COMP225: Algorithms and Data Structures

Advanced Trees

Mark Dras

Mark.Dras@mq.edu.au

E6A380

Outline

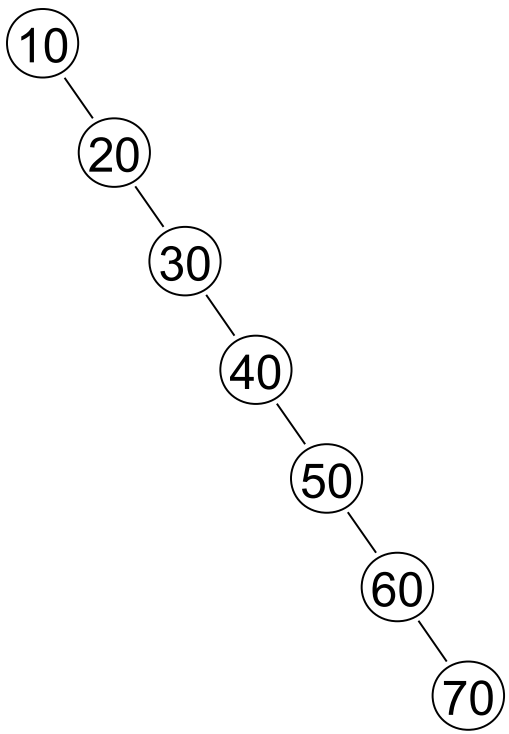
- AVL Trees
- B-Trees
- External Storage

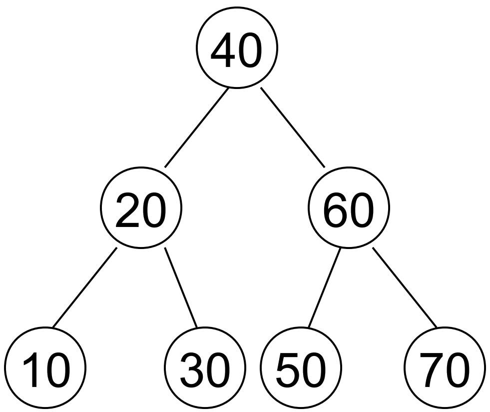
Balanced Search Trees

- Binary Search Trees can be efficient for retrieving data
  – the longest path that needs to be searched is the height of the tree
  – in a balanced tree, the height is $\lceil \log_2(n+1) \rceil$
  – the worst case is when the tree linear, so height is n

Balanced Search Trees

- Insert 10, 20, 30, 40, 50, 60, 70, vs 40, 20, 60, 10, 30, 50, 70





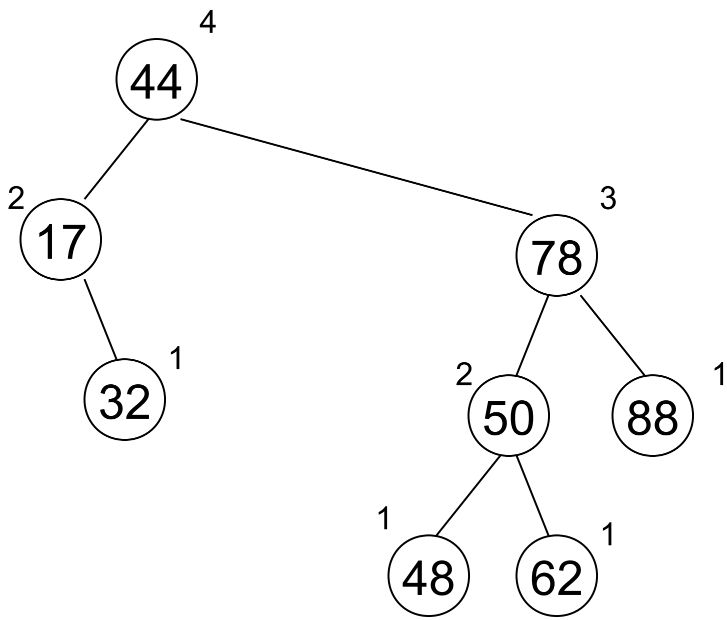
Balanced Search Trees

- Insertions and deletions can change the shape away from being balanced
- So, there are other types of trees with the same ordering properties as Binary Search Trees which remain balanced

AVL trees

- Start with notion of the height of a tree
  - height of leaf node is 1
  - height of non-leaf node is 1 + max of its children
- Can define a balanced tree by **height-balance** property:
  - for every node v of T, the heights of the children differ by at most 1
  - same definition for individual nodes
- Idea is to rebalance the tree whenever an insertion or a deletion occurs

AVL tree

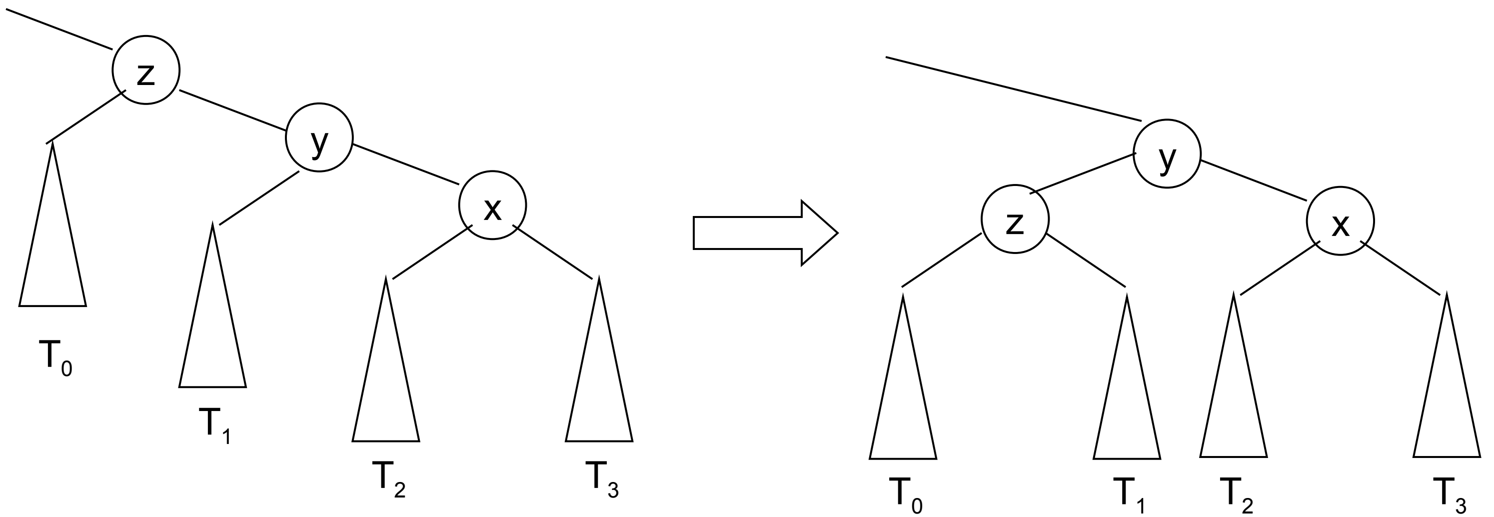


AVL: insertion

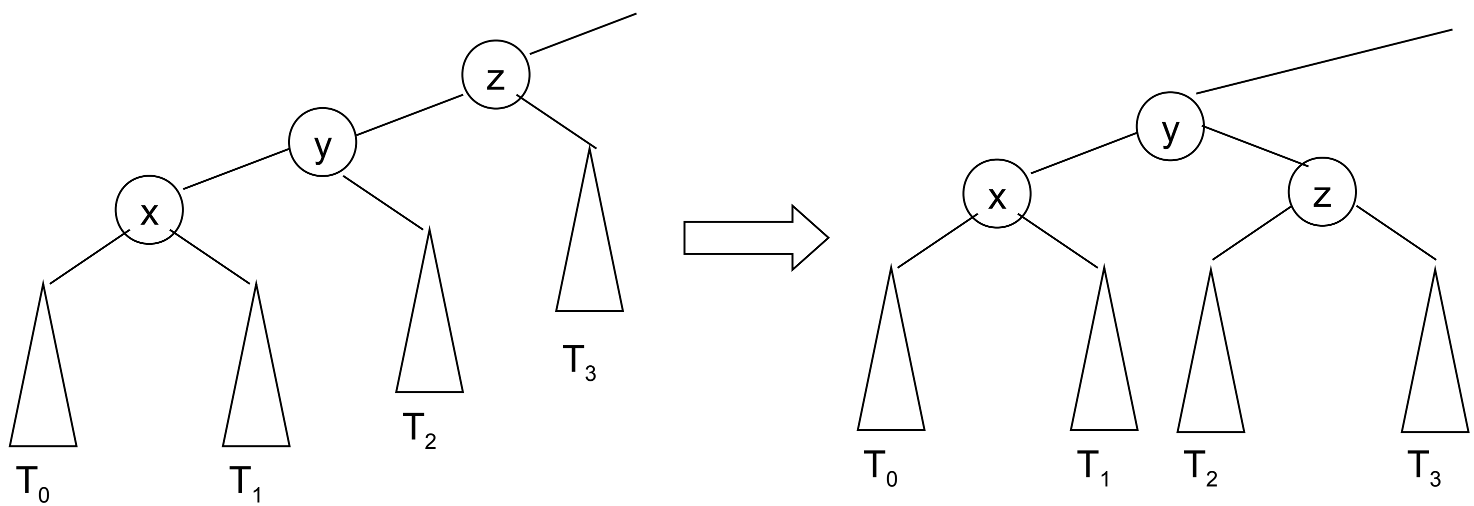
- Insert as normal for a BST
- check whether tree is unbalanced
  - start from inserted node w
  - move up the tree finding first unbalanced node z
  - restructure at node z via "single rotation" or "double rotation"
  - use y (child of z with greater height) and x (child of y with greater height)

AVL: single rotations

S1:

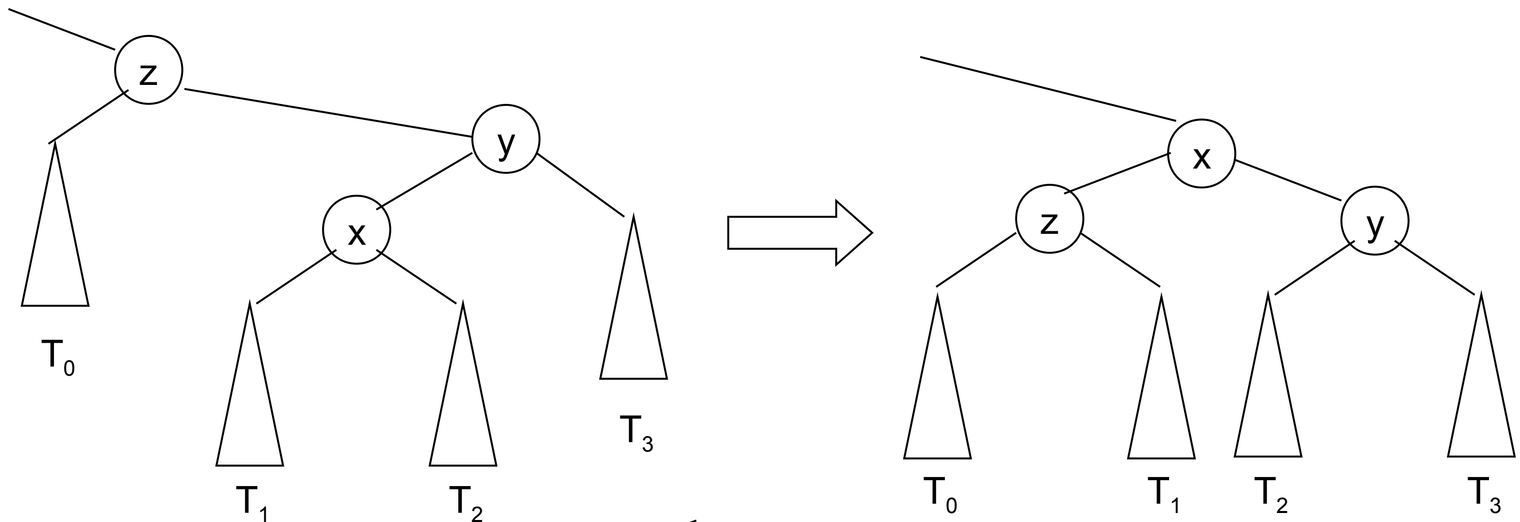


S2:

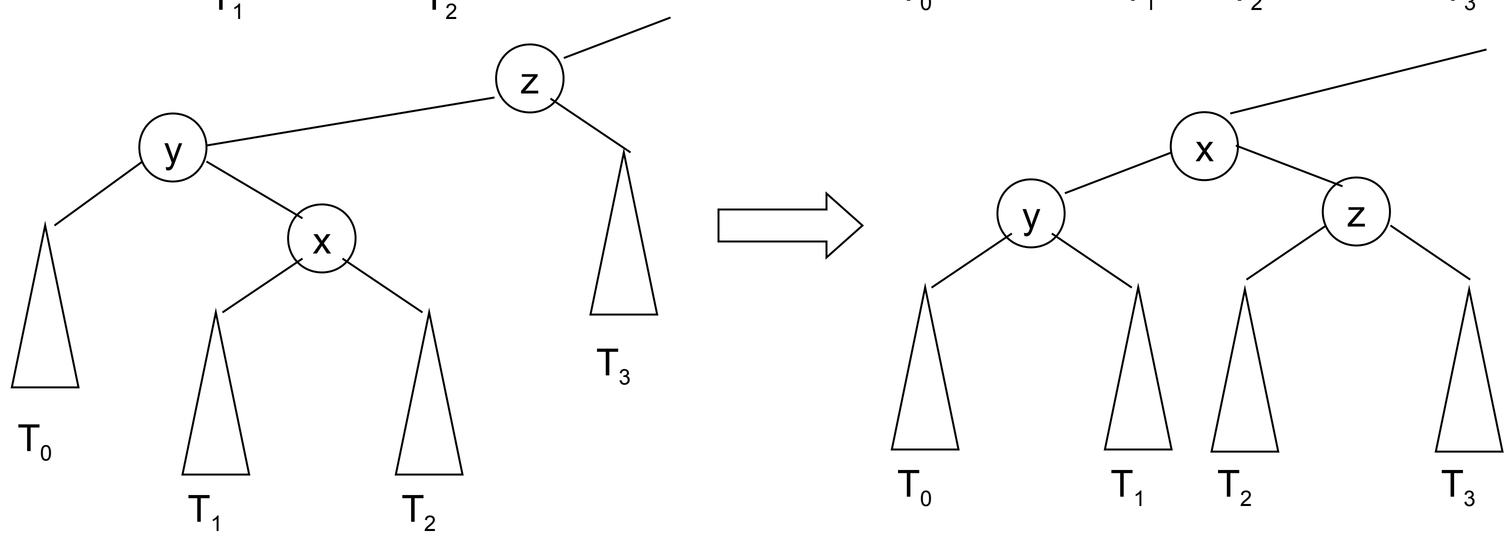


AVL: double rotations

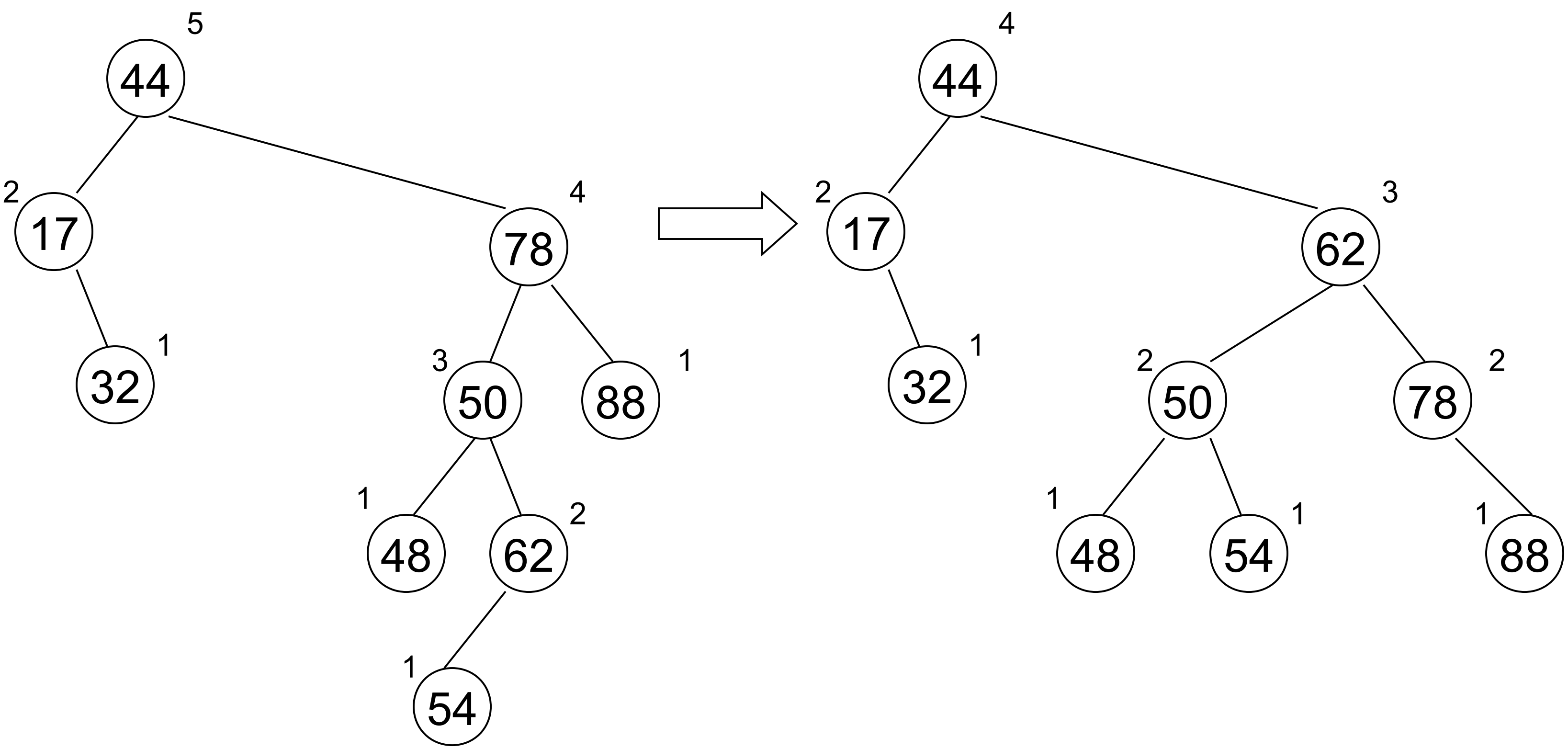
D1:



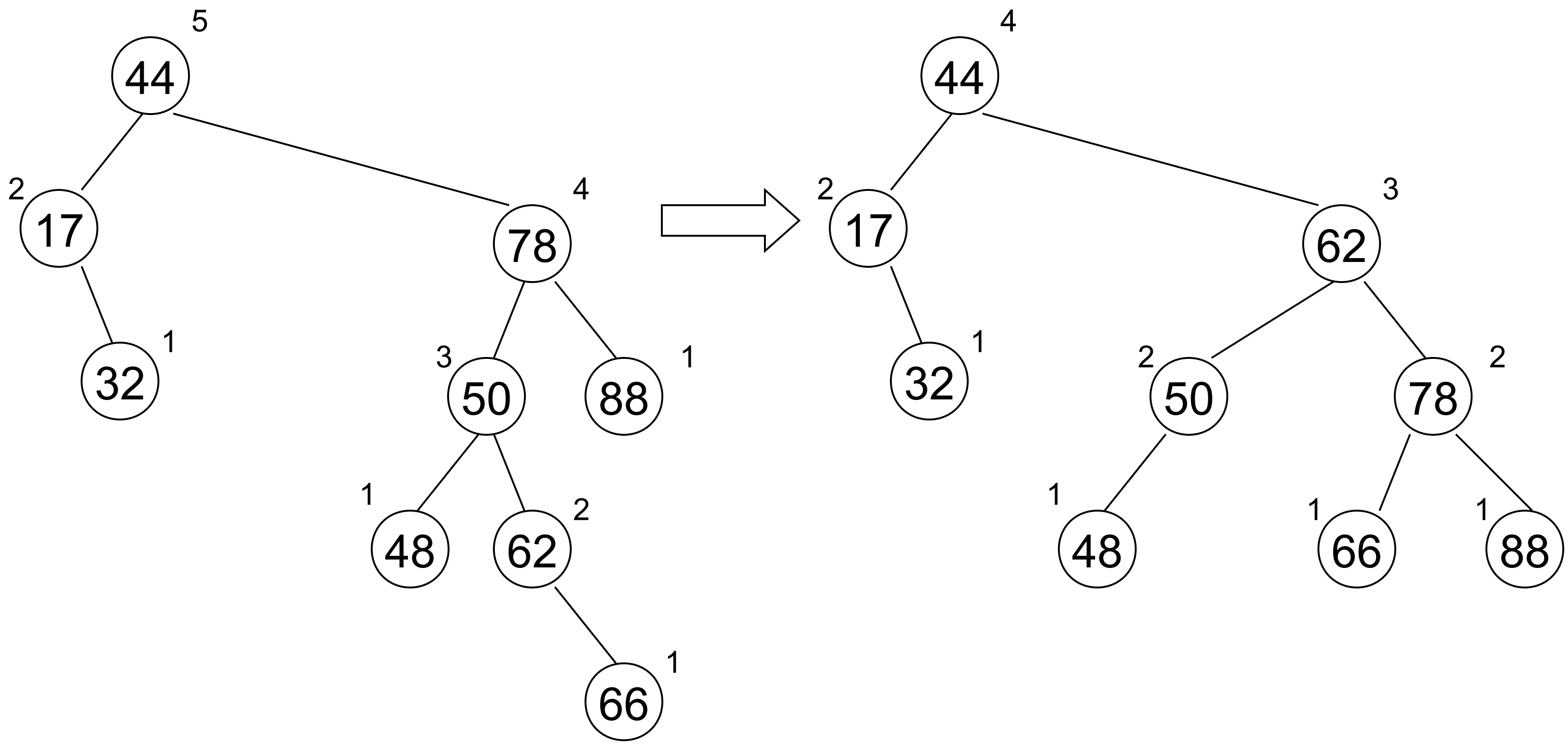
D2:



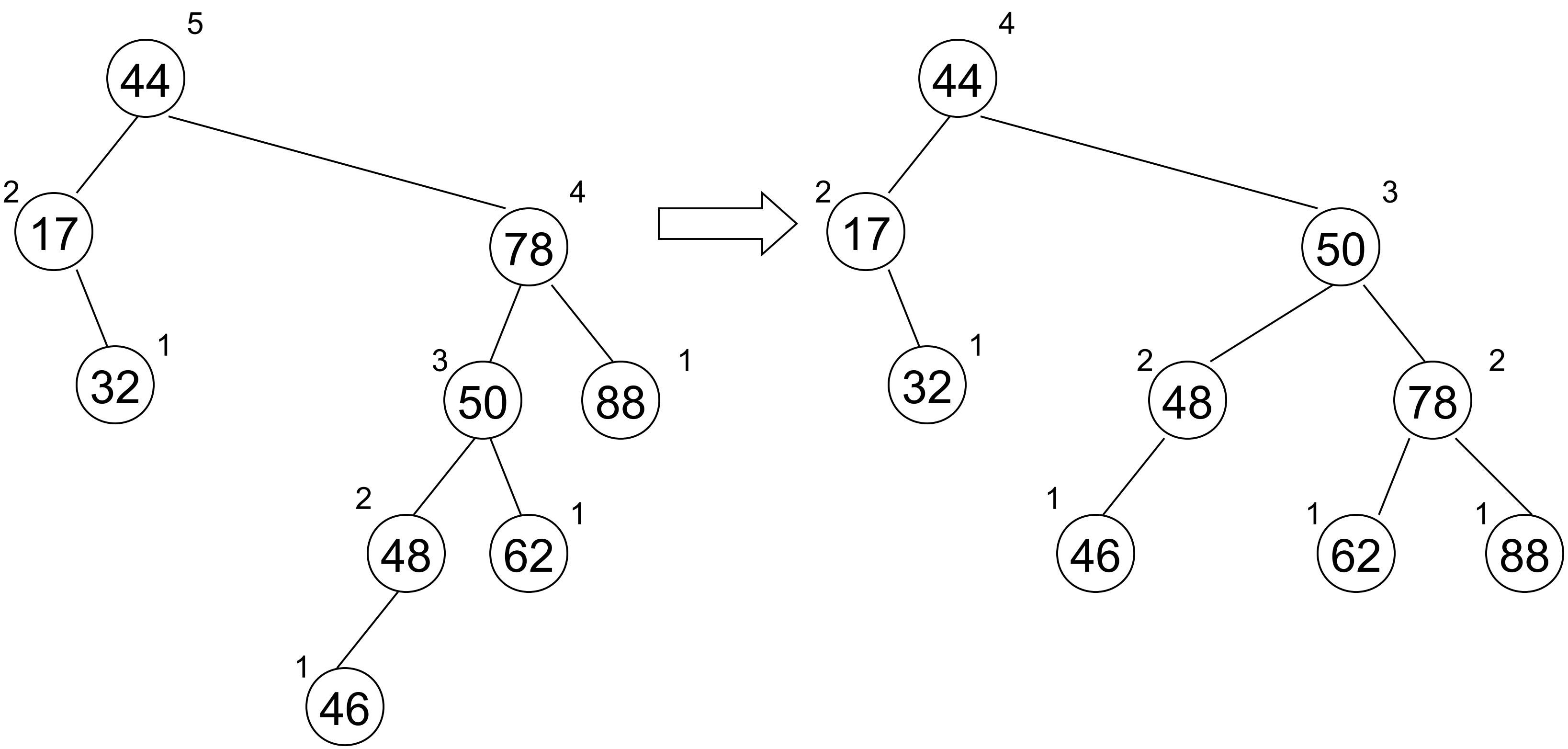
AVL: insert example (D2)



