B-Trees

An *index* is a very general concept: it is an ordered collection of keys used to gain efficient access to some target data. We attempt to organize an index in such a way as to maximize the efficiency of finding the key we are searching for (and subsequently, the relevant target data).

If the index is small enough to fit into RAM, we can use all the fancy data structures and algorithms we know to search for keys and to maintain order in the index. But sometimes, the index is too big to fit into RAM...

Topic	***Folk & Zoellick***
• Binary Searching in Big Indexes	§ 8.2
• Binary Search Trees for Big Indexes	§ 8.3
• Paged Binary Trees	§ 8.5
• M-Way Search Trees	§ N/A
• B-Trees	§ 8.7
• Operations on B-Trees	§§ 8.8, 8.13

Binary Searching in Big Indexes

When an index file is too big to fit in RAM, we end up searching it on disk. The most efficient searching algorithm we know (for general sorted records) is the binary search.

Let's look at a big sorted index. The index occupies many different clusters on disk. The $64,000 question is:

How expensive is the binary search?

We'll assume that clusters are 2,000 bytes. Records in the index file are 20 bytes, so there are 100 index records in each cluster. Let's say there are a grand total of 3,160 records, occupying 32 clusters.

Searching Binarily

Q: If the record we're looking for turns out to be record 1,440, how many clusters get read during the binary search?

A:

Q: If the record we're looking for is number 1,351, how many clusters get read during the binary search?

A:

Q: If the record we're looking for is in cluster 1,098, how many clusters get read during the binary search?

A:

Q: Do you see what I'm getting at?

A:

Binary Search Besmirched

In our example of 3,120 records in the index file, the binary search *always* starts by reading record 1,560 in cluster 16. Then (assuming 1,560 is not the record we're looking for) it always reads either record 780 in cluster 8 or record 2,340 in cluster 24, etc.

For our search to find record 1,440, here are the records we read and the clusters they're in:

record	1,560	780	1,170	1,365	1,463	etc.
cluster	16	8	12	14	15	15

For our search to find record 1,351, here are the records we read and the clusters they're in:

record	1,560	780	1,170	1,365	1,268	1,316	etc.
cluster	16	8	12	14	13	14	14

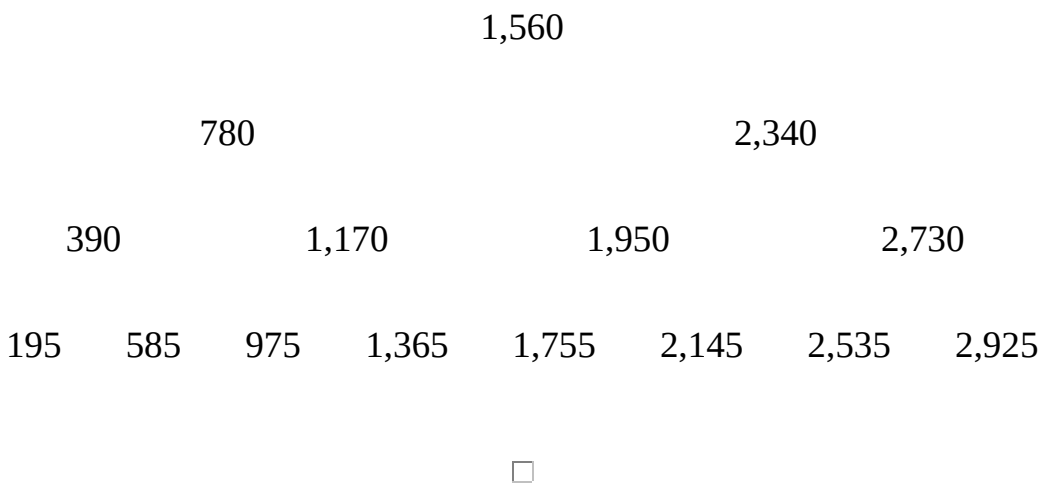
Both searches begin by looking at the exact same records. Yet each of those single records requires a different cluster to be read.

There must be a better way!

Binary Trees

In the previous example, the records we were looking for (1,440 and 1,351) were quite close together. That's why searching for them involved reading mostly the same records.

But even in the general case, the path of records read in the binary search is quite predictable:



Q: What does this look like?

A:

Q: Ya, I know, but what *kind*?

A:

Binary Search Trees

So in an index of close to 3,200 records, a binary search will read at most about $\log_2(3{,}200) = 12$ records in order to find the desired record.

