DATA STRUCTURES and ALGORITHMS in C++

FOURTH EDITION

Chapter 7: Multiway Trees

Objectives

Looking ahead – in this chapter, we'll consider

- The Family of B-Trees
- Tries

Introductory Remarks

- In chapter 6, a tree was defined as either an empty structure or one whose children were the disjoint trees t_1, … t_m
- Although we focused on binary trees and binary search trees, we can use this definition to describe trees with multiple children
- Such trees are called ***multiway trees of order m*** or ***m-way trees***
- Of more use is the case where an order is imposed on the keys in each node, creating a ***multiway search tree of order m***, or an ***m-way search tree***

Introductory Remarks (continued)

- This type of multiway tree exhibits four characteristics:
  - Each node has m children and $m - 1$ keys
  - The keys in each node are in ascending order
  - The keys in the first i children are smaller than the ith key
  - The keys in the last $m - i$ children are larger than the ith key
- *M*-way search trees play an analogous role to binary search trees; fast information retrieval and updates
- However, they also suffer from the same problems
- Consider the tree in Figure 7.1; it is a 4-way tree where accessing the keys may require different numbers of tests
- In particular, the tree is unbalanced

Introductory Remarks (continued)

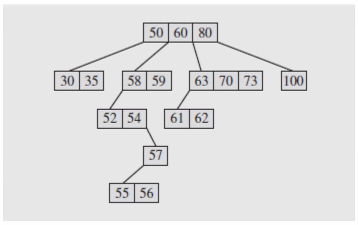


Fig. 7.1 A 4-way tree

- The balance problem is of particular importance in applying these trees to secondary storage access, which can be costly
- Thus, building these trees requires a more careful strategy

The Family of B-Trees

- The basic unit of data transfer associated with I/O operations on secondary storage devices is the block
- Blocks are transferred to memory during read operations and written from memory during write operations
- The process of transferring information to and from storage involves several operations
- Since secondary storage involves mechanical devices, the time required can be orders of magnitude slower than memory operations
- So any program that processes information on secondary storage will be significantly slowed

The Family of B-Trees (continued)

- Thus the properties of the storage have to be taken into account in designing the program
- A binary tree, for example, may be spread over a large number of blocks on a disk file, as shown in Figure 7.2
- In this case two blocks would have to be accessed for each search conducted
- If the tree is used frequently, or updates occur often, the program's performance will suffer
- So in this case the binary tree, well-suited for memory processing, is a poor choice for secondary storage access

The Family of B-Trees (continued)

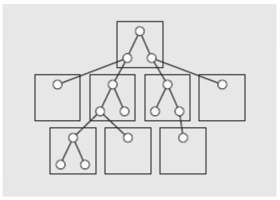


Fig. 7.2 Nodes of a binary tree can be located in different blocks on a disk

- It is also better to transfer a large amount of data all at once than to have to jump around on the disk to retrieve it
- This is due to the high cost of accessing secondary storage
- If possible, data should be organized to minimize disk access

The Family of B-Trees (continued)

- B-Trees
  - We can conclude from the previous section that the time to access secondary storage can be reduced by proper choice of data structures
  - ***B-trees***, developed by Rudolf Bayer and Edward McCreight in 1972, are one such structure; generalization of 2-3 trees by Hopcroft
  - B-trees work closely with secondary storage and can be adjusted to reduce issues associated with this type of storage
  - For example, the size of a B-tree node can be as large as a block
  - The number of keys can vary depending on key sizes, data organization, and block sizes
  - Summary: A B-Tree is a self-balancing tree that is a variation on binary search trees, but allows for more than 2 child nodes. They are particularly useful for handling data that can't fit into main memory.

