B+ Trees

Bringing it all together

- sequential data files
- indexes
- tree structures
- buffering

Topic	***Folk & Zoellick***
• Postponing Splitting	§ 8.14
• B* Trees	§ 8.15
• Virtual B-Trees	§ 8.16
• Sequential Access Revisited	§§ 9.2, 9.3
• B+ Trees	§§ 9.3, 9.10, 9.11

To Split?

The good performance of B-trees for large index files is largely due to two things:

1. pruning the search space to $1/K^{th}$ of its previous size every time a cluster is read
2. the localized effect of inserting and deleting keys

K is the average number of keys in a node and...

$$(M-1)/2 \leq K \leq (M-1)$$

We can make two vital observations here:

1. the bigger K is, the better performance we get from the B-tree
2. splitting a node into two nodes (from the B-tree insertion algorithm) makes K smaller

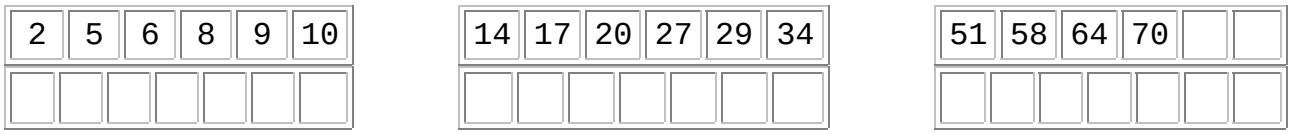
Q to make sure you're listening: Why does splitting a node make K smaller?

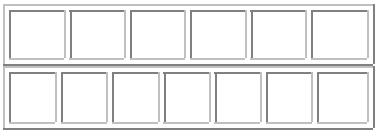
A:

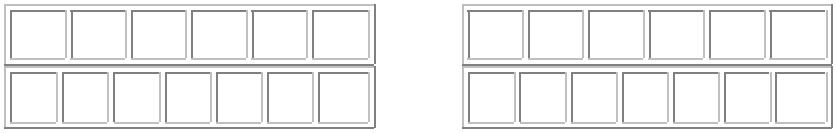
A:

Or Not

Let’s do another example. Insert the key with value 19 into the following B-tree (of order 7):