The first four records read will always be among the fifteen records above.

In fact, out of 3,200 records, the first 7 reads in a binary search to find any record will always be among the same 128 records. Our clusters can hold almost 128 records.

Why don't we put all those records in the same cluster!

Paged Binary Trees

If we store our index as a tree instead of as a sorted sequence of records we can minimize the number of clusters that need to be read in order to find any given record.

But to store the index as a tree, we need to keep a bit of extra information in each record:

primary key	*reference field*	*left child RRN*	*right child RRN*

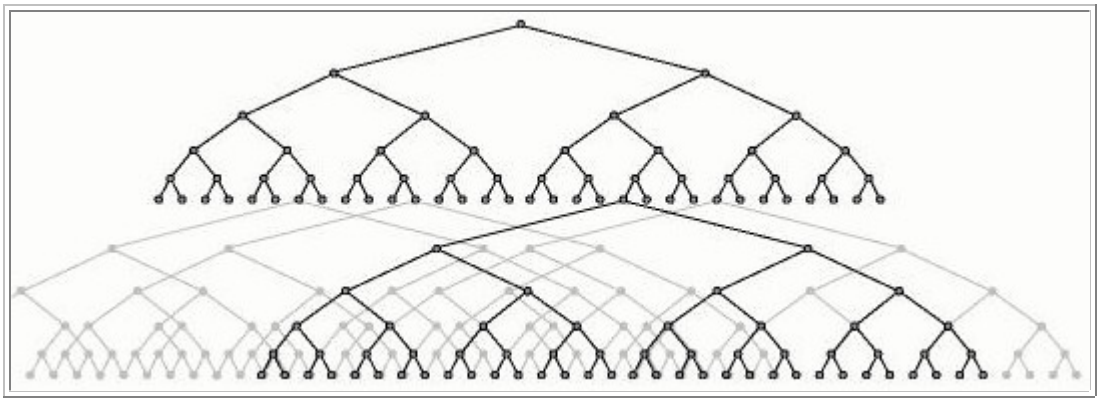
This way, we can place the records exactly where we want them.

If our cluster size is 2,048 bytes and our records are 32 bytes, we can fit 64 records in each cluster. That's the first *six* records on *every* binary search path in one cluster ($2^6 \times 32 = 64 \times 32 = 2{,}048$).

If each of the leaves in the tree is considered the root of another tree, each of those subtrees can store the remaining six records of every binary search path (starting from the current root) in one cluster as well.

I think we need a picture.

Paged Binary Trees (page 2)



Q: If the top six levels of records in the tree are contained in one cluster, and the tree containing six levels of records starting at each leaf is contained in one cluster, how many clusters are occupied by the entire index?

A:

Q: What is the maximum number of clusters that will be read in any binary search through the index?

A:

The Good

The number of comparisons required for a binary search is $O(\log_2 N)$, where N is the number of records being compared.

If we do a straight binary search through our sorted sequence of index records, the *number of clusters read* is almost the same as the number of comparisons (since subsequent comparisons are usually from different clusters). In fact, the number of clusters read in the straight binary search is about $\frac{1}{2}\log_2 N$, which is $O(\log_2 N)$.

If we used a paged binary tree instead of the sorted sequence, the number of comparisons is still $O(\log_2 N)$, but the number of clusters read is $O(\log_K N)$, where K is the number of records per cluster.

In our example, the number of clusters read in the binary search on the sorted sequence is about

$$\frac{1}{2}\log_2(3{,}120) = 6$$

If we use the paged binary tree instead... K is 64, so the number of clusters read per search is

$$\log_{64}(3{,}120) = 2$$

The Bad

In order to benefit from the notion of binary search, we must cut our search space in half every time we make a comparison. In the paged binary tree, that means that roughly half the descendants of every node should be in the left subtree, and half should be in the right.

That is, the tree should be balanced.

But even if we created the paged binary tree to be balanced, one single record addition or deletion could throw it out of balance.

The Ugly

Data structures courses tell us how to balance an unbalanced tree by performing fancy rotations, swapping, etc. Unfortunately, these operations could result in a large number of cluster reads and rewrites, making tree maintenance impractical.

:-(

We need a new solution.