AVL: insert example (D2)



AVL: insert example (S2)

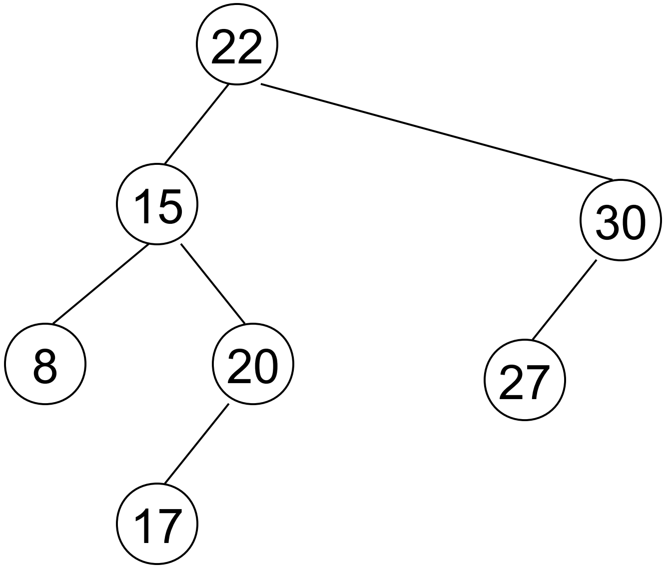


AVL: insertion

- Only need one rotation to rectify height-balance

AVL: insert exercise

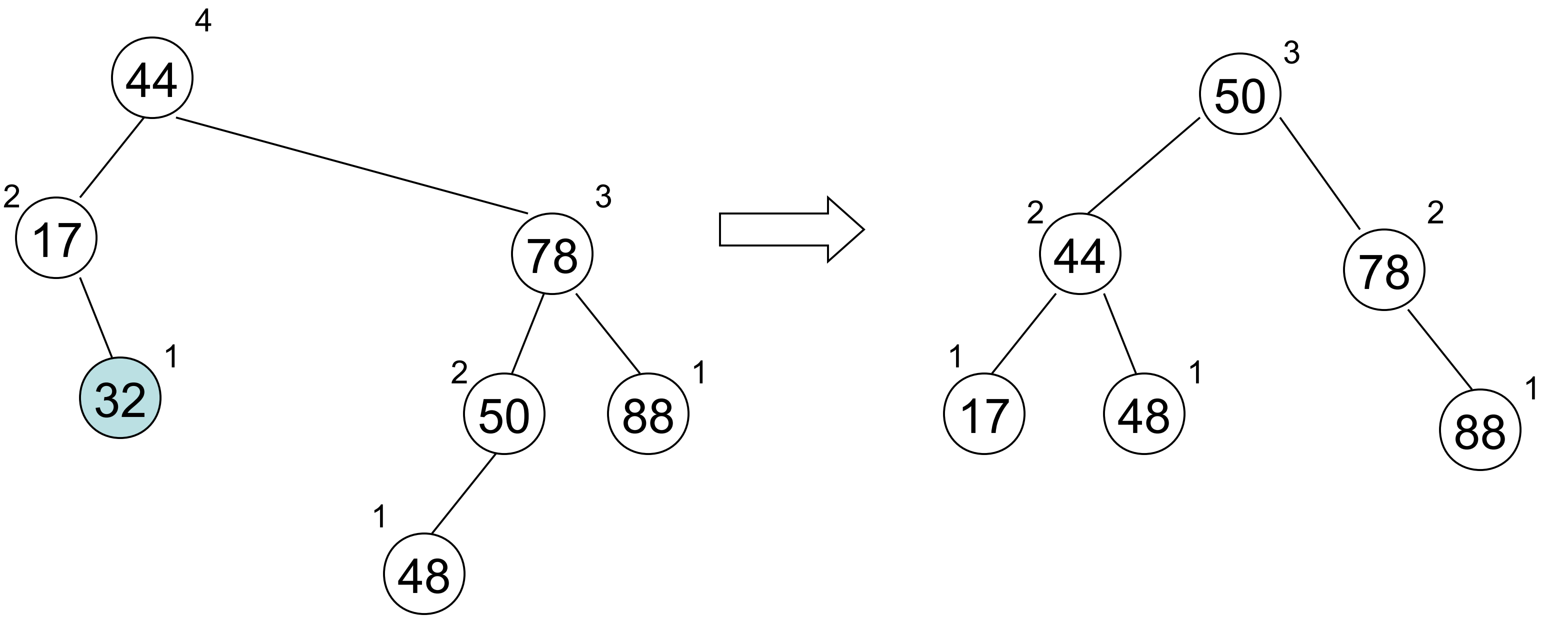
- Insert 16



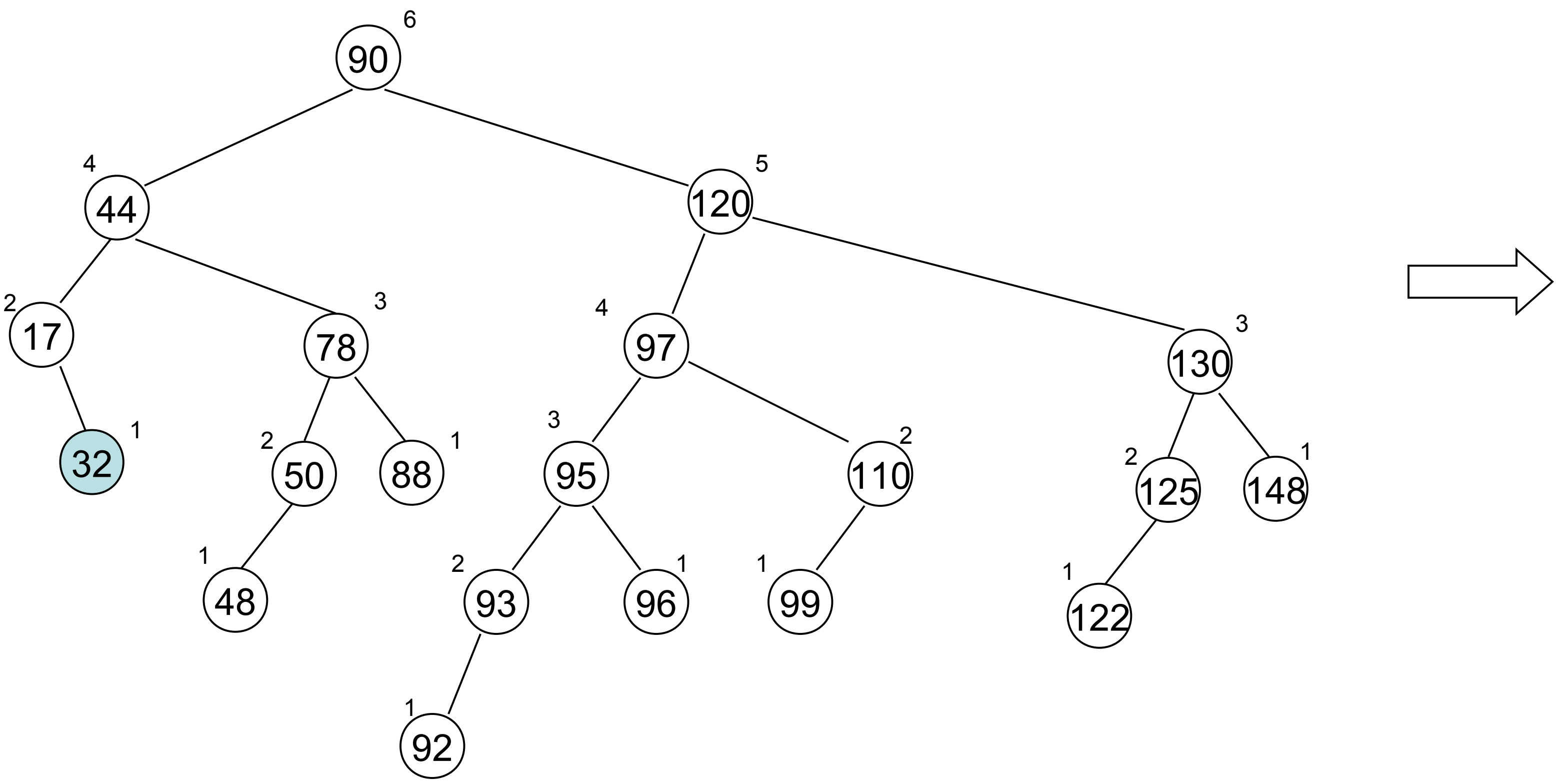
AVL: deletion

- Use same rotation operations as insertion
- Define z, y, x as before
  - find z from deleted node w