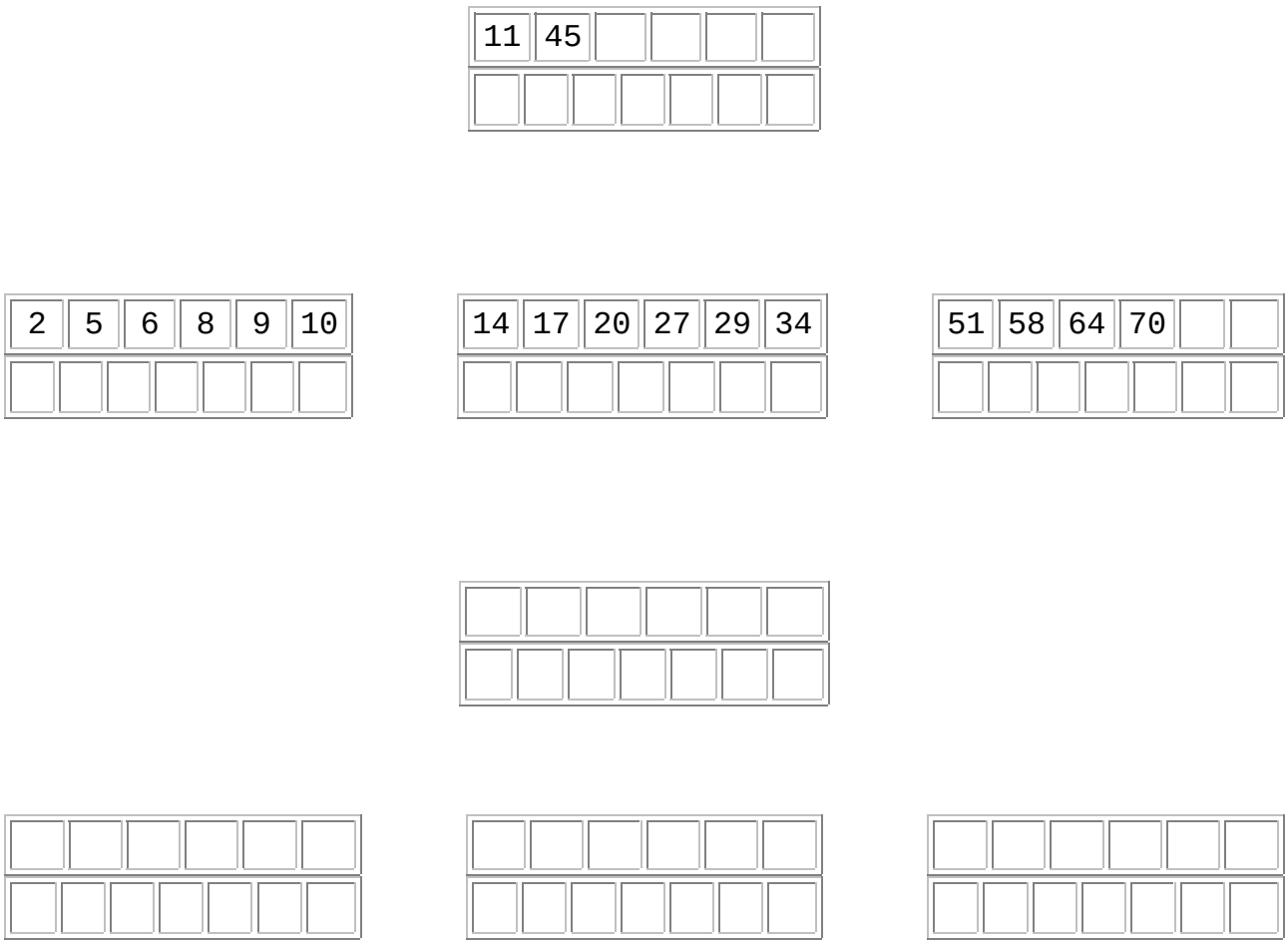


Q: What was the original K? What’s the new K?

A:

Not

Insert the key with value 19 into the following B-tree without splitting:

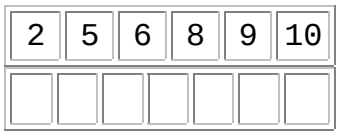


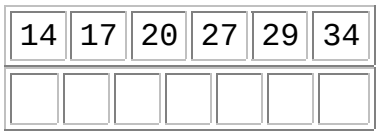
Q: What was the original K? What's the new K?

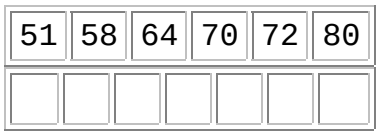
A:

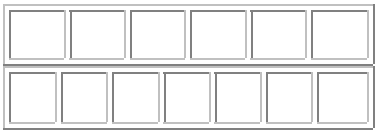
B* Trees Example

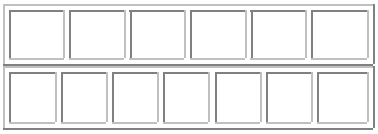
Insert the key with value 19 into the following B-tree:

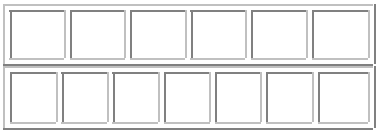












B* Trees

Recall the definition of a B-tree:

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)

□

A B* tree is a B-tree with one tiny difference:

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 *two thirds* full (from (2M-1)/3 to M-1 values)
3. the root may contain any number of values (from 1 to M-1)

□

Observation: if we postpone splitting a node until it and its neighbours are full, we can combine it with a full neighbour (and their parent key) and split the combined keys into three nodes, each one of them guaranteed to be two thirds full (preserving the B* tree property).

Virtual B-Trees (An Aside)

Remember the idea behind *paged binary trees*?

The first *d* comparisons in a binary search are always among the same 2^d-1 records. Why not put those 2^d-1 records in the same cluster on disk so that the first *d* comparisons can all be done at the cost of *one* cluster read.

In fact, RAM can usually hold many clusters at the same time. For example, if we have 32MB of RAM free and our clusters are 8KB, we could fit 4,000 clusters in RAM at the same time.

In a B-tree, each node occupies an entire cluster's worth of keys. But it's still just an M-way search tree, meaning that every search through the B-tree starts at the root and proceeds down to the leaves.

Virtual B-Trees (An Aside Continued)

Q: Wouldn't it make sense to keep the root node of the B-tree in RAM all the time?

A:

Q: In fact, if we could fit 4,000 clusters in 32MB of RAM that's not being used for anything else, wouldn't it make sense to keep the top $\log_2(4{,}000)$ levels of the B-tree in RAM all the time?

A:

Q: Ok then, smartie, which nodes *should* we keep in RAM?

A:

□

Q: What's a *virtual B-tree*?

