B-TREES I

Contents of today's lecture:

- Introduction to multilevel indexing and B-trees.
- Insertions in B trees.

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Sections 9.1-9.6.

Introduction to Multilevel Indexing and B-Trees

Problems with **simple indexes** that are kept in disk:

1. Seeking the index is still slow (binary searching):

   We don't want more than 3 or 4 seeks for a search.
   So, here $\log_2(N+1)$ is still slow:

N	$\log_2(N+1)$
15 keys	4
1,000	$\sim$10
100,000	$\sim$17
1,000,000	$\sim$20

2. Insertions and deletions should be as fast as searches:
   In simple indexes, insertion or deletion take O(n) disk accesses (since index should be kept sorted)

Indexing with Binary Search Trees

We could use **balanced** binary search trees:

- **AVL Trees**

  Worst-case search is $1.44 \log_2(N+2)$

  1,000,000 keys $\rightarrow$ 29 levels

  Still prohibitive...

- **Paged Binary Trees**

  Place subtrees of size K in a single page

  Worst-case search is $\log_{K+1}(N+1)$

K=511, N=134,217,727

Binary trees: 27 seeks
Paged binary tree: 3 seeks

This is good but there are lots of difficulties in **maintaining** (doing insertions and deletions in) a paged binary tree.

Multilevel Indexing

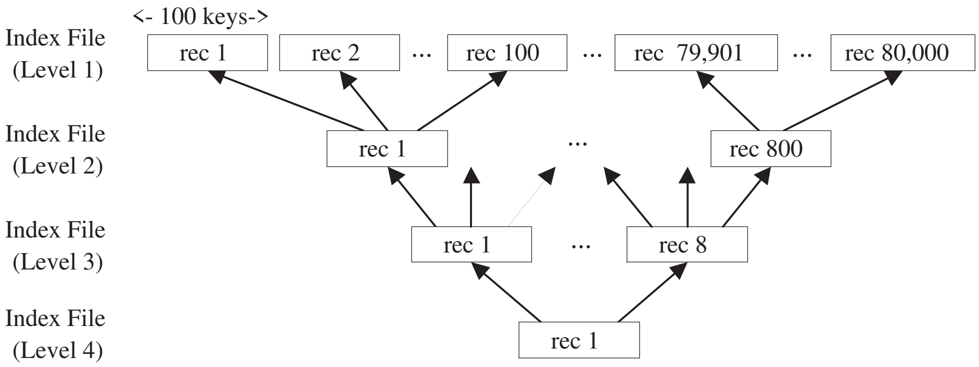
Consider our 8,000,000 example with keysize = 10 bytes.
Index file size = 80 MB
Each **record** in the **index** will contain 100 pairs (key, reference)
A **simple index** would contain: 80,000 records. Too expensive to search (still $\sim$ 16 seeks)

Multilevel Index

Build an index of an index file:

How:

- Build a simple index for the file, sorting keys using the method for external sorting previously studied.
- Build an index for this index.
- Build another index for the previous index, and so on.



Note: That the index of an index stores the largest key in the record it is pointing to.

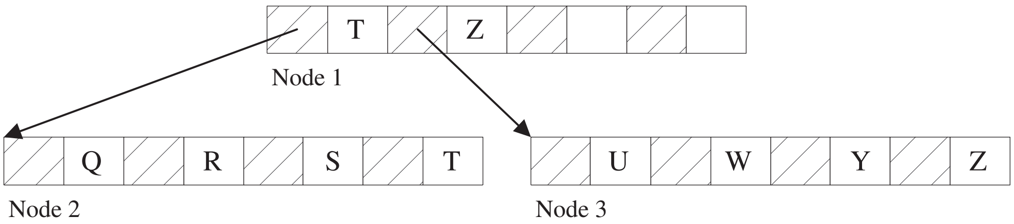
B-Trees - Working Bottom-Up

- Again an index record may contain 100 keys.
- An index record may be half full (each index record may have from 50 to 100 keys).
- When insertion in an index record causes it to overfull:

  - Split record in two

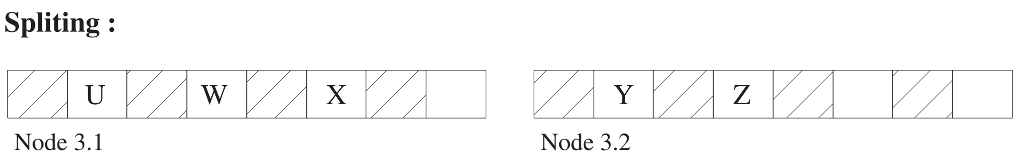
  - "Promote" the largest key in one of the records to the upper level

Example for order = 4 (instead of 100).

