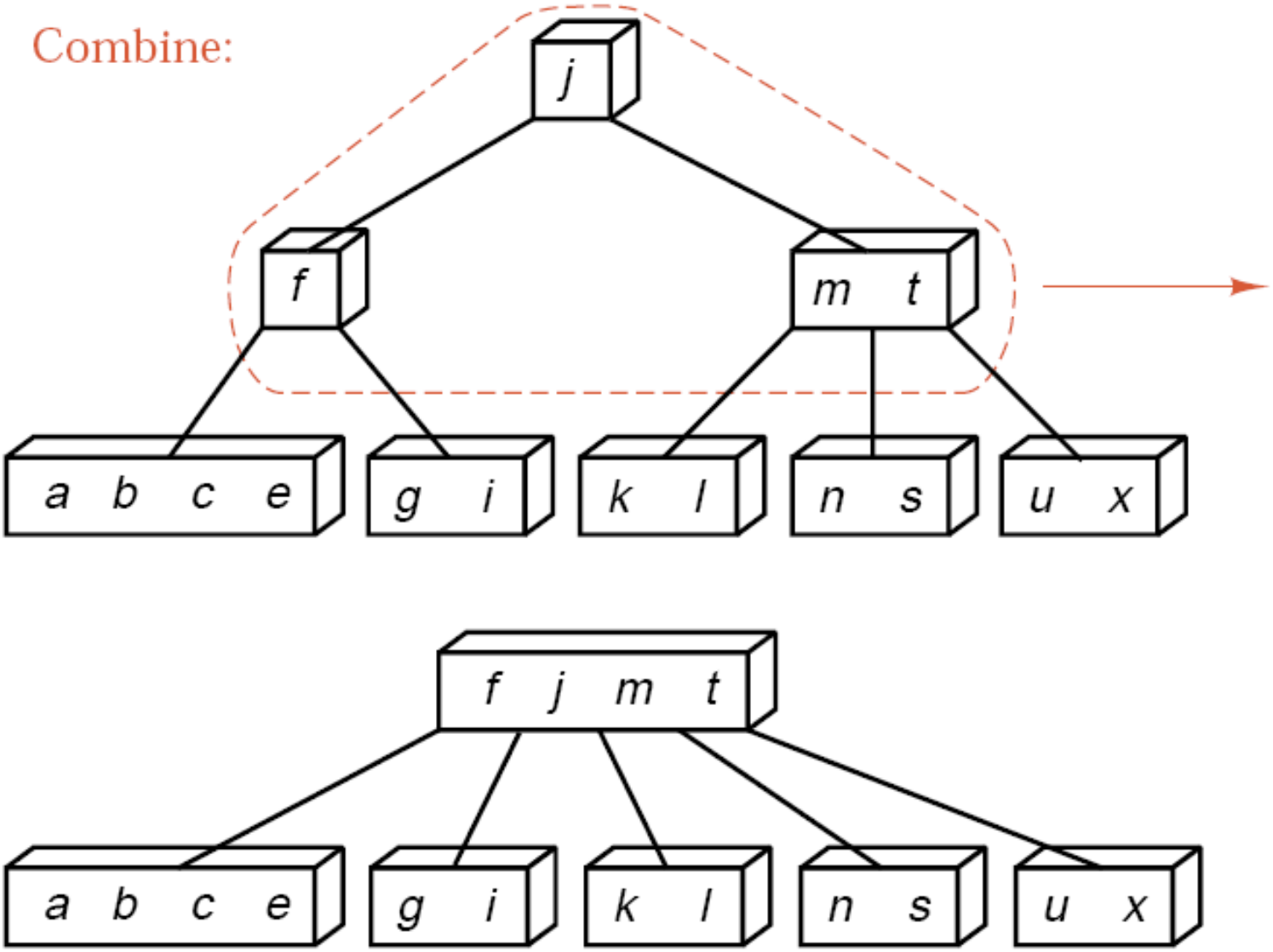
11.3.6 Deletion from a B-Tree



11.3.6 Deletion from a B-Tree



11.3.6 Deletion from a B-Tree



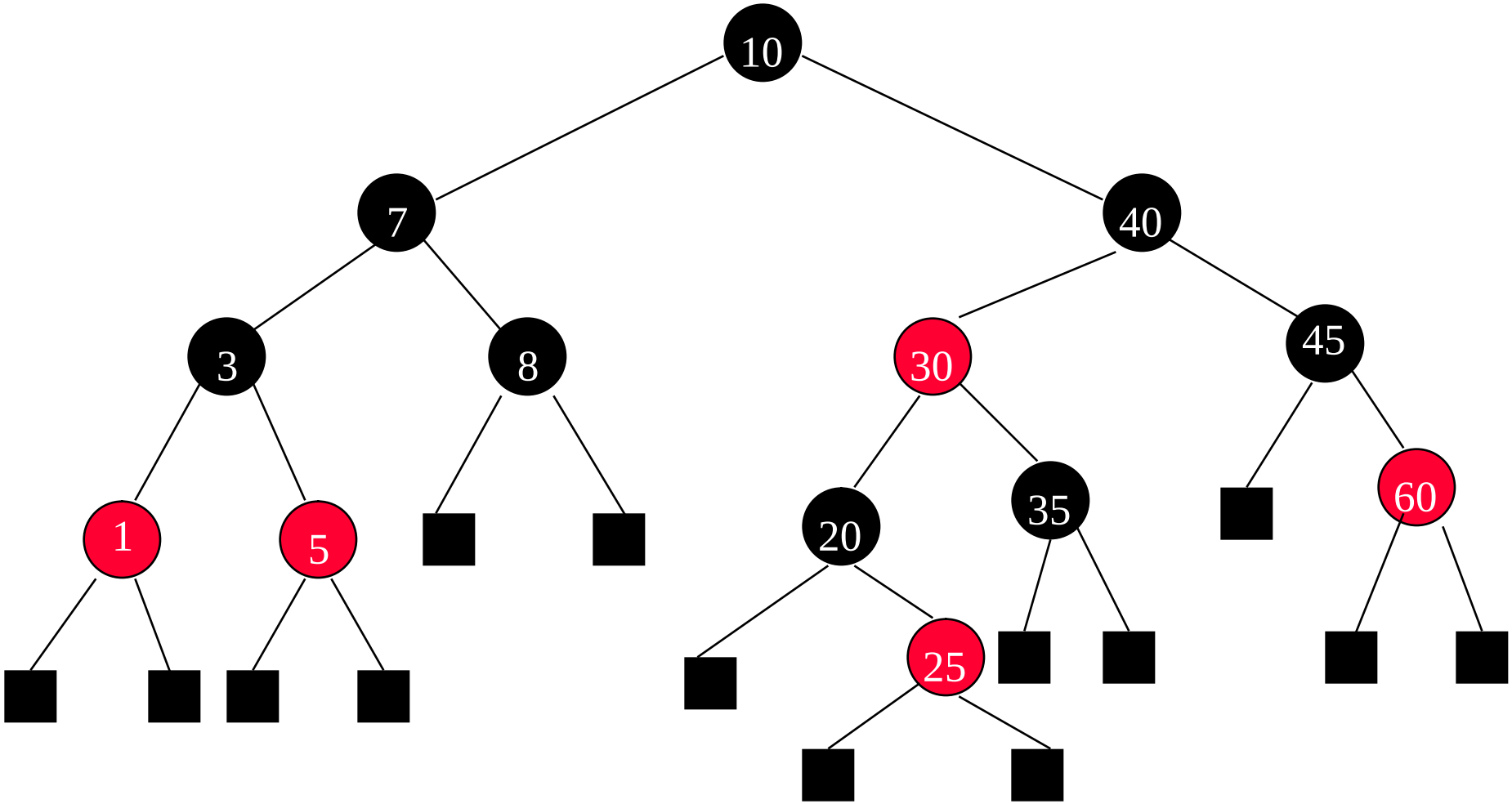
11.4 Red-Black Trees

Colored Nodes Definition

- Binary search tree.
- Each node is colored red or black.
- Root and all external nodes are black.
- No root-to-external-node path has two consecutive red nodes.
- All root-to-external-node paths have the same number of black nodes

A red-black tree is a binary search tree.

