University of Dublin
Trinity College

Index Structures for Files
Multi-Level Indexes

Owen.Conlan@cs.tcd.ie

Multi-Level Indexes

Because a single-level index is an ordered file, we can create a primary index *to the index itself*

- the original index file is called the *first-level index*
- the index to the index is called the *second-level index*

We can repeat the process, creating a 3rd, 4th,.... top level until all entries in the *top level* fit in one disk block

Multi-Level Indexes

Indexing schemes so far have looked at an ordered index file

Binary search performed on this index to locate pointer to disk block or record

Requires approximately $\log_2 n$ accesses for index with n blocks

Base 2 chosen because binary search reduces part of index to search by factor of 2

The idea behind multi-level indexes is to reduce the part of the index to search by bfr , the blocking factor of the index (which is bigger than 2!)

Fan-Out

The value of bfr for the index is called the fan-out

We will refer to it using the symbol fo

Searching a multi-level index requires approximately

$$\log_{fo} n$$

block accesses

This is smaller than binary search for fo > 2

Multi Level Indexes

A multi-level index can be created for any type of first-level index (primary, secondary, clustering) as long as the first-level index consists of *more than one* disk blocks

Such a multi-level index is a form of *search tree* ; however, insertion and deletion of new index entries is a severe problem because every level of the index is an *ordered file*

Hence most multi-level indexes use B-tree or B^+ tree data structures, which leave space in each tree node disk block) to allow for new index entries

Multi Level Indexes

A multi-level index considers the index file as an ordered file with a distinct entry for each K(i)
- First level

We can create a primary index for the first level
- Second level
- Because the second level uses block anchors we only need an entry for each block in the first level

We can repeat this process for the second level
- Third level would be a primary level for the second level

And so on … until all the entries of a level fit in a single block

Example

Imagine we have a single level index with entries across 442 blocks and a blocking factor of 68

Blocks in first level,

$n_1 = 442$

Blocks in second level,

$n_2 = \text{ceil}(n_1/\text{fo}) = \text{ceil}(442/68) = 7$

Blocks in third level,

$n_3 = \text{ceil}(n_2/\text{fo}) = \text{ceil}(7/68) = 1$

t = 3

t+1 access to search multi-level index.

Binary search of a single level index of 442 blocks takes 9 +1 accesses

How many levels?

The top level index (where all the entries fit in one block) is the t^{th} level

Each level reduces the number of entries at the previous level by a factor of fo, the index fan-out

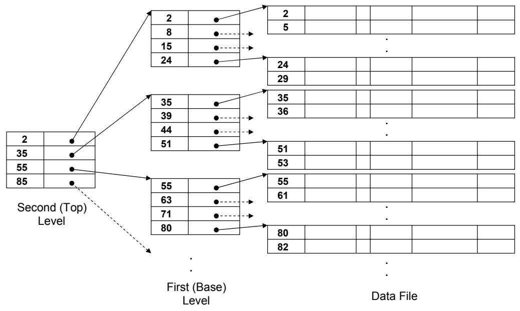
Therefore,

$1 \le (n/(\text{fo}^t))$

We want t,

$t = \text{round}(\log_{\text{fo}} n)$

Two-Level Index



Search Algorithm

For searching a multi-level primary index with t levels

```
p ← address of top level block of index;
for j ← t step -1 to 1 do
  begin
  read the index block (at j^th index level) whose address is p;
  search block p for entry I such that $K_j(i) \le K < K_j(i+1)$ (if
        $K_j(i)$ the last entry in the block it is sufficient to
        satisfy $K_j(i) \le K$);
  p ← $P_j(i)$ (* picks appropriate pointer at j^th index level *);
  end;
read the data file block whose address is p;
search block p for record with key = K;
```

The Invention of the B-Tree

It is hard to think of a major general-purpose file system that is not built around B-tree design

They were invented by two researchers at Boeing, R. Bayer and E. McCreight in 1972

By 1979 B-trees were the "de facto, the standard organization for indexes in a database system“

B-trees address the problem of speeding up indexing schemes that are too large to copy into memory

The Problem

The fundamental problem with keeping an index on secondary storage is that accessing secondary storage is slow. Why?