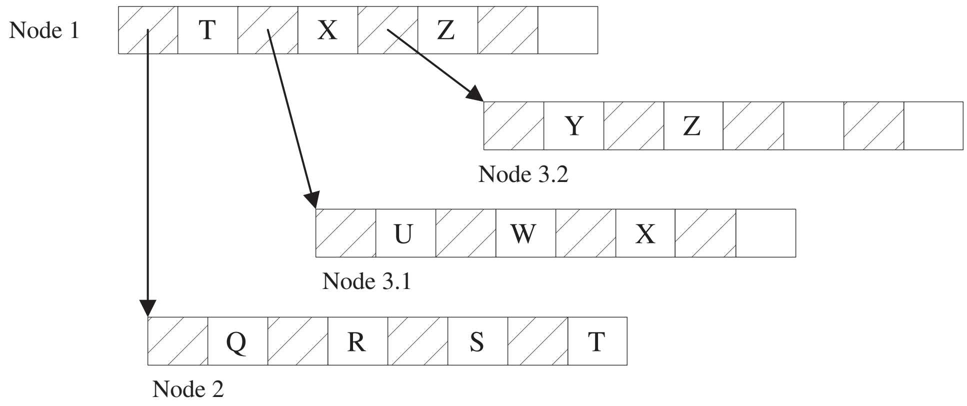


Inserting X

X is between T and Z: insertion in node 3 splits it and generates a promotion of node X.



Promoting largest of Node 3.1.

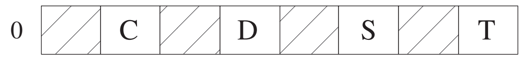


Important: If Node 1 was full, this would generate a new split-promotion of Node 1. This could be propagated up to the root.

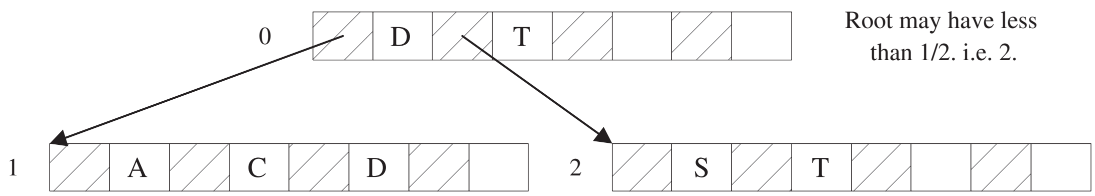
An example showing insertions:

Inserting keys: order = 4

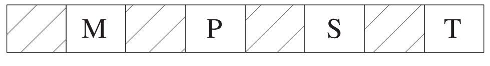
C, S, D, T, A, M, P, I, B, W, N, G, U, R, K, E, H, O, L, J, Y, Q, Z, F, X, V



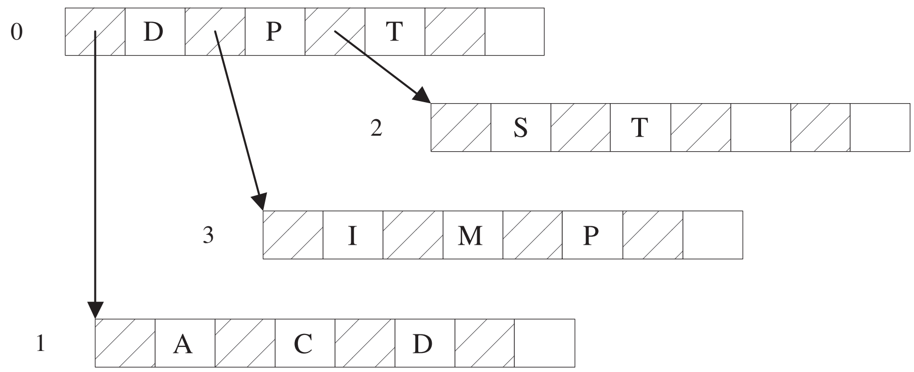
Inserting A: Split and promotion



Inserting M,P only changes Node 2 to :



But inserting “I” Splits and promotes “P”.



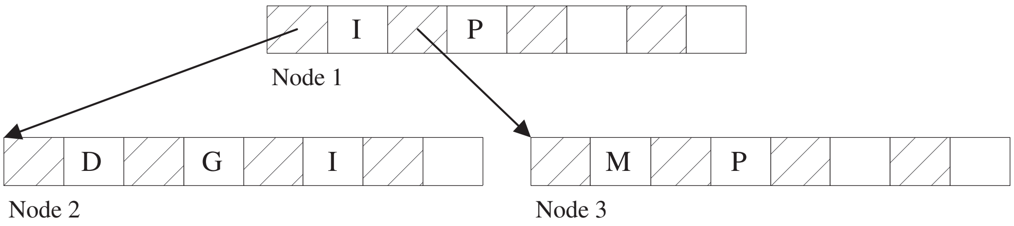
There is room for one more node at level 2. After that the root may get split!

Complete the above example as an exercise.

Important: At each level, no more than 2 nodes are affected. We got search and updates with cost equal to the height of the tree !!!

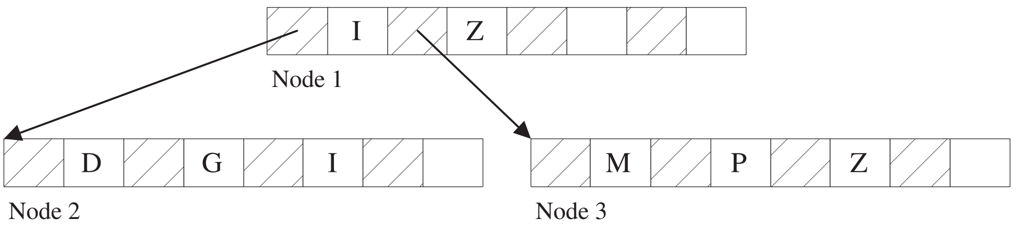
Note regarding insertions in B-trees

Special case of larger key:



Inserting Z

Z is larger than P but P is larger in Node 1, so the place for Z is Node 3.



