Extra: B+ Trees

Motivations

Many times you want to minimize the disk accesses while doing a search. A binary search tree allows two keys per node – a B+ tree allows as many values that will fit on a page.

Differences between BST and B+

- B+ is a balanced tree
  - Same distance for every path through the tree
  - Not true for BST
- B+ tree keeps data only in the leaf nodes
  - The index nodes are simply for traffic control
  - BST has data with the keys
- B+ tree has many keys per node
  - BST has 1

Objectives

- To understand the basic structure of a B+ Tree
- Understanding how the way keys are stored allows for efficient searching
- To understand the insertion algorithm
- To understand the deletion algorithm
- To see how rotation minimizes the cost of insertion and deletion algorithms

What is a B+ tree?

- B+ tree – a dynamic structure that adjusts to changes in the file gracefully. It is the most widely used structure because it adjusts well to changes and supports both equality and range queries.
- It is a balanced tree in which the internal nodes direct the search and the leaf nodes contain the data entries. The leaf nodes are organized into a doubly linked list allowing us to easily traverse the leaf pages in either direction.

Example of a B+ tree
Advantages / Disadvantages

- Advantage of B⁺-tree index files:
  - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.

- (Minor) disadvantage of B⁺-trees:
  - extra insertion and deletion overhead, space overhead.

- Advantages of B⁺-trees outweigh disadvantages
  - B⁺-trees are used extensively

B⁺-Tree Index Files (Cont.)

A B⁺-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between \([n/2]\) and \(n\) children.
- A leaf node has between \([(n-1)/2]\) and \(n-1\) values
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and \((n-1)\) values.

B⁺-Tree Node Structure

- Typical node

<table>
<thead>
<tr>
<th>(P_1)</th>
<th>(K_1)</th>
<th>(P_2)</th>
<th>...</th>
<th>(P_{n-1})</th>
<th>(K_{n-1})</th>
<th>(P_n)</th>
</tr>
</thead>
</table>

- \(K_i\) are the search-key values
- \(P_i\) are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).

- The search-keys in a node are ordered
  \(K_1 < K_2 < K_3 < \ldots < K_{n-1}\)

(Initially assume no duplicate keys, address duplicates later)

Non-Leaf Nodes in B⁺-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with \(m\) pointers:
  - All the search-keys in the subtree to which \(P_i\) points are less than \(K_i\)
  - For \(2 \leq i \leq n-1\), all the search-keys in the subtree to which \(P_i\) points have values greater than or equal to \(K_{i-1}\) and less than \(K_i\)
  - All the search-keys in the subtree to which \(P_e\) points have values greater than or equal to \(K_{n-1}\)

<table>
<thead>
<tr>
<th>(P_1)</th>
<th>(K_1)</th>
<th>(P_2)</th>
<th>...</th>
<th>(P_{n-1})</th>
<th>(K_{n-1})</th>
<th>(P_n)</th>
</tr>
</thead>
</table>

Example of B⁺-tree

- Leaf nodes must have between 2 and 4 values \((n-1)/2\) and \(n-1\), with \(n = 5\).
- Non-leaf nodes other than root must have between 3 and 5 children \((n/2)\) and \(n\) with \(n = 5\).
- Root must have at least 2 children.

Observations about B⁺-trees

- Operations (insert, delete) on the tree keep it balanced. \(\log N\) cost where \(f\)-fanout, \(N = \theta\) of leaf pages.
- Minimum occupancy of 50% is guaranteed for each node except the root node if the deletion algorithm we will present is used. (in practice, deletes just delete the data entry because files usually grow, not shrink). Each node contains \(m\) entries where \(d \leq m \leq 2d\) entries.
- Search for a record is just a traversal from the root to the appropriate leaf. This is the height of the tree – because it is balanced is consistent. Because of the high fan-out, the height of a B⁺-tree is rarely more than 3 or 4.
Queries on B⁺-Trees

1. Find record with search-key value V.
   1. C=root
   2. While C is not a leaf node {
      1. Let i be least value such that V ≤ Kᵢ
      2. If no such exists, set C = last non-null pointer in C
      3. Else if (V=Kᵢ) Set C = Pᵢ₊₁ else set C = Pᵢ
   } //Now C is in leaf node containing Kᵢ
3. Let i be least value such that Kᵢ = V
4. If there is such a value, follow pointer Pᵢ to the desired record.
5. Else no record with search-key value k exists.

Queries on B⁺-Trees (Cont.)

1. If there are K search-key values in the file, the height of the tree is no more than \( \lceil \log_{2}K \rceil \).
2. A node is generally the same size as a disk block, typically 4 kilobytes.
3. With 1 million search key values and \( n = 100 \) at most \( \log_{2}(1,000,000) = 4 \) nodes are accessed in a lookup.
4. Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup — above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds.

