Chapter 10
2-3-4 Trees and External Storage

• Introduction to 2-3-4 Trees
• The Tree234 Workshop Applet
• Java Code for a 2-3-4 Tree
• 2-3-4 Trees and Red-Black Trees
• Efficiency of 2-3-4 Trees
• 2-3 Trees
• External Storage
• Summary

Overview

• In a binary tree, each node has one data item and can have up to two children.
• If we allow more data items and children per node, the result is a multiway tree.
• 2-3-4 trees are multiway trees that can have up to four children and three data items per node.
• 2-3-4 trees are interesting for several reasons.
  – They're balanced trees like red-black trees.
  – They're slightly less efficient than red-black trees, but easier to program.
  – Most importantly, they serve as an easy-to-understand introduction to B-trees.
• A B-tree is another kind of multiway tree that's particularly useful for organizing data in external storage.
  – (External means external to main memory; usually this is a disk drive.)
  – A node in a B-tree can have dozens or hundreds of children.
  – We'll discuss external storage and B-trees in the second part of this chapter.

Introduction to 2-3-4 Trees

• Figure 10.1 shows a small 2-3-4 tree.
  – Each lozenge-shaped node can hold one, two, or three data items.
• The top three nodes have children, and the six nodes on the bottom row are all leaf nodes, which by definition have no children.
• In a 2-3-4 tree
  – all the leaf nodes are always on the same level.

What's in a Name?

• A leaf node, by contrast, has no children, but it can nevertheless contain one, two, or three data items.
• Empty nodes are not allowed.
• Because a 2-3-4 tree can have nodes with up to four children, it's called a multiway tree of order 4.

• Why isn't a 2-3-4 tree called a 1-2-3-4 tree?
  – Can't a node have only one child, as nodes in binary trees can?
    – A binary tree (Chapters 8, "Binary Trees," and 9, "Red-Black Trees") can be thought of as a binary tree of order 2 because each node can have up to two children.
    – However, there's a difference (besides the maximum number of children) between binary trees and 2-3-4 trees.
      – In a binary tree,
        - a node can have up to two child links.
        - A single link, to its left or to its right child, is also perfectly permissible.
        - The other link has a null value.
Introduction to 2-3-4 Trees

What’s in a Name?

• In a 2-3-4 tree, on the other hand,
  – nodes with a single link are not permitted.
  – A node with one data item must always have two links, unless it's a leaf, in which case it has no links.

• Figure 10.2 shows the possibilities.
  – A node with two links is called a 2-node,
  – a node with three links is a 3-node,
  – and a node with 4 links is a 4-node,
  – but there is no such thing as a 1-node.

Introduction to 2-3-4 Trees

2-3-4 Tree Organization

• An important aspect of any tree’s structure is the relationship of its links to the key values of its data items.
  – In a binary tree, all children with keys less than the node’s key are in a subtree rooted in the node’s left child, and all children with keys larger than or equal to the node’s key are rooted in the node’s right child.
  – In a 2-3-4 tree the principle is the same, but there’s more to it: (See Figure 10.3 in the next slide)
    • All children in the subtree rooted at child 0 have key values less than key 0.
    • All children in the subtree rooted at child 1 have key values greater than key 0 but less than key 1.
    • All children in the subtree rooted at child 2 have key values greater than key 1 but less than key 2.
    • All children in the subtree rooted at child 3 have key values greater than key 2.

Introduction to 2-3-4 Trees

Searching

• Finding a data item with a particular key is similar to the search routine in a binary tree.
  – You start at the root, and, unless the search key is found there, select the link that leads to the subtree with the appropriate range of values.
  – For example, to search for the data item with key 64 in the tree in Figure 10.1, you start at the root.
    • You search the root, but don’t find the item.
    • Because 64 is larger than 50, you go to child 1, which we will represent as 60/64/66. (Remember that child 1 is on the right, because the numbering of children and links starts at 0 on the left.)
    • You don’t find the data item in this node either, so you must go to the next child.
    • Here, because 64 is greater than 60 but less than 70, you go again to child 1. This time you find the specified item in the 62/64/66 link.
• New data items are always inserted in leaves, which are on the bottom row of the tree.
• If items were inserted in nodes with children,
  – then the number of children would need to be changed to maintain the structure of the tree,
  • which stipulates that there should be one more child than data items in a node.

Introduction to 2-3-4 Trees
Insertion
• Insertion into a 2-3-4 tree is sometimes quite easy and sometimes rather complicated.
  – In any case the process begins by searching for the appropriate leaf node.
  – If no full nodes are encountered during the search, insertion is easy.
    • When the appropriate leaf node is reached, the new data item is simply inserted into it.
    – Figure 10.4 shows a data item with key 18 being inserted into a 2-3-4 tree.
    – Insertion may involve moving one or two other items in a node so the keys will be in the correct order after the new item is inserted.
      • In this example the 23 had to be shifted right to make room for the 18.

Introduction to 2-3-4 Trees
Insertion
• Insertion becomes more complicated if a full node is encountered on the path down to the insertion point.
  – When this happens, the node must be split.
  – It’s this splitting process that keeps the tree balanced.
  – The kind of 2-3-4 tree we’re discussing here is often called a top-down 2-3-4 tree
    • because nodes are split on the way down to the insertion point.

Introduction to 2-3-4 Trees
Node Splits
• Let’s name the data items in the node that’s about to be split A, B, and C.
• Here’s what happens in a split. (We assume the node being split is not the root; we’ll examine splitting the root later.)
  – A new, empty node is created. It’s a sibling of the node being split, and is placed to its right.
  – Data item C is moved into the new node.
  – Data item B is moved into the parent of the node being split.
  – Data item A remains where it is.
  – The rightmost two children are disconnected from the node being split and connected to the new node.

Introduction to 2-3-4 Trees
Node Splits
• An example of a node split is shown in Figure 10.5.
• Another way of describing a node split is to say that a 4-node has been transformed into two 2-nodes.
• Notice that the effect of the node split is to move data up and to the right.
  – It’s this rearrangement that keeps the tree balanced.
• Here the insertion required only one node split, but more than one full node may be encountered on the path to the insertion point.
  – When this is the case there will be multiple splits.

Introduction to 2-3-4 Trees
Splitting the Root
• When a full root is encountered at the beginning of the search for the insertion point, the resulting split is slightly more complicated:
  – A new node is created that becomes the new root and the parent of the node being split.
  – A second new node is created that becomes a sibling of the node being split.
  – Data item C is moved into the new node.
  – Data item B is moved into the new sibling.
  – Data item A remains where it is.
  – The two rightmost children of the node being split are disconnected from it and connected to the new right-hand node.
• Figure 10.6 shows the root being split.
  – This process creates a new root that’s at a higher level than the old one.
  – Thus the overall height of the tree is increased by one.
• Another way to describe splitting the root is to say that a 4-node is split into three 2-nodes.
• Following a node split, the search for the insertion point continues down the tree.
• In Figure 10.6, the data item with a key of 41 is inserted into the appropriate leaf.

Because all full nodes are split on the way down,
  – a split can’t cause an effect that ripples back up through the tree.
  – The parent of any node that’s being split is guaranteed not to be full, and can therefore accept data item B without itself needing to be split.
  – Of course, if this parent already had two children when its child was split, it will become full.
  – However, that just means that it will be split when the next search encounters it.
• Figure 10.7 shows a series of insertions into an empty tree.
  – There are four node splits, two of the root and two of leaves.