B-Tree Properties

Properties of a B-tree of order m:

1. Every node has a maximum of m children.
2. Every node, except for the root and the leaves, has at least $\lceil m/2 \rceil$ children.
3. The root has at least two children (unless it is a leaf).
4. All the leafs appear on the same level.
5. The leaf level forms a complete index of the associated data file.

Worst-case search depth

The worst-case depth occurs when every node has the minimum number of children.

Level	Minimum number of keys (children)
1 (root)	2
2	$2 \cdot \lceil m/2 \rceil$
3	$2 \cdot \lceil m/2 \rceil \cdot \lceil m/2 \rceil = 2 \cdot \lceil m/2 \rceil^2$
4	$2 \cdot \lceil m/2 \rceil^3$
...	...
d	$2 \cdot \lceil m/2 \rceil^{d-1}$

If we have N keys in the leaves:

$$N \geq 2 \cdot \lceil m/2 \rceil^{d-1}$$

So, $d \leq 1 + \log_{m/2}(N/2)$

For $N = 1,000,000$ and order $m = 512$, we have

$$d \leq 1 + \log_{256} 500,000$$
$$d \leq 3.37$$

There is at most 3 levels in a B-tree of order 512 holding $1,000,000$ keys.

B-TREES II

Contents of today's lecture:

- Outline of `Search` and `Insert` algorihtms
- Deletions in B trees.

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Sections 9.8, 9.9, 9.10, 9.11, 9.12

Outline of Search and Insert algorithms

`Search (keytype key)`

1) Find leaf: find the leaf that could contain `key`, loading all the nodes in the path from root to leaf into an array in main memory.
2) Search for `key` in the leaf which was loaded in main memory.

`Insert (keytype key, int datarec_address)`

1) Find leaf (as above).

2) Handle special case of new largest key in tree: update largest key in all nodes that have been loaded into main memory and save them to disk.

3) Insertion, overflow detection and splitting on the update path:
`currentnode` = leaf found in step 1)
`recaddress` = `datarec_address`

   3.1) Insert the pair (`keys, recaddress`) into `currentnode`.

   3.2) If it caused overflow

   - Create `newnode`
   - Split contents between `newnode, currentnode`
   - Store `newnode, currentnode` in disk
   - If no parent node (root), go to step 4)
   - `currentnode` becomes parent node
   - `recaddress` = address in disk of `newnode`
   - `key` = largest key in new node
   - Go back to 3.1)

4) Creation of a new root if the current root was split.

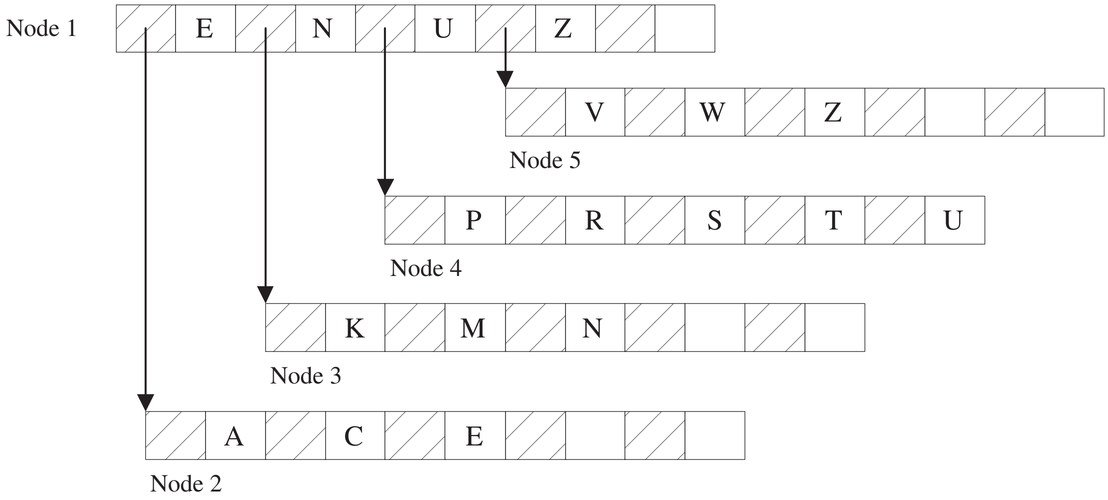
   Create root node pointing to `newnode` and `currentnode`. Save new root to disk.

Deletions in B-Trees

The rules for deleting a key K from a node n in a B-tree:

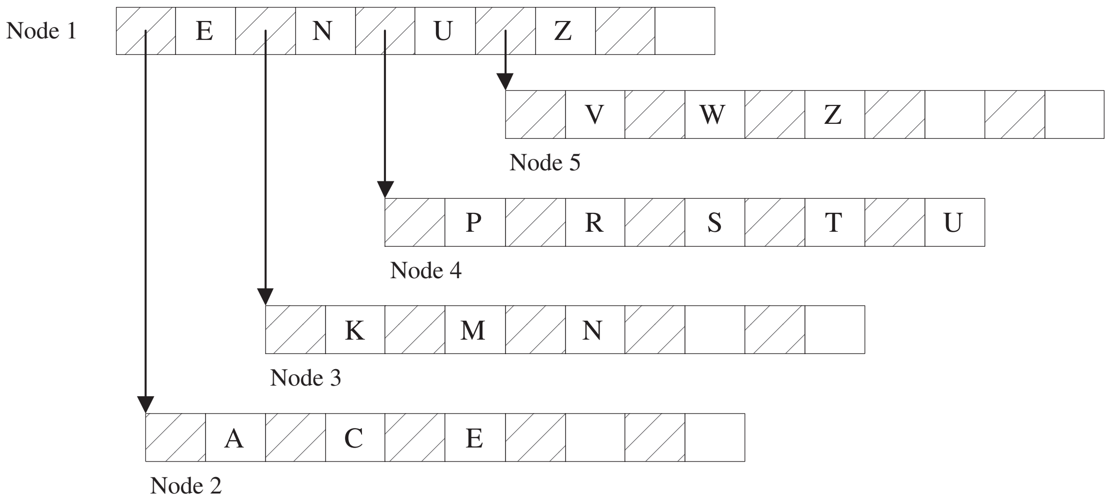
1. If n has more than the minimum number of keys and K is not the largest key in n, simply delete K from n.
2. If n has more than the minimum number of keys and K is the largest key in n, delete K from n and modify the higher level indexes to reflect the new largest key in n.
3. If n has exactly the minimum number of keys and one of the siblings has "few enough keys", **merge** n with its sibling and delete a key from the parent node.
4. If n has exactly the minimum number of keys and one of the siblings has extra keys, **redistribute** by moving some keys from a sibling to n, and modify higher levels to reflect the new largest keys in the affected nodes.

Consider the following example of a B-tree of **order 5** (minimum allowed in node is 3 keys)

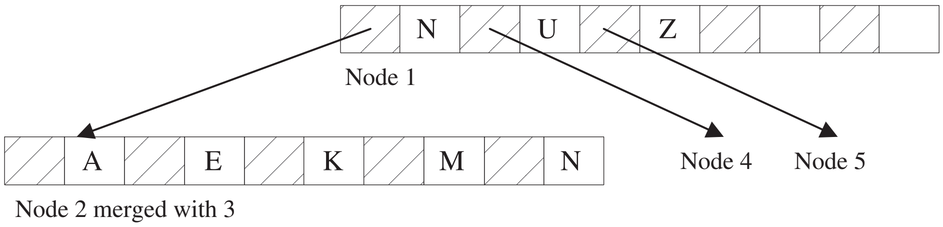


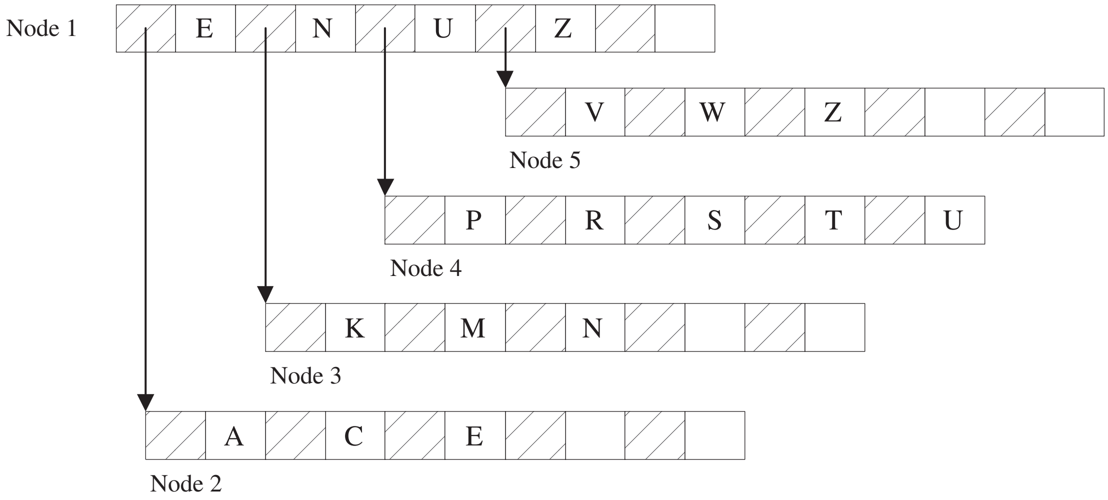
We consider the following 5 alternative modifications on the previous tree:

- Deleting “T” falls into case 1)
- Deleting “U” falls into case 2)

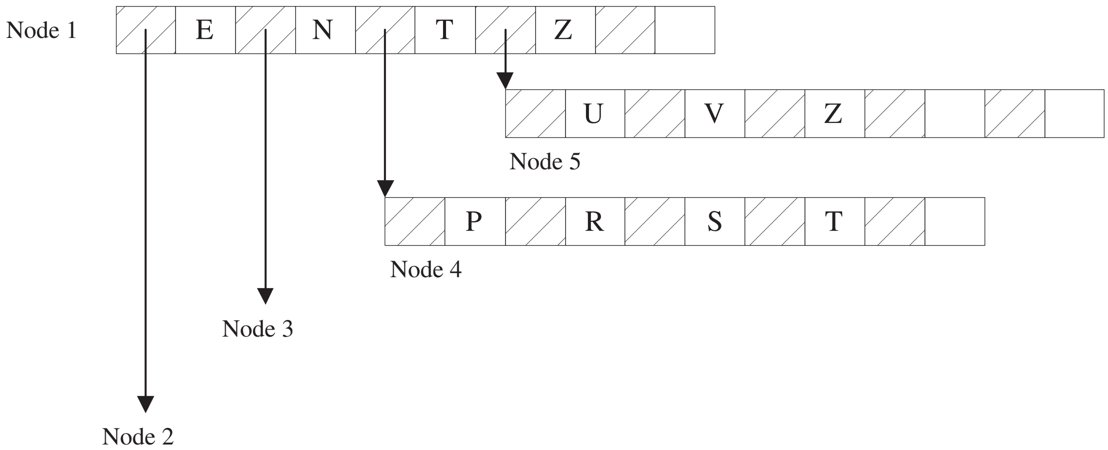


- Deleting “C” falls into case 3) merging with sibling:





