B-Tree
Motivation for studying Multi-way and B-trees

- A disk access is very expensive compared to a typical computer instruction (mechanical limitations) - One disk access is worth about 200,000 instructions.
- Thus, When data is too large to fit in main memory the number of disk accesses becomes important.
- Many algorithms and data structures that are efficient for manipulating data in primary memory are not efficient for manipulating large data in secondary memory because they do not minimize the number of disk accesses.
- For example, AVL trees are not suitable for representing huge tables residing in secondary memory.
- The height of an AVL tree increases, and hence the number of disk accesses required to access a particular record increases, as the number of records increases.
What is a Multi-way tree?

- A multi-way (or m-way) search tree of order m is a tree in which
  - Each node has at-most \( m \) subtrees, where the subtrees \textit{may be empty}.
  - Each node consists of at least 1 and at most \( m-1 \) keys
  - The keys in each node are sorted.

- The keys and subtrees of a non-leaf node are ordered as:
  \[
  T_0, k_1, T_1, k_2, T_2, \ldots, k_{m-2}, T_{m-2}, k_{m-1}\]
  such that:
  - All keys in subtree \( T_0 \) are less than \( k_1 \).
  - All keys in subtree \( T_i \), \( 1 \leq i \leq m - 2 \), are greater than \( k_i \) but less than \( k_{i+1} \).
  - All keys in subtree \( T_{m-1} \) are greater than \( k_{m-1} \).
Each element in subtree number 0 is less than 66
Each element in subtree number 1 is between 66 and 88.
Each element in subtree number 2 is greater than 88.
Dictionary for Secondary storage

• The AVL tree is an excellent dictionary structure when the entire structure can fit into the main memory.
  – following or updating a pointer only requires a memory cycle.
• When the size of the data becomes so large that it cannot fit into the main memory, the performance of AVL tree may deteriorate rapidly
  – Following a pointer or updating a pointer requires accessing the disk once.
  – Traversing from root to a leaf may need to access the disk \( \log_2 n \) time.
    • when \( n = 1048576 = 2^{20} \), we need 20 disk accesses. For a disk spinning at 7200rpm, this will take roughly 0.166 seconds. 10 searches will take more than 1 second! This is way too slow.
Introduction of B-tree

- B-tree: proposed by Bayer and McCreight 1972
- A B-tree operates closely with secondary storage and can be tuned to reduce the impediments imposed by this storage
- One important property of B-trees is the size of each node which can be made as large as the size of the block. (the basic unit of I/O operations associated with a disk is a block)
- a B-tree of order $t$ is a multiway search tree.
• A B-tree is not a binary tree because B-tree has many more than two children
• B-trees may be formulated to store a set of elements or a bag of elements. (a given elements can occur many times in a bag but only once in a set)
• A B-tree is balanced.
• Every leaf in a B-tree has the same depth
• 2-3-4 tree (discussed by Rudolf Bayer): a B-tree of order 4 (min degree=2)
A B-tree of order m (or branching factor m), where m > 2, is either an empty tree or a multiway search tree with the following properties:

- The root is either a leaf or it has at least two non-empty subtrees and at most m non-empty subtrees.
- Each non-leaf node, other than the root, has at least \([m/2]\) non-empty subtrees and at most m non-empty subtrees. (Note: \([x]\) is the lowest integer > x).
- The number of keys in each non-leaf node is one less than the number of non-empty subtrees for that node.
- All leaf nodes are at the same level; that is the tree is perfectly balanced.
What is a B-tree? (cont’d)

For a non-empty B-tree of order $m$:

<table>
<thead>
<tr>
<th></th>
<th>Root node</th>
<th>Non-root node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum number of keys</td>
<td>1</td>
<td>$\lceil m/2 \rceil - 1$</td>
</tr>
<tr>
<td>Minimum number of non-empty subtrees</td>
<td>2</td>
<td>$\lceil m/2 \rceil$</td>
</tr>
<tr>
<td>Maximum number of keys</td>
<td>$m - 1$</td>
<td>$m - 1$</td>
</tr>
<tr>
<td>Maximum number of non-empty subtrees</td>
<td>$m$</td>
<td>$m$</td>
</tr>
</tbody>
</table>

This may be zero, if the node is a leaf as well

These will be zero if the node is a leaf as well
More on Why B-Trees

- B-trees are suitable for representing huge tables residing in secondary memory because:
  1. With a large branching factor \( m \), the height of a B-tree is low resulting in fewer disk accesses.

  Note: As \( m \) increases the amount of computation at each node increases; however this cost is negligible compared to hard-drive accesses.

  2. The branching factor can be chosen such that a node corresponds to a block of secondary memory.

  3. The most common data structure used for database indices is the B-tree. An **index** is any data structure that takes as input a property (e.g. a value for a specific field), called the search key, and **quickly** finds all records with that property.
Comparing B-Trees with AVL Trees

• The height $h$ of a B-tree of order $m$, with a total of $n$ keys, satisfies the inequality:
  \[ h \leq 1 + \log_{\frac{m}{2}} \left( \frac{n + 1}{2} \right) \]
• If $m = 300$ and $n = 16,000,000$ then $h \approx 4$.
• Thus, in the worst case finding a key in such a B-tree requires 3 disk accesses (assuming the root node is always in main memory).
• The average number of comparisons for an AVL tree with $n$ keys is $\log n + 0.25$ where $n$ is large.
• If $n = 16,000,000$ the average number of comparisons is 24.
• Thus, in the average case, finding a key in such an AVL tree requires 24 disk accesses.
Definition

• The height of a B-tree
  – Worst-case height
    • $h \leq \log_t(n+1)/2$
  – Proof

\[
\begin{align*}
n \geq & \quad 1 + (t - 1) \sum_{i=1}^{h} 2t^{i-1} \\
= & \quad 1 + 2(t - 1) \left( \frac{t^{h} - 1}{t - 1} \right) \\
= & \quad 2t^{h} - 1.
\end{align*}
\]
Insertion in B-Trees

• **OVERFLOW CONDITION:**
  A root-node or a non-root node of a B-tree of order \( m \) overflows if, after a key insertion, it contains \( m \) keys.

• Insertion algorithm:

  If a node overflows, split it into two, propagate the "middle" key to the parent of the node. If the parent overflows the process propagates upward. If the node has no parent, create a new root node.

• Note: Insertion of a key always **starts** at a leaf node.
Insertion in B-Trees

- Insertion in a B-tree of odd order

Example: Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in this order in an initially empty B-tree of order 3.
• Insert the following keys into an initially empty 5-way B-Tree
  • $a \ g \ f \ b \ k \ d \ h \ m \ j \ e \ s \ i \ r \ x \ c \ l \ n \ t \ u \ p$