Insertion Algorithm

1. Place the record in sorted position in the appropriate leaf.
2. Split the leaf page including the inserted record.
3. Middle key in the index page in sorted order.
4. Left leaf page contains records with keys below the middle key.
5. Right leaf page contains records with keys equal to or greater than the middle key.

Insertion

Examples of insertion with B⁺ tree with minimum size = 1.

Starting with a tree looking like this:

Insert 28

Insert 28 into the below tree:
Insert 25

Our next insertion is at 25. We look at the leaf node it would go in and find there is no room. We split the node, and roll the middle value to the index mode above it.

Insert 8

Our next case occurs when we want to add 8. The leaf node is full, so we split it and attempt to roll the index to the index node. It is full, so we must split it as well.

Insert 15

Our last case occurs when we want to add 15. This is going to result in the root node being split. The leaf node is full, as are the two index nodes above it. This gives us:
Delete Algorithm

<table>
<thead>
<tr>
<th>Leaf page less than 1/2?</th>
<th>Index page less than 1/2?</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>No</td>
<td>Delete the record from the leaf page. Arrange keys in ascending order to fill void. If the key of the deleted record appears in the index pages, use the next key to replace it.</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>Combine the leaf page and its sibling. Change the index page to reflect the change.</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>1. Combine the leaf page and its sibling. 2. Adjust the index page to reflect the change. 3. Combine the index page with its sibling. 4. Delete entry from parent. Continue combing index pages until you reach a page with the correct fill factor or you reach the root page.</td>
</tr>
</tbody>
</table>

Delete

Let’s take our tree from the insert example with a minor modification (we have added 30 to give us an index node with 2 indexes in it):

Let’s delete 18 from the below tree:

Our next delete is similar, except the index appears in a index node. In that case, the next index replaces the one in the index node. Let’s delete 25 from the previous slide.

Delete 25

Our first delete is of 18. Simplest case is that it is not an index and in a leaf node that deleting it will not take you below 1/2.
Delete 28
Delete 28 from the below tree:

Our next case takes the node below d. Let’s delete 28. For this one we combine the leaf page (in our case it is empty) with its sibling and update the index appropriately. That gives us:

Delete 30
Delete 30 from the below tree:

Next we delete 30. This takes us below d for the index. We combine the indexes, which has the effect of taking the index above below d. This continues to the root.

Woah. That seemed like magic. What process got us to that?

Ok – let’s go through it. When we deleted 30, which took the data entry node that 30 was in below d. Now we have to merge with the sibling. When we merge – it’s to the sibling on the left, which means pointer in the index above is no longer valid. We remove it (which leaves it less than d), pull down the index from above and merge the index node with its sibling.
Delete 30

Repeating the process gets us back to

```
Liang, Introduction to Java Programming, Tenth Edition, (c) 2015 Pearson Education, Inc. All
rights reserved.
```

Delete 5

```
Delete 5 from the below tree:
```

```
Liang, Introduction to Java Programming, Tenth Edition, (c) 2015 Pearson Education, Inc. All
rights reserved.
```

Delete 5

```
Our last example deletes 5. This takes the node and the index
above it below d. We remove the leaf node and combine the
index with its neighbor.
```

```
Liang, Introduction to Java Programming, Tenth Edition, (c) 2015 Pearson Education, Inc. All
rights reserved.
```

Rotation

```
It is also possible to rebalance a tree to reduce the number of
splits – called rotation. If you are trying to insert, and a leaf
page is full, but its sibling isn’t – you can move an index to
a sibling and avoid splitting. Let’s go back to a tree from
our insert example:
```

```
Liang, Introduction to Java Programming, Tenth Edition, (c) 2015 Pearson Education, Inc. All
rights reserved.
```

Add 3

```
We want to add 3 – but in this case we check the sibling to see if
it has room. It does, so we move a record to it adjusting the
index. Now we have:
```

```
Liang, Introduction to Java Programming, Tenth Edition, (c) 2015 Pearson Education, Inc. All
rights reserved.
```
The same concept works with deletes.

Bulk Loading of a B+ Tree

- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.
- **Bulk Loading** can be done much more efficiently.
- **Initialization**: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.

Bulk Loading (Contd.)

- Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up, it splits. (Split may go up right-most path to the root.)
- Much faster than repeated inserts.

Summary of Bulk Loading

- Option 1: multiple inserts.
  - Slow.
  - Does not give sequential storage of leaves.
- Option 2: **Bulk Loading**
  - Has advantages for concurrency control.
  - Fewer I/Os during build.
  - Leaves will be stored sequentially (and linked, of course).
  - Can control “fill factor” on pages.