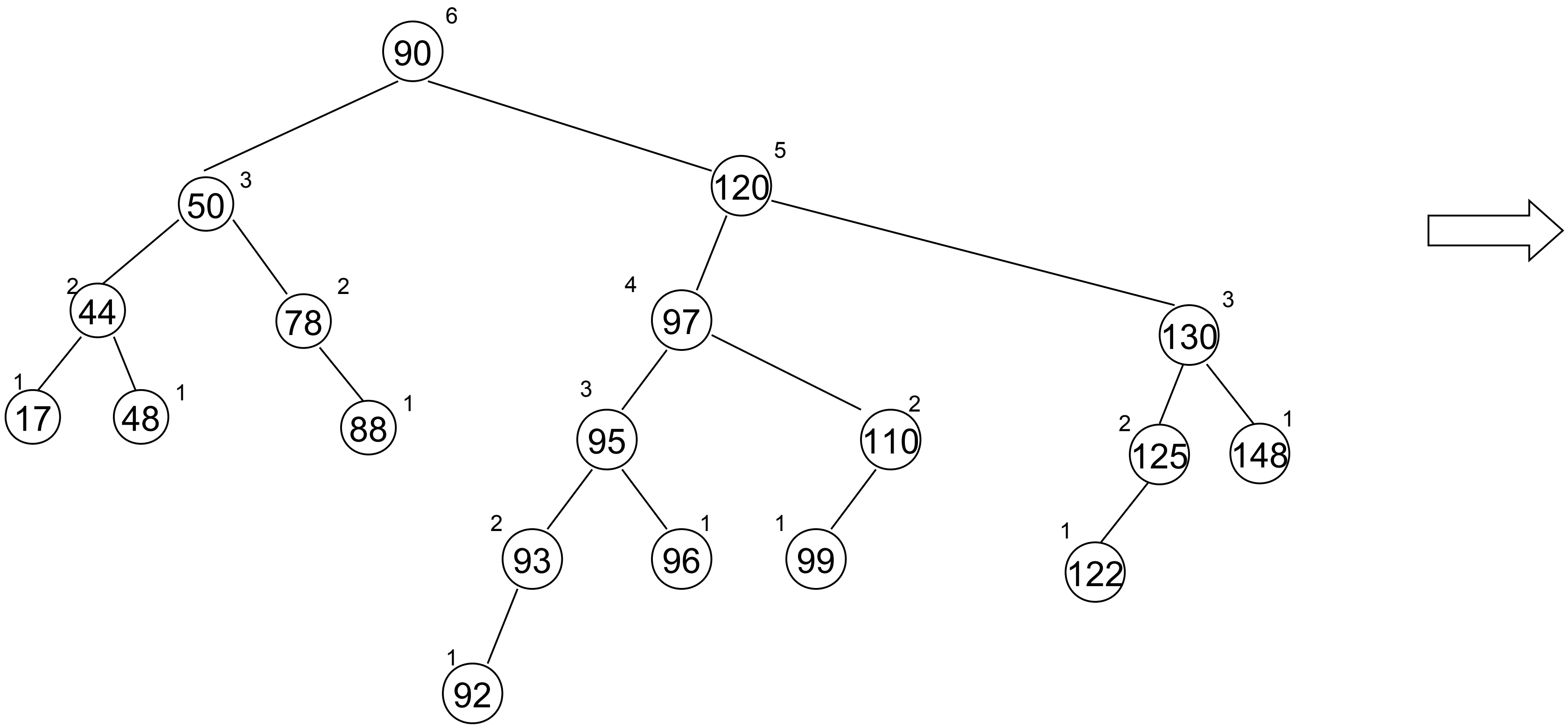
AVL: delete example



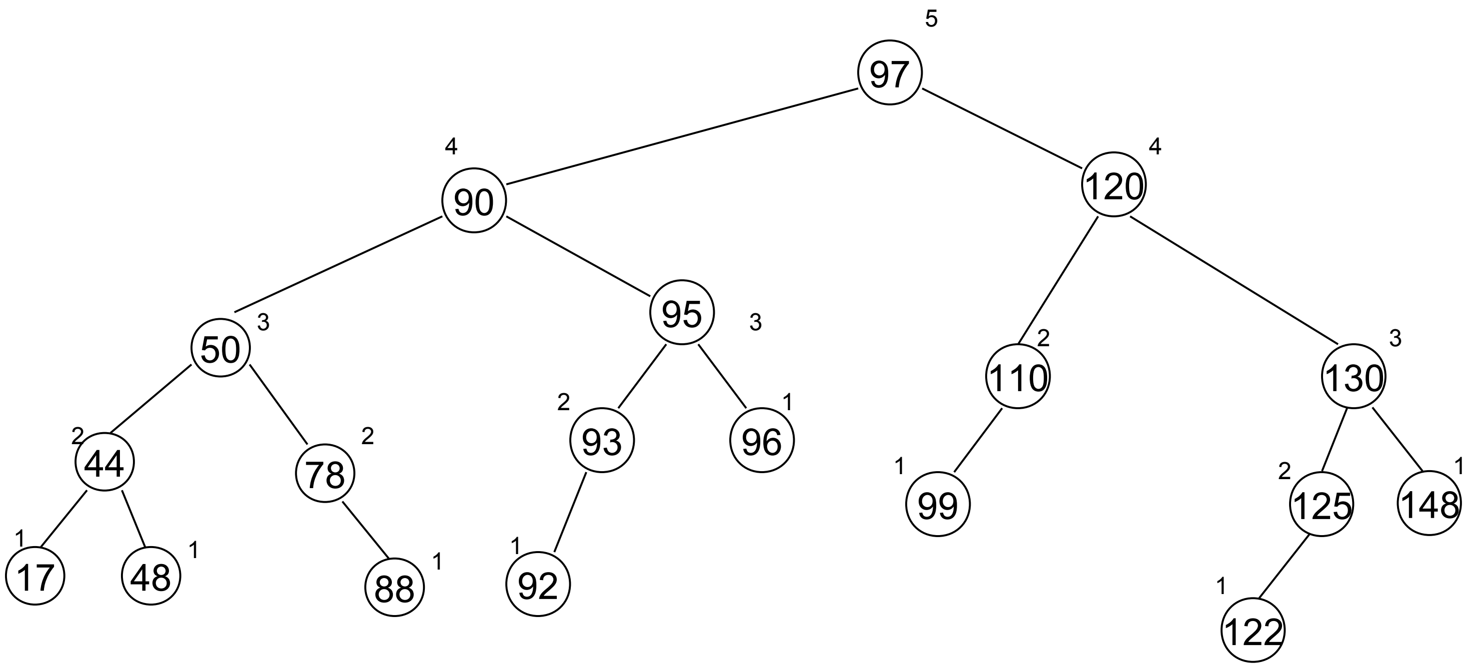
AVL: delete example



AVL: delete example



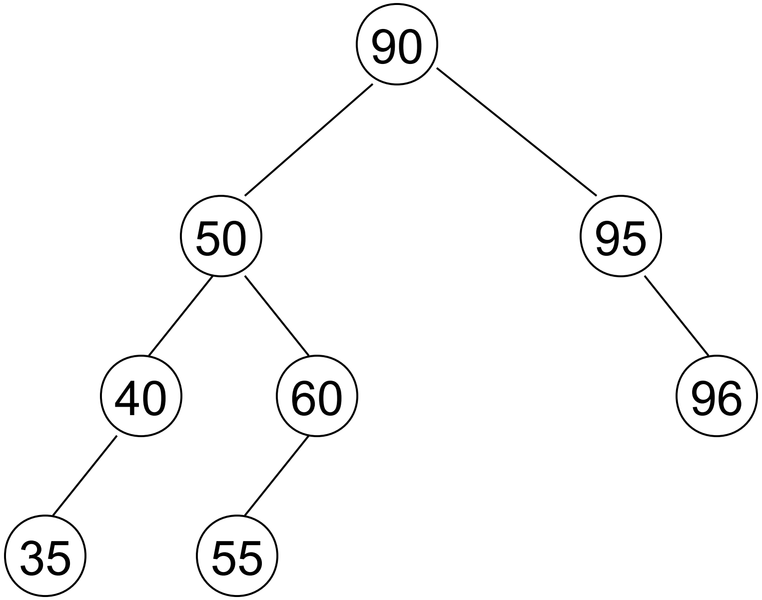
AVL: delete example



AVL: deletion

- Note that may need multiple rotations to restore height-balance
- Goes up from initial node z to root
- Different from insertion: why?
  - with insertion, unbalancing occurs by adding an element (i.e. increasing the height of a subtree)
  - however, deletion shrinks height of subtree
  - what do rotations do?

AVL: delete exercise



Outline

- AVL Trees
- B-Trees
- External Storage

B-Trees

- A B-tree of order m is a multiway search tree where
  - the root has at least two subtrees unless it is a leaf
  - each nonroot and each nonleaf node holds k-1 keys and k references to subtrees, where $\lceil m/2 \rceil$ <= k <= m
  - each leaf node holds k-1 keys where $\lceil m/2 \rceil$ <= k <= m
  - all leaves are on the same level

B-Trees

- A B-tree, as a result,
  - is always at least half full,
  - has few levels, and
  - is perfectly balanced

B-Trees

- Worst-case height
  - assume n elements, smallest allowable number of references per non-root node q = $\lceil m/2 \rceil$
  - maximum height is

$$h <= (\log_q (n+1)/2) + 1$$

  - derivation in Drozdek (p304)

B-Trees

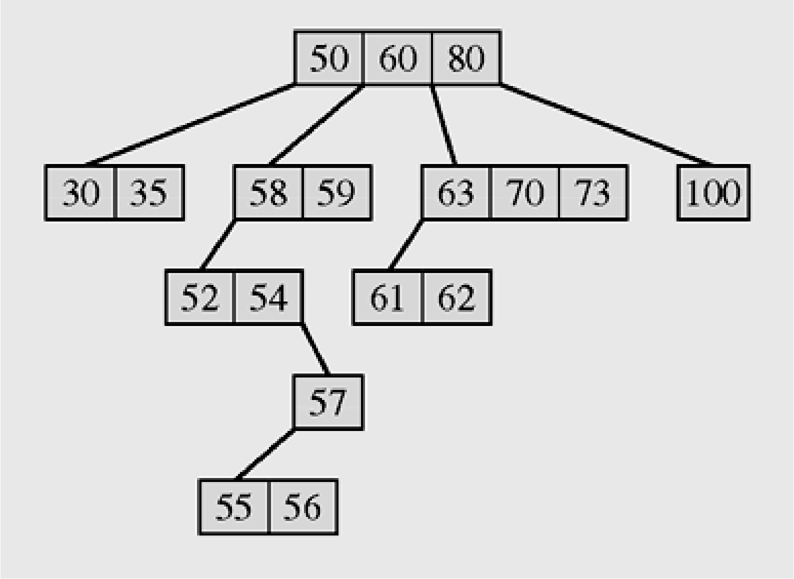


Figure 7-1 A 4-way tree

B-Tree Node Definition

```
public class BTreeNode {
  int m = 4;
  boolean leaf = true;
  int keyTally = 1;
  int keys[] = new int[m-1];
  BTreeNode references[] = new BTreeNode[m];

  BTreeNode(int key) {
    keys[0] = key;
    for (int i = 0; i < m; i++)
      references[i] = null;
    }

}
```

Recall: BST Search

```
public IntBSTNode search(int el) {
    return search(root,el);
}
protected IntBSTNode search(IntBSTNode p, int el) {
    while (p != null)
        if (el == p.key)
             return p;
        else if (el < p.key)
             p = p.left;
        else p = p.right;
    return null;
}
```

B-Tree Search

```
public BTreeNode BTreeSearch(int key) {
  return BTreeSearch(key, root);
}

protected BTreeNode BTreeSearch(int key, BTreeNode node) {
  if (node != null) {
    int i = 1;
    for ( ; i <= node.keyTally && node.keys[i-1] < key; i++);
    if (i > node.keyTally || node.keys[i-1] > key)
      return BTreeSearch(key, node.references[i-1]);
    else return node;
  }
  else return null;
}
```

Inserting a Key into a B-Tree

- There are three common situations encountered when inserting a key into a B-tree:
  - A key is placed in a leaf that still has some room
  - The leaf in which a key should be placed is full
  - If the root of the B-tree is full then a new root and a new sibling of the existing root have to be created

BTreeInsert

- Item is placed in leaf
  - if leaf not full, OK (Fig 7.5)
  - if leaf is full, split into two and percolate up to parent (Fig 7.6)
  - if root is full, split and create new root (Fig 7.7)

Inserting a Key into a B-Tree (continued)

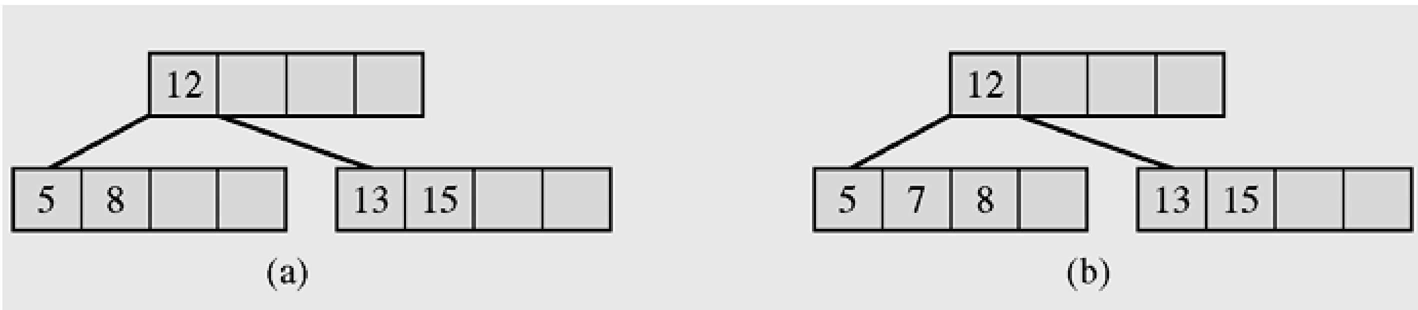


Figure 7-5 A B-tree (a) before and (b) after insertion of the number 7 into a leaf that has available cells

Inserting a Key into a B-Tree (continued)