Mary Search Trees

In a *binary* search tree

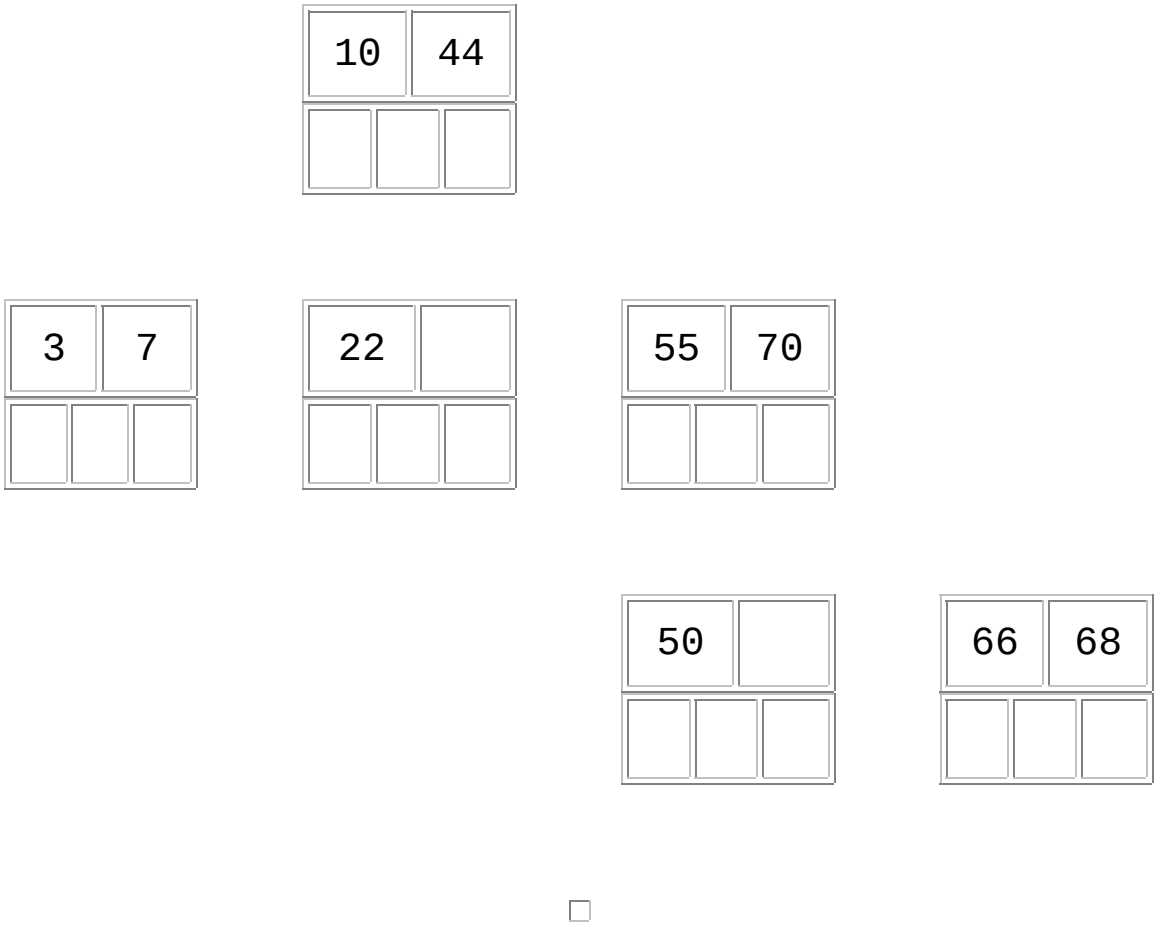
- every node has one value (V) and two pointers (L, R)
- all values in the subtree pointed to by L are less than V
- all values in the subtree pointed to by R are greater than V

In an *M-ary* search tree

- every node has M-1 values ($V_1 ... V_{M-1}$) and M pointers ($P_1 ... P_M$)
- the values in each node are sorted ($V_1 < V_2 < ... V_{M-1}$)
- all values in the subtree pointed to by P_i are $< V_i$
- all values in the subtree pointed to by P_{i+1} are $> V_i$

M-ary Search Tree Example

What I’m really trying to say is:



We call this search tree a *3-way search tree*

M-ary Search Tree Searching

Searching for *k* in a *binary* search tree

- If *k* = the value in this node, done!
- If *k* < the value in this node, look for *k* in the left subtree
- If *k* > the value in this node, look for *k* in the right subtree

Searching for *k* in an *M-ary* search tree

- If *k* = one of the values V_i in this node, done!
- If *k* < value V_1, look for *k* in the subtree pointed to by P_1
- If *k* > value V_{M-1}, look for *k* in the subtree pointed to by P_M
- If *k* > value V_{i-1} and *k* < value V_i, look for *k* in the subtree pointed to by P_i