Binary searching requires too many seeks:

Searching for a key on a disk often involves seeking to different disk tracks. If we are using binary search, on average about 9.5 seeks is required to find a key in an index of 1000 items using a binary search

The Problem

It can be very expensive to keep the index in sorted order

If inserting a key involves moving a large number of other keys in the index, index maintenance is very nearly impractical on secondary storage for indexes with large numbers of keys

We need to find a way to make insertions and deletions to indexes that will not require massive reorganization

Using B-trees and B+ trees as dynamic multi-level indexes

These data structures are variations of search trees that allow efficient insertion and deletion i.e. are good in dynamic situations

Specifically designed for disk

- each node corresponds to a disk block
- each node is kept between half full and completely full

Using B-trees and B+ trees as dynamic multi-level indexes

An insertion into a node that is not full is very efficient; if a node is full then insertion causes the node to split into two nodes

Splitting may propagate upwards to other tree levels

Deletion is also efficient as long as a node does not become less than half full; if it does then it must be merged with neighbouring nodes

Definition of a B-Tree

A B-Tree, of order `m`, is a multi-way search lexicographic search tree where:

- every node has

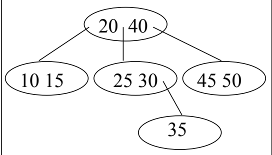
  `CEIL[m/2]- 1 ≤ k ≤ m-1` keys

  appearing in increasing order from left to right; an exception is the root which may only have one key

- a node with `k` keys *either* has `k +1` pointers to children, which correspond to the partition induced on the key-space by those `k` keys, *or* it has all its pointers null, in which case it is terminal

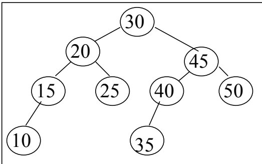
- all terminal nodes are at the same level

Comparison



Total search time
= 2*1 + 6*2 + 1*3
= 17

3-way tree



Total search time
= 1 + 2*2 + 4*3 + 2*4
= 25

2-way (binary) tree

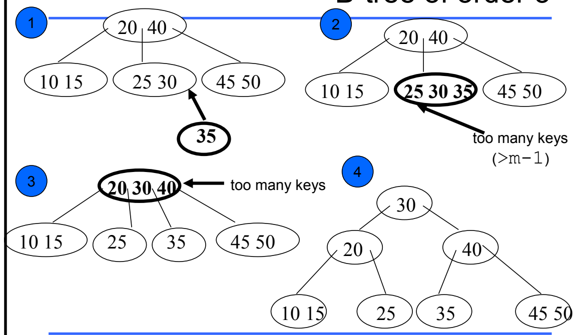
Insertion into a B-Tree

The *terminal* node where the key should be placed is found and the addition (in appropriate place lexicographically) is made

If overflow occurs (i.e. `>m-1` keys), the node splits in two and middle key (along with pointers to its newly created children) is passed up to the next level for insertion and so on

At worst splitting will propagate along a path to the root, resulting in the creation of a new root

Example of insertion of key 35 into B-tree of order 3



final B-tree

Deletion from a B-Tree

Delete key

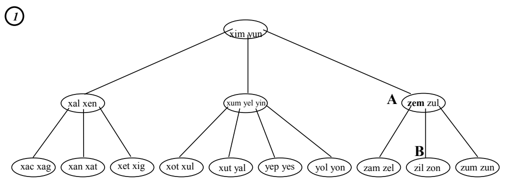
If node still has at least `CEIL[m/2]-1` keys then OK

If not -

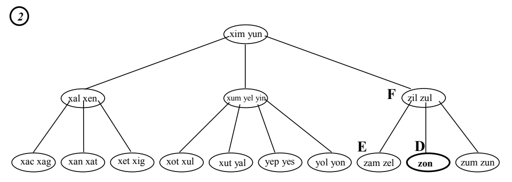
- If there is a lexicographic successor (i.e. node with deleted key is not a leaf node) – promote it.
- If any node is left with less than `CEIL[m/2]-1` keys merge that node with left or right hand sibling and common parent

This may leave parent node with too few keys so continue merging which may result in leaving the root empty in which case it is deleted, thereby reducing the number of levels in the tree by 1

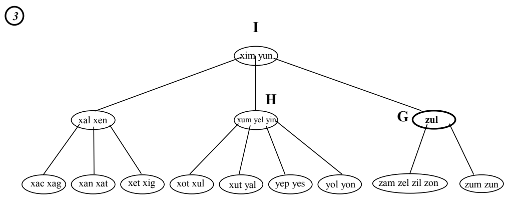
Deletion from B-Tree (Case 1)



