B Tree

- B-tree is a specialized multiway tree designed especially for use on disk
- B-Tree consists of a root node, branch nodes and leaf nodes containing the indexed field values in the ending (or leaf) nodes of the tree
- Rudolf Bayer and Ed McCreight invented the B-tree while working at Boeing Research Labs in 1971 (Bayer & McCreight 1972), but they did not explain what, if anything, the B stands for. Douglas Comer explains:
  > The origin of "B-tree" has never been explained by the authors. As we shall see, "balanced," "broad," or "bushy" might apply. Others suggest that the "B" stands for Boeing. Because of his contributions, however, it seems appropriate to think of B-trees as "Bayer"-trees. (Comer 1979, p. 123)

B-Tree Characteristics

- In a B-tree each node may contain a large number of keys
- B-tree is designed to branch out in a large number of directions and to contain a lot of keys in each node so that the height of the tree is relatively small
- Constraints that tree is always balanced
- Space wasted by deletion, if any, never becomes excessive
- Insert and deletions are simple processes
  > Complicated only under special circumstances
  > Insertion into a node that is already full or a deletion from a node makes it less then half full

Aspects of B Tree

- Specs of 'd' order B Tree:
  1. Internal nodes (except for root) has at least d, at most 2d key
  2. Root (if it is not a leaf) has at least 2 children
  3. All leafs have same level (balanced tree)
  4. Level increase and decreases are handled to the up (not down)
  5. An internal node has at least d key and d+1 children
Suppose we start with an empty B-tree and keys arrive in the following order:
1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45

We want to construct a B-tree of order 5

The first four items go into the root:

To put the fifth item in the root would violate condition 5
Therefore, when 25 arrives, pick the middle key to make a new root

6, 14, 28 get added to the leaf nodes:

Adding 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf

7, 52, 16, 48 get added to the leaf nodes

Adding 68 causes us to split the rightmost leaf, promoting 48 to the root, and adding 3 causes us to split the leftmost leaf, promoting 3 to the root; 26, 29, 53, 55 then go into the leaves

Adding 45 causes a split of 25, 26, 28 and promoting 28 to the root then causes the root to split
Constructing a B-tree (contd.)

![B-tree Diagram]

Inserting into a B-Tree

- Attempt to insert the new key into a leaf
- If this would result in that leaf becoming too big, split the leaf into two, promoting the middle key to the leaf’s parent
- If this would result in the parent becoming too big, split the parent into two, promoting the middle key
- This strategy might have to be repeated all the way to the top
- If necessary, the root is split in two and the middle key is promoted to a new root, making the tree one level higher

Structure of Btree Node

- **Bucket Factor**: number of records inserted into one node
- **Fan-out**: number of children sourced from one node
- The size of each node is equal to size of the page/block

Exercise in Inserting a B-Tree

- Insert the following keys to a 5-way B-tree:
  - 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56
Removal from a B-tree

During insertion, the key always goes into a leaf. For deletion we wish to remove from a leaf. There are three possible ways we can do this:

1. If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted.
2. If the key is not in a leaf then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf — in this case we can delete the key and promote the predecessor or successor key to the non-leaf deleted key's position.

Type #1: Simple leaf deletion

Delete 2: Since there are enough keys in the node, just delete it.

Type #2: Simple non-leaf deletion

Note when printed the slide is animated
**Type #4: Too few keys in node and its siblings**

12 29 36

7 9 15 22 31 43 69 72

Join back together.

Too few keys!

Note when printed this slide is animated.

---

**Type #4: Too few keys in node and its siblings**

12 29

7 9 15 22 31 43 56 69

Note when printed this slide is animated.

---

**Type #3: Enough siblings**

12 29

7 9 15 31 43 56 69

Demote root key and promote leaf key.

Delete 22.

Note when printed this slide is animated.

---

**Type #3: Enough siblings**

12 31

7 9 15 29 43 56 69

Note when printed this slide is animated.
Exercise in Removal from a B-Tree

- Given 5-way B-tree created by these data (last exercise):
  - 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56
- Add these further keys: 2, 6, 12
- Delete these keys: 4, 5, 7, 3, 14

Analysis of B-Trees

- The maximum number of items in a B-tree of order \( m \) (2\( d+1 \)) and height \( h \):
  - root \( m - 1 \)
  - level 1 \( m(m - 1) \)
  - level 2 \( m^2(m - 1) \)
  - \( \ldots \)
  - level \( h \) \( m^h(m - 1) \)
- So, the total number of items is
  \[
  (1 + m + m^2 + m^3 + \ldots + m^h)(m - 1) = \\
  [(m^{h+1} - 1) / (m - 1)](m - 1) = m^{h+1} - 1
  \]
- When \( m = 5 \) and \( h = 2 \) this gives \( 5^3 - 1 = 124 \)

Reasons for using B-Trees

- When searching tables held on disc, the cost of each disc transfer is high but doesn’t depend much on the amount of data transferred, especially if consecutive items are transferred
  - If we use a B-tree of order 101, say, we can transfer each node in one disc read operation
  - A B-tree of order 101 and height 3 can hold 101^4 – 1 items (approximately 100 million) and any item can be accessed with 3 disc reads (assuming we hold the root in memory)
- If we take \( m = 3 \), we get a 2-3 tree, in which non-leaf nodes have two or three children (i.e., one or two keys)
  - B-Trees are always balanced (since the leaves are all at the same level), so 2-3 trees make a good type of balanced tree

B+ Tree: node structures

- \( P_i \): child pointer
- \( K_i \): search key
- \# of search key = \( n \), \# of pointers = \( n+1 \)

leaf

- \( L \): pointers to left and right neighbours
What is a B+ Tree

Answer:
1. Both tree in slide 3 and slide 4 are valid; how you store data in B+ Tree depend on your algorithm when it is implemented.

2. As long as the number of data in each leaf are balanced, it doesn’t matter how many data you stored in the leaves. For example: in the previous question, the n can be 3 or 4, but can not be 5 or more than 5.

2. The searching time in a B+ tree is much shorter than most of other kinds of trees. For example: To search a data in one million key-values, a balanced binary requires about 20 block reads, in contrast only 4 block reads is required in B+ Tree.
   (The formula to calculate searching time can be found in the book. Page 492-493)
Searching

- Since no structure change in a B+ tree during a searching process, so just compare the key value with the data in the tree, then give the result back.

  For example: find the value 45, and 15 in below tree.

```
[25
  5 10 15 20
  25 30
  50 55 60 65
  75 80 85 90]
```

Searching

- Result:
  1. For the value of 45, not found.
  2. For the value of 15, return the position where the pointer located.

Insertion

- Since insert a value into a B+ tree may cause the tree unbalance, so rearrange the tree if needed.

- Example #1: insert 28 into the below tree.

```
[25
  5 10 15 20
  25 30
  50 55 60 65
  75 80 85 90]
```

Insertion

- Result:

```
[25 15 75
  5 10 15 20
  25 28 30
  50 55 60 65
  75 80 85 90]
```
**Insertion**

- Example #2: insert 70 into below tree

```
  25  50  75
  5  10  15  20  25  28  30  50  55  60  65  75  80  85  90
```

**Process: split the tree**

```
  5  10  15  20  25  28  30  50  55  60  65  75  80  85  90

  50  55

  60  65

  70
```

Violates the 50% rule, split the leaf

**Result:** chose the middle key 60, and place it in the index page between 50 and 75.

```
  5  10  15  20  25  28  30  50  55  60  65  75  80  85  90
```

**The insert algorithm for B+ Tree**

<table>
<thead>
<tr>
<th>Date Page Full</th>
<th>Index Page Full</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>NO</td>
<td>Place the record in sorted position in the appropriate leaf page.</td>
</tr>
<tr>
<td>YES</td>
<td>NO</td>
<td>Split the leaf page.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1. Place Middle Key in the index page in sorted order.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Leaf leaf page contains records with keys below the middle key.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. Right leaf page contains records with keys equal to or greater than the middle key.</td>
</tr>
<tr>
<td>YES</td>
<td>YES</td>
<td>Split the leaf page.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1. Records with keys &lt; middle key go to the left leaf page.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Records with keys &gt;= middle key go to the right leaf page.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Split the index page.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Split the index page.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. Keys &lt;= middle key go to the left index page.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. Keys &gt; middle key go to the right index page.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5. The middle key goes to the next (higher level) index.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IF the next level index page is full, continue splitting the index page.</td>
</tr>
</tbody>
</table>
Insertion

Exercise: add a key value 95 to the below tree.

Violation the 50% rule, split the leaf.

Deletion

Same as insertion, the tree has to be rebuild if the deletion result violate the rule of B+ tree.

Example #1: delete 70 from the tree

Result:

Insertion

Result: again put the middle key 60 to the index page and rearrange the tree.
Deletion

Example #2: delete 25 from below tree, but 25 appears in the index page.

But...

This is OK.

Deletion

Example #3: delete 60 from the below tree

Deletion

Result: replace 28 in the index page.

Add 28

Deletion

Result: delete 60 from the index page and combine the rest of index pages.
## Deletion

- Delete algorithm for B+ trees

<table>
<thead>
<tr>
<th>Leaf Page Below Fill Factor</th>
<th>Index Page Below Fill Factor</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>NO</td>
<td>Delete the record in the leaf page. Arrange keys in ascending order to fill void. If the key of the deleted record appears in the index page, use the next key to replace.</td>
</tr>
<tr>
<td>YES</td>
<td>NO</td>
<td>Combine the leaf page and the sibling. Change the index page to reflect the change.</td>
</tr>
</tbody>
</table>
| YES                         | YES                           | 1. Combine the leaf page and the sibling.  
2. Adjust the index page to reflect the change.  
3. Combine the index page with the sibling.  
Continue combining index pages until you reach a page with the correct fill factor or reach the end page. |

- Deletion algorithm for B+ trees

- NO: Delete the record in the leaf page. Arrange keys in ascending order to fill void. If the key of the deleted record appears in the index page, use the next key to replace.

- YES: Combine the leaf page and the sibling. Change the index page to reflect the change.