The Family of B-Trees (continued)

- B-Trees (continued)
  - A ***B-tree of order m*** (you can think of **m** as the maximum number of children) is defined as a multiway search tree with these characteristics:
    - The root has a minimum of two subtrees (unless it is a leaf)
    - Each nonroot and nonleaf node stores $k - 1$ keys and k pointers to subtrees where $\lceil m/2 \rceil \leq k \leq m$
    - Each leaf node stores $k - 1$ keys, where $\lceil m/2 \rceil \leq k \leq m$
    - All leaves are on the same level
  - Based on this definition, a B-tree is always at least half-full, has few levels, and is perfectly balanced
  - This definition is also based on the order of the B-tree specifying the maximum number of children; it can also specify the minimum, in which case the number of keys and pointers change

The Family of B-Trees (continued)

- B-Trees (continued)
  - The value for *m* is usually large so the information in one page or block of secondary storage can fit in one node

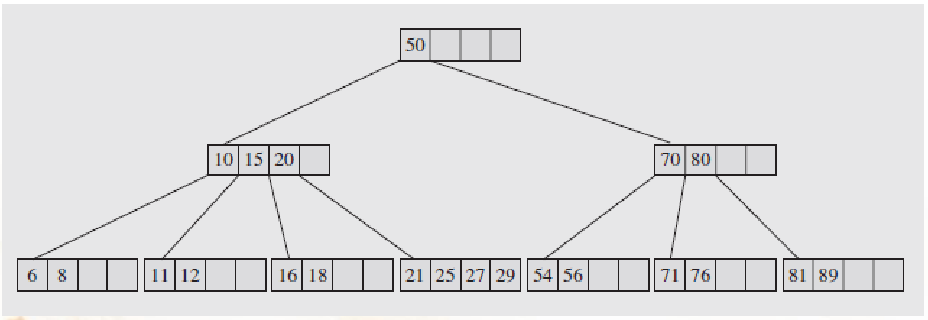
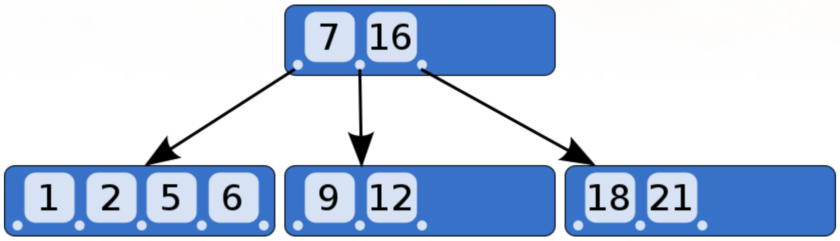


Fig. 7.4 A B-tree of order 5 shown in an abbreviated form

The Family of B-Trees (continued)



M=5

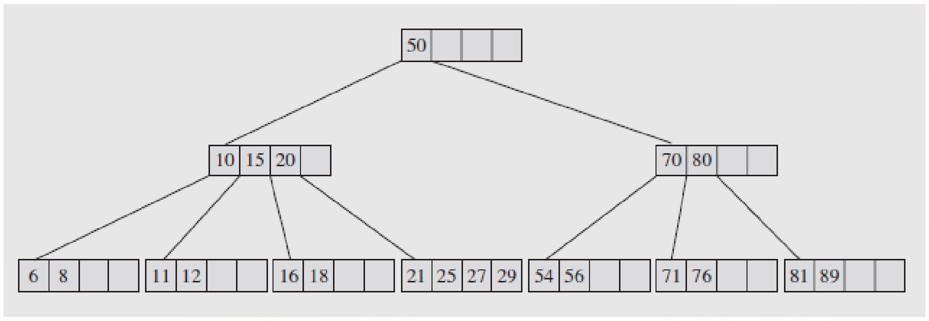
The Family of B-Trees (continued)

- B-Trees – Searching in a B-Tree (continued)
  - Finding a key in a B-tree is a straightforward extension to searching a regular BST; an algorithm for this is:

```
BTreeNode *BTreeSearch(keyType k, BTreeNode *node) {
  if (node != 0) {
    for (i=1, i <= node->keyTally && node->keys[i-1] < K; i++);
    if (i > node->keyTally || node->keys[i-1] > K)
      return BTreeSearch(K, node->pointers[i-1]);
    else return node;
  }
  else return 0;
}
```

