深入理解并实现基本的 B 树数据结构

杨其臻

May 08, 2025

在传统二叉搜索树中，每个节点只能存储一个关键字并拥有最多两个子节点。这种结构在内存中表现良好，但面对磁盘存储时，频繁的随机 I/O 会导致性能急剧下降。B 树通过多路平衡的设计，将多个关键字和子节点集中在单个节点中，使得一次磁盘读取可以获取更多有效数据。这种特性使其成为数据库、文件系统等场景的核心数据结构。本文将解析 B 树的核心原理，并基于 Python 实现一个支持插入、删除和查找的 B 树结构。

1 B 树的基础理论

1.1 B 树的定义与特性

B 树是一种多路平衡搜索树，其核心特征在于「平衡性」与「多路分支」。每个节点最多包含 $m-1$ 个关键字（m 为阶数），非根节点至少包含 $\lceil m/2 \rceil - 1$ 个关键字。所有叶子节点位于同一层，确保从根节点到任意叶子节点的路径长度相同。例如，一个 3 阶 B 树中，根节点可能包含 1-2 个关键字，非根节点至少包含 1 个关键字。
与二叉搜索树相比，B 树通过减少树的高度降低了磁盘访问次数。而相较于 B+ 树，B 树允许在内部节点存储数据，这使得某些场景下的查询效率更高，但牺牲了范围查询的性能。

1.2 关键操作逻辑

查找操作从根节点开始，逐层比较关键字以确定下一步搜索的子节点。例如，若当前节点关键字为 [10, 20]，查找值 15 时，会选择第二个子节点（对应区间 $10 < 15 \leqslant 20$）。
插入操作需维护节点的关键字数量上限。当节点关键字数超过 $m-1$ 时，需进行分裂：将中间关键字提升至父节点，左右两部分形成两个新子节点。若分裂传递到根节点，则树的高度增加。
删除操作更为复杂。若删除关键字后节点仍满足最小关键字数要求，则直接删除；否则需通过「借位」从兄弟节点获取关键字，或与兄弟节点「合并」以维持平衡。

2 B 树的实现细节

2.1 节点结构与初始化

B 树的节点需包含关键字列表、子节点列表以及是否为叶子节点的标志。以下 Python 代码定义了节点类：

```
class BTreeNode:
    def __init__(self, leaf=False):
        self.keys = [] # 存储关键字的列表
```

```
        self.children = [] # 存储子节点的列表
        self.leaf = leaf # 是否为叶子节点
```

初始化 B 树时需指定阶数 m，并创建空的根节点。例如，阶数为 3 的 B 树初始状态为一个空根节点。

2.2 插入操作的实现

插入操作的核心是递归查找插入位置，并在必要时分裂节点。以下代码展示了插入逻辑的关键片段：

```
def insert(self, key):
    root = self.root
    if len(root.keys) == (2 * self.m) - 1: # 根节点已满
        new_root = BTreeNode(leaf=False)
        new_root.children.append(root)
        self._split_child(new_root, 0) # 分裂原根节点
        self.root = new_root # 更新根节点
    self._insert_non_full(self.root, key)
```

_split_child 方法负责分裂子节点：

```
def _split_child(self, parent, index):
    child = parent.children[index]
    new_node = BTreeNode(leaf=child.leaf)
    mid = len(child.keys) // 2
    parent.keys.insert(index, child.keys[mid]) # 中间关键字提升至父节点
    new_node.keys = child.keys[mid+1:] # 右半部分成为新节点
    child.keys = child.keys[:mid] # 左半部分保留
    if not child.leaf:
        new_node.children = child.children[mid+1:]
        child.children = child.children[:mid+1]
    parent.children.insert(index+1, new_node) # 插入新子节点
```

此代码中，mid 变量确定分裂位置，原节点的右半部分被分离为独立节点，中间关键字提升至父节点。

3 代码实现与测试

3.1 查找方法的实现

查找操作通过递归遍历树结构实现：

```
def search(self, key, node=None):
    if node is None:
        node = self.root
    i = 0
```

```
    while i < len(node.keys) and key > node.keys[i]:
        i += 1
    if i < len(node.keys) and key == node.keys[i]:
        return True # 找到关键字
    elif node.leaf:
        return False # 到达叶子节点未找到
    else:
        return self.search(key, node.children[i])
```

时间复杂度为 $O(\log n)$，其中 n 为关键字总数。

3.2 删除操作的边界测试

删除根节点是特殊场景。例如，当根节点无关键字且只有一个子节点时，需将子节点设为新根：

```
def delete(self, key):
    self._delete(self.root, key)
    if len(self.root.keys) == 0 and not self.root.leaf:
        self.root = self.root.children[0] # 降低树高度
```

测试时需验证删除后所有节点仍满足 B 树的平衡条件，例如通过遍历检查每个节点的关键字数量是否在允许范围内。

4 B 树的应用与优化

4.1 实际应用场景

在 MySQL 的 InnoDB 引擎中，B+ 树作为索引结构，其叶子节点通过链表连接以支持高效范围查询。而原始 B 树因内部节点可存储数据，适用于需要频繁随机访问的场景，如某些文件系统的元数据管理。

4.2 优化方向

B+ 树通过将数据仅存储在叶子节点，减少了内部节点的大小，从而在相同磁盘页中容纳更多关键字。此外，Blink-Tree 通过添加「右兄弟指针」支持并发访问，允许在修改节点时其他线程继续读取旧版本数据。

B 树通过巧妙的多路平衡设计，在磁盘存储场景中展现出卓越性能。尽管其实现复杂度较高，但理解其核心原理并动手实现，是掌握高级数据结构的必经之路。读者可进一步探索 B+ 树或并发 B 树变种，以应对更复杂的工程需求。