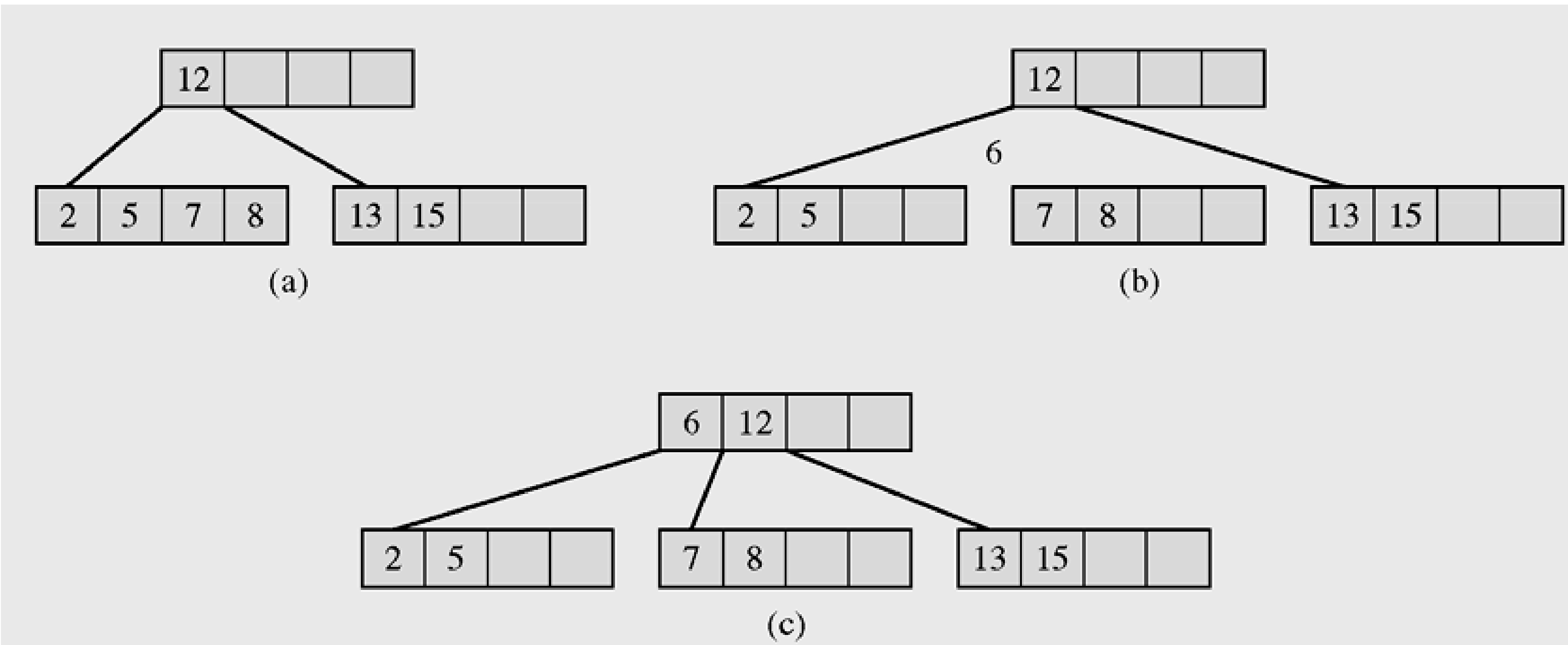


Figure 7-6 Inserting the number 6 into a full leaf

Inserting a Key into a B-Tree (continued)

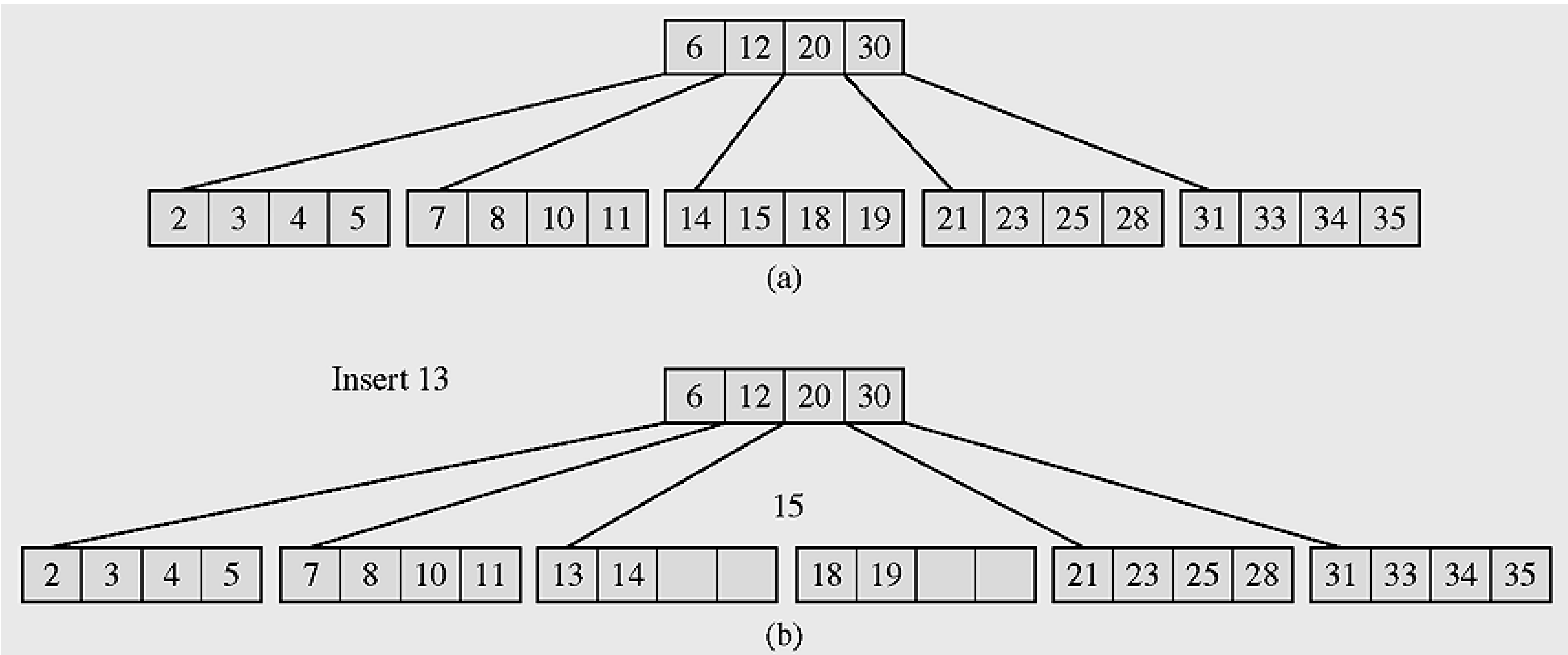


Figure 7-7 Inserting the number 13 into a full leaf

Inserting a Key into a B-Tree (continued)

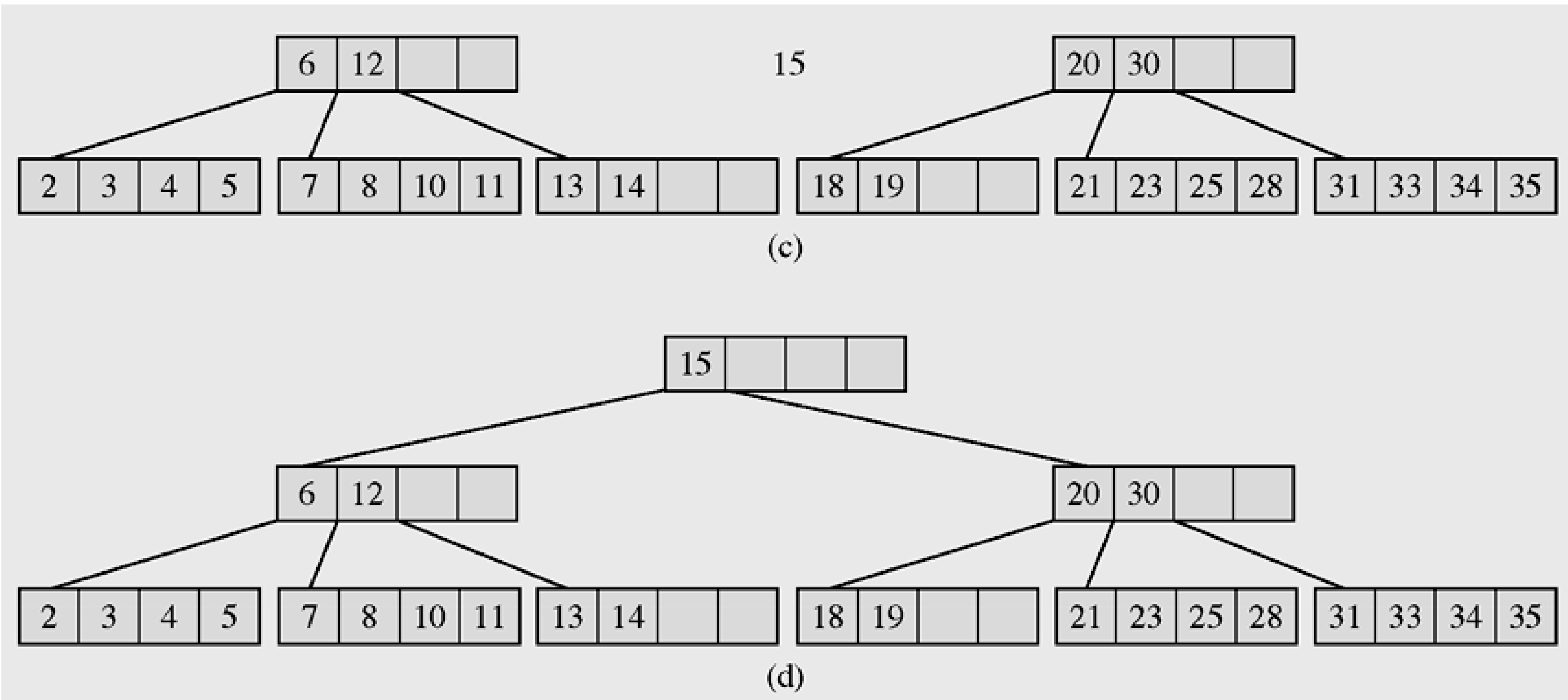


Figure 7-7 Inserting the number 13 into a full leaf (continued)

BTreeInsert

```
BTreeInsert (K)
  find a leaf node to insert K
  while (true)
    find a proper position in keys for K;
    if node is not full
      insert K and increment keyTally;
      return;
    else split node into node1, node2 // node1 = node, node2 is new
      distribute keys and references evenly between node1 and node2;
      initialise properly keyTally for node1, node2;
      K = middle key;
      if node was the root
        create a new node as parent of node1, node2;
        put K and references to node1, node2 in the root, set keyTally=1;
        return;
      else
        node = its parent // now process the node's parent
```

Inserting a Key into a B-Tree (continued)

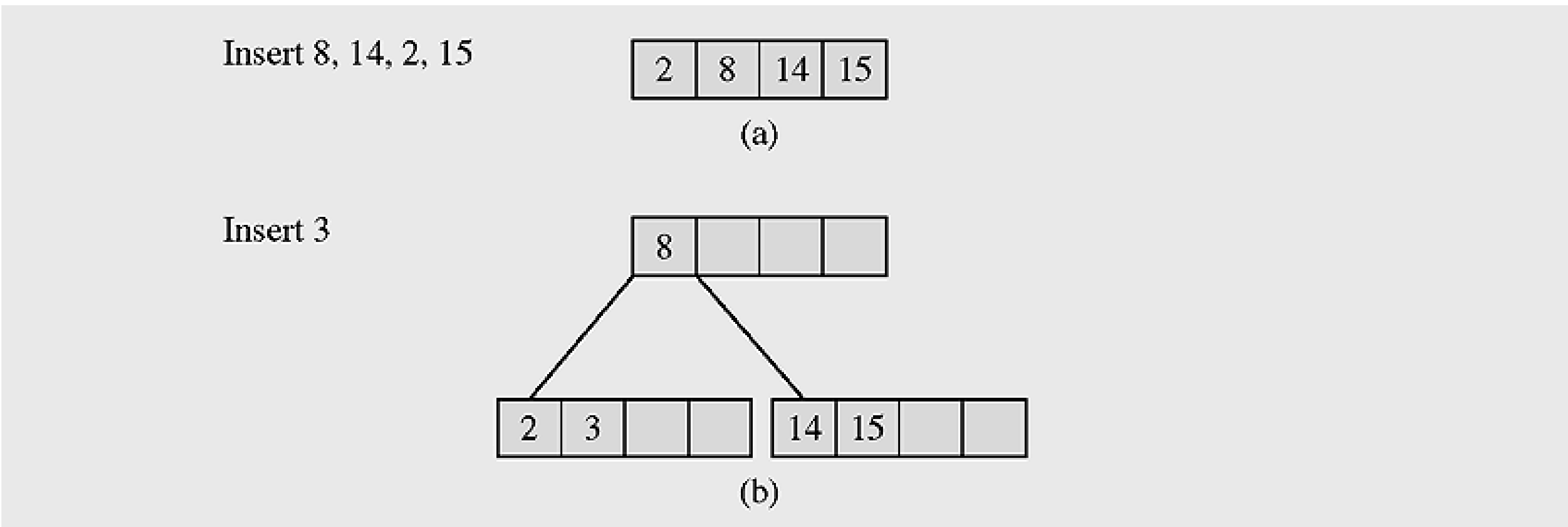


Figure 7-8 Building a B-tree of order 5 with the `BTreeInsert()` algorithm

Inserting a Key into a B-Tree (continued)