  - The worst case occurs when the tree has the smallest number of pointers per nonroot node ($q = \lceil m/2 \rceil$), and the search has to reach a leaf

Searching a B-Tree

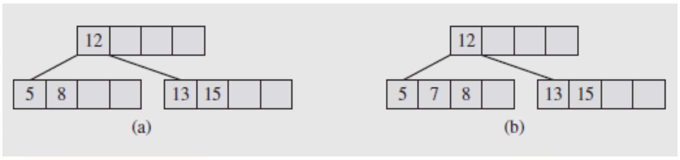


The Family of B-Trees (continued)

- B-Trees – Searching in a B-Tree (continued)
  - For a sufficiently large order, *m*, the height is small
  - If *m* = 200 and *n* (the number of keys) is 2,000,000, for instance, the value of $h \leq 4$
  - This means that even in the worst case, finding a key in the B-tree would require 4 seeks
  - In addition, if the root of the tree can be retained in memory, only three seeks need be performed in secondary storage

Inserting into a B-Tree

- B-Trees – Inserting a Key Into a B-Tree (continued)
  - Search from the root until you find the node for the new item
  - Case 1 : If the node is not at capacity, just add the item in the proper sorted order



A B-tree (a) before and (b) after insertion of the number 7 into a leaf that has available cells

Inserting into a B-Tree

- B-Trees – Inserting a Key Into a B-Tree (continued)
  - Case 2 : The node is at capacity. Split it into two, and push the middle value up into the parent node.

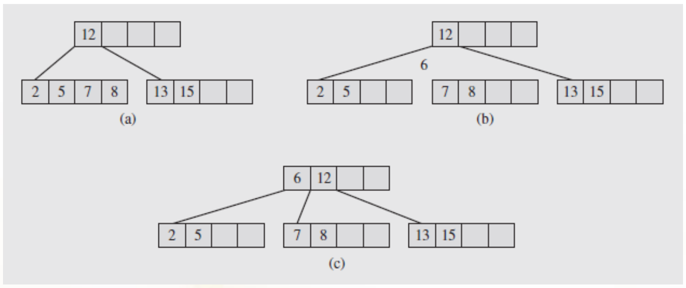
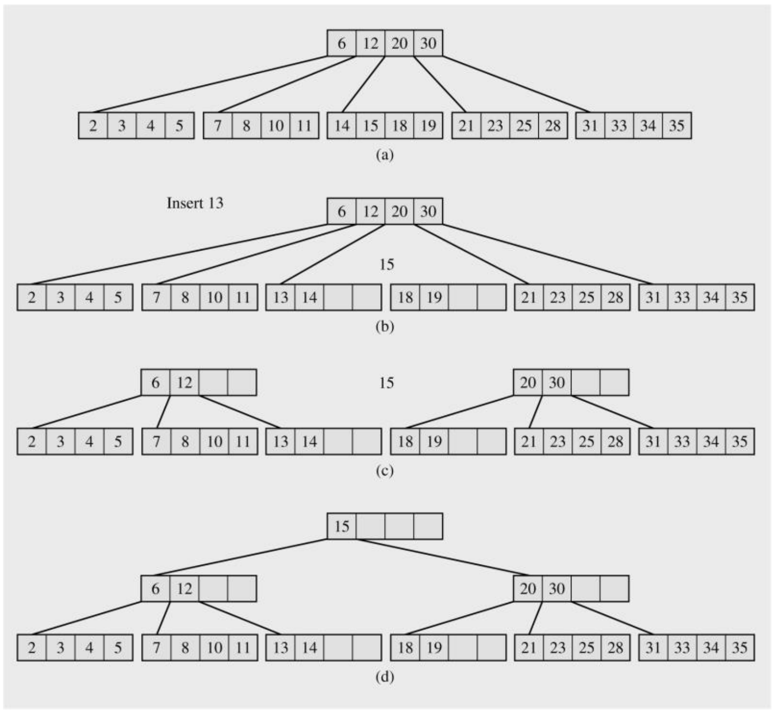


Fig. 7.6 Inserting the number 6 into a full leaf

Inserting into a B-Tree

- B-Trees – Inserting a Key Into a B-Tree (continued)
  - Case 3 : The node is at capacity. Split it into two, and push the middle value up into the parent node. But if the parent is full, we have to recursively repeat and split the parent, possibly creating a new root node.