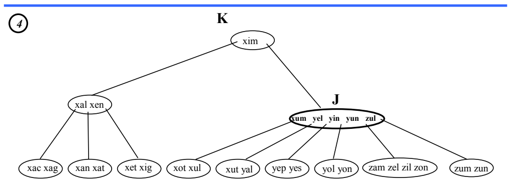
For B tree of order 5:
Maximum number of keys per node = m -1 = 5 -1 = 4
Minimum number of keys per node = CEIL [$^m/_2$] -1 = CEIL [$^5/_2$] -1 =2
Replace 'zem' in node A by its lexicographic succesor 'zil' in node B



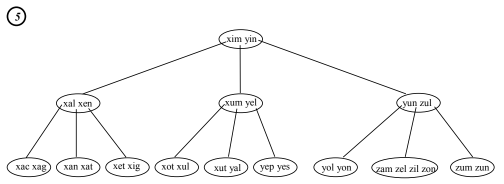
Node D has too few keys and is therefore merged with left-hand sibling, node E, and their common parent 'zil' in node F



Node G now contains too few nodes and is therefore merged with left-hand sibling, node H, and their common parent 'yun' in node I



Node J now contains too many keys and is therefore split and the middle key 'yin' passed up to the root, node K, for insertion.

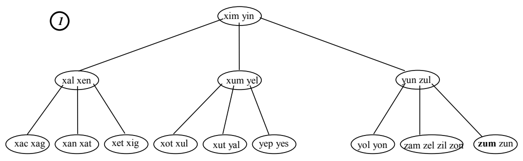


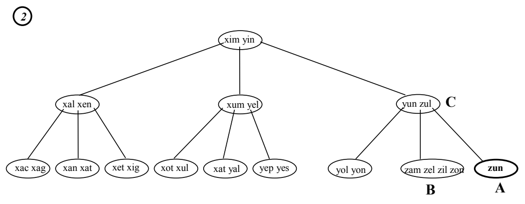
The deletion of 'zem' has now been completed.

Deletion from B-Tree (Case 2)

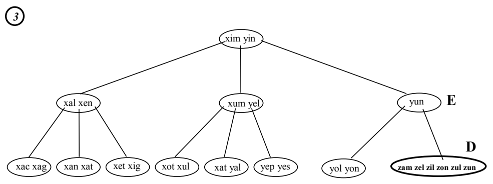
Deletion of key from terminal node where resulting node has less than minimum number of keys

Delete 'zum' from previous tree

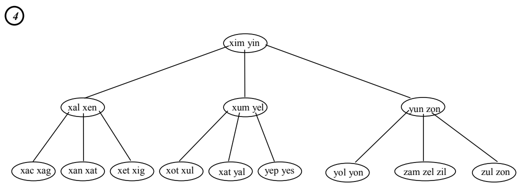




Node A now contains too few keys and is merged with left-hand sibling, node B, and their common parent 'zul' in node C.



Node D is now too full, it therefore splits in two and the middle key, say 'zon', is passed up to E for insertion



The deletion of 'zum' has now been completed

B-trees as Primary file organisation technique

entry in B-tree used as a dynamic multi-level index consists of:

<search key, record pointer, tree pointer>
i.e. data records are stored separately

B-tree can also be used as a primary file organisation technique; each entry consists of:
<search key, *data record*, tree pointer>

works well for small files with small records;
otherwise fan-out and number of levels becomes too great for efficient access

B-Trees: Bottom up

Key insight of B-trees:

- *We can build tree upwards from the bottom instead of downwards from the top*
- Bayer and McCreight recognised that the decision to work down from the root was, of itself, the problem
- B-trees allow the root to emerge rather than set it up and then find ways to change it

Splitting and Promoting

In a B-tree a page node consists of an ordered sequence of keys and corresponding pointers

There is no explicit tree within a page as there is in paged trees

The number of pointers always exceeds the number of keys by one

Many different definitions of the order of a B-tree. We will go with the following:

Splitting and Promoting

A B-tree of capacity order *d* has:

- d <= number of keys <= 2d in each node except the root
- 1 <= number of keys <= 2d in the root

All leaf nodes are on the same level

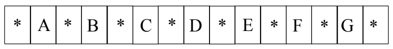
- This means that the tree is always balanced.
- There are no deeper levels on one branch of the tree than another

Splitting and Promoting

Building the first page is easy enough:

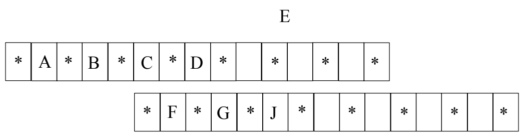
- As we insert new keys, we use a single disk access to read the page into memory and, working in memory, insert the key into its place in the page
- Since we are working in memory, this insertion is cheap