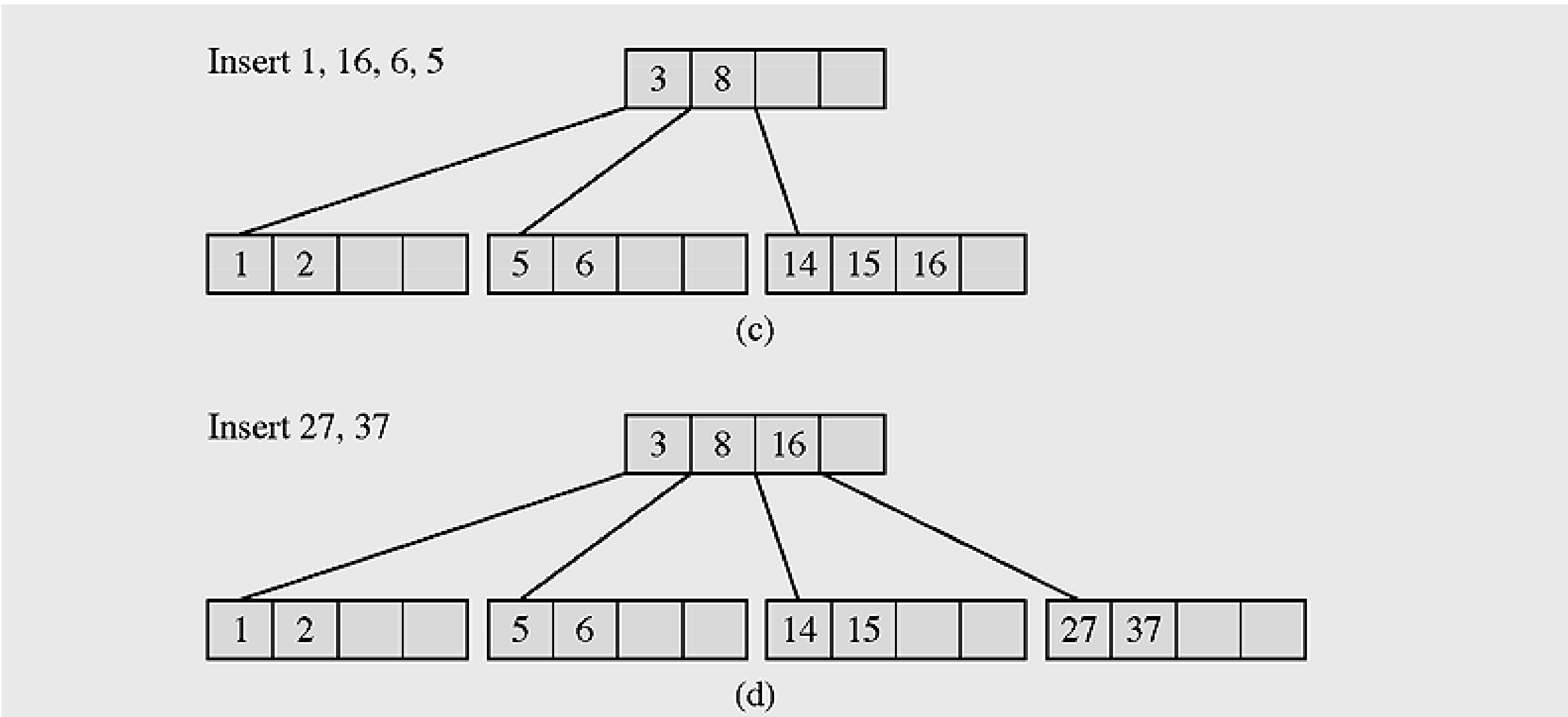


Figure 7-8 Building a B-tree of order 5 with the `BTreeInsert()` algorithm (continued)

Inserting a Key into a B-Tree (continued)

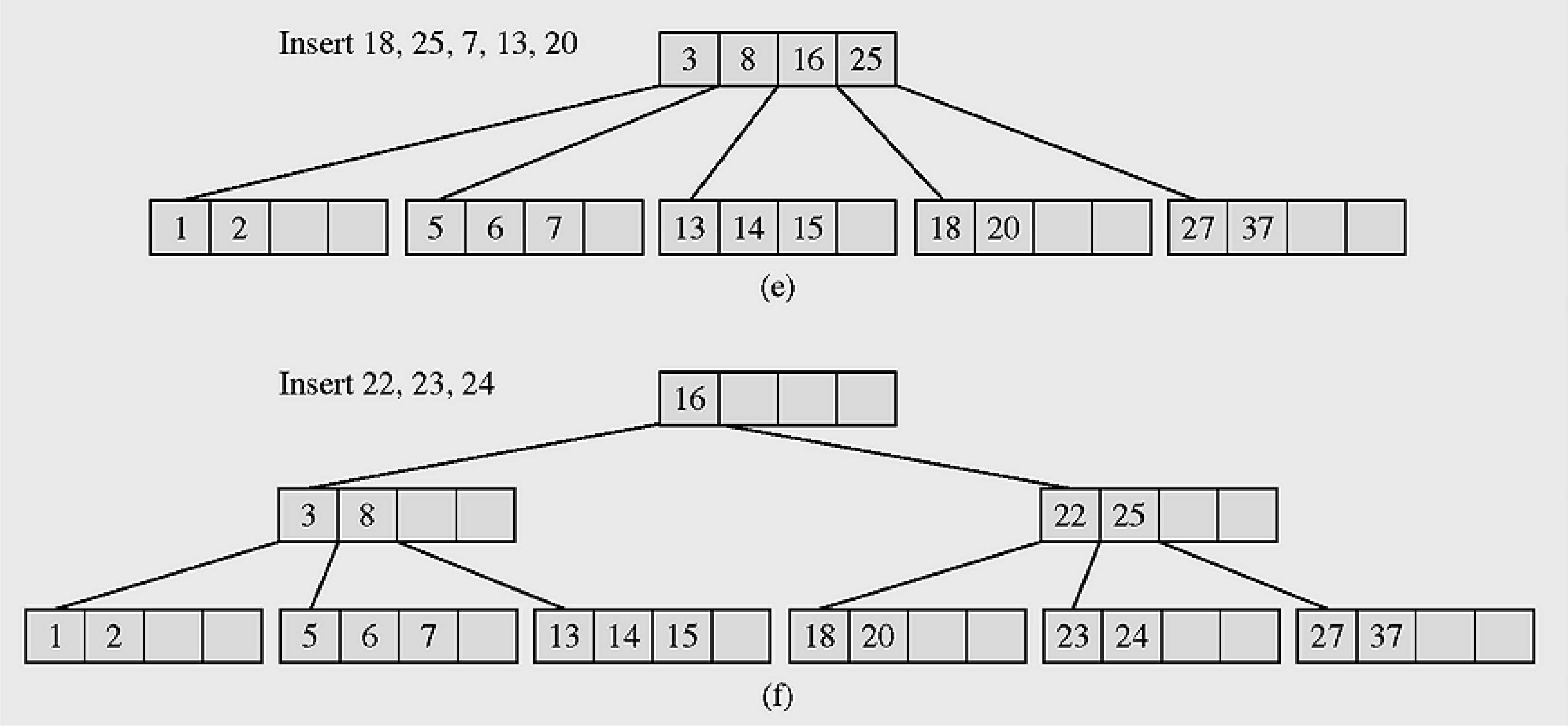


Figure 7-8 Building a B-tree of order 5 with the `BTreeInsert()` algorithm (continued)

Exercise

- Insert 10, 20, 50, 40, 30 into an empty B-tree of order 3

Deleting a Key from a B-Tree

- Avoid allowing any node to be less than half full after a deletion
- In deletion, there are two main cases:
  - Deleting a key from a leaf
  - Deleting a key from a nonleaf node

BTreeDelete

- Deleting from a leaf
  - If leaf is half-full, OK (Fig 7.9a-b)
  - If leaf underflows (< m/2-1)
    - if a sibling has more than m/2-1, redistribute (Fig 7.9b-c)
    - if not, leaf and a sibling are merged; may percolate up (Fig 7.9c-d)
      - special case for root (Fig 7.9c-e)
- Deleting from a non-leaf (Fig7.9e-f)
  - Reduces to swapping predecessor from leaf, deleting swapped item in leaf

BTreeDelete

```
BTreeDelete(K)
  node = BTreeSearch(K, root);
  if (node != null)
    if node is not a leaf
      find a leaf with the closest predecessor S of K;
      copy S over K in node;
      node = the leaf containing S;
      delete S from node;
    else delete K from node;
    while (true)
      if node does not underflow
        return;
      else if there is a sibling of node with enough keys
        redistribute the keys between node and its sibling;
        return;
      else if node's parent is the root
        if the parent has only one key
          merge node, its sibling and the parent to form a new root;
        else merge node and its sibling;
        return;
      else merge node and its sibling;
        node = its parent;
```

Deleting a Key from a B-Tree (continued)

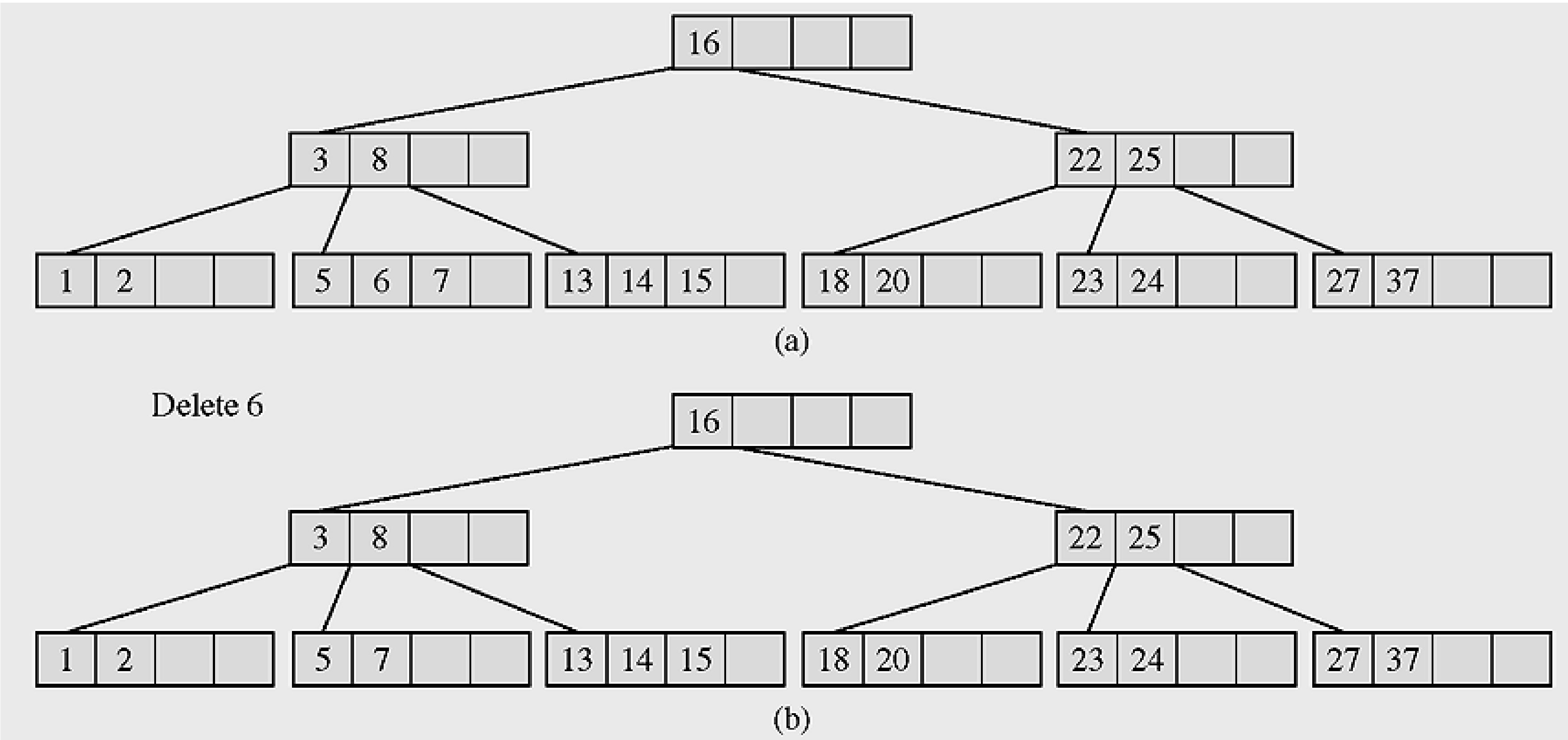


Figure 7-9 Deleting keys from a B-tree

Deleting a Key from a B-Tree (continued)

