Multi-way Search Trees

Introduction

Every node in a binary search tree contains one value and two pointers, **left** and **right**, which point to the node's left and right sub-trees, respectively. The same concept is us in an **M-way** search tree which has $M - 1$ values per node and M sub-trees. In such a tree M is called the degree of the tree. Every internal node of an M-way search tree consists of pointers to M sub-trees and contains $M - 1$ keys, where $M > 2$.

$$P_0 \quad K_0 \quad P_1 \quad K_1 \quad P_2 \quad K_2 \quad \cdots \quad P_{n-1} \quad K_{n-1} \quad P_n$$

All the key values are stored in ascending order. That is, $K_1 < K_{i+1}$ for $0 \leq i \leq n - 2$

In an M-way search tre, it is not compulsory that every node has exactly $M - 1$ values and M sub-trees. Rather, the node can have anywhere from 1 to $M - 1$ values, and the number of sub-trees can bary from 0 (for a leaf node) to $i + 1$, where i is the number of key values in the node. M is thus a fixed upper limit that defines how many key values can be stored in the node.

Basic properties of M-way search tree.

- Note that the key values in the sub-tree pointed by P_0 are less than the key value K_0. Similarly, all the key values in the sub-tree pointed by P_1 are less than K_1, so on and so forth. Thus, the generalized rule is that all the key values in the sub-tree pointed by P_i are less than K_i, where $0 \leq i \leq n - 1$

- Note that the key values in the sub-tree pointed by P_1 are greater than the key value K_0. Similarly, all the key values in the sub-tree pointed by P_2 are greater than K_1, so on and so forth. Thus, the generalized rule is that all the key values in the sub-tree pointed by P_i are greater than K_{i-1}, where $0 \leq i \leq n - 1$.

In an M-way search tree, every sub-tree is also an M-way search tree and follows the same rules.

B Trees

A B-tree is a specialized M-way tree. A B-tree of order m can have a maximum of $m - 1$ keys and m pointers to its sub-trees. A B-tree may contain a large number of key values and pointers to sub-trees. Storing a large number of keys in a single node keeps the height of the tree relatively small.

A B-tree is designed to store data and allows search, insertion, and deletion operations to be performed in logarithmi amortized time. A B-tree of order m (the maximum number of children that each node can have) is a tree with all the properties of an M-way search tree. In addition it has the following properties:

(1) Every node in the B-tree has at most (maximum) m children.

(2) Every node in the B-tree except the root node and leaf node has at least (minimum) $\frac{m}{2}$ children. This condition helps to keep the tree bushy so that the path from the root node to the leaf is very short, even in a tree that stores a lot of data.

(3) The root node has at least two children if it is not a terminal (leaf) node.

(4) All leaf nodes are at the same level.

An internal node in the B-tree can have n number of children, where $0 \leq n \leq m$ It is not necessary that every node has the same number of children, but the only restriction is that the node should have at least $\frac{m}{2}$ children. While performing insertion and deletion operations in a B-tree, the number of child nodes may change. So, in order to maintain a minimum number of children, the internal nodes may be joined or split.

Searching for an Element in a B-Tree

Searching for an element in a B-tree is similar to that in binary search trees. Since the running time of the search operation depends upon the height of the tree, the algorithm to search for an element in a B-tree takes $O(\log_t n)$ time to execute.

Inserting a New Element in a B-Tree

In a B-tree, all insertions are done at the leaf node level. A new value is inserted in the B-tree using the algorithm below.

1. Search the B-tree to find the leaf node where the new key value should be inserted.

2. If the leaf node is not full, that is, it contains less than $m-1$ key values, then insert the new element in the node keeping the node's elements ordered.

3. If the leaf node is full, that is, the leaf node already contains $m-1$ key values, then

(a) insert the new value in order into the existing set of keys,

(b) split the node at its median into two nodes (note that the split nodes are half full), and

(c) push the median element up its parent's node. If the parent's node is already full, then split the parent node by following the same steps.

Deleting an Element from a B-Tree

Like insertion, deletion is also done from the leaf nodes. There are two cases of deletion. In the first case, a leaf node has to be deleted. In the second case, an internal node has to be deleted. Let us first see the steps involved in deleting a leaf node.

1. Locate the leaf node which has to be deleted.

2. If the leaf node contains more than the minimum number of key values (more than $\frac{m}{2}$ elements), then dekete the value.

3. Else if the leaf node does not contain $\frac{m}{2}$ elements, then fill the node by taking an element either from the left or from the right sibling.

(a) If the left sibling has more than the minimum number of key values, push its largest key into its parent's node where the key is deleted.