Suppose we want to add the key J to the B-tree:



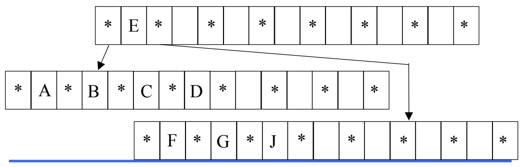
Splitting and Promoting

The single leaf we have is full, so we split the leaf into two leaves, distributing the keys as evenly as possible between the old leaf node and the new one, using one key to split the keys



Splitting and Promoting

Since we now have two leaves, we need to create a higher level in the tree to enable us to choose between the leaves when searching. We need to create a new root



Algorithms for B-trees

We have seen how B-trees work on paper, but how do they work on computer?

Page Structure:

- Here we show how the nodes of a B-tree should be represented in memory:

```
class BTNode {
        int         KeyCount;              /* no of keys stored in page */
        char        Key [d * 2];           /* the actual keys */
        int         Children [d * 2 + 1]; /* record nos of children */
}
```

Algorithms for B-Trees

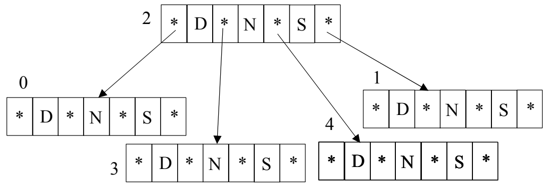
Searching

- Searching illustrates the charactistic aspects of most B-tree algorithms
  - *They are recursive*
  - *They work in two stages, working alternatively on entire pages and then* within *pages.*
- The searching procedure calls itself recursively, seeking to a page, and then searching through that page
- It looks for a key at successively lower levels of the tree until it finds the key, or it has reached the bottom of the tree

Algorithms for B-trees

```
Function: search(NodePtr, Key)
    if ( NodePtr == NULL )
        return NOT FOUND
    else
        Node = load tree node pointed to by NodePtr
        Pos = binary search node for key
        if ( found )
            return FOUND
        else
            return search(Node.pointers[Pos], Key)
        endif
    endif
end function
```

Algorithms for B-trees



Let's work through the function by hand searching for the key K in the above tree

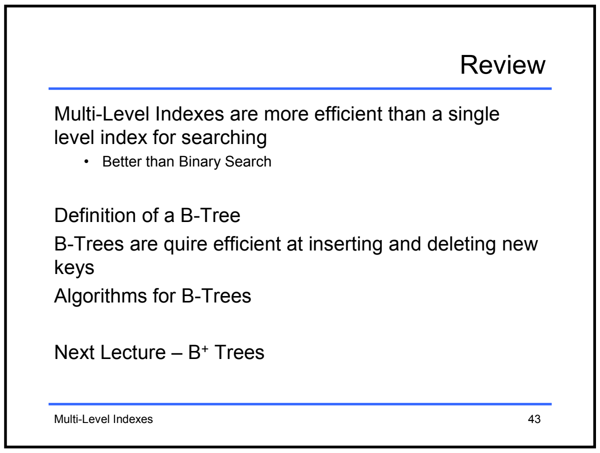
- We begin with `NodePtr` equal to the record number of the root of the B-tree (2)

Algorithms for B-trees

- `NodePtr` is not NULL, so the function reads the root into the variable `Node`.
- The binary search does not find K, so `Pos` is set to the index of the pointer between D and N, that is `Node.pointers[0]`.
- The value of `Node.pointers[0]` is 3, which is the record number of the top of the sub-tree which contains keys between D and N.
- The function search is called again with parameters of `NodePtr` = 3 and `Key` = K.
- Again, `NodePtr` is not NULL, so the function reads node 3 into the variable `Node`.

Algorithms for B-trees

- The binary search does not find K, so `Pos` is set to the index of the pointer between I and M, that is `Node.pointers[0]`.
- The value of `Node.pointers[0]` is NULL. There is no sub-tree which contains keys between D and N.
- The function search is called again with parameters of `NodePtr` = NULL and `Key` = K.
- `NodePtr` is NULL, so the function returns `not found`.
- The value `not found` is passed back through the levels of return statements until the code that originally calls search receives this information.



Review

Multi-Level Indexes are more efficient than a single level index for searching

- Better than Binary Search

Definition of a B-Tree

B-Trees are quire efficient at inserting and deleting new keys

Algorithms for B-Trees

Next Lecture – B^+ Trees