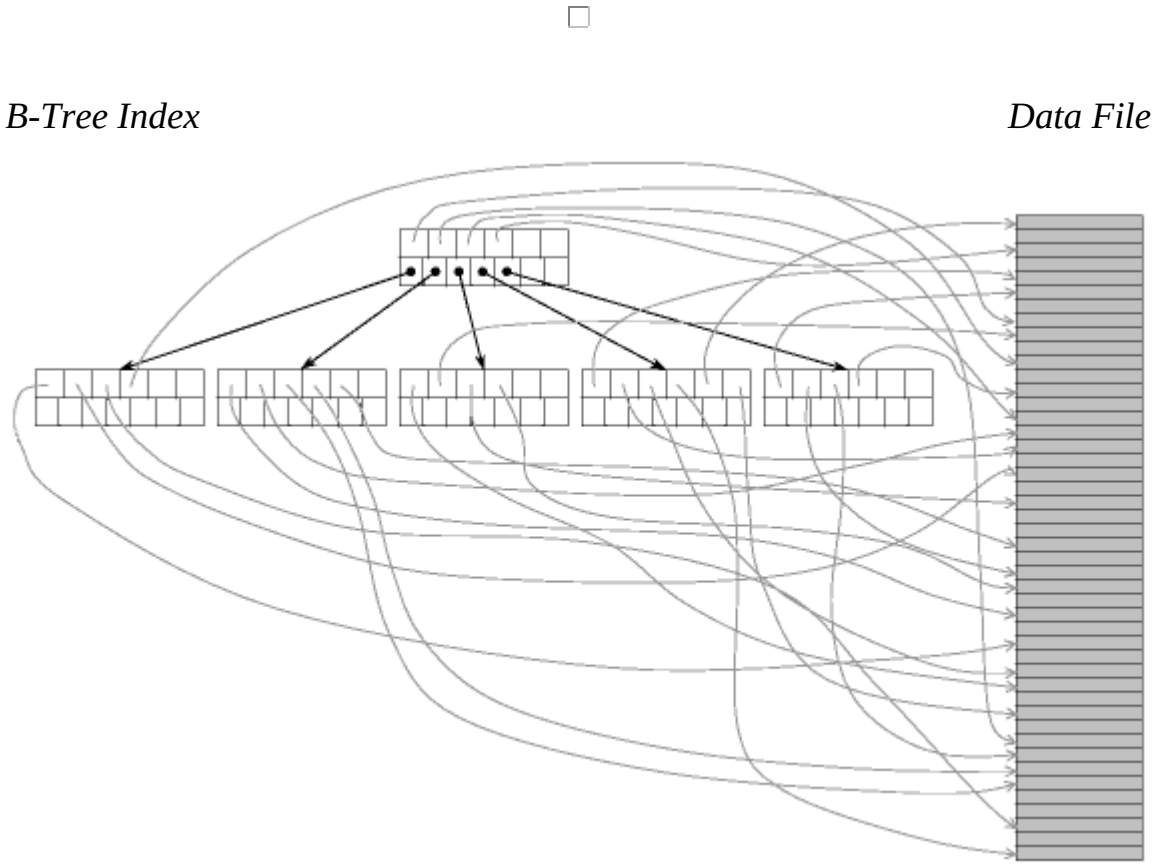
A:

Sequential Access Revisited

Imagine a big unsorted data file full of records. Suppose that the data file has a big index file which is stored as a B-tree. Now let's say we want to print out a report containing all the records in the data file in sorted order.

Q: How expensive is it to access every record in the data file in order using the B-tree index?

A:



Sequential Access Revisited Continued

Does anybody remember the motivation for using B-trees in the first place? I sure do. Part of the motivation is that the process of inserting records and deleting records in a plain vanilla binary search tree is too expensive.

But think about the B* tree for just a second. Each node resides in a different cluster on disk. Yet we add and delete keys all the time without affecting too many clusters, and without disturbing the order in each cluster. This was made possible through careful redistribution of keys in each cluster.

*What if we did the same thing with each clusterful of records in the **data file***

Redistribution in the Data File

We postpone splitting the nodes in a B* tree by redistributing keys from an overflowing node to a neighbour node through the parent node. This is possible because when a new node is created it is only two thirds full.

Imagine a sorted data file on disk. Each cluster occupied by the file contains some number *M* of records. We could start off with each cluster only two thirds full of records. If a new record is added or deleted in the middle of the file, there will only be one cluster affected.

There are several problems with this idea, but none of them are too serious.

P: If the current cluster is already full and we want to add a record, what happens to the clusters following the current cluster?

P: If the current cluster has only one record and we want to delete it, what happens to the clusters following the current cluster?

P: Isn't this a big waste of space?

P: This scheme might be ok for sequential access, but what about searching?