(b) Else, if the right sibling has more than the minimum number of key values, push its node o the leaf node where the key is deleted.

4. Else, if both left and right siblings contain only the minimum number of elements, then create a new leaf node by combining the two leaf nodes and the intervening element of the parent node (ensuring that the number of elements does not exceed the maximum number of elements a node can have, that is m). If pulling the intervening element from the parent node leaves it with less than the minimum number of keys in the node, then propagate the process upwards, thereby reducing the height of the B-tree.

To delete an internal node, promote the successor or predecessor of the key to be deleted to occupy the position of the deleted key. this predecessor or successor will always be in the leaf node. So the processing will be done as if a value from the leaf node has been deleted.

B+ Trees

A B+ Tree is a variant of a B-tree which stores data in a way that allows for efficient insertion, retrieval, and removal of records, each of which is identified by a **key**. While a B-tree can store both keys and records in its interior nodes, a B+ tree, in contrast, stores all the records at the leaf level of the tree; only keys are stored in the interior nodes.

The leaf nodes of a B+ tree are often linked to one another in a linked list. Typically, B+ trees are used to stored large amounts of data that cannot be stored in the main memory. With B+ trees, the secondary storage (magnetic disk) is used to store the leaf nodes of trees and the internal nodes of trees are stored in the main memory.

B+ trees store data only in the leaf nodes. All other nodes (internal nodes) are **index nodes** or i-nodes and store index values. This allows us to traverse the tree from the root down to the leaf node that stores the desired data item.

Many database systems are implemented using B+ tree structure because of its simplicity. Since all the data appear in the leaf nodes and are ordered, the tree is always balanced and makes searching for data efficient.

A B+ tree can be thought of as a multi-level index in which the leaves make up a dense index and the non-leaf nodes make up a sparse index. The advantage of B+ trees can be given as:

(1) Records can be fetched in equal number of disk accesses

(2) It can be used to perform a wide range of queries easily as leaves are linked to nodes at the upper level

(3) Height of the tree is less balanced

(4) Supports both random and sequential access to records

(5) Keys are used for indexing

Comparison Between B-trees and B+ Trees

B-trees

- **(1)** Seach keys are not repeated
- **(2)** Data is stored in ineternal or leaf nodes
- **(3)** Searching takes more time as data may be found in a leaf or non-leaf node
- **(4)** Deletion of non-leaf nodes is very complicated
- **(5)** Leaf nodes cannot be stored using linked lists
- **(6)** The strucutre and operations are complicated

B+ Tree

- **(1)** Stores redundant search key
- **(2)** Data is stored only in leaf nodes
- **(3)** Searching data is very easy as the data can be found in leaf nodes only
- **(4)** Deletion is very simply because data will be in the leaf node
- **(5)** Leaf node data are ordered using sequential linked lists
- **(6)** The structure and operations are simple

Inserting a New Element in a B+ Tree

A new element is simply added in the leaf node if there is space for it. If the data node in the tree where insertion has to be done is full, then that node is split into two nodes. This call for adding a new index value in the parent index node so that future queries can arbitrate between the two nodes.

Adding the new index value in the parent node may cause it, in turn, to split. In factm all the nodes on the path from a leaf to the root may split when a value is added to a leaf node. If the root node splits, a new leaf node is created and the tree grows by one level. Steps to insert a node in a B+ Tree

Step 1: Insert the new node as the leaf node.

Step 2: If the leaf node overflows, split the node and copy the middle element to next index node.

Step 3: If the index node overflows, split that node and move the middle element to next index page.

Deleting an Element from a B+ Tree

As in B-trees, deletion is always done from a leaf node. If deleting a data element leaves that node empty, then the neighboring nodes are examined and merged with the **underfull** node.

This process calls for the deletiong of an index value from the parent node which, in turn, may cause it to become empty. Similar to the insertion proces, deletiong may cause a merge-delete wave to run from a leaf node all the way up to the root. This leads to shrinking of the tree by one level. step to delete a node from a B+ Tree

Step 1: Delete the key and data from the leaves.

Step 2: If the leaf node underflows, merge that node wit the sibling and delete the key in between them.

Step 3: If the index node underflows, merge that node with the sibling and move down the key in between them.

2-3 Trees

In a 2-3 tree, each interior node has either two or three children:

- Nodes with two children are called 2-nodes. The 2-nodes have one data value and two children
- Nodes with three children are called 3-nodes. The 3-nodes have two data values and three children (left child, middle child, and a right child)

This means that a 2-3 tree is not a binary tree. In this tree, all the leaf nodes are at the same level (bottom level).