Inserting into a B-Tree

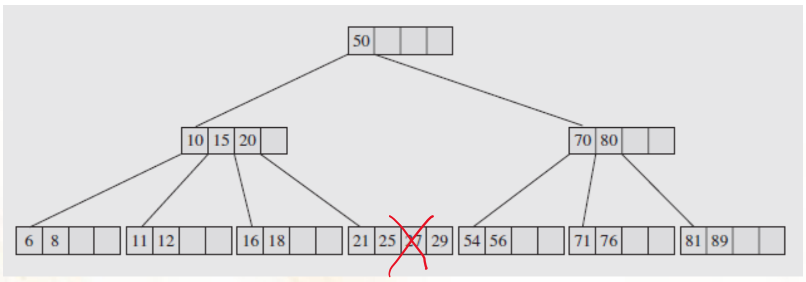
- Visualization
- https://www.cs.usfca.edu/~galles/visualization/BTree.html

The Family of B-Trees (continued)

- B-Trees – Inserting a Key Into a B-Tree (continued)
  - Figure 7.8 on page 318 shows how a B-tree grows during the insertion of new keys
  - Notice how it remains perfectly balanced at all times
  - There is a variation of this called ***presplitting*** which can be used
  - In this case, as a search is conducted top-down for a specific key, any full nodes are split; this way no split is propagated upward
  - Complexity is O($\log_m n$) but could have some larger constant value for operations needed to manipulate keys per node

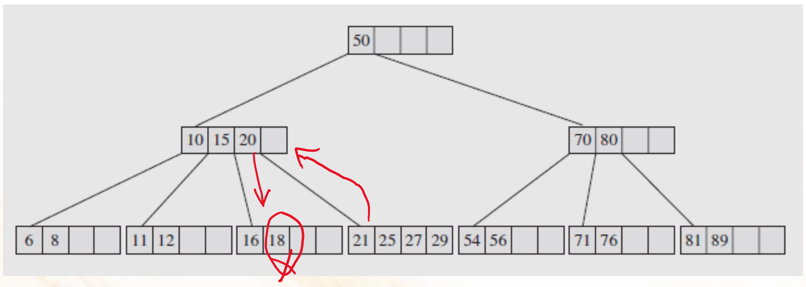
Deleting a Key from a B-Tree

- B-Trees – Deleting a Key From a B-Tree (continued)
  - Deletion is basically the reverse of insertion, although there are several special cases to be considered
  - Case 1: The key is in a leaf and there is still m/2 keys after deleting it, then we can just delete it straight up



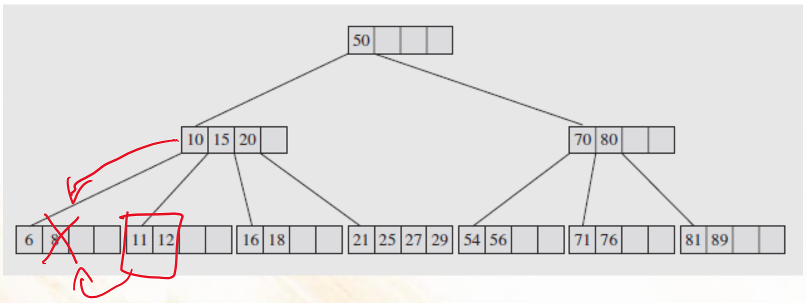
Deleting a Key from a B-Tree

- B-Trees – Deleting a Key From a B-Tree (continued)
  - Case 2a: The key is in a leaf and there are no longer m/2 keys after deleting it.
  - Find the closest key in a sibling, if the sibling can give up a key, rotate it to the parent and move the parent key down in place of the deleted key