- Deleting “W” falls into case 4) redistribution with sibling:



- Deleting “M” allows for two possibilities: case 3) or 4)
  - Merge Node 3 with Node 2; or
  - Redistribute keys between Node 3 and Node 4

Note that “sibling” here refers only to nodes that have the same parent and are **next to each other**.

B+ TREES I

Contents of today's lecture:

- Maintaining a sequence set.
- A simple prefix B+ tree.

Reference : Folk, Zoellick and Riccardi, File Structures, 1998. Sections 10.1 - 10.5

Motivation

Some applications require two views of a file :

Indexed view :	**Sequential view :**
Records are indexed by a key	Records can be sequentially accessed in order by key
Direct, indexed access	Sequential access (physically contiguous records)
Interactive, random access	Batch processing (Ex: co-sequential processing)

Example of applications

- Student record system in a university :
  - Indexed view : access to individual records
  - Sequential view : batch processing when posting grades or when fees are paid
- Credit card system :
  - Indexed view : interactive check of accounts
  - Sequential view : batch processing of payment slips

We will look at the following two aspects of the problem :

1. Maintaining a **sequence set** : keeping records in sequential order
2. Adding an **index set** to the sequence set

Maintaining a Sequence Set

Sorting and re-organizing after insertions and deletions is out of question.
We organize the sequence set in the following way:

- Records are grouped in **blocks**
- Blocks should be **at least half full**.
- **Link fields** are used to point to the preceding block and the following block (similarly to doubly linked lists)
- Changes (insertion/deletion) are localized into blocks by performing :
  - **Block Splitting** when **insertion** causes overflow
  - **Block Merging** or **Redistribution** when **deletion** causes underflow

Example:

Block size = 4

key : **Last Name**

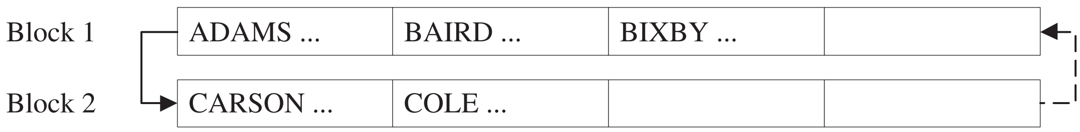
→ Forward Pointer

- - → Backward Pointer

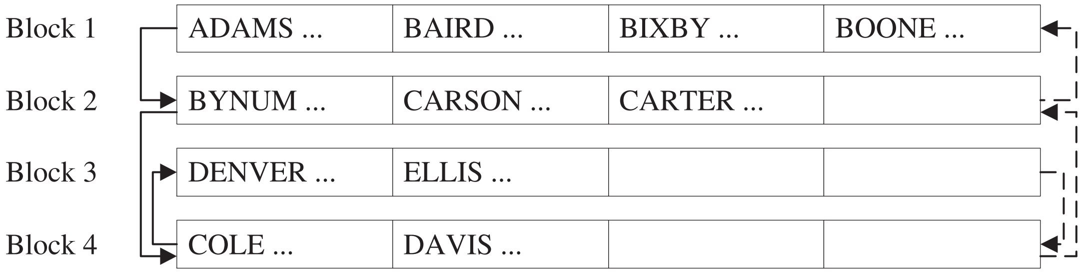
• Insertion with overflow:

Block 1	ADAMS ...	BIXBY ...	CARSON ...	COLE ...

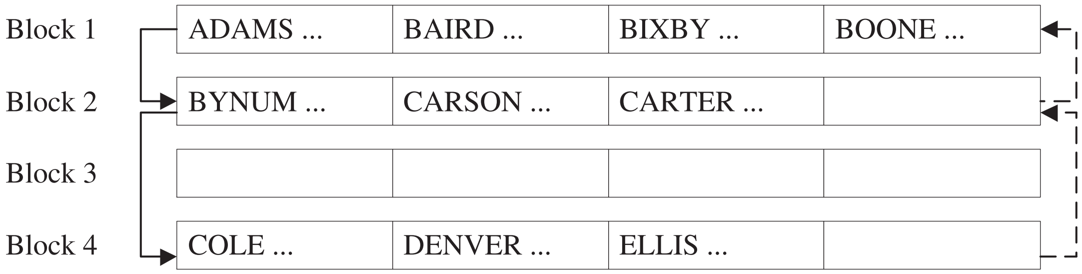
Insert "BAIRD ..."