Searching for an Element in a 2-3 Tree

The search operation is used to determine whether a data value x is present in a 2-3 tree T. The process of searching a value in a 2-3 tree is very similar to searching a binary search tree.

The search for a data value x starts at the root. If k_1 and k_2 are two values stored in the root node, then

- if $x < k_1$, move to the left child
- if $x \geq k_1$ and the node has only two children, move to the right child.
- if $x \geq k_1$ and the node has three children, then move to the middle child if $x < k_2$ else to the right child if $x \geq k_2$

At the end of this process, the node with data value x is reached if and only if x is at this leaf.

Inserting a New Element in a 2-3 Tree

To insert a new value in the 2-3 tree, an apppropriate position of the value is located in one of the leaf nodes. If after insertion of the value, the properties of the 2-3 tree do not get violated then insertion is over. Otherwise, if any property is violated then the violating node must be split.

Splitting a node

A node is split when it has three value data values and four children.

Deleting an Element from a 2-3 Tree

In the deletion process, a specified data value is deleted from the 2-3 tree. If deleting a value from a node violates the property of a tree, that is, if a node is left with less than one data value then two nodes must be merged together to preserve the general properties of a 2-3 tree.

In insertion,the new value had to be added in any of the leaf nodes but in deletion it is not necessary that the value has to be deleted from a leaf node. The value can be deleted from any of the nodes. To delete a value x, it is replaced by its in-order successor and then removed. If a node becomes empty after deleting a value, it is then merged with another node to restore the property of the tree.

Trie A trie is an ordered tree data structure. Trie stores keys that usualy strings. It is basically a k-ary position tree.

In contrast to binary search trees, nodes in a trie do not store the keys associated with them. Rather, a node's position in the tree represents the key associated with that node. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string.

Advantages Relative to Binary Search Tree

When compared with a binary search tree, the trie data structure has the following advantages:

Faster Search

Searching for keys is faster, as searching a key of length m takes $O(m)$ time in the worst case. On the other hand, a binary search tree performs $O(\log n)$ comparisons of keys, where n is the number of nodes in the tree. Since search time depends on the height of the tree which is logarithmic in the number of keys (if the tree is balanced), the worst case may take $O(m \log n)$ time. In addition to this m approaches $\log(n)$ in the worst case. Hence, a trie data structure provides a faster search mechanism.

Less space

Trie occupies less space, especially when it contains a large number of short strings. Since keys are not stored explicitly and nodes are shared between the keys with common initial subsequences, a trie calls for less space as compared to a binary search tree.

Longest Prefix-Matching

Trie facilitates the longest prefix-matching which enables us to find the key sharing the longest possible prefix of all unique characters. Since trie provides more advantages, it can be though of as a good replacement for binary search trees.

Advantages Relative to hash Table

Trie can also be used to replace a **hash table** as it provides the following advantages:

• Searching for data in a trie is faster in the worst case, $O(m)$ time, compared to an imperfect hash table, which may have numerous key collisions. Trie is free from collision of keys problem.

• Unlike a hash tablem there is no need to choose a hash function or to change it when more keys are added to a trie.

• A trie can sort the keys using a predetermined alphabetical ordering.

Disadvantages

The disadvantages of having a trie:

• In some cases, tries can be slower than hash tables while searching data. This is true in cases when the data is directly accessed on a hard disk drive or some other secondary storage device that has high randome access time as compated to the main memory.

s • All the key values cannot be easily represented as strings.

Applications

Tries are commonly used to store a dictionary. These applications take advantage of a trie's ability to quickly search, insert, and delete the entries. Tries are also used to implement approximate matching algorithms, including those used in spell-checking software.

Points to Remember

• An M-way search tree has $M - 1$ values per node and M sub-trees. In such a tree M is called the degree of the tree. M-way search tree consists of pointers to m sub-trees and contains $M - 1$ keys, where $M > 2$.

• A B-tree of order m can have a maximum of $m - 1$ keys and m pointers to its sub-trees. A B-tree may contain a large number of key values and pointers to its sub-trees.

• A B+ tree is a variant of B-tree which stores sorted data in a way that allows for efficient insertion, retrieval and removal of records, each of which is identified by a key. B+ tree record data at the leaf level of the tree; only keys are stored in interior nodes.

• A trie is an ordered tree data structure which stores keys that are usually strings. It is basically a k-ary position tree.

• In contrast to binary search trees, nodes in a trie do not store the keys associated with them. Rather, a node's position in the tree represents the key associated with that node.

• In a 2-3 tree, each interior node has either two or three children. This means that a 2-3 tree is not a binary tree.