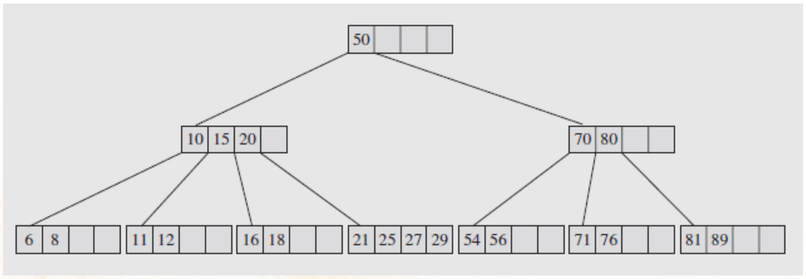
Deleting a Key from a B-Tree

- B-Trees – Deleting a Key From a B-Tree (continued)
  - Case 2b: The key is in a leaf and there are no longer m/2 keys after deleting it.
  - Find the closest key in a sibling, if the sibling can't give up a key, merge the nodes, move parent down, and then delete the key



Deleting a Key from a B-Tree

- B-Trees – Deleting a Key From a B-Tree (continued)
  - Case 3: The key is in an internal node
  - Skipping details, but more special cases that involve taking a key from a child with enough keys and moving it up, or merging child nodes together



The Family of B-Trees (continued)

- B^*-Trees
  - Since each B-tree node is a block of secondary storage, accessing the node means accessing secondary memory
  - This is time expensive, compared to accessing keys in nodes in main memory
  - So if we can cut down on the number of nodes that are created, performance may improve
  - A ***B***-tree* is a variant of a B-tree in which the nodes are required to be two-thirds full, with the exception of the root
  - Specifically, for a B-tree of order *m*, the number of keys in all nonroot nodes is *k* for $\lfloor (2m-1)/3 \rfloor \le k \le m - 1$
  - Higher utilization than B-tree

The Family of B-Trees (continued)

- B^+-Trees
  - Since nodes in B-trees represent pages or blocks, the transition from one node to another requires a page change, which is a time consuming operation
  - If for some reason we needed to access the keys sequentially, we could use an inorder traversal, but accessing nonterminal nodes would require substantial numbers of page changes
  - For this reason, the ***B^+-tree*** was developed to improve sequential access to data
  - Any node in a B-tree can be used to access data, but in a B^+-tree only the leaf nodes can refer to data
  - The internal nodes, called the ***index set***, are indexes used for fast access to the data

The Family of B-Trees (continued)

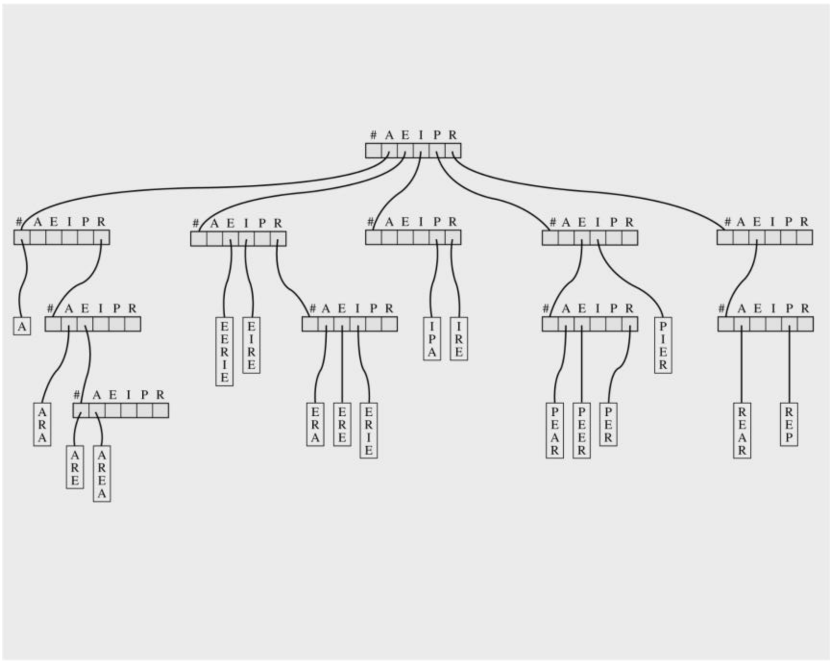
- B^+-Trees (continued)
  - The leaves are structured differently from the other nodes, and usually are linked sequentially to form a ***sequence set***
  - That way scanning the list of leaves produces the data in ascending order
  - So the name B^+-tree is appropriate, since it is implemented as a B-tree plus a linked list