Operating the Tree234 Workshop applet provides a quick way to see how 2-3-4 trees work.
  – When you start the applet you’ll see a screen similar to Figure 10.8.
  – Within each non-leaf node, the algorithm examines each data item, starting on the left, to see which child it should go to next.
  – In a leaf node it examines each data item to see if it contains the specified key.
  – If it can’t find such an item in the leaf node, the search fails.
• In the Tree234 Workshop applet it’s important to complete each operation before attempting a new one.
  – Continue to click the button until the message says Press any button.
  – This is the signal that an operation is complete.

When it’s first started, the Tree234 Workshop applet inserts 10 data items into the tree.
• You can use the Fill button to create a new tree with a different number of data items from 0 to 45.
  – Click Fill and type the number into the field when prompted.
  – Another click will create the new tree.
• The tree may not look very full with 45 nodes, but more nodes require more levels, which won’t fit in the display.
  – If it encounters a full node along the way, it splits it before continuing on.
  – Then try inserting at the end of a path that includes a full node, either at the root, at the leaf, or somewhere in between.
  – Watch how new nodes are formed and the contents of the node being split are distributed among three different nodes.
• One of the problems with 2-3-4 trees is that there are a great many nodes and data items just a few levels down. The Tree234 Workshop applet supports only four levels, but there are potentially 64 nodes on the bottom level, each of which can hold up to three data items.

• It would be impossible to display so many items at once on one row, so the applet shows only some of them: the children of a selected node. (To see the children of another node, you click on it; we’ll discuss that in a moment.)

• To see a zoomed-out view of the entire tree, click the Zoom button. Figure 10.9 shows what you’ll see.

• In this view:
  – nodes are shown as small rectangles;
  – data items are not shown.
  – Nodes that exist and are visible in the zoomed-in view (which you can restore by clicking Zoom again) are shown in green.
  – Nodes that exist but aren’t currently visible in the zoomed-out view are shown in magenta.
  – and nodes that don’t exist are shown in gray.
  – View the applet on a color monitor to make sense of the display.

• Using the Zoom button to toggle back and forth between the zoomed-out and zoomed-in views – allows you to see both the big picture and the details, and hopefully put the two together in your mind.

• In the zoomed-in view you can always see all the nodes in the top two rows:
  – there’s only one, the root, in the top row,
  – and only four in the second row.
  – Below the second row things get more complicated because there are too many nodes to fit on the screen:
    • 16 on the third row,
    • 64 on the fourth.
  – However, you can see any node you want by clicking on its parent, or sometimes its grandparent and then its parent.

• A blue triangle at the bottom of a node shows where a child is connected to a node.
  – If a node’s children are currently visible, the lines to the children can be seen running from the blue triangles to them.
  – If the children aren’t currently visible, there are no lines, but the blue triangles indicate that the node nevertheless has children.
  – If you click on the parent node, its children and the lines to them will appear.
  – By clicking the appropriate nodes you can navigate all over the tree.

• For convenience, all the nodes are numbered, starting with 0 at the root and continuing up to 85 for the node on the far right of the bottom row.
  – The numbers are displayed to the upper right of each node, as shown in Figure 10.8
  – Nodes are numbered whether they exist or not, so the numbers on existing nodes probably won’t be contiguous.

• Figure 10.10 shows a small tree with four nodes in the third row.
  – The user has clicked on node 1, so its two children, numbered 5 and 6, are visible.

• If the user clicks on node 2,
  – its children 9 and 10 will appear, as shown in Figure 10.11.

• These figures show how to switch among different nodes in the third row by clicking nodes in the second row.

• To switch nodes in the fourth row
  – you’ll need to click first on a grandparent in the second row, then on a parent in the third row.

• During searches and insertions with the Find and Ins buttons, – the view will change automatically to show the node currently being pointed to by the red arrow.

• The Tree234 Workshop applet offers a quick way to learn about 2-3-4 trees.

• Try inserting items into the tree.
  – Watch for node splits.
  – Stop before one is about to happen, and figure out where the three data items from the split node are going to go.
  – Then press Ins again to see if you’re right.

• As the tree gets larger you’ll need to move around it to see all the nodes.
  – Click on a node to see its children (and their children, and so on).
  – If you lose track of where you are, use the Zoom key to see the big picture.
• How many data items can you insert in the tree?
  – There’s a limit because only four levels are allowed.
  – Four levels can potentially contain \(1 + 4 + 16 + 64\) nodes, for a total of 85 nodes (all visible on the zoomed-out display).
  – Assume a full 3 items per node gives 255 data items.
  – However, the nodes can’t all be full at the same time.
  • Long before they fill up, another root split, leading to five levels, would be necessary, and this is impossible because the applet supports only four levels.
  • You can insert the most items by deliberately inserting them into nodes that lie on paths with no full nodes, so that no splits are necessary.
  • This is not a reasonable procedure with real data.
  • For random data you probably can’t insert more than about 50 items into the applet.
  • The Fill button allows only 45, to minimize the possibility of overflow.

• In this section we’ll examine a Java program that models a 2-3-4 tree.
  • We’ll show the complete `tree234.java` program at the end of the section.
  • This is a relatively complex program, and the classes are extensively interrelated, so you’ll need to peruse the entire listing to see how it works.
  • There are four classes: `DataItem`, `Node`, `Tree234`, and `Tree234App`. We’ll discuss them in turn.

Objects of this class represent the data items stored in nodes.
• In a real-world program each object would contain an entire personnel or inventory record;
  – but here there’s only one piece of data, of type `double`, associated with each `DataItem` object.
• The only actions that objects of this class can perform are to initialize themselves and display themselves.
  • The display is the data value preceded by a slash: `/27`. (The display routine in the `Node` class will call this routine to display all the items in a node.)

• We’ve chosen to store the number of items currently in the node (`numItems`) and the node’s parent (`parent`) as fields in this class.
  – Neither of these is strictly necessary, and could be eliminated to make the nodes smaller.
  – However, including them clarifies the programming, and only a small price is paid in increased node size.
• Various small utility routines are provided in the `Node` class to manage the connections to child and parent and to check if the node is full and if it is a leaf.
  – However, the major work is done by the `findItem()`, `insertItem()`, and `removeItem()` routines.
  • These handle individual items within the node.
  • They search through the node for a data item with a particular key;
  • insert a new item into the node;
  • moving existing items if necessary;
  • and remove an item, again moving existing items if necessary.
  • Don’t confuse these methods with the `find()` and `insert()` routines in the `Tree234` class, which we’ll look at next.
An object of the `Tree234` class represents the entire tree. The class has only one field: `root`, of type `Node`. All operations start at the root, so that's all a tree needs to remember.

### Searching

Searching for a data item with a specified key is carried out by the `find()` routine.
- It starts at the root, and at each node calls that node's `findItem()` routine to see if the item is there.
  - If so, it returns the index of the item within the node's item array.
  - If `find()` is at a leaf and can't find the item, the search has failed, so it returns -1.
  - If it can't find the item in the current node, and the current node isn't a leaf, `find()` calls the `getNextChild()` method, which figures out which of a node's children the routine should go to next.

### Inserting

The `insert()` method starts with code similar to `find()`, except that if it finds a full node it splits it.
- Also, it assumes it can't fail;
- it keeps looking, going to deeper and deeper levels, until it finds a leaf node.
- At this point it inserts the new data item into the leaf.
- (There is always room in the leaf, otherwise the leaf would have been split.)