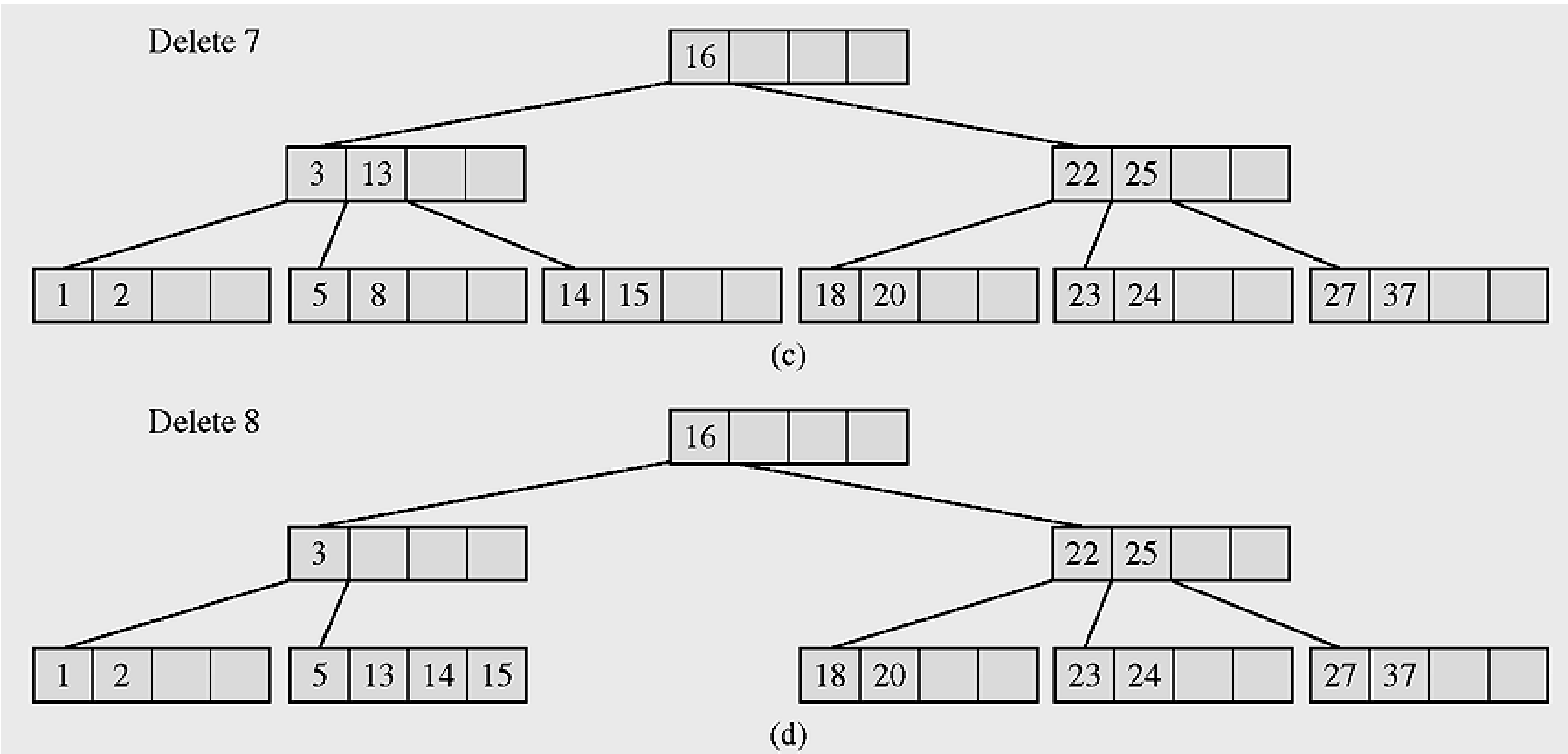


Figure 7-9 Deleting keys from a B-tree (continued)

Deleting a Key from a B-Tree (continued)

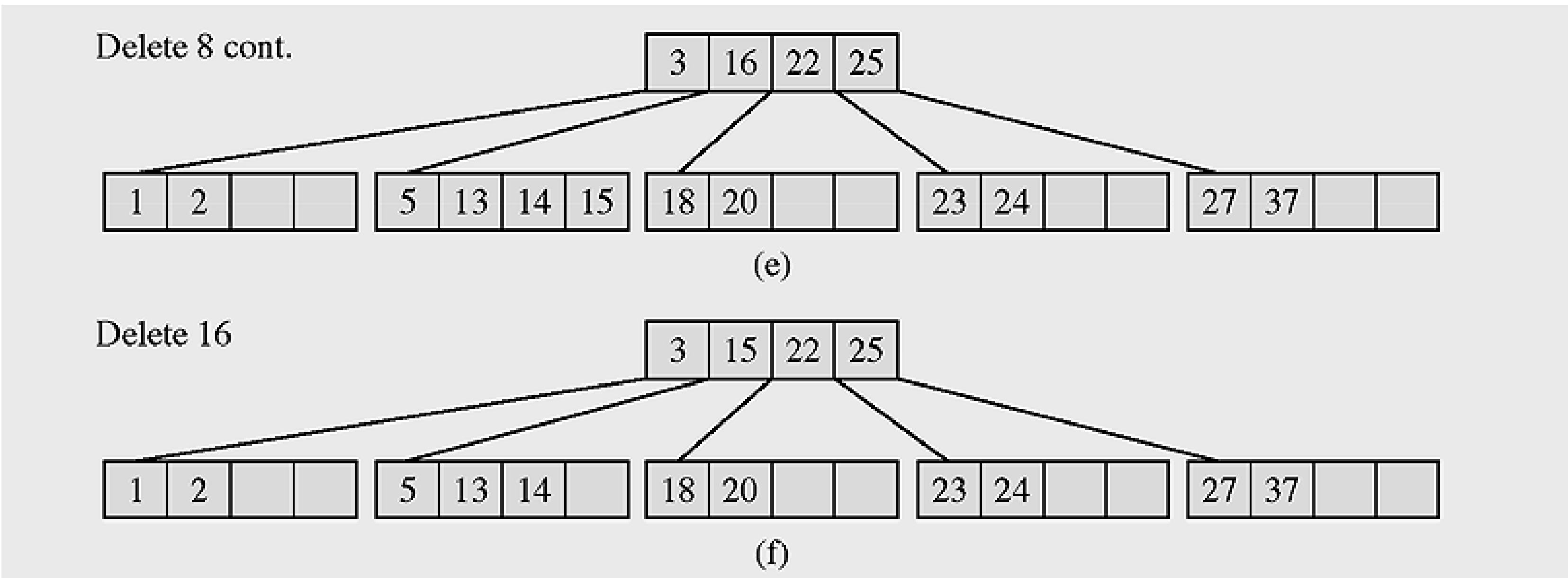


Figure 7-9 Deleting keys from a B-tree (continued)

BST vs B-Tree

- Shorter path to leaf, but tradeoff with more comparisons at each node
  - in general don't want nodes to be too large
  - however, see later ...
- Big advantage is that the tree stays balanced

Exercise

- Delete 50 from the final tree of the previous exercise

Outline

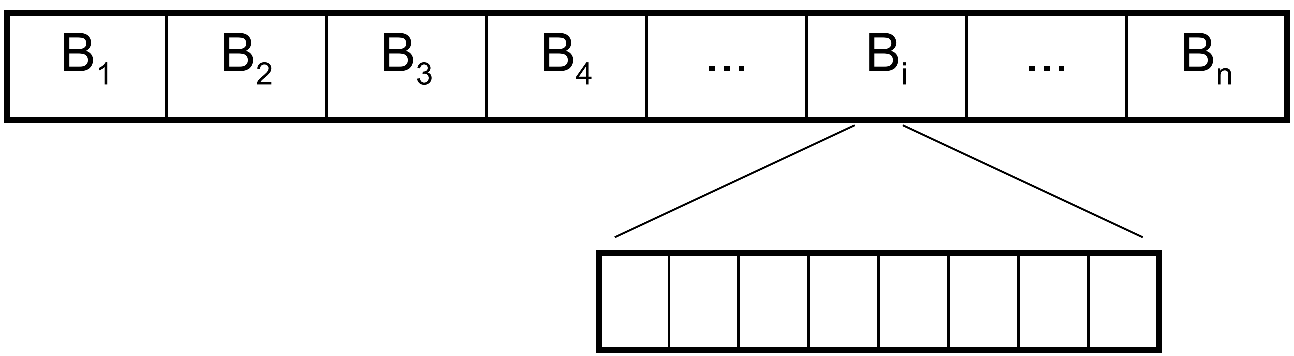
- AVL Trees
- B-Trees
- External Storage

External Storage

- For large amounts of data, not all can be held in memory
- Files in external storage can be either sequential access or direct access
  - sequential like a linked list
  - direct like an array

Files

- Consist of data records
- Are organised into blocks
  - k records per block



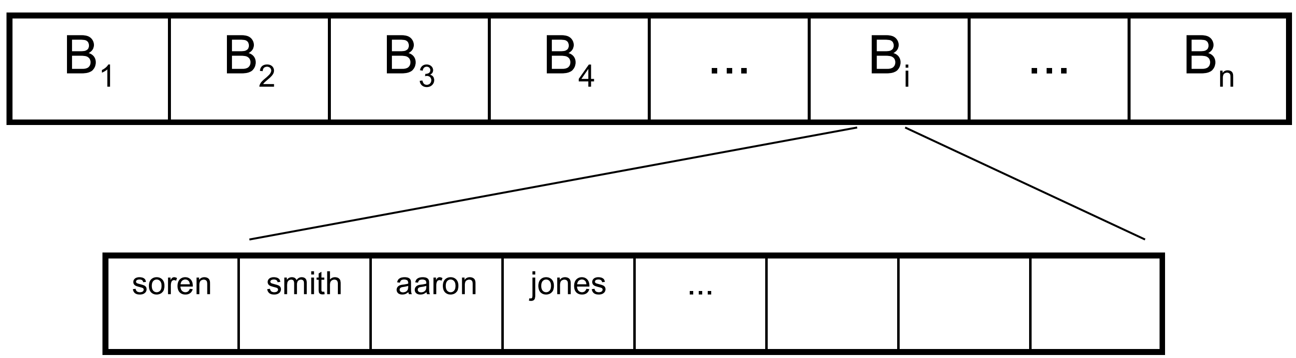
Operating on a Block

- I/O is at the block level

  buf.readBlock(dataFile, i)

  buf.writeBlock(dataFile, i)

- Need to get whole block for operations
  - e.g. increasing salary in an employee record



Operating on a Block

```
// read block i from file dataFile into buffer buf
buf.readBlock(dataFile, i)

// find entry buf.getRecord(j) that contains the right search key
(buf.getRecord(j)).setSalary((buf.getRecord(j)).getSalary() + 1000)

// write changed block back to file dataFile
buf.writeBlock(dataFile, i)
```

External Storage

- Most expensive part is accessing storage
  - might take the same amount of time to read and write a single block as to process all the records in that block

Sorting Data

- Problem is that data is too large to fit into memory all at once
- Therefore, good idea is to use a divide-and-conquer style of sorting algorithm
  - e.g. mergesort
  - sort as much as you can read into memory, then merge later

Sorting Data

- Example

  An external record contains 1600 employee records. You want to sort these records by social security number. Each block contains 100 records, and thus the file contains 16 blocks B_1, B_2 and so on to B_{16}. Assume that the program can access only enough internal memory to manipulate 300 records (3 blocks' worth) at one time.