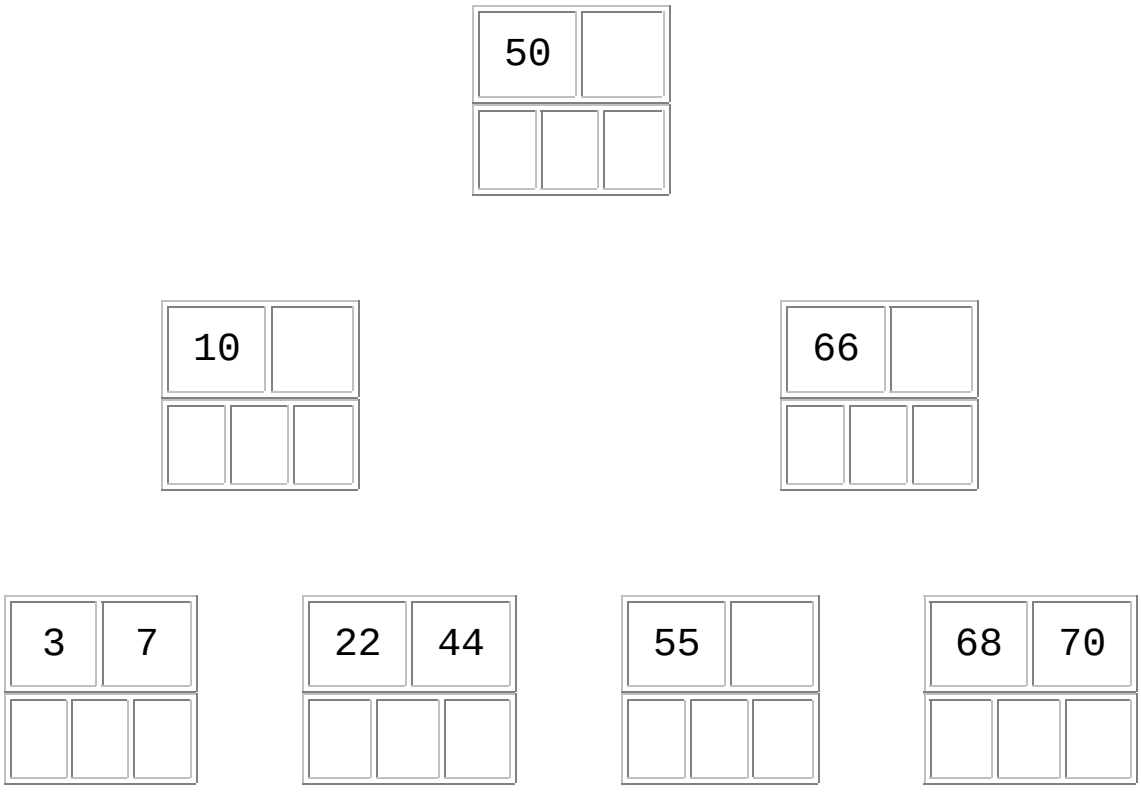
B-Trees

B-Tree

An M-way search tree with three special properties

1. every leaf node is on the same level (*perfectly balanced*)
2. every node except the root is at least half full (from (M-1)/2 to M-1 values)
3. the root may contain any number of values (from 1 to M-1)

Here is a B-tree containing the same values as our previous 3-way search tree. This is known as a *B-tree of order 3*.



B-Tree Insertion

One of the problems with regular old BSTs was that they become easily unbalanced. Balanced tree data structures (such as AVL trees) require heavy maintenance to preserve balance on inserting and deleting nodes.

Inserting a key X in a B-tree of order M

1. do an M-way search tree search to find the leaf node where X should be inserted

2. add X to this leaf node in the correct position

3. if there are now ≤ M-1 values in this node, done!

   if there are M values, this node has *overflowed* so:

   i. split the node in 3: Left, Middle and Right:

      Left becomes a new node containing the first (M-1)/2 values

      Middle is just the middle value

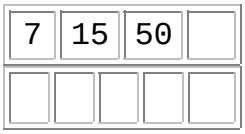
      Right becomes a new node containing the last (M-1)/2 values

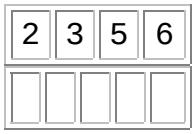
   ii. promote Middle to this node's parent, making Left and Right its left and right children

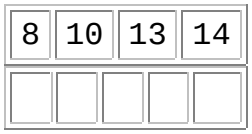
   repeat step 3 for this node's parent node