### Splitting

The `split()` method is the most complicated in this program.
- It is passed the node that will be split as an argument.
  - First, the two rightmost data items are removed from the node and stored.
  - Then the two rightmost children are disconnected; their references are also stored.
  - A new node, called `newRight`, is created.
    - It will be placed to the right of the node being split.
    - If the node being split is the root, an additional new node is created: a new root.
  - Next, appropriate connections are made to the parent of the node being split.
    - It may be a pre-existing parent, or if the root is being split it will be the newly created root node.
  - Assume the three data items in the node being split are called A, B, and C.
    - Item B is inserted in this parent node.
    - If necessary, the parent's existing children are disconnected and reconnected one position to the right to make room for the new data item and new connections.
    - The `newRight` node is connected to this parent. (Refer to Figures 10.5 and 10.6.)
  - Now the focus shifts to the `newRight` node.
    - Data item C is inserted in it, and child 2 and child 3, which were previously disconnected from the node being split, are connected to it.
  - The split is now complete, and the `split()` routine returns.

In the `Tree234App` class, the `main()` routine inserts a few data items into the tree.
- It then presents a character-based interface for the user, who can enter s to see the tree, i to insert a new data item, and f to find an existing item. Here's some sample interaction:
  - Enter first letter of show, insert, or find: s
  - level=0 child=0 /50/
  - level=1 child=0 /30/40/
  - level=1 child=1 /60/70/
  - Enter first letter of show, insert, or find: i
  - Enter value to insert: 40
  - Insert successful
  - Enter first letter of show, insert, or find: f
  - Enter value to find: 40
  - Found 40
(Cont’d on the next slide.)
Enter first letter of show, insert, or find: i
Enter value to insert: 20
Enter first letter of show, insert, or find: s
level=0 child=0 /50/
level=1 child=0 /20/30/40/
level=1 child=1 /60/70/
Enter first letter of show, insert, or find: i
Enter value to insert: 10
Enter first letter of show, insert, or find: s
level=0 child=0 /30/50/
level=1 child=0 /10/20/
level=1 child=1 /40/
level=1 child=2 /60/70/

Java Code for a 2-3-4 Tree

The Tree234App Class

- The output is not very intuitive, but there's enough information to draw the tree if you want.
- The level, starting with 0 at the root, is shown, as well as the child number.
- The display algorithm is depth-first,
  - so the root is shown first,
  - then its first child and the subtree of which the first child is the root,
  - then the second child and its subtree,
  - and so on.
- The output shows two items being inserted, 20 and 10.
- The second of these caused a node (the root's child 0) to split.

Figure 10.12 (next slide) depicts the tree that results from these insertions, following the final press of the s key.

---

Java Code for a 2-3-4 Tree

Listing for tree234.java

```java
// tree234.java
// demonstrates 234 tree
// to run this program C>java Tree234App
import java.io.*;

class DataItem
{
    public long dData;          // one data item
    public DataItem(long dd)    // constructor
    { dData = dd; }
    public void displayItem()   // display item, format "/27"
    { System.out.print("/\"+dData); }
}

class Node
{
    private static final int ORDER = 4;
    private int numItems;
    private Node parent;
    private Node childArray[] = new Node[ORDER];
    private DataItem itemArray[] = new DataItem[ORDER-1];

    public void connectChild(int childNum, Node child)
    { childArray[childNum] = child;
      if(child != null)
        child.parent = this;
    }

    public Node disconnectChild(int childNum)
    { Node tempNode = childArray[childNum];
      childArray[childNum] = null;
      return tempNode;
    }

    public Node getChild(int childNum)
    { return childArray[childNum]; }

    public boolean isLeaf()
    { return (childArray[0]==null) ? true : false; }

    public boolean isFull()
    { return (numItems==ORDER-1) ? true : false; }

    public Node getFirstChild()
    { return childArray[0]; }

    public Node getNextChild() // returns child after childNum
    { if(childNum+1 < ORDER)
      return childArray[childNum+1];
    }

    public boolean isDone() // returns true if all children are displayed
    { return (childNum==ORDER-1) ? true : false; }

    public boolean isDoneDisplay() // returns true if all children are displayed
    { return isDone(); }

    public DataItem insertItem(DataItem item, int childNum)
    { if(isFull())
      isLeaf = true;
      return null;
    }

    public DataItem getNextItem() // returns next item in tree
    { return itemArray[childNum]; }

data: 20 10
```

Listing for tree234.java (1)

Listing for tree234.java (2)

Listing for tree234.java (3)
public void insert(long dValue) // insert a DataItem
    {
        DataItem tempItem = new DataItem(dValue);
        Node curNode = root;
    
        while(true)
            {
                if(curNode.isFull()) // if node full,
                    {
                        split(curNode); // split it
                        // search once
                        break;
                    }
                else if(curNode.isLeaf()) // if node is leaf,
                    {
                        curNode.insertItem(tempItem); // insert new DataItem
                        return;
                    }
                else
                    {
                        childNumber = curNode.findItem(dValue); // find item
                        if((childNumber=curNode.findItem(key)) != -1) // if found,
                            {
                                if(newKey < itsKey) // if it's smaller
                                    itemArray[j] = itemArray[j+1]; // shift it right
                                else if(newKey > itsKey) // if it's bigger
                                    itemArray[j] = tempItem; // insert new item
                                itemArray[j+1] = null; // disconnect it
                                return j+1; // return index to insert new item
                            }
                        else // search deeper
                            curNode = getNextChild(curNode, key); // go left one cell
                    }
            }
        return -1; // can't find it
    }

private Node root = new Node(); // make root node

    public int find(long key) // search once
    {
        if(( childNumber=curNode.findItem(key) ) != -1) // if found,
            return childNumber; // return index to search deeper
        else // can't find it
            return -1; // can't find it
    }

    public int insertItem(DataItem newItem) // insert new item
    { // assumes node is not full
        if(newItem == null) // null item
            return 0; // return index to insert
        else if(itemArray[0] == null) // item null
            { // examine items
                for(int j=ORDER-2; j>=0; j--) // on right,
                    { // search once
                        itemArray[j].displayItem(); // /56/
                        if(( childNumber=itemArray[j].findItem(key) ) != -1) // if found,
                            return j; // return index to insert new item
                    }
            }
        else if(itemArray[ORDER-1].dData == key) // if item matches
            return ORDER-1; // return index to insert new item
        else // non-matching item
            { // search once
                if(itemArray[0] == null) // node not full
                    return 0; // return index to insert
                else // non-null node
                    return -1; // can't find it
            }
    }

private Node root = new Node(); // make root node
    public int find(long key) // search once
    {
        if(( childNumber=root.findItem(key) ) != -1) // if found,
            return childNumber; // return index to search deeper
        else // can't find it
            return -1; // can't find it
    }

    public int insertItem(DataItem newItem) // insert new item
    { // assumes node is not full
        if(newItem == null) // null item
            return 0; // return index to insert
        else if(itemArray[0] == null) // item null
            { // examine items
                for(int j=ORDER-2; j>=0; j--) // on right,
                    { // search once
                        itemArray[j].displayItem(); // /56/
                        if(( childNumber=itemArray[j].findItem(key) ) != -1) // if found,
                            return j; // return index to insert new item
                    }
            }
        else if(itemArray[ORDER-1].dData == key) // if item matches
            return ORDER-1; // return index to insert new item
        else // non-matching item
            { // search once
                if(itemArray[0] == null) // node not full
                    return 0; // return index to insert
                else // non-null node
                    return -1; // can't find it
            }
    }