Sorting Data

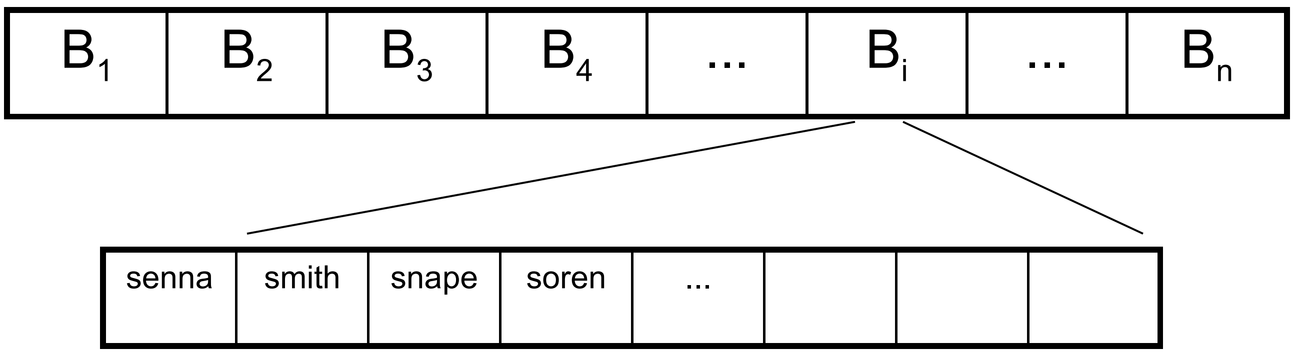
- Mergesort
  - read in each block and sort them in turn
    - result in 16 sorted blocks
  - divide memory into three chunks: in1, in2, out
    - load two sorted blocks into in1, in2
    - merge into out
    - "flush" out when it gets full
    - read in new blocks to in1, in2 when empty
  - each time, double the number of records is sorted

External Lookup

- Idea is to organise records in external storage for efficient operations: traversal, retrieval, insertion and deletion
- For traversal and retrieval:
  - best case is if records in file are all in sorted order
  - can sort using previously mentioned mergesort

External Lookup

- For traversal and retrieval:
  - for retrieval, can do binary search on sorted file
  - will minimise disk access



External Lookup

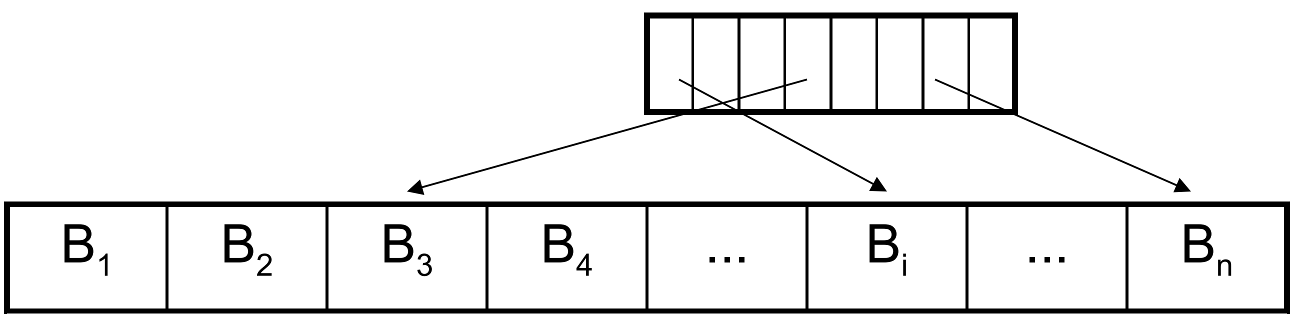
- Insertion and deletion will mess things up
  - if inserting at the end of the file, no longer in sorted order
  - can find insertion site using binary search
  - however, will need to shuffle all items higher in the order to the right
  - similarly for deletion: shuffle all items higher in the order to the left
  - lots of disk accesses in the shuffle

Index Files

- Idea of index files is like a library catalogue
  - indexes just point to data
  - indexes are a lot smaller than data
  - if sufficiently small, can be held in memory
  - even if on disk, will minimise disk accesses
- Other advantages
  - ordering of file records isn't important
    - can just add new records to end
  - can maintain several indexes

Index Files

- Index contains two parts
  - a key: same value as the search key in file record
  - a pointer: shows the number of the data block that contains the record (just an integer)



Index Files

- Can keep the index file in any sort of order
- Assume (to start):
  - that index records are 10% the size of file record;
  - that there are 1000 blocks;
  - that index records are in sequential order
- Using index for retrieve reduces number of operations from $\log_2 1000$ (≈ 10) to about $1 + \log_2 100$ (≈ 9)

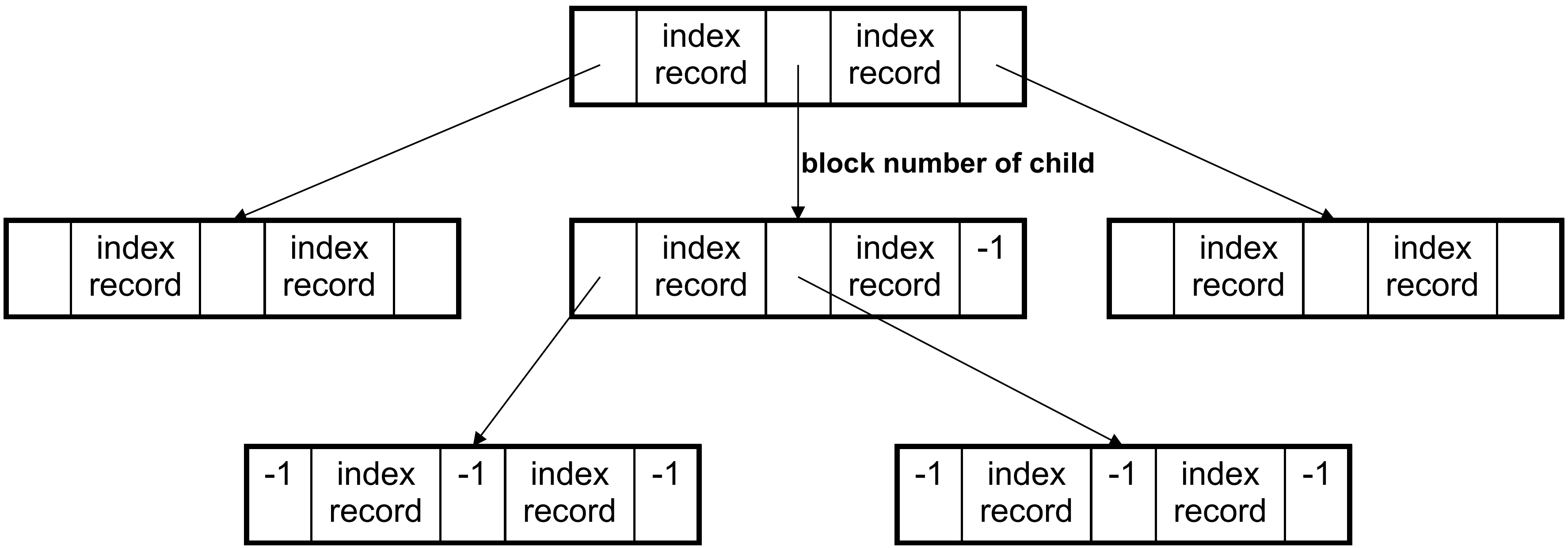
Index Files

- Much more significant for insert and delete
  - without index, have to shift half of all blocks on average (500 blocks)
  - with index, only have to shift half of all indexes on average (50 blocks)
- Still not great
  - can improve by external hashing or external B-Trees

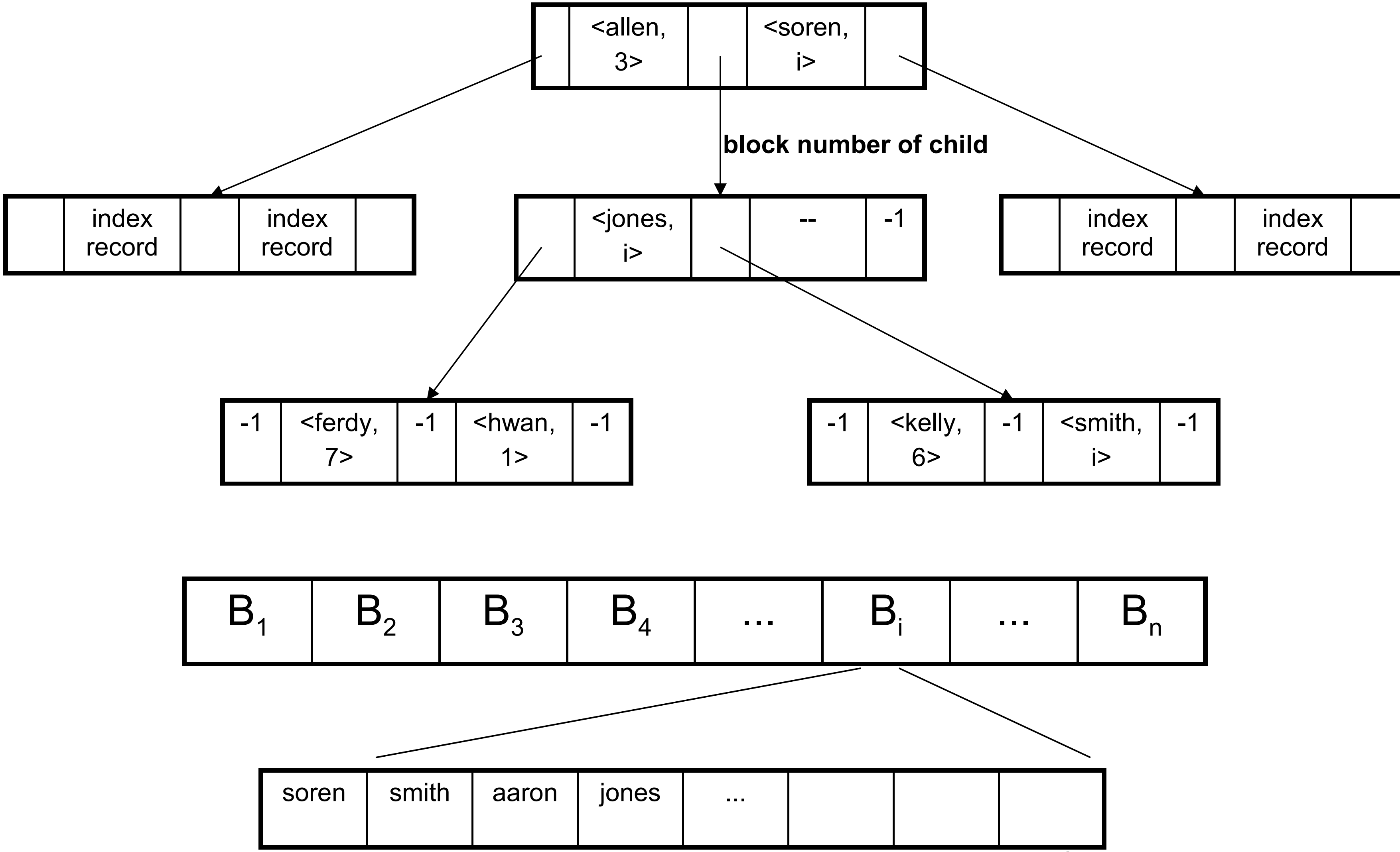
B-Trees for External Storage

- Index file can be arranged as an external search tree
- Nodes in the external search tree contain index records of the form <key,pointer>, plus child pointers
  - pointers and child pointers aren't the same thing
  - pointers point to the file record that's indexed
  - child pointers are index-internal

B-Trees for External Storage

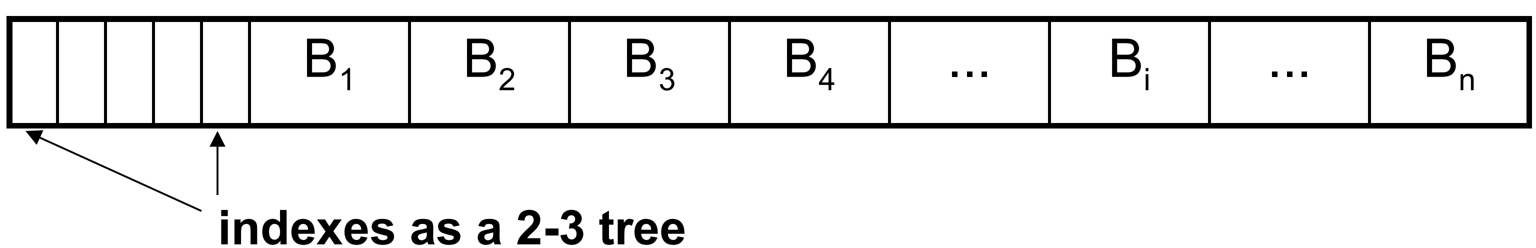


B-Trees for External Storage



B-Trees for External Storage

- Previous example is an instance of a B-tree of order 3 for indexing file access
- Child pointers contain block ID (an integer) of the child index entry
  - the value -1 plays the role of NULL
  - remember index is also stored on disk, e.g.



B-Trees for External Storage

- Recall trade-off in BST vs B-Tree comparison
  - for external trees, there's an extra cost
  - since each B-tree node is stored on disk, getting a new B-tree node (when following child pointers) is expensive relative to comparisons
  - therefore, it's OK to have more elements in a node, and more children

B-Trees for External Storage

- What *m* to choose?
  - best if a single B-tree node takes up a whole block
  - *m* should be the largest integer such that *m* child pointers (i.e. integer-sized values) and *m-1* <key,pointer> records can fit into a block

Effect of B-trees

- Imagine the situation of Woolworths keeping all purchase records (e.g. to analyse best selling products, trends, etc.)
  - what’s the difference in time to find an individual record, given no indexing vs B-tree indexing, as a rough estimate?