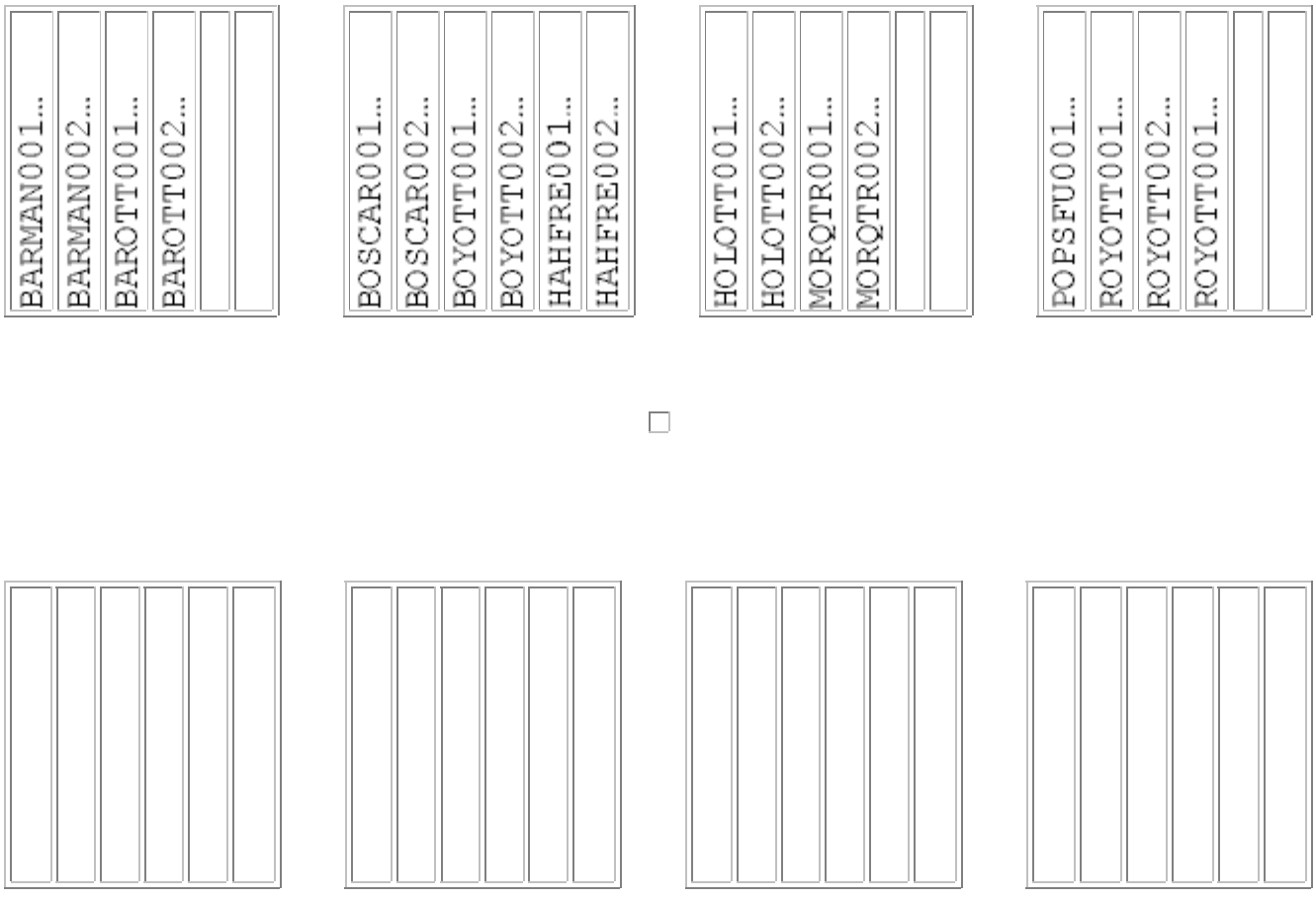
Linking the Clusters

If we try to add a record to a cluster that's already full, we can redistribute records to a neighbouring cluster that isn't full. If the neighbours are also full, we can do a two-to-three split, like we do with B* trees.

Of course, adding a cluster into the middle of the file would be way too expensive. So we'll allow clusters to be out of order, and keep track of the logical ordering with pointers.