private Node root = new Node(); // make root node
    public int find(long key) // search once
    {
        if(( childNumber=root.findItem(key) ) != -1) // if found,
            return childNumber; // return index to search deeper
        else // can't find it
            return -1; // can't find it
    }

    public int insertItem(DataItem newItem) // insert new item
    { // assumes node is not full
        if(newItem == null) // null item
            return 0; // return index to insert
        else if(itemArray[0] == null) // item null
            { // examine items
                for(int j=ORDER-2; j>=0; j--) // on right,
                    { // search once
                        itemArray[j].displayItem(); // /56/
                        if(( childNumber=itemArray[j].findItem(key) ) != -1) // if found,
                            return j; // return index to insert new item
                    }
            }
        else if(itemArray[ORDER-1].dData == key) // if item matches
            return ORDER-1; // return index to insert new item
        else // non-matching item
            { // search once
                if(itemArray[0] == null) // node not full
                    return 0; // return index to insert
                else // non-null node
                    return -1; // can't find it
            }
    }

private Node root = new Node(); // make root node
    public int find(long key) // search once
    {
        if(( childNumber=root.findItem(key) ) != -1) // if found,
            return childNumber; // return index to search deeper
        else // can't find it
            return -1; // can't find it
    }

    public int insertItem(DataItem newItem) // insert new item
    { // assumes node is not full
        if(newItem == null) // null item
            return 0; // return index to insert
        else if(itemArray[0] == null) // item null
            { // examine items
                for(int j=ORDER-2; j>=0; j--) // on right,
                    { // search once
                        itemArray[j].displayItem(); // /56/
                        if(( childNumber=itemArray[j].findItem(key) ) != -1) // if found,
                            return j; // return index to insert new item
                    }
            }
        else if(itemArray[ORDER-1].dData == key) // if item matches
            return ORDER-1; // return index to insert new item
        else // non-matching item
            { // search once
                if(itemArray[0] == null) // node not full
                    return 0; // return index to insert
                else // non-null node
                    return -1; // can't find it
            }
    }

private Node root = new Node(); // make root node
    public int find(long key) // search once
    {
        if(( childNumber=root.findItem(key) ) != -1) // if found,
            return childNumber; // return index to search deeper
        else // can't find it
            return -1; // can't find it
    }

    public int insertItem(DataItem newItem) // insert new item
    { // assumes node is not full
        if(newItem == null) // null item
            return 0; // return index to insert
        else if(itemArray[0] == null) // item null
            { // examine items
                for(int j=ORDER-2; j>=0; j--) // on right,
                    { // search once
                        itemArray[j].displayItem(); // /56/
                        if(( childNumber=itemArray[j].findItem(key) ) != -1) // if found,
                            return j; // return index to insert new item
                    }
            }
        else if(itemArray[ORDER-1].dData == key) // if item matches
            return ORDER-1; // return index to insert new item
        else // non-matching item
            { // search once
                if(itemArray[0] == null) // node not full
                    return 0; // return index to insert
                else // non-null node
                    return -1; // can't find it
            }
    }

private Node root = new Node(); // make root node
    public int find(long key) // search once
    {
        if(( childNumber=root.findItem(key) ) != -1) // if found,
            return childNumber; // return index to search deeper
        else // can't find it
            return -1; // can't find it
    }

    public int insertItem(DataItem newItem) // insert new item
    { // assumes node is not full
        if(newItem == null) // null item
            return 0; // return index to insert
        else if(itemArray[0] == null) // item null
            { // examine items
                for(int j=ORDER-2; j>=0; j--) // on right,
                    { // search once
                        itemArray[j].displayItem(); // /56/
                        if(( childNumber=itemArray[j].findItem(key) ) != -1) // if found,
                            return j; // return index to insert new item
                    }
            }
        else if(itemArray[ORDER-1].dData == key) // if item matches
            return ORDER-1; // return index to insert new item
        else // non-matching item
            { // search once
                if(itemArray[0] == null) // node not full
                    return 0; // return index to insert
                else // non-null node
                    return -1; // can't find it
            }
    }

private Node root = new Node(); // make root node
    public int find(long key) // search once
    {
        if(( childNumber=root.findItem(key) ) != -1) // if found,
            return childNumber; // return index to search deeper
        else // can't find it
            return -1; // can't find it
    }

    public int insertItem(DataItem newItem) // insert new item
    { // assumes node is not full
        if(newItem == null) // null item
            return 0; // return index to insert
        else if(itemArray[0] == null) // item null
            { // examine items
                for(int j=ORDER-2; j>=0; j--) // on right,
                    { // search once
                        itemArray[j].displayItem(); // /56/
                        if(( childNumber=itemArray[j].findItem(key) ) != -1) // if found,
                            return j; // return index to insert new item
                    }
            }
        else if(itemArray[ORDER-1].dData == key) // if item matches
            return ORDER-1; // return index to insert new item
        else // non-matching item
            { // search once
                if(itemArray[0] == null) // node not full
                    return 0; // return index to insert
                else // non-null node
                    return -1; // can't find it
            }
    }

private Node root = new Node(); // make root node
    public int find(long key) // search once
    {
        if(( childNumber=root.findItem(key) ) != -1) // if found,
            return childNumber; // return index to search deeper
        else // can't find it
            return -1; // can't find it
    }

    public int insertItem(DataItem newItem) // insert new item
    { // assumes node is not full
        if(newItem == null) // null item
            return 0; // return index to insert
        else if(itemArray[0] == null) // item null
            { // examine items
                for(int j=ORDER-2; j>=0; j--) // on right,
                    { // search once
                        itemArray[j].displayItem(); // /56/
                        if(( childNumber=itemArray[j].findItem(key) ) != -1) // if found,
                            return j; // return index to insert new item
                    }
            }
        else if(itemArray[ORDER-1].dData == key) // if item matches
            return ORDER-1; // return index to insert new item
        else // non-matching item
            { // search once
                if(itemArray[0] == null) // node not full
                    return 0; // return index to insert
                else // non-null node
                    return -1; // can't find it
            }
    }
Java Code for a 2-3-4 Tree
Listing for tree234.java (10)

```java
// deal with parent
itemIndex = parent.insertItem(itemB); // item B to parent
int n = parent.getNumItems();         // total items?
for(int j=n-1; j>itemIndex; j--){          // move parent's
    Node temp = parent.disconnectChild(j); // one child
    parent.connectChild(j+1, temp);        // to the right
}
// connect newRight to parent
parent.connectChild(itemIndex+1, newRight);
// deal with newRight
newRight.insertItem(itemC);       // item C to newRight
newRight.connectChild(0, child2); // connect to 0 and 1
newRight.connectChild(1, child3); // on newRight
}
```
2-3-4 Trees and Red-Black Trees

- At this point 2-3-4 trees and red-black trees (Chapter 9) probably seem like entirely different entities.
- However, it turns out that in a certain sense they are completely equivalent.
  - One can be transformed into the other by the application of a few simple rules.
  - The operations needed to keep them balanced are equivalent.
  - Mathematicians would say they were isomorphic.
- You probably won’t ever need to transform a 2-3-4 tree into a red-black tree,
  - but equivalence of these structures casts additional light on their operation and is useful in analyzing their efficiency.
- Historically
  - the 2-3-4 tree was developed first;
  - later the red-black tree evolved from it.

