Performance Boosts of Using a B-Tree: Takeaways

Syntax

- Retrieving a row of data from a file:

```
import linecache
row = linecache.getline(file_name, line_number)
print(row)
```

- Creating a simple B-Tree and inserting into a node:

```
class Node:
    def __init__(self, keys=None, children=None):
        self.keys = keys or []
        self.children = children or []
    def is_leaf(self):
        return len(self.children) == 0
    def __repr__(self):
     # Helpful method to keep track of Node keys.
        return "".format(self.keys)
class BTree:
    def __init__(self, t):
        self.t = t
        self.root = None
    def insert(self, key):
        self.insert_non_full(self.root, key)
```

- Searching a B-Tree:

```
class BTree(BaseBTree):
    def search(self, node, term):
        if not self.root:
            return False
        index = 0
        for key in node.keys:
            if key == term:
                return True
            if term > key:
               index += 1
        if node.is_leaf():
            return False
    return self.search(node.children[index], term)
```

Concepts

- A B-Tree is a sorted and balanced tree that contains nodes with multiple keys and children.

- An index is a data structure that contains a key and a direct reference to a row of data.
- The degree of a B-Tree is a property of the tree designed to bound the number of keys in a tree.
    - The minimum degree must be greater than or equal to two.
- The maximum number of children we can have per node is $2t$ where t is the degree of the tree. We call this property the order of the tree.
- The height of the B-Tree is given by the equation $\log_m(n) = h$, where m is order of the tree, n is the number of entries, and h is the height of the tree.
- The time complexity for inserting data into a B-Tree is $O(\log_m(n))$.

Resources

- B-Tree
- Degree and order of a tree