Example Red Black Tree

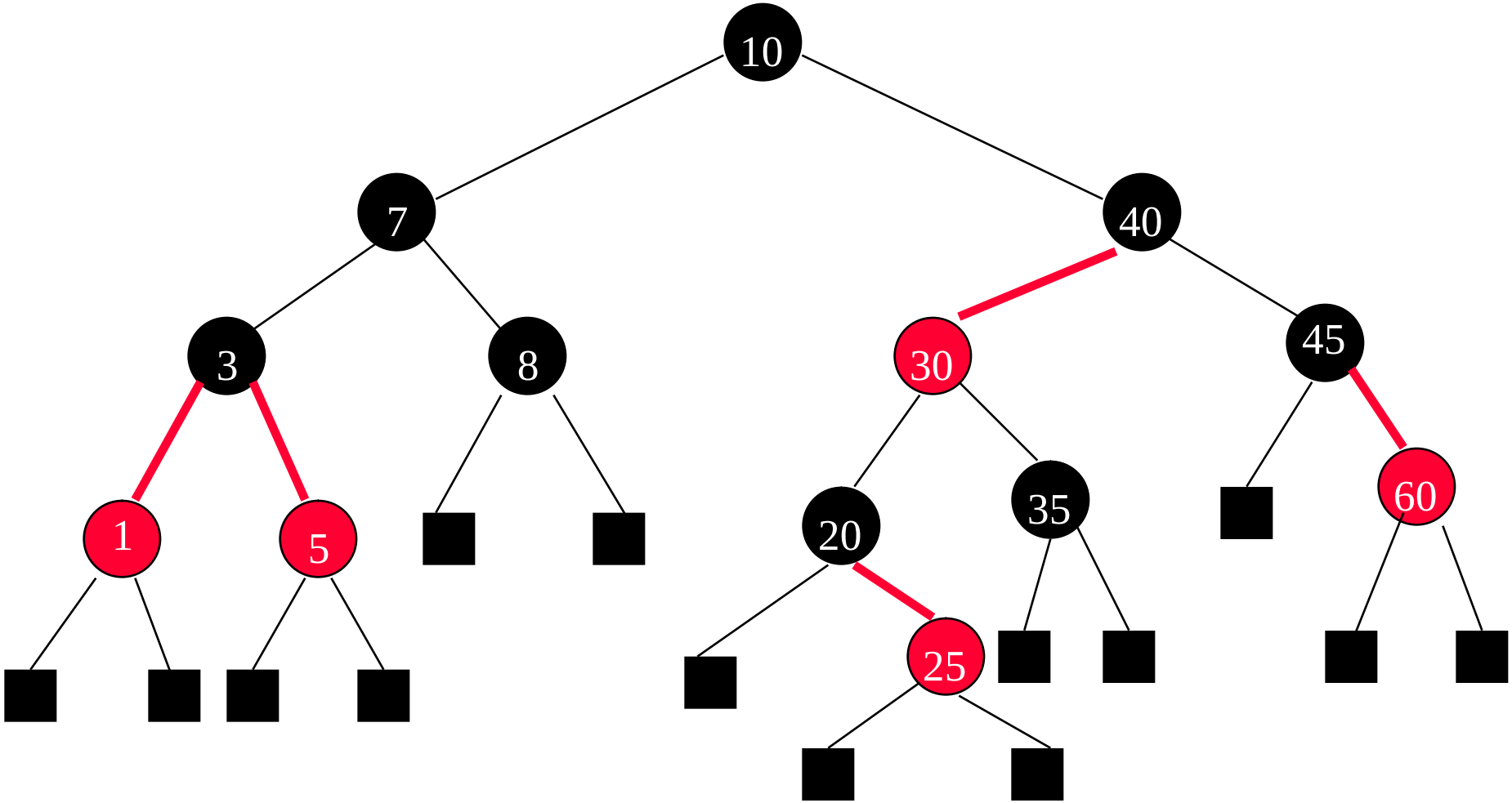


Red Black Trees

Colored Edges Definition

- Binary search tree.
- Child pointers are colored red or black.
- Pointer to an external node is black.
- No root to external node path has two consecutive red pointers.
- Every root to external node path has the same number of black pointers.

Example Red Black Tree



Red Black Tree

- Lemma 11.1 Let the length of a root-to-extrenal path be the number of pointers on the path. If P and Q are two root-to-external-node paths in a red-black tree, the length(P)$\leq$2length(Q).

proof:

1) 任一支路上黑指针的数量为*rank*(r)，且都相等；

2)最后一个指针是黑色；

3)没有两个连续的红指针,即红指针的数量一定不大于黑指针数量。

Red Black Tree

- Lemma 11.2 Let h be the height of a red-black tree (excluding the external nodes), let n be the number of internal nodes in the tree, and let r be the rank of the root.

a) $h \leq 2r$

b) $n \geq 2^r - 1$

c) $h \leq 2\log_2(n+1)$

Rank为每一条支路上黑指针的数量.如***rank***为***r***，则二叉树（不包含外部节点）的高度至少为***r*** 。外部节点的***rank***为**0.**

Red Black Tree

- The height of a red black tree that has n (internal) nodes is between $\log_2(n+1)$ and $2\log_2(n+1)$.
- Notice that the worst-case height of a red-black tree is more than the worst-case height (appraximately $1.44\log_2(n+2)$ of a AVL tree with the same number of internal nodes)