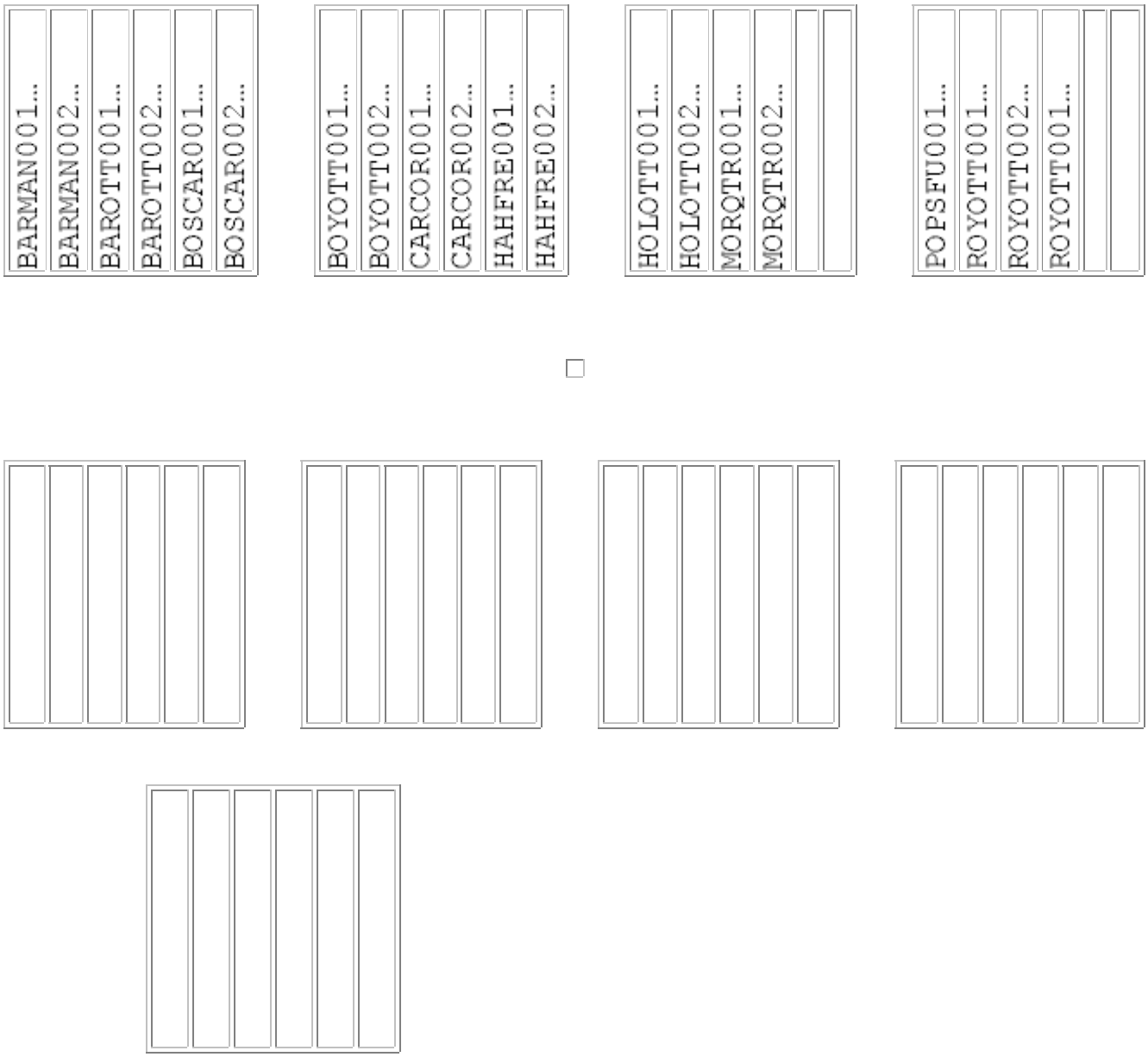
Redistributing Records

We've added two new records to this data file so the second cluster is full. Now we want to add the record with key value `CARCOR001`. We must redistribute some of the records to a neighbouring cluster to make room.



Splitting Clusters

We've added yet another record to the file so now the first two clusters are full. Now we want to add the record with key value `BAROTT003`.



Searching the Sequence Set

The set of clusters containing our data file is called a *sequence set*. If we want to access every record in the file in order, all we have to do is start with the first cluster, read every record, then skip to the next cluster, etc. But what if we want to access one particular record directly?

Why not keep an index!

Primary Key	***Cluster Number***
BOSCAR001	5
CARCOR001	2
HOLOTT001	3
POPSFU001	4

To find the record we're looking for, we compare its primary key to the keys in the index. If our key is ≥ a key in the index (and < the next key in the index), the record is in the cluster named in the index. If our key is < a key in the index, the record is in the previous cluster (back links will help!).

Q: What if this index is too big to fit into RAM?

A: :-)

B+ Trees

If we use a B-tree as an index for a sequence set, we call the whole magilla a *B+ Tree*. Here's a B+ tree of order 3 for our previous example.