- **Deletion with merging:**



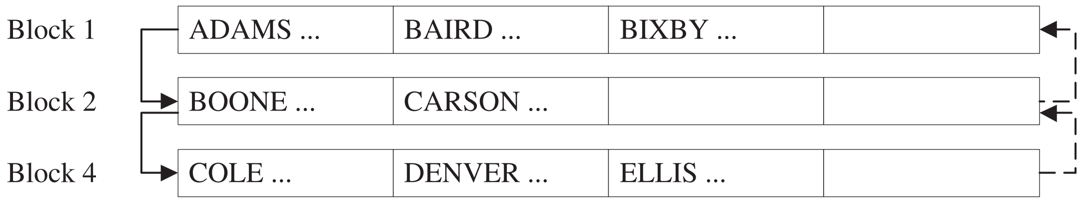
Delete "DAVIS ..." (Merging)



Block 3 is available for re-use

Delete ‘BYNUM’:
Just remove it from Block 2
Then, delete ‘CARTER’ :
We can either merge Block 2 and 4 or redistribute records among Blocks 1 and 2.

- **Deletion with redistribution:**



When previous and next blocks are full then redistribution is the only option.

Advantages and disadvantages of the scheme described

Advantages:

- No need to re-organize the whole file after insertions/deletions.

Disadvantages:

- File takes more space than unblocked files (since blocks may be half full).
- The order of the records is not necessarily **physically** sequential (we only guarantee physical sequentiality within a block).

Choosing Block Size

Consider :

- Main memory constraints (must hold at least 2 blocks)
- Avoid seeking within a block (Ex: in sector formatted disks choose block size equal to cluster size).

Adding an Index Set to the Sequential Set

Index will Contain Separators Instead of Keys

Choose the **Shortest Separator** (a prefix)

```
Block     Range of Keys  Separator
-----     -------------  ---------
  1       ADAMS - BERNE
                         BO
  2       BOLEN - CAGE
                         CAM
  3       CAMP  - DUTTON
                         E
  4       EMBRY - EVANS
                         F
  5       FABER - FOLK
                         FOLKS
  6       FOLKS - GADDIS
```

How can we find a separator for `key1`= "CAGE" and `key2` ="CAMP"?

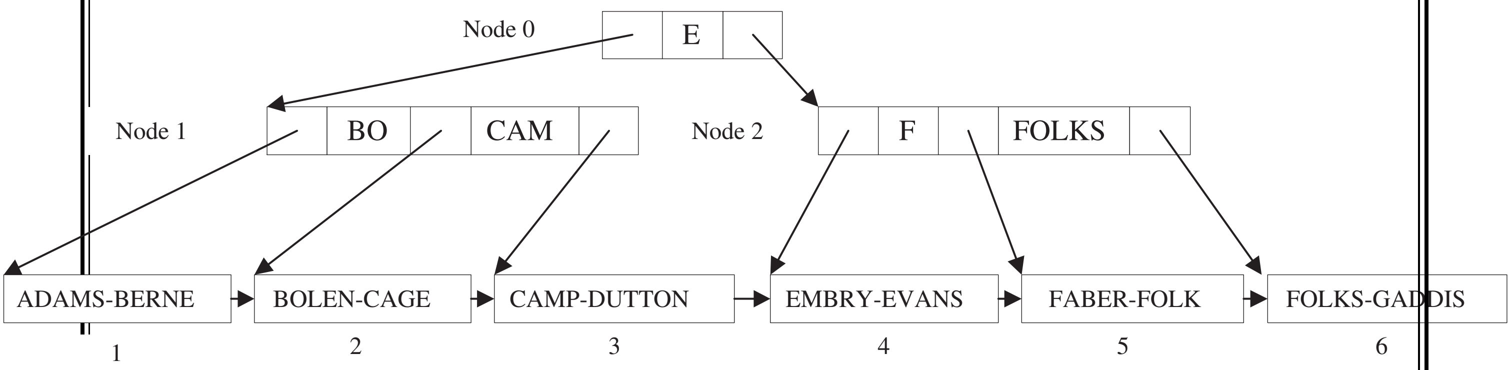
Find the smallest prefix of `key2` that is not a prefix of `key1`.
In this example, the separator is "CAM".

The Simple Prefix B+ Tree

The **simple prefix B+ tree** consists of :

- **sequence set** (as previously seen).
- **index set**: similar to a B-tree index, but storing the shortest separators (prefixes) for the sequence set.

Note : If a node contains N separators, it will contain N+1 children. Using separators slightly modifies the operations in the B-tree index.
Example :
Order of the index set is 3 (i.e. maximum of 2 separators and 3 children). Note: The order is usually much larger, but we made it small for this example.



Search in a simple prefix B+ tree: Search for "EMBRY":

- Retrieve Node 0 (root).
- "EMBRY" > "E", so go right, and retrieve Node 2.
- Since "EMBRY" < "F" go left, and retrieve block number 4.
- Look for the record with key "EMBRY" in block number 4.

B+ TREES II

Contents of today's lecture:

- Simple Prefix B+ Tree Maintenance: Insertions and Deletions

Reference : Folk, Zoellick and Riccardi, File Structures, 1998. Sections 10.6 - 10.7 (overview 10.8 - 10.11).

From B trees to B+ trees

A clarification regarding the book's treatment of B-trees and B+ trees: the transition from understanding B trees to understanding the index set of the simple prefix B+ tree may be facilitated by the following interpretation.

- Look at the B + tree index set as a B-tree in which the smallest element in a child is stored at the parent node in order:

key1, pointer1, key2, pointer2, ..., keyN, pointerN.