Transformation from 2-3-4 to Red-Black

Figure 10.14 shows a 2-3-4 tree and the corresponding red-black tree obtained by applying these transformations.
- Dotted lines surround the subtrees that were made from 3-nodes and 4-nodes.
- The red-black rules are automatically satisfied by the transformation.
  - Check that this is so:
    - two red nodes are never connected,
    - and there is the same number of black nodes on every path from root to leaf (or null child).

Operational Equivalence

- Not only does the structure of a red-black tree correspond to a 2-3-4 tree,
  - but the operations applied to these two kinds of trees are also equivalent.
- In a 2-3-4 tree the tree is kept balanced using node splits
- In a red-black tree the two balancing methods are color flips and rotations.

4-Node Splits and Color Flips

- As you descend a 2-3-4 tree searching for the insertion point for a new node,
  - you split each 4-node into two 2-nodes.
- In a red-black tree you perform color flips.
- How are these operations equivalent?
4-Node Splits and Color Flips
- In Figure 10.15-a, a 4-node in a 2-3-4 tree before it is split.
- Figure 10.15-b shows the situation after the split.
  - The 2-node that was the parent of the 4-node becomes a 3-node.
- In Figure 10.15-c, we show the red-black equivalent to the 2-3-4 tree in 10.15-a.
  - The dotted line surrounds the equivalent of the 4-node.
- A color flip results in the red-black tree of Figure 10.15-d.
  - Now nodes 40 and 60 are black and 50 is red.
  - Thus 50 and its parent form the equivalent of a 3-node, as shown by the dotted line.
  - This is the same 3-node formed by the node split in Figure 10.15-b.
- Splitting a 4-node during the insertion process in a 2-3-4 tree is equivalent to performing color flips during the insertion process in a red-black tree.

3-Node Splits and Rotations
- Although these arrangements are equally valid, the tree in b) is not balanced, while that in c) is.
- Given the red-black tree in b),
  - we would want to rotate it to the right (and perform two color changes) to balance it.
  - Amazingly, this rotation results in the exact same tree shown in c).
- Thus we see an equivalence between rotations in red-black trees and the choice of which node to make the parent when transforming 2-3-4 trees to red-black trees.
- Although we don’t show it, a similar equivalence can be seen for the double rotation necessary for inside grandchildren.

Efficiency of 2-3-4 Trees
- In a red-black tree,
  - one node on each level must be visited during a search, whether to find an existing node or insert a new one.
  - The number of levels in a red-black tree (a balanced binary tree) is about log₂(N+1), so search times are proportional to this.
- In a 2-3-4 tree,
  - one node must be visited at each level as well,
  - but the 2-3-4 tree is shorter (has fewer levels) than a red-black tree with the same number of data items.
- Refer to Figure 10.14, where the 2-3-4 tree has three levels and the red-black tree has five.
Efficiency of 2-3 Trees

Speed
- On the other hand, there are more items to examine in each node, which increases the search time.
- Because the data items in the node are examined using a linear search, this multiplies the search times by an amount proportional to M, the average number of items per node.
- The result is a search time proportional to M log N.
- Some nodes contain 1 item, some 2, and some 3.
- If we estimate that the average is 2, search times will be proportional to 2 log N.
- This is a small constant number that can be ignored in Big O notation.
- Thus, for 2-3 trees the increased number of items per node tends to cancel out the decreased height of the tree.
- The search times for a 2-3 tree and for a balanced binary tree such as a red-black tree are approximately equal, and are both O(log N).

Efficiency of 2-3 Trees

Storage Requirements
- Each node in a 2-3-4 tree contains storage for three references to data items and four references to its children.
- This space may be in the form of arrays, as shown in tree234.java, or of individual variables.
- Not all this storage is used.
- A node with only one data item will waste two thirds of the space for data and half the space for children.
- A node with two data items will waste one third of the space for data and one quarter of the space for children.
- To put it another way, it will use 5/7 of the available space.
- If we take two data items per node as the average utilization, about 2/7 of the available storage is wasted.
- One might imagine using linked lists instead of arrays to hold the child and data references, but the overhead of the linked list compared with an array, for only 3 or 4 items, would probably not make this a worthwhile approach.
- Because they’re balanced, red-black trees contain fewer nodes that have only one child, so almost all the storage for child references is used.
- Also, every node contains the maximum number of data items, which is 1. This makes red-black trees more efficient than 2-3-4 trees in terms of memory usage.

External Storage

Accessing External Data
- The data structures we’ve discussed so far are all based on the assumption that data is stored entirely in main memory (often called RAM, for Random Access Memory).
- However, in many situations the amount of data to be processed is too large to fit in main memory all at once.
- In this case a different kind of storage is necessary.
- Disk files generally have a much larger capacity than main memory;
  - this is made possible by their lower cost per byte of storage.
- Of course, disk files have another advantage: their permanence.
  - When you turn off your computer (or the power fails), the data in main memory is lost.
  - Disk files can retain data indefinitely with the power off.
  - However, it’s mostly the size difference that we’ll be involved with here.
- The disadvantage of external storage is that it’s much slower than main memory.
  - This speed difference means that different techniques must be used to handle it efficiently.

External Storage

As an example of external storage,
- imagine that you’re writing a database program to handle the data found in the phone book for a medium-sized city; perhaps 500,000 entries.
  - Each entry includes a name, address, phone number, and various other data used internally by the phone company.
  - Let’s say an entry is stored as a record requiring 512 bytes.
  - The result is a file size of 500,000x512, which is 256,000,000 bytes or 256 megabytes.
  - We’ll assume that on the target machine this is too large to fit in main memory, but small enough to fit on your disk drive.
- Thus you have a large amount of data on your disk drive.
  - How do you structure it to provide the usual desirable characteristics: quick search, insertion, and deletion?
- Keep in mind two facts:
  - First, accessing data on a disk drive is much slower than accessing it in main memory.
  - Second, you must access many records at once. Let’s explore these points.
Very Slow Access

- A computer’s main memory works electronically.
  - Any byte can be accessed just as fast as any other byte, in a fraction of a microsecond (a millionth of a second).
- Things are more complicated with disk drives.
  - Data is arranged in circular tracks on a spinning disk, something like the tracks on a compact disc (CD) or the grooves in an old-style phonograph record.
  - To access a particular piece of data on a disk drive, the read-write head must first be moved to the correct track.
    - This is done with a stepping motor or similar device; it’s a mechanical activity that requires several milliseconds (thousandths of a second).

External Storage
Accessing External Data

Very Slow Access

- Once the correct track is found,
  - the read-write head must wait for the data to rotate into position.
    - On the average, this takes half a revolution.
    - Even if the disk is spinning at 10,000 revolutions per minute, about 3 more milliseconds pass before the data can be read.
    - Once the read-write head is positioned, the actual reading (or writing) process begins; this might take a few more milliseconds.
- Thus, disk access times of around 10 milliseconds are common.
  - This is something like 10,000 times slower than main memory.
  - Technological progress is reducing disk access times every year, but main memory access times are being reduced faster,
    - so the disparity between disk access and main memory access times will grow even larger in the future.