Other Trees

- K-d trees
- Bit trees
- R-trees
- 2-4 trees

Tries

- As we've seen, traversing binary trees is accomplished through full key comparisons
- Prefix B-trees, however, showed us that only a portion of the key is actually necessary to traverse a path
- However, the determination of prefixes, and maintaining them through insertions and deletions, complicated matters
- Trees that use parts of keys to effect traversals and searches are called ***tries*** (pronounced "try")
- The keys in tries are sequences of characters, and the trie is structured around these as opposed to entire keys
- In the examples, the keys will be constructed from five capital letters – A, E, I, P, and R



Tries (continued)

- A trie that has words stored in vertical rectangles is shown in Figure 7.38 on page 365
- These rectangles represent the leaf nodes of the trie
- The internal nodes can be looked at as arrays of pointers to subtries
- So at each level of the trie, *i*, we check the *i*th letter of the key that is being processed
- If the pointer at that position is null, then either a search has failed, or an insertion can be done
- Otherwise, we continue the process until a leaf which contains the key is found

Tries (continued)

- So searching for a word is a matter of using the letters in the word to search through the trie until we locate the leaf
- There is a problem with this, however, what happens is a word is a prefix of another word, also in the tree?
- In Figure 7.38, for example, we store the words “ARE” and “AREA”
- Without some mechanism to distinguish them, a search for “ARE” will not stop, because there is a pointer to another node containing the final “A”
- This is handled by using a special character not used in any word (in this example, the “#” symbol) in each node

Tries (continued)

- As we process the word, we check the link associated with “#” to see if the word is linked to it
- After processing the “A”, “R”, and “E” or “ARE”, we are out of letters, and the “#” link in that node points to the word
- So we can conclude the word is in the tree and the search is successful
- This raises another issue – do we need to store entire words in the trie?
- In the case of “ARE”, when we reach the node and see the “#” link isn’t null, we are performing an unnecessary step
- Again, prefix B-trees suggest a solution, and we could have the leaf nodes store only the unprocessed suffixes of words

Tries (continued)

- The examples have used only five letters, but in a realistic setting there would be 27, one for each letter plus "#"
- Since the height of the trie is based on the length of the longest prefix, in English this isn't a long string
- Typically, 5 – 7 searches will be adequate; this was true for 100,000 English words
- A perfectly balanced binary tree for 100,000 words has a height of 17; this is about twice the number of searches in a corresponding trie
- So for speed of access in applications, a trie makes a good choice

Tries (continued)

- Since a trie has two types of nodes, insertion is somewhat more complicated than in a binary tree
- The algorithm on page 367 demonstrates this
- However, in comparing tries to binary trees, we find the order of key insertion is irrelevant, whereas this dictates the shape of the binary tree
- This doesn't absolve tries of structure problems, though
- Tries can become skewed by the prefixes of the words inserted
- In fact, the length of the longest identical prefix in two words determines the height of the trie

Tries (continued)

- The height of a trie is the length of the longest identical prefix plus one for a level to discriminate between the two entries, plus one for the leaf
- In Figure 7.8, the trie is of height five, because the longest identical prefix, "ARE", is three letters long
- The biggest problem posed by tries is space requirements
- A given node may only point to a few entries, but still needs to store pointers for all 27 possibilities
- Because this is a main memory issue, there is a need to reduce wasted space
- One possibility would be to store only those pointers in use, as shown in Figure 7.39