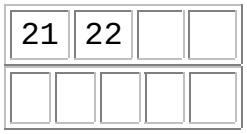
B-Tree Insertion Example

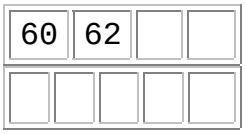
Here’s an example of a B-tree of order 5 (M = 5):



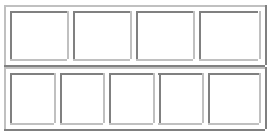


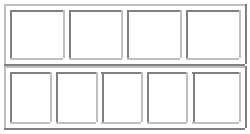


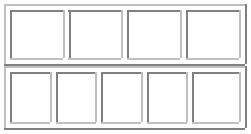


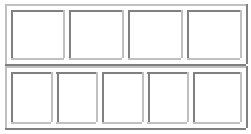


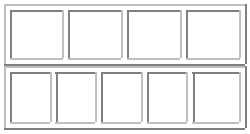
Let’s insert the key with value 29:





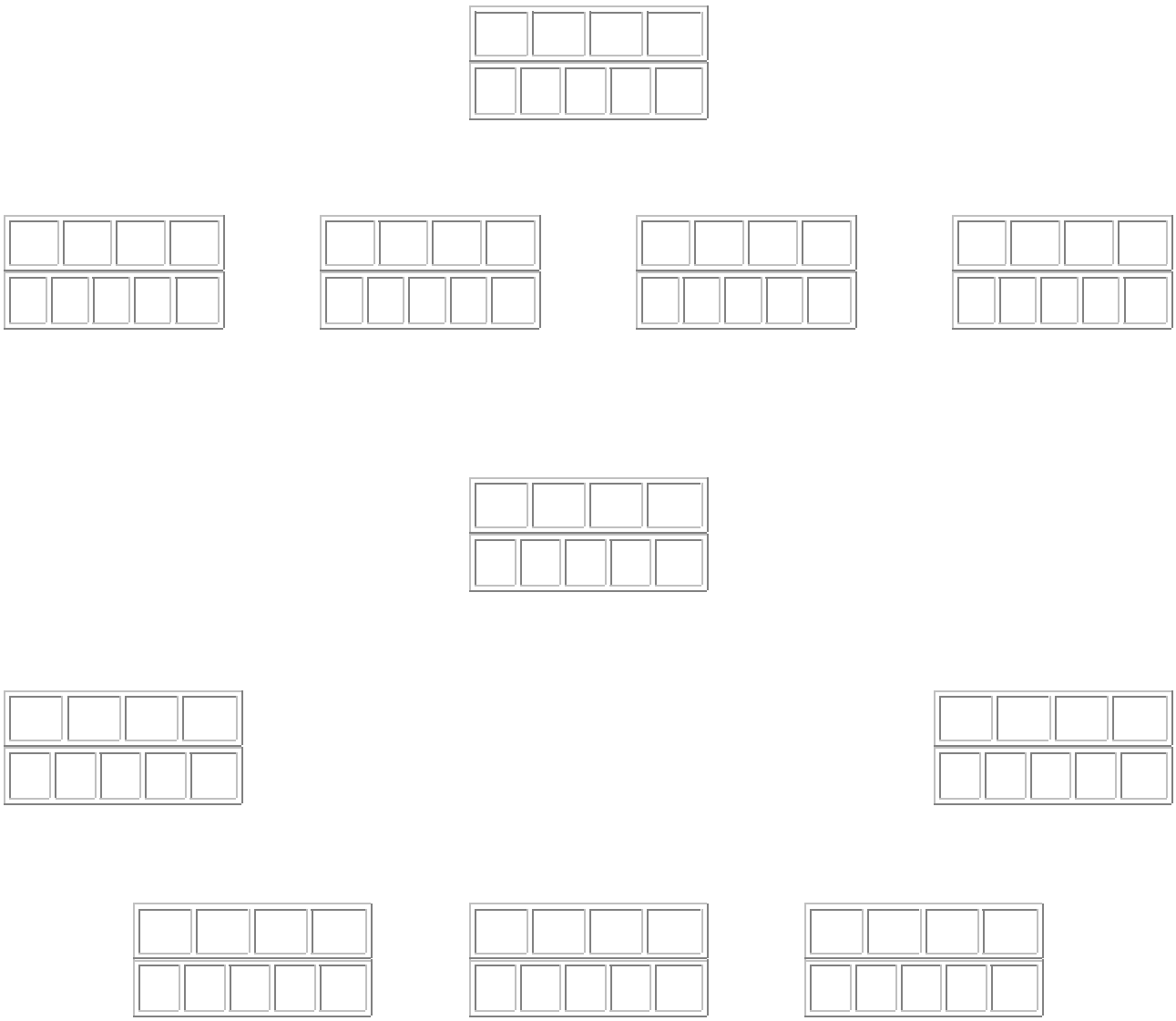






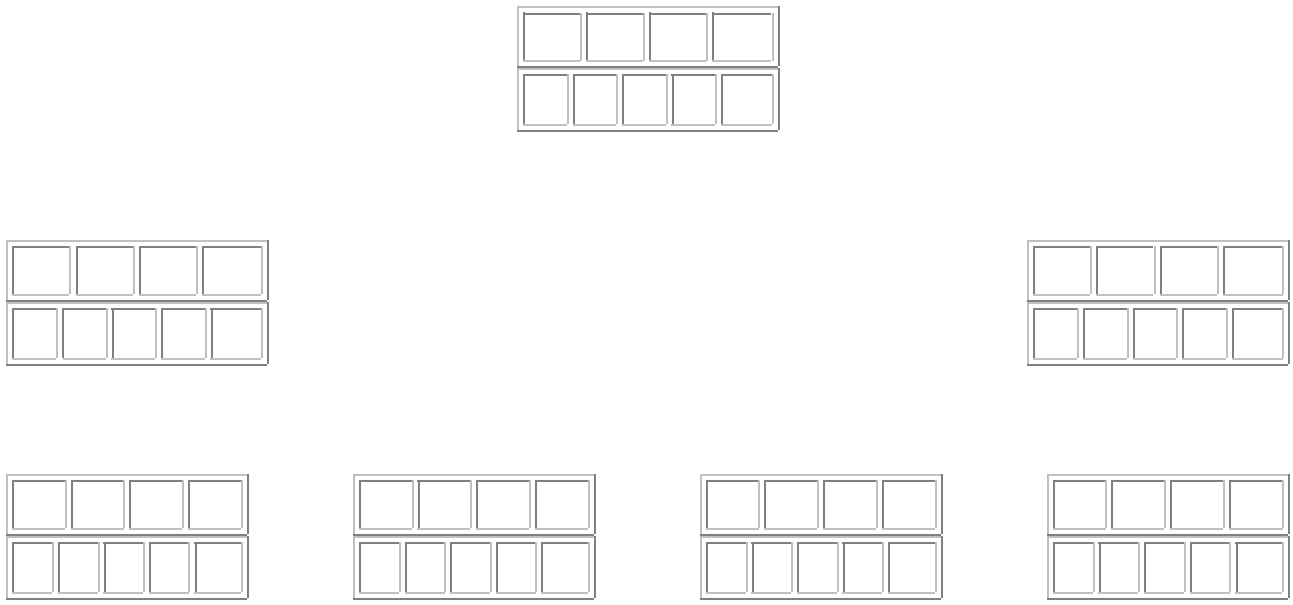
B-Tree Insertion Example (cont.)

Now let’s insert the key with value 12:



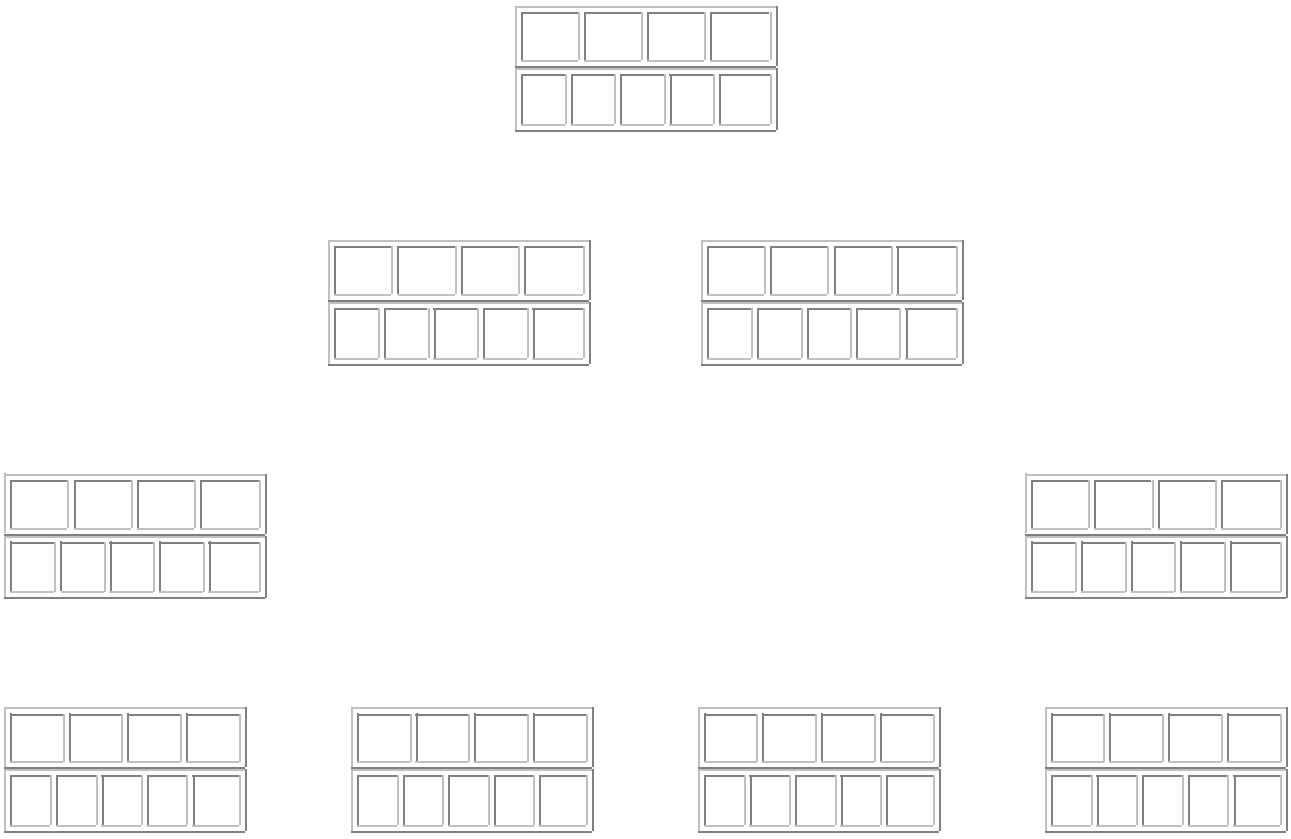
B-Tree Insertion Example (cont.)

Now let’s insert the key with value 4:



Oops!

B-Tree Insertion Example (cont.)



B-Tree Deletion

So how about deleting a key from a B-tree?

Remember, every node (except the root) in a B-tree of order M must contain at most M-1 keys and at least (M-1)/2 keys. The algorithm for deleting a key X from a B-tree may cause a node to *underflow*.

Deleting a key X from a B-tree

1. if X is not in a leaf node, swap it with the largest key in its left subtree; now X is in a leaf node
2. delete X from this node
3. if there are still ≥ (M-1)/2 values in this node, done!

   if there are (M-1)/2 - 1 values, this node has *underflowed* and it must be repaired.

Caught in the Underflow

Here's how we repair an underflowing node in a B-tree of order M.

1. combine this node U with its more populous neigbour V and their parent k_P, maintaining order in the resulting combination W

2. if W has M-1 values, it replaces nodes U and V and k_P; the parent node may have underflowed, so repeat this algorithm with the parent node

   if W has N > M-1 values, split it into Left, Middle and Right:

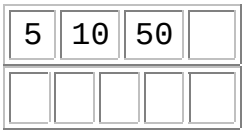
   Left replaces node U and contains the first (N-1)/2 values

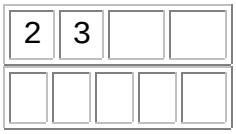
   Middle replaces k_P in the parent node

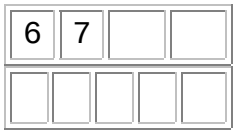
   Right replaces node V and contains the last (N-1)/2 values

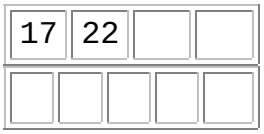
B-Tree Deletion Example

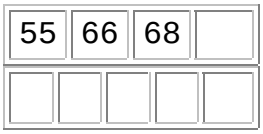
Delete the key with value 5
from this B-tree (of order 5).