One Block at a Time

- Once it is correctly positioned and the reading (or writing) process begins,
  - a disk drive can transfer a large amount of data to main memory fairly quickly.
    - For this reason, and to simplify the drive control mechanism, data is stored on the disk in chunks called blocks, pages, allocation units, or some other name, depending on the system.
      - We’ll call them blocks.
- The disk drive always reads or writes a minimum of one block of data at a time.
- Block size varies, depending on the operating system, the size of the disk drive, and other factors, but it is usually a power of 2.
  - For our phone book example, let’s assume a block size of 8,192 bytes (2\(^{13}\)).
  - Thus our phone book database will require 256,000,000 bytes divided by 8,192 bytes per block, which is 31,250 blocks.
- Your software is most efficient when it specifies a read or write operation that’s a multiple of the block size.
  - If you ask to read 100 bytes, the system will read one block, 8,192 bytes, and throw away all but 100.
  - Or if you ask to read 8,200 bytes, it will read two blocks, or 16,384 bytes, and throw away almost half of them.
  - By organizing your software so that it works with a block of data at a time you can optimize its performance.
  - Assuming our phone book record size of 512 bytes, you can store 16 records in a block (8,192 divided by 512), as shown in Figure 10.17.
  - Thus for maximum efficiency it’s important to read 16 records at a time (or multiples of this number).
  - It’s also useful to make your record size a multiple of 2.
    - That way an integral number of them will always fit in a block.

Sequential Ordering

- One way to arrange the phone book data in the disk file would be to order all the records according to some key, say alphabetically by last name.
- The record for Joseph Aardvark would come first, and so on.
  - Shown in Figure 10.18.
Searching

• To search a sequentially ordered file for a particular last name such as Smith,
  – you could use a binary search. You would start by reading a block of records from the middle of the file.
  – The 16 records in the block are all read at once into a 8,192-byte buffer in main memory.
• If the keys of these records are too early in the alphabet (Keller, for example),
  – you’d go to the 3/4 point in the file (Prince) and read a block there;
  – if the keys were too late, you’d go to the 1/4 point (DeLeon).
  – By continually dividing the range in half you’d eventually find the record you were looking for.

As we saw in Chapter 2, a binary search in main memory takes \( \log_2 N \) comparisons, which for 500,000 items is about 19.

– If every comparison took, say 10 microseconds, this would be 190 microseconds, or about 2/10,000 of a second; less than an eye blink.
– However, we’re now dealing with data stored on a disk.
  – Because each disk access is so time consuming, it’s more important to focus on how many disk accesses are necessary than on how many individual records there are.
  – The time to read a block of records will be very much larger than the time to search the 16 records in the block once they’re in memory.
– Disk accesses are much slower than memory accesses, but on the other hand we access a block at a time, and there are far fewer blocks than records.
  – In our example there are 31,250 blocks. \( \log_2 \) of this number is about 15, so in theory we’ll need about 15 disk accesses to find the record we want.

In practice this number is reduced somewhat because we read 16 records at once.

– In the beginning stages of a binary search it doesn’t help to have multiple records in memory because the next access will be in a distant part of the file.
  – However, when we get close to the desired record, the next record we want may already be in memory because it’s part of the same block of 16.
  – This may reduce the number of comparisons by two or so.
  – Thus we’ll need about 13 disk accesses (15 – 2), which at 10 milliseconds per access requires about 130 milliseconds, or 1/7 second
  – This is much slower than in-memory access, but still not too bad.

Insertion

• Unfortunately the picture is much worse if we want to insert (or delete) an item from a sequentially ordered file.
  – Because the data is ordered, both operations require moving half the records on the average, and therefore about half the blocks.
  – Moving each block requires two disk accesses, one read and one write.
  – Once the insertion point is found, the block containing it is read into a memory buffer.
  – The last record in the block is saved, and the appropriate number of records are shifted up to make room for the new one, which is inserted.
  – Then the buffer contents are written back to the disk file.

  • How can the records of a file be arranged to provide fast search, insertion, and deletion times?
  – Trees are a good approach to organizing in-memory data.
  – Will trees work with files?
  • Yes, but a different kind of tree must be used for external data.
    – A multiway tree somewhat like a 2-3-4 tree, but with many more data items per node;
      • it’s called a B-tree
      • B-trees were first conceived as appropriate structures for external storage by R. Bayer and E. M. McCreight in 1972.
One Block Per Node

• Why do we need so many items per node?
  – We’ve seen that disk access is most efficient when data is read or written one block at a time.
  – In a tree, the entity containing data is a node.
  – It makes sense then to store an entire block of data in each node of the tree.
    • This way, reading a node accesses a maximum amount of data in the shortest time.

• How much data can be put in a node?
  – When we simply stored the 512-byte data records for our phone book example, we could fit 16 into a 8,192-byte block.

In a tree,
  – Also need to store the links to other nodes (which means links to other blocks, because a node corresponds to a block).

For block numbers we can use a field of type int, a 4-byte type, which can point to more than 2 billion possible blocks; probably enough for most files.

We could reduce the number of records to 15 to make room for the links, but it’s most efficient to have an even number of records per node, so after appropriate negotiation with management we reduce the record size to 507 bytes.

There will be 17 child links (one more than the number of data items) so the links will require 68 bytes (17x4).

A block in such a tree, and the corresponding node representation, is shown in Figure 10.19.

Within each node the data is ordered sequentially by key, as in a 2-3-4 tree.

In fact, the structure of a B-tree is similar to that of a 2-3-4 tree,
  – except that there are more data items per node and more links to children.

The order of a B-tree is the number of children each node can potentially have.
  – In our example this is 17, so the tree is an order 17 B-tree.

Carried out in much the same way it is in an in memory 2-3-4 tree.
  – First, the block containing the root is read into memory.
  – The search algorithm then starts examining each of the 15 records (or, if it’s not full, as many as the node actually holds), starting at 0.
  – When it finds a record with a greater key, it knows to go to the child whose link lies between this record and the preceding one.
  – This process continues until the correct node is found.
  – If a leaf is reached without finding the specified key, the search is unsuccessful.

Somewhat different than it is in a 2-3-4 tree.

Recall that
  – In a 2-3-4 tree many nodes are not full, and in fact contain only one data item.
    • In particular, a node split always produces two nodes with one item in each.
      – This is not an optimum approach in a B-tree.
  – In a B-tree it’s important to keep the nodes as full as possible so that each disk access, which reads an entire node, can acquire the maximum amount of data.
    • To help achieve this end, the insertion process differs from that of 2-3-4 trees in three ways:
      • A node split divides the data items equally: half go to the newly created node, and half remain in the old one.
      • Node splits are performed from the bottom up rather than from the top down.
      • It’s not the middle item in a node that’s promoted upward, but the middle item in the sequence formed from the items in the node plus the new item.

We’ll demonstrate these features of the insertion process by building a small B-tree, as shown in Figure 10.20.

There isn’t room to show a realistic number of records per node, so we’ll show only four; thus the tree is an order 5 B-tree.

There’s room to show a realistic number of records per node, so we’ll show only four; thus the tree is an order 5 B-tree.

– shows a root node that’s already full; items with keys 20, 40, 60, and 80 have already been inserted into the tree.
– A new data item with a key of 70 is inserted, resulting in a node split.

A node in a B-tree of order 17

A node in a B-tree of order 17

A node in a B-tree of order 17
Although searching is faster in B-trees than in sequentially ordered disk files, it’s for insertion and deletion that B-trees show the greatest advantage.

**Efficiency of B-Trees**

- The more records there are in a node, the fewer levels there are in the tree.
  - We’ve seen that there are 6 levels in our B-tree, even though the nodes hold only 16 records.
- In contrast, a binary tree with 500,000 items would have about 19 levels, and a 2-3-4 tree would have 10.
- If we use blocks with hundreds of records, we can reduce the number of levels in the tree and further improve access times.
- Although searching is faster in B-trees than in sequentially ordered disk files, it’s for insertion and deletion that B-trees show the greatest advantage.

**External Storage**

**B-Trees**

**Insertion**

- Because it’s the root that’s being split, two new nodes are created (as in a 2-3 tree):
  - a new root and a new node to the right of the one being split.
- To decide where the data items go, the insertion algorithm arranges their 5 keys in order, in an internal buffer.
  - Four of these keys are from the node being split, and the fifth is from the new item being inserted.
  - In Figure 10.20, these 5-item sequences are shown to be 10, 40, 60, 70, 80.
  - In this first step the sequence 20, 40, 60, 70, 80 is shown.
  - The center item in this sequence, 60, in this first step, is promoted to the new root node.
    - (In the figure, an arrow indicates that the center item will go upward.)
  - All the items to the left of center remain in the node being split, and all the items to the right go into the new right-hand node.
  - The result is shown in Figure 10.20-a. (In our phone book example, 8 items would go into each child node, rather than the 2 shown in the figure.)

- This is a major improvement over the 500,000 accesses required for a total of 12.
- In our phone book example, there are 500,000 records.
- Operations on B-trees are very fast, considering that the data is stored on disk.
  - In our phone book example there are 500,000 records.
  - All the nodes in the B-tree are at least half full, so they contain at least 8 records and 9 links to children.
  - The height of the tree is thus somewhat less than log9N (logarithm to the base 9 of N), where N is 500,000.
    - This is 5.972, so there will be about 6 levels in the tree.
  - Thus, using a B-tree, only six disk accesses are necessary to find any record in a file of 500,000 records.
    - At 10 milliseconds per access, this takes about 60 milliseconds, or 6/100 of a second.
  - Dramatically faster than the binary search of a sequentially ordered file.

- Next let’s see how things look if a node must be split.
  - The newly created node must be written to the disk, and the parent must be written back to disk.
  - The new root is promoted upward into the tree.
  - The newly created child is promoted to the new root node.

**Insertion**

- The next item to be inserted, 15, splits this left child, with the result shown in Figure 10.20-b.
  - Here the 20 has been promoted upward into the tree.
  - The second item is split, causing the creation of a new node and the promotion of the middle item, 30, to the root.
  - The result is shown in Figure 10.20-c.

- Again three items, 25, 35, and 50, are added to the tree.
  - The first two items fill up the second child, and the third one splits it, causing the creation of a new node and the promotion of the middle item, 35, to the root.
  - As shown in Figure 10.20-d.
  - The next item to be inserted, 32, does cause a split.
    - In fact it causes two of them.
    - The second node child is full, so it’s split.
    - As shown in Figure 10.20-e.
  - However, the 27, promoted from this split, has no place to go because the root is full.
    - Therefore, the root must be split as well, resulting in Figure 10.20-f.
  - Notice that throughout the insertion process no node (except the root) is ever less than half full.
  - As we noted, this promotes efficiency because a file access that reads a node always acquires a substantial amount of data.

**Insertion**

- The next item to be inserted, 15, splits this left child, with the result shown in Figure 10.20-c.
  - Here the 20 has been promoted upward into the tree.
  - The second item is split, causing the creation of a new node and the promotion of the middle item, 30, to the root.
  - The result is shown in Figure 10.20-c.

- Again three items, 25, 35, and 50, are added to the tree.
  - The first two items fill up the second child, and the third one splits it, causing the creation of a new node and the promotion of the middle item, 35, to the root.
  - As shown in Figure 10.20-d.

- The next item to be inserted, 32, does cause a split.
  - In fact it causes two of them.
  - The second node child is full, so it’s split.
  - As shown in Figure 10.20-e.

- However, the 27, promoted from this split, has no place to go because the root is full.
  - Therefore, the root must be split as well, resulting in Figure 10.20-f.

- Notice that throughout the insertion process no node (except the root) is ever less than half full.
- As we noted, this promotes efficiency because a file access that reads a node always acquires a substantial amount of data.

**External Storage**

**B-Trees**

**Insertion**

- Now the root is full.
  - However, subsequent insertions don’t necessarily cause a node split, because nodes are split only when a new item is inserted into a full node, not when a full node is encountered in the search down the tree.
- Thus 22 and 27 are inserted in the second child without causing any splits.
  - As shown in Figure 10.20-g.

- However, the next item to be inserted, 32, does cause a split.
  - In fact it causes two of them.
    - The second node child is full, so it’s split.
    - As shown in Figure 10.20-h.
  - However, the 27, promoted from this split, has no place to go because the root is full.
    - Therefore, the root must be split as well, resulting in Figure 10.20-i.
  - Notice that throughout the insertion process no node (except the root) is ever less than half full.
  - As we noted, this promotes efficiency because a file access that reads a node always acquires a substantial amount of data.
External Storage

B-Trees

Efficiency of B-Trees

- In some versions of the B-tree, only leaf nodes contain records. Non-leaf nodes contain only keys and block numbers.
  - This may result in faster operation because each block can hold many more block numbers.
  - The resulting higher-order tree will have fewer levels, and access speed will be increased.
  - However, programming may be complicated because there are two kinds of nodes: leaves and non-leaves.

External Storage

Indexing

- A different approach to speeding up file access is to store records in sequential order but use a file index along with the data itself.
- A file index is a list of key/block pairs, arranged with the keys in order.
- Recall that in our original phone book example we had 500,000 records of 512 bytes each, stored 16 records to a block, in 31,250 blocks.
- Assuming our search key is the last name, every entry in the index contains two items:
  - The key, such as Jones.
  - The number of the block where the Jones record is located within the file. These numbers run from 0 to 31,249.

External Storage

Indexing

- Let's say we use a string 28 bytes long for the key (big enough for most last names), and 4 bytes for the block number (a type `int` in Java).
- Each entry in our index thus requires 32 bytes.
  - This is only 1/16 the amount necessary for each record.
- The entries in the index are arranged sequentially by last name.
- The original records on the disk can be arranged in any convenient order.
- This usually means that new records are simply appended to the end of the file, so the records are ordered by time of insertion.
- This arrangement is shown in Figure 10.21.

External Storage

Indexing

- The index-in-memory approach allows much faster operations on the phone book file than are possible with a file in which the records themselves are arranged sequentially.
  - For example,
    - a binary search requires 19 index accesses. At 20 microseconds per access, that's only about 4/10,000 of a second.
    - Then there's (inevitably) the time to read the actual record from the file, once its block number has been found in the index. However, this is only one disk access of (say) 10 milliseconds.