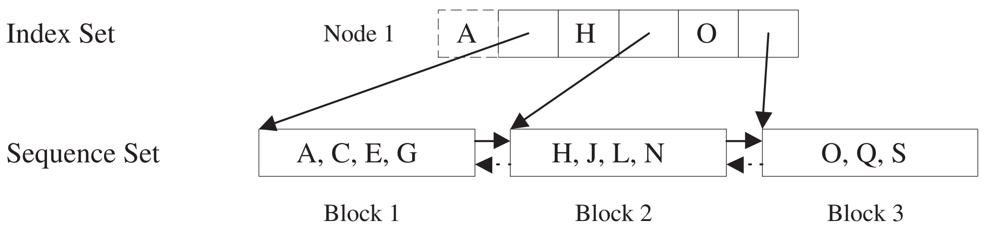
- Then, remember that each key is a separator. And remove key1 from the node's representation. It may help if you think that "key1" is still there, but is "invisible" for understanding purposes.

With this in mind, it should become clear that the updates on the B + tree index set are the same as in regular B-trees.

Simple Prefix B+ Tree Maintenance

Example:

- Sequence set has blocking factor 4
- Index set is a B tree of order 3

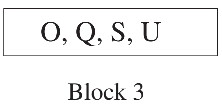


Note that "A" is not really there.

1. Changes which are local to single blocks in the sequence set

   Insert "U":

   - Go to the root
   - Go to the right of "O"
   - Insert "U" to block 3:
     The only modification is



   - There is no change in the index set

Delete “O”:

- Go to the root
- Go to the right of “O”
- Delete “O” from block 3:
  The only modification is

Q, S, U

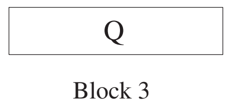
Block 3

There is no change in the index set: “O” is still a perfect separator for blocks 2 and 3.

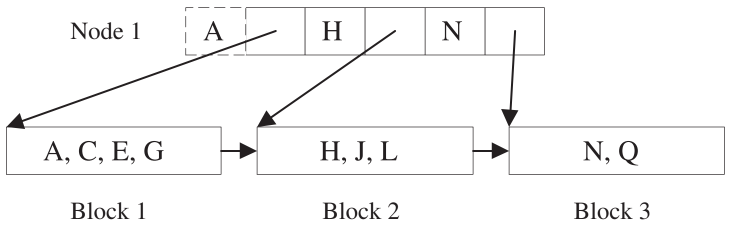
2. Changes involving multiple blocks in the sequence set.

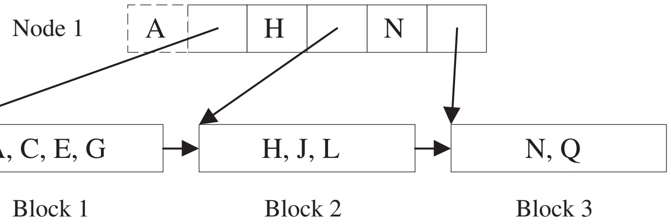
   Delete "S" and "U":

   Now block 3 becomes less than 1/2 full (underflow)



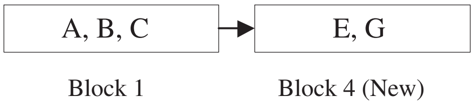
   Since block 2 is full, the only option is **re-distribution** bringing a key from block 2 to block 3:
   We must update the separator "O" to "N".
   The new tree becomes:



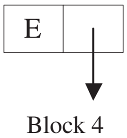


Insert "B":

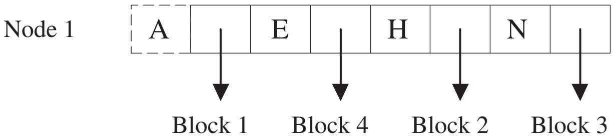
- Go to the root
- Go to the left of "H" to block 1
- Block 1 would have to hold A,B,C,E,G
- Block 1 is split:



Promote new separator "E" together with pointer to new block 4

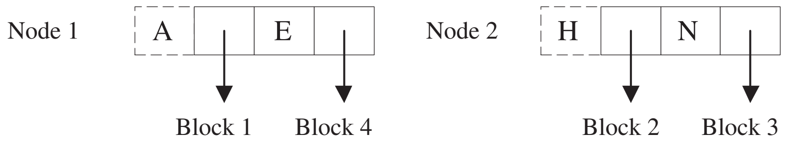


We wished to have:

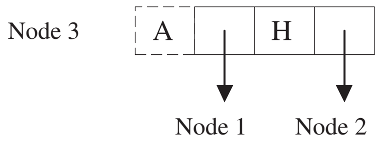


But the order of the index set is 3 (3 pointers, 2 keys).

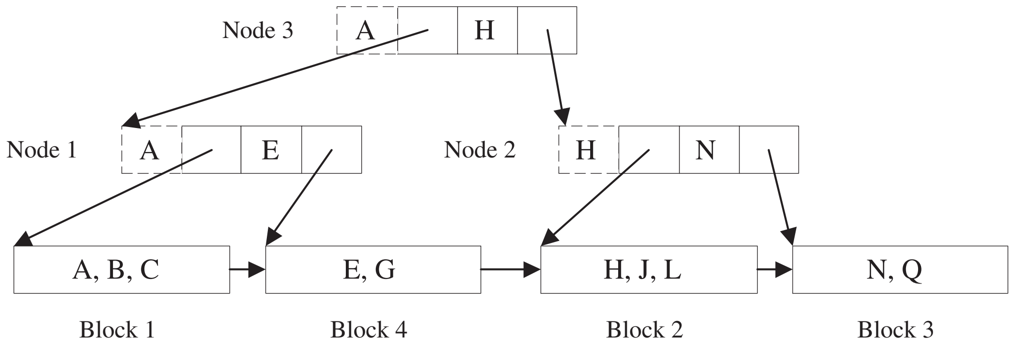
So this causes node to split:



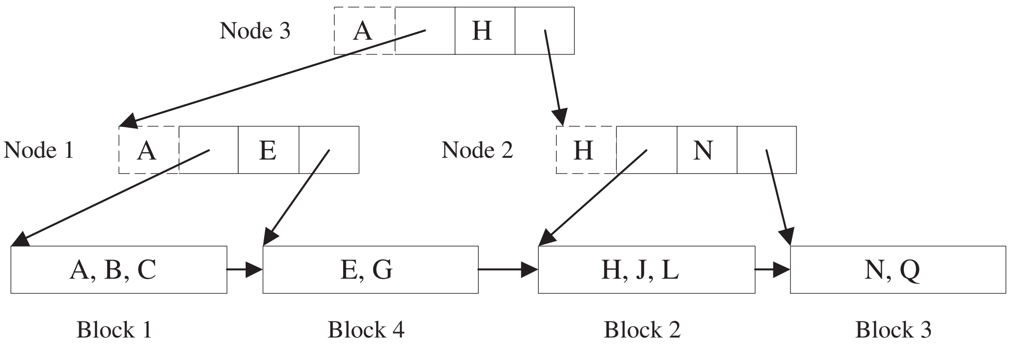
Create a new root to point to both nodes:



The new tree is:

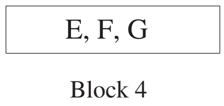


Remember that “A” is not really present in nodes 1 an 3 and that “H” is not really present in node 2.



Insert “F”

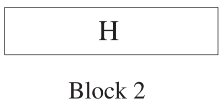
- Go to root
- Go to left of “H”
- Go to right of “E” in Node 1
- Insert “F” in block 4
- Block 4 becomes:



- Index set remains unchanged

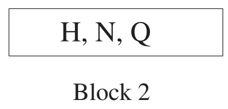
Delete “J” and “L”

Block 2 would become:

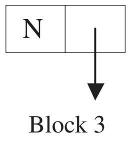


But this is an underflow.
One may get tempted to redistribute among blocks 4 and 2: E, F, G and H would become E, F and G, H.
Why this is not possible ?
Block 4 and block 2 are not siblings! They are cousins.

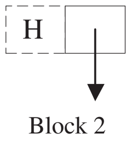
The only sibling of block 2 is block 3.
Redistribution in not possible between H and N,Q, so the only possibility is **merging** blocks 2 and 3 :



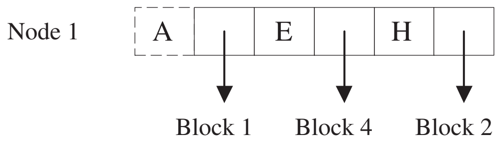
- Send block 3 to `AVAIL LIST`
- Remove the following from node 2



- This causes an underflow in Node 2

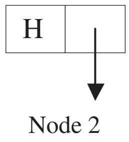


- The only possibility is a merge with its sibling (Node 1):

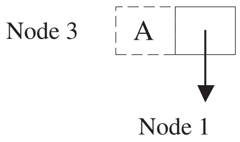


- In our interpretation “H” becomes “visible”; In the textbook’s interpretation “H” is brought down from the root.
- Send node 2 to `AVAIL LIST`

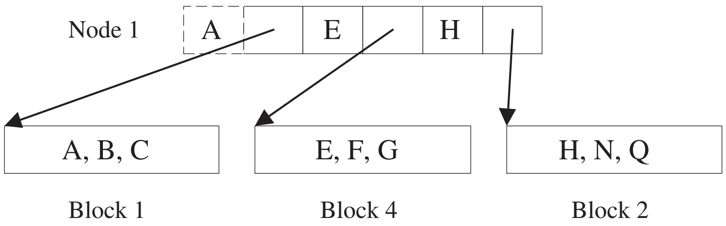
- Now remove



from the root (Node 3). This causes underflow on the root:



- Underflow on the root causes removal of the root (Node 3) and Node 1 becomes the new root :



Blocks were reunited as a big happy family again !!

Compare it with the original tree.

Note : Remember that a B+ tree may be taller, so that splittings or mergings of nodes may propagate for several levels up to the root.

Advanced Observations for Meditation

1. Usually a node for the index set has the same physical size (in bytes) than a block in the sequence set. In this case, we say "index set block" for a node.
2. Usually sequence set blocks and index set blocks are mingled inside the same file.
3. The fact that we use separators of variables length suggests the use of B trees of variable order. The concepts of underflow and overflow become more complex in this case.
4. Index set blocks may have a complex internal structure in order to store variable length separators and allow for binary search on them (see Figure 10.12 on page 442).
5. Building a B+ tree from an existing file already containing many records can be done more efficiently than doing a sequence of insertions into an initially empty B+ tree. This is discussed in Section 10.9 (Building a Simple Prefix B+ Tree).
6. Simple prefix B+ trees or regular B+ trees are very similar. The difference is that the latter stores actual keys rather than shortest separators or prefixes.