# Goals for Today

- Reason about tradeoffs between clustered vs. unclustered tree indexes
- Understand the difference and tradeoffs between **static** and **dynamic** tree-based indexes
- Learn the **algorithms** for search, insert, and delete for both types of tree indexes

---

## I/O Operation Cost

<table>
<thead>
<tr>
<th>Operation</th>
<th>Unclustered Alt-2 Tree Index</th>
<th>Clustered Alt-2 Tree Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan all records</td>
<td>(B) (ignore index... why?)</td>
<td>(1.5B) (ignore index)</td>
</tr>
<tr>
<td>Equality Search</td>
<td>((\log_2 0.5B) + 1 + 1)</td>
<td>((\log_2 0.5B) + 1 + 1)</td>
</tr>
<tr>
<td>Unique key</td>
<td>(\text{assume an index or data entry is } 1/3 \text{ size of a record, so } # \text{ pages at leaf level} = 0.33 \times 1.5B = 0.5B)</td>
<td>(# \text{ matching_leaf_pages} + # \text{ matching records})</td>
</tr>
<tr>
<td>Range Search</td>
<td>((\log_2 0.5B) + # \text{ matching_leaf_pages} + # \text{ matching records})</td>
<td>((\log_2 0.5B) + # \text{ matching_leaf_pages} + # \text{ matching pages})</td>
</tr>
<tr>
<td>Insert</td>
<td>(\text{search } + 1 + 2 (2 \text{ for } r/w \text{ in heap}))</td>
<td>(\text{search } + 1 + 2)</td>
</tr>
<tr>
<td>Delete</td>
<td>same as insert</td>
<td>same as insert</td>
</tr>
</tbody>
</table>

---

## Example: Indexes

Student relation organized as a Heap:

```
CREATE CLUSTERED index sidIndex ON Students(sid)
USING B-tree;
```

```
CREATE Index syntax (and options)
varies between DBMSs!!
```

- Exercise 2

```
Students(sid, name, class)
```

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>class</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Erin, JR</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Carl, FF</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Alice, SR</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Frank, JR</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Bob, FR</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Dina, SO</td>
<td></td>
</tr>
</tbody>
</table>

---

In some (many?) systems, specifying CLUSTERED implies Alternative 1

- (this is a canonical example)
Want Index on name too!

- Cross off which options are not possible for the new index (given our existing Alt 1 tree index on sid)
  - Clustered
  - Unclustered
  - Tree-based
  - Hash-based

- Alt 1 (data entries are data records)
- Alt 2 (data entries are pairs of key → record id)
- Alt 3 (data entries are pairs of key → {record ids})

Tree Indexes: Indexed Sequential Access Method

- ISAM is an old-fashioned idea
  - B+ trees are usually better, as we’ll see
    - Though not always
  - But, it’s a good place to start
  - Simpler than B+ tree, but many of the same ideas

- Upshot
  - Don’t brag about being an ISAM expert
  - Do understand how they work, and tradeoffs with B+ trees
Example ISAM Tree

- **Index entries:** `<search key value, page id>` they direct search to data entries *in leaves.*
- Example where each node can hold 2 entries

---

ISAM has a STATIC Index Structure

*Index File creation:*
1. Allocate leaf pages sequentially
2. Sort records by search key
3. Allocate and fill index entry pages (now the structure is ready for use)
4. Allocate overflow pages as needed

**Static tree structure:** inserts/deletes affect only leaf nodes of tree.
Example: Insert 23*, Delete 51*

After deletion 51 will still appear in index levels, but not in leaf!

Exercise: (3) on worksheet
Insert 21*, 13*, 16*, 32*, 29*

B+ Tree: The Most Widely Used Index

Insert/delete at log N cost; keep tree height-balanced.

- Each node (except for root) contains \( m \) entries: \( d \leq m \leq 2d \) entries.
- “\( d \)” is called the order of the tree.
  (so maintain 50% min occupancy)
- Supports equality and range-searches efficiently.

As in ISAM, all searches go from root to leaves, but structure is dynamic.

Example B+ Tree

- Search begins at root page, and key comparisons direct it to a leaf (as in ISAM)
- Search for 5*, 15*, all data entries \( \geq 24* \) ...

Based on the search for 15*, we know it is not in the tree!
B+Tree Insertions and Deletions

• Important goals for tree modification:

1. Maintain balanced nature of tree! (non-leaf pages at least half-full)

2. Maintain correctness of pointers

3. Only leaf pages contain data entries

Example B+ Tree – Inserting 23

Example B+ Tree – Inserting 8

Notice that root was split, leading to increase in height.
B+Tree: Inserting a Data Entry

- Find correct leaf $L$.
- Put data entry into $L$.
  - If $L$ has enough space, done!
  - Else, must split $L$ (into $L$ and a new node $L_2$)
    - Redistribute entries evenly, copy up middle key.
    - Insert index entry pointing to $L_2$ into parent of $L$.

- This can happen recursively
  - To split index node, redistribute entries evenly, but push up middle key. (Contrast with leaf splits.)

- Splits “grow” tree; root split increases height.
  - Tree growth: gets wider or one level taller at top.

Leaf vs. Index Page Split
(from previous example of inserting “8”)

- Minimum occupancy is guaranteed in both leaf and index page splits

- Note difference between copy-up and push-up;

Example Tree - Delete 19*

Entry to be inserted in parent node. (Note that 5 is copied up and continues to appear in the leaf.)

Entry to be inserted in parent node. (Note that 17 is pushed up and only appears once in the index. Contrast this with a leaf split.)
Example Tree – Now, Delete 20*

Under-occupancy!
Need to re-distribute.

Example Tree – Delete 20*

Take from sibling

Example Tree – Then Delete 24*

Too few entries!
Can’t redistribute,
Must merge...

Example Tree – Delete 24*

Too few entries!

Merge adjacent nodes, pull down 17
B+ Tree: Deleting a Data Entry

- Find correct leaf L.
- Remove the entry.
  - If L is at least half-full, done!
  - If L has only d-1 entries,
    - Try to re-distribute, borrowing from sibling (adjacent node with same parent as L).
    - If re-distribution fails, merge L and sibling.
- If merge occurred:
  - If merging leaf pages must delete entry (pointing to L or sibling) from parent of L.
  - Else if merging non-leaf pages, must pull down parent entry
- Merge could propagate to root, decreasing height.

Be sure to update the “differentiating entry” between the two siblings.

HeapPages in SimpleDb

- Bits are just bits (zeroes and ones)
  - The software we write imposes meaning on them
  - E.g., 00000110
    - could mean the number 6
    - could mean slots 1, 2 in a heap page are occupied!
  - Note how we read the bits from right to left
    - I.e., the least significant bit is the right-most bit

- Header bytes in HeapPage

SimpleDb HeapPage

- Example: Slot 10’s bit would be in the second byte (byte 1)
  - Generally, slot i in byte floor( i / 8 )
  - (other ways of computing this too)

- Bitwise operators!
  - <<, &
  - Check if a bit is 0:
    headerByte & (1 << headerBit) == 0

Java Exceptions

- So far in Lab 1:
  - Possibly seen java.lang.NullPointerException!!
  - Followed documentation to throw exceptions, e.g., throw new DbException();

- Coming up:
  - May need to catch exceptions, e.g., catch (IOException e) {...}
  - In general, poor design (and hides bugs!) to catch multiple exceptions just by one catch clause that catches the parent class Exception