External Storage

Indexing

- Index File in Memory
  - Because it's so much smaller than the file containing actual records, it may be that the index is small enough to fit entirely in main memory.
  - In our example there are 500,000 records. Each one has a 32-byte entry in the index, so the index will be 32x500,000 or 1,600,000 bytes long (1.6 megabytes).
  - In modern computers there's no problem fitting this in memory.
  - The index can be stored on the disk, but read into memory whenever the database program is started up.
  - From then on, operations on the index can take place in memory.
  - At the end of the day (or perhaps more frequently) the index can be written back to disk for permanent storage.

External Storage

Indexing

- Insertion
  - To insert a new item in an indexed file two steps are necessary.
    - We first insert its full record into the main file; then we insert an entry, consisting of the key and the block number where the new record is stored, into the index.
  - Because the index is in sequential order, to insert a new item we need to move half the index entries, on the average.
    - Figuring 2 microseconds to move a byte in memory, we have 250,000 times 32 bytes 2, or about 16 seconds to insert a new entry.
    - This compares with five minutes for the unindexed sequential file. (Note that we don't need to move any records in the main file; we simply append the new record at the end of the file.)
External Storage
Indexing

**Insertion**
- Of course, you can use a more sophisticated approach to storing the index in memory.
- You could store it as a binary tree, 2-3-4 tree, or red-black tree, for example.
- Any of these would significantly reduce insertion and deletion times.
- In any case the index-in-memory approach is much faster than the sequential-file approach.
- In some cases it will also be faster than a B-tree.
- The only actual disk accesses necessary for an insertion into an indexed file involve the new record itself.
- Usually the last block in the file is read into memory, the new record is appended, and the block is written back out.
  - This involves only two file accesses.

**Multiple Indexes**
- An advantage of the indexed approach is that multiple indexes, each with a different key, can be created for the same file.
- In one index the keys can be last names, in another telephone numbers, in another addresses.
- Because the indexes are small compared with the file, this doesn't increase the total data storage very much.
- Of course, it does present more of a challenge when items are deleted from the file, because entries must be deleted from all the indexes, but we won't get into that here.

**Index Too Large for Memory**
- If the index is too large to fit in memory, it must be broken into blocks and stored on the disk.
- For large files it may then be profitable to store the index itself as a B-tree.
- In the main file the records are stored in any convenient order.
- This arrangement can be very efficient.
  - Appending records to the end of the main file is a fast operation, and the index entry for the new record is also quick to insert because the index is a tree.
  - The result is very fast searching and insertion for large files.
- Note that when an index is arranged as a B-tree, each node contains a number of child pointers and one fewer data items.
- The child pointers are the block numbers of other nodes in the index.
- The data items consist of a key value and a pointer to a block in the main file.
- Don't confuse these two kinds of block pointers.

**Complex Search Criteria**
- In complex searches the only practical approach may be to read every block in a file sequentially.
  - Suppose in our phone book example we wanted a list of all entries in the phone book with first name Frank, who lived in Springfield, and who had a phone number with three “7” digits in it. (These were perhaps clues found scrawled on a scrap of paper clutched in the hand of a victim of foul play.)
- A file organized by last names would be no help at all.
  - Even if there were index files ordered by first names and cities, there would be no convenient way to find which files contained both Frank and Springfield.
- In such cases (which are quite common in many kinds of databases) the fastest approach is probably to read the file sequentially, block by block, checking each record to see if it meets the criteria.

External Storage
Sorting External Files

- Mergesort is the preferred algorithm for sorting external data.
  - Because, more so than most sorting techniques, disk accesses tend to occur in adjacent records rather than random parts of the file.
  - Recall from Chapter 6, “Recursion,” that mergesort works recursively by calling itself to sort smaller and smaller sequences.
    - Once two of the smallest sequences (one byte each in the internal-memory version) have been sorted, they are then merged into a sorted sequence twice as long.
    - Larger and larger sequences are merged, until eventually the entire file is sorted.
  - The approach for external storage is similar (Next slide)
### External Storage

#### Sorting External Files

- **Figure 10.22** shows the mergesort process on an external file.
- The file consists of four blocks of four records each, for a total of 16 records.
- Only three blocks can fit in internal memory. (Of course all these sizes would be much larger in a real situation.)
- Figure 10.22-a shows the file before sorting; the number in each record is its key value.

#### Internal Sort of Blocks

- In the first phase
  - all the blocks in the file are sorted internally.
  - This is done by reading the block into memory and sorting it with any appropriate internal sorting algorithm, such as Quicksort (or for smaller numbers of records, shellsort or insertion sort).
  - The result of sorting the blocks internally is shown in Figure 10.22-b.
  - The dotted lines in the figure separate sorted records; solid lines separate unsorted records.
  - A second file may be used to hold the sorted blocks, and we assume that availability of external storage is not a problem.
  - It’s often desirable to avoid modifying the original file.

#### Merging

- In the second phase
  - we want to merge the sorted blocks.
  - In the first pass
    - we merge every pair of blocks into a sorted 2-block sequence.
    - Thus the two blocks 2-9-11-14 and 4-12-13-16 are merged into 2-9-11-12-13-14-16.
    - Also, 3-5-10-15 and 1-6-7-8 are merged into 1-3-5-6-7-8-10-15.
    - The result is shown in Figure 10.22-c.
  - A third file is necessary to hold the result of this merge step.
- In the second pass,
  - the two 8-record sequences are merged into a 16-record sequence, and the sort is complete.
- Of course more merge steps would be required to sort larger files; the number of such steps is proportional to log_2 N. The merge steps can alternate between two files.

#### Internal Arrays

- The following lists show the details of each of the three mergesorts.
  - **Mergesort 1:**
    1. Read 2-9-11-14 into arr1
    2. Read 4-12-13-16 into arr2
    3. Merge 2, 4, 9, 11 into arr3; write to disk
    4. Merge 12, 13, 14, 16 into arr3; write to disk

  - **Mergesort 2:**
    1. Read 3-5-10-15 into arr1
    2. Read 1-6-7-8 into arr2
    3. Merge 1, 3, 5, 6 into arr3; write to disk
    4. Merge 7, 8, 10, 15 into arr3, write to disk
Internal Arrays

Sorting External Files

- Mergesort 3:
  1. Read 2-4-9-11 into arr1
  2. Read 1-3-5-6 into arr2
  3. Merge 1, 2, 3, 4 into arr3; write to disk
  4. Merge 5, 6 into arr3 (arr2 is now empty)
  5. Read 7-8-10-15 into arr1
  6. Merge 7, 8 into arr3; write to disk
  7. Merge 9, 10, 11 into arr3 (arr1 is now empty)
  8. Read 12-13-14-16 into arr1
  9. Merge 12 into arr3; write to disk
  10. Merge 13, 14, 15, 16 into arr3; write to disk

Figure 10.22: Mergesort on an external file

Helpful to examine the details of the array contents as the steps are completed.

Figure 10.23 shows how these arrays look at various stages of Mergesort 3.

External Storage

- This last sequence of 10 steps is rather lengthy.
- External storage means storing data outside of main memory; usually on a disk.
- External storage is larger, cheaper (per byte), and slower than main memory.
- Data in external storage is typically transferred to and from main memory a block at a time.
- Data can be arranged in external storage in sequential key order. This gives fast search times but slow insertion (and deletion) times.
- A B-tree is a multiway tree in which each node may have dozens or hundreds of keys and children.
- There is always one more child than there are keys in a B-tree node.
- For the best performance, a B-tree is typically organized so that a node holds one block of data.
- If the search criteria involve many keys, a sequential search of all the records in a file may be the most practical approach.