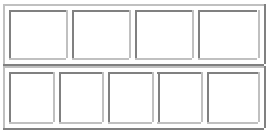


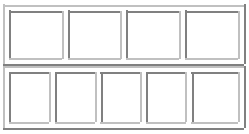


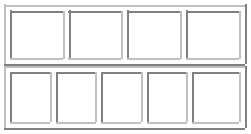


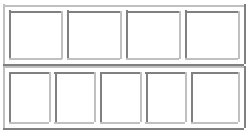


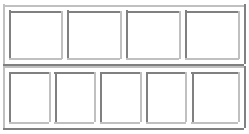


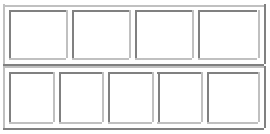


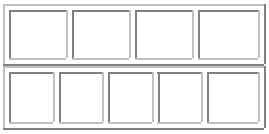


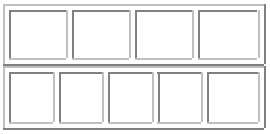


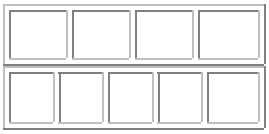






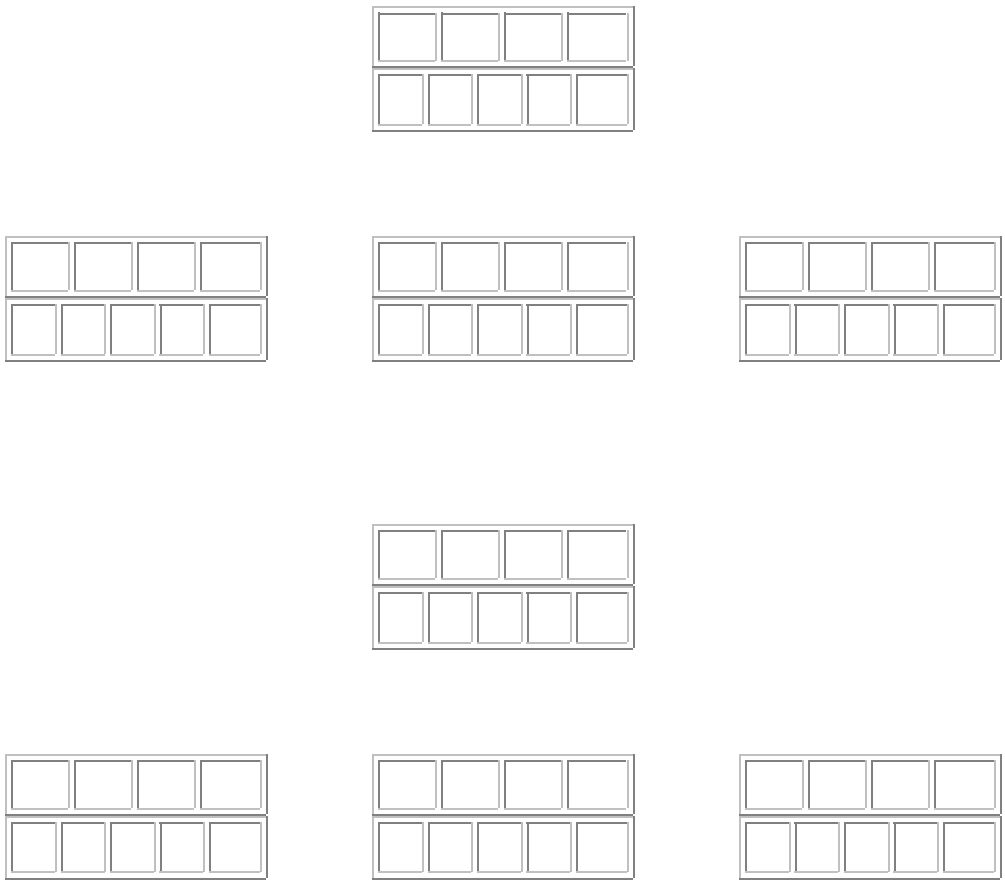






B-Tree Deletion Example (cont.)

How about deleting the key with value 22 now.



So What?

You'll recall, the reason we're doing all of this is because $\frac{1}{2}\log_2 N$ (the number of clusters read in a straight binary search) is too slow for searching for a key value in an index that resides on disk.

Using a *paged binary tree* we reduced the number of cluster reads to $\log_K N$, where K is the number of records that fit in a cluster. The bad news was that inserting and deleting records in a paged binary tree was going to be *way* too expensive.

Q: How can we get the same search performance out of a B-tree that we get from paged binary trees? (Hint: how big should we make our B-tree nodes?)

A:

Q: How expensive is inserting and deleting in a B-tree?

A: