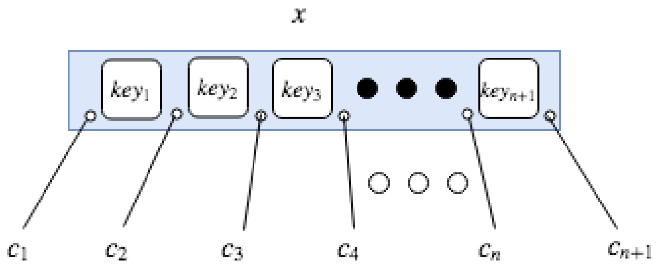
B-Trees

Introduction

If you have read my tutorials on 2-3 trees (https://algorithmtutor.com/Data-Structures/Tree/2-3-Trees/) and 2-3-4 trees (https://algorithmtutor.com/Data-Structures/Tree/2-3-4-Trees/), you know that a node in these balanced search trees have more than 1 keys. These trees are the special cases of a much generalized search tree called a **B-tree**. In B-tree, a node can have n keys where n is the positive integer ≥ 2. A node with n keys have $n+1$ child nodes. A typical B-tree node x has following information.

1. The number of keys $x.n$
2. The array of keys $[x.key_1, x.key_2, \dots, x.key_{x.n}]$. These keys are sorted in ascending order i.e. $x.key_1 \leq x.key_2 \leq \dots \leq x.key_{x.n}$.
3. The array of pointers to the $x.n+1$ child nodes $[x.c_1, x.c_2, \dots, x.c_{x.n+1}]$. If x is a leaf node, this array is empty as leaf nodes do not have children.
4. An identifier $x.leaf$ that gives if x is a leaf node.

Figure 1 shows a visual representation of x.



(data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAASwAAAEsCAMA

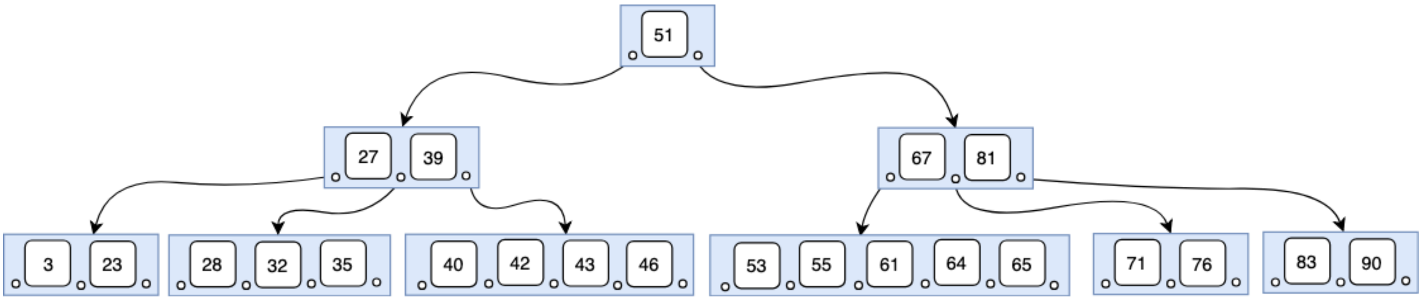
Figure 1: A typical B-tree node.

Besides the regular search tree invariants, B-tree has following additional invariants.

1. All the leaf nodes must be on the same level (perfect balance).

2. All the nodes except the root node must have at $t - 1$ keys where t is called *minimum order*. If $t = 3$, all the nodes except root nodes must have at least 2 keys. The value of t should be at least 2.
3. All the nodes can have at most $2t - 1$ keys.
4. A node with n keys must have $n + 1$ child nodes.
5. If a root node is a non-leaf node, it must have at least 2 child nodes.

An example of a B-tree with $t = 3$ is given in Figure 2.



(data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAASwAAAEsCAMA

Figure 2: An example of a B-tree of minimum degree 3

Since it is a perfectly balanced search tree, all the operations run in $O(\log n)$ time. The base of the logarithm is higher than 2.

The in-order traversal of the B-tree should yield keys in sorted order. For this, keys in a subtree c_i must be between k_{i-1} and k_i inclusive.

Why do we need B-trees?

We already have binary search trees that guarantee $O(\log n)$ running time for all the operations and are much more easy to implement. Why do we still need B-trees? Before answering this question, let us consider a scenario. Suppose we want to store very large information into a tree. The information is so large that it doesn't fit into the device's main memory. If it is the case, the program can not load the whole tree into the memory. What it does is it only load one node at a time from the disk and then process it. Reading from and writing to the disk is an expensive operation. It takes more time to read a node from the disk than to process it.

In order to lower the overall running time, we need to perform as few as possible disk read and write operations and this is where B-trees come into play. B-trees node has more branching factor meaning the node has more than 2 child nodes which in turn makes the height small. When a tree has a small height, it takes less number of read operations to go from root to any leaf node. A binary search tree (AVL or Red-Black) is much deeper than a B-tree with the same number of keys. This means, if we store the large information into a binary search tree, we need to perform much more disk read/write operations which make it much much slower than the B-trees.

Operations

Search

Searching a B-tree is similar to searching a binary search tree. In BST, we make a binary branching decision on every node. We compare a target key with the node's key and move either to the left subtree or to the right subtree. In the B-tree we need to make multi-way branching decisions as it may have more than 2 child nodes. The pseudo-code for the search operation is given below. Here `x` is the node being compared and `k` is the target key.

```
B-TREE-SEARCH ($x$, $k$)
   $i$ = 1
   while $i \le x.n$ and $k \ge x.key_i$
      i = i + 1
   if $i \le x.n$ and $k == x.key_i$
      return $(x, i)$
   else if $x$ is a leaf node
      return "Not Found"
   else
      read $x.c_i$ from the disk
      B-TREE-SEARCH($x.c_i$, $k$)
```

If the key is found in the node x at index i, it returns the node and corresponding index. Otherwise it reports not found message.

Insertion

The insertion operation is much involved than the search operation. To insert a key k into a B-tree, we first find the appropriate leaf node. If the leaf node has less than $2t-1$ keys i.e. not full, we simply insert k into its appropriate index. If the node is full, we can not insert the key into this node. In this case, we split the node from the middle (median key), move the median key to its parent node and insert k. The split operation is illustrated in figure 3.

(data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAASwAAAEsCAMA

Figure 3: Illustrating the split operation

As you see in figure 3, the median key 12 is moved upward to the parent node and the remaining keys are split into two nodes. If the parent node is full, we split the parent node as well and repeat the same process.

Let us illustrate the insertion by inserting 56 into the B-tree given in figure 2. The first step is to find the appropriate leaf node. For this, we perform a search operation and we stop when we reach the leaf node. Figure 3 highlights the appropriate leaf node to insert 56.

(data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAASwAAAEsCAMA

Figure 4: Inserting 56. The appropriate leaf node is highlighted.

The minimum order of the tree is 3 that means it can hold a maximum of 5 keys. The leaf node has already 5 keys and that is why it is full. To insert 56, we need to split it first. Figure 5 shows the tree after the split.

(data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAASwAAAEsCAMA

Figure 5: The B-tree after the split

The median of the keys is 61. So we move 61 to its parent node. The parent node now has 3 keys which is perfectly fine. Now we are ready to insert 56. The appropriate node to insert 56 is the left of 61. Figure 6 shows the final B-tree after insertion.

(data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAASwAAAEsCAMA

Figure 6: The final B-tree after inserting 56

Deletion

Deletion is similar to the insertion operation but bit more complex. Please follow the deletion operation in 2-3-4 (https://algorithmtutor.com/Data-Structures/Tree/2-3-4-Trees/#Delete-Operation) tree. The deletion in B-tree is very similar to the deletion in 2-3-4 trees. To delete a key k from the B-tree, we do the following. (All the illustrations assume the minimum degree of the B-tree is 3)

1. Search the tree and find the node x containing k.
2. If x is a leaf node, we need to consider the following cases.
   a. If x has at least t keys, we delete k from x. This is illustrated in figure 7 with $t = 3$.

(data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAASwAAAEsCA

Figure 7: Deleting from a leaf node with $> t - 1$ keys.

b. If x has only $t - 1$ keys, we look at its immediate sibling nodes. If one of the siblings has at least t keys, we steal one key from the sibling. To steal the key, we move the key to the parent node and an appropriate key from the parent node is moved down to x. Now x has t keys, we can delete k from x. This is illustrated in figure 8.

(data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAASwAAAEsCA

Figure 8: Deletion from a leaf node with $t - 1$ keys. The immediate sibling has at least t keys.

c. If x and its immediate sibling nodes have $t - 1$ keys, we need to look at its parent node. If the parent node has at least t keys, we merge one of the keys from the parent node with x and one of its sibling node. The merge operation decreases the number of keys in the parent node by 1. This is illustrated in figure 9.

(data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAASwAAAEsC

Figure 9: Deletion form a leaf node. The immediate sibling nodes have only $t - 1$ keys but the parent has at least t keys.

d. If x, its immediate sibling nodes, and the parent node all have only $t - 1$ keys (but parent node has more than one key), we follow step 2c and merge the nodes. But his leaves the parent node with less than $t - 1$ keys. This violates the B-tree invariants. To correct this we follow the steps 2b and 2c on the parent node until the tree becomes a valid B-tree. This is illustrated in figure 10.

(data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAASwAAAEsCA

Figure 10: Deletion from a leaf node. Node x its immediate siblings and the parent node all have $t - 1$ keys.

e. If x satisfies case 2d and its parent node has only one key, the parent node must be a root node. In this case we shrink the tree by merging the root node with its two child nodes. This is illustrated in figure 11.

(data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAASwAAAEsCA

Figure 11: Illustrating the shrinking of the tree

3. If x is an internal node, we find the predecessor key k’ of k. We exchange k and k’. Now k’ is in the leaf node. We follow step 2 to delete k’ from the leaf node.

References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (n.d.). Introduction to algorithms (3rd ed.). The MIT Press.

‹ Splay Trees (/Data-Structures/Tree/Splay-Trees/)

Randomly Built Binary Search Trees (/Data-Structures/Tree/Randomly-Built-Binary-Search-Trees/) ›