Tries (continued)

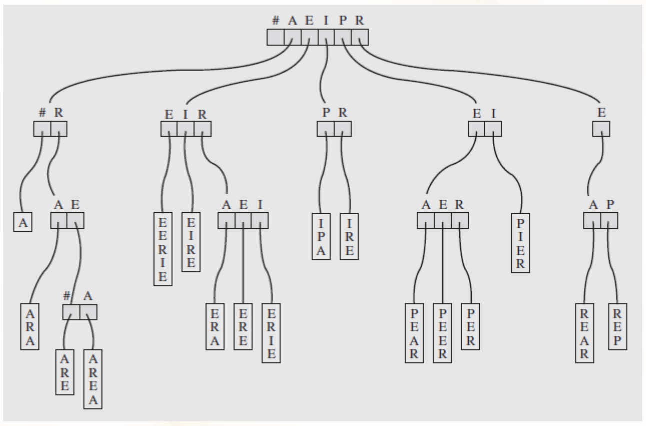
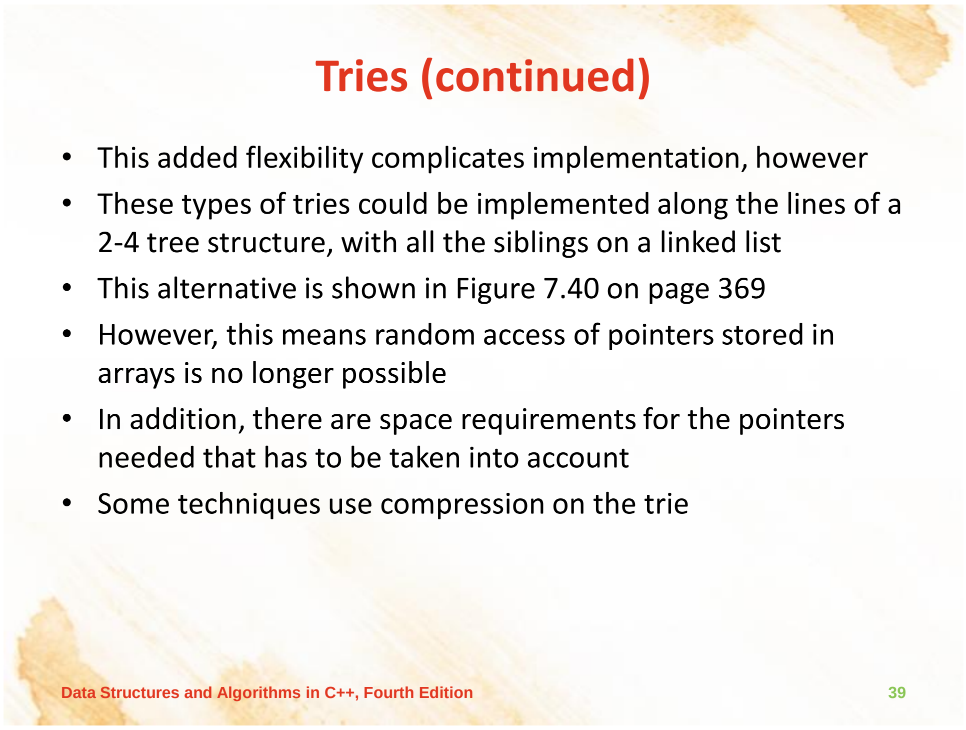


Fig. 7.39 The trie in Figure 7.38 with all unused pointer fields removed



Tries (continued)

- This added flexibility complicates implementation, however
- These types of tries could be implemented along the lines of a 2-4 tree structure, with all the siblings on a linked list
- This alternative is shown in Figure 7.40 on page 369
- However, this means random access of pointers stored in arrays is no longer possible
- In addition, there are space requirements for the pointers needed that has to be taken into account
- Some techniques use compression on the trie