1.题目名称

B-树应用

2.代码行数

378行

3.算法思想

1.素数筛

素数筛将[1,1e4]的素数筛出来，存入数组。

2.插入操作

如果是叶子节点，则直接插入数据。如果是内部节点，则找到正确的子节点并递归插入，若节点满则进行分裂节点再插入。

3.创建B-树

如果根节点为空，直接赋值，如果不为空则判断节点是否满，未满的话则直接进行插入操作，否则先分裂节点再插入。

4.寻找节点

如果找到目标节点返回true，反之递归进入合适的子节点继续查找。

5.删除节点

如果当前节点不是叶子节点且没有找到要删除的键，那么递归地在子节点中查找并删除该键

叶子节点中的删除:
如果当前节点是叶子节点，则直接从键数组中删除该键。

非叶子节点中的删除:
如果当前节点不是叶子节点，那么进行以下操作：
a. 找到前驱节点和后继节点。
b. 根据前驱节点和后继节点的数据量，决定如何删除目标数据。如果前驱节点有足够的数据，则用前驱节点的键替换目标键并递归删除该键；如果后继节点有足够的数据，则用后继节点的键替换目标键并递归删除该键；如果前驱和后继节点都没有足够的数据，则合并这两个节点并删除目标键。

4.主要/核心函数分析

Insert

```
void B_Tree::Insert(int num) {
    if (root == nullptr) {
        // 如果根节点为空，创建一个新的根节点并插入数据
        root = new Node;
        root->Is_Leaf = true;
        root->keys.push_back(num);
    } else {
        if (Find(num, root)) {
            // 如果数据已经存在于树中，直接返回
```

```
            return;
        }
        if (root->keys.size() == M - 1) {
            // 如果根节点已满，创建一个新的根节点，并将原根节点作为子节点进行分裂
            Node* newRoot = new Node;
            newRoot->Child.push_back(root);
            splitChild(newRoot, 0);
            InsertNonFull(newRoot, num);
            root = newRoot;
        } else {
            InsertNonFull(root, num);
        }
    }
}
```

这个方法负责向B树中插入一个新的整数。首先，它检查根节点是否为空。如果根节点为空，则创建一个新的根节点并插入该数字。如果根节点不为空，它会检查该数字是否已经存在于树中。如果已经存在，则直接返回。否则，它会检查根节点是否已满。如果根节点已满，则创建一个新的根节点，并将原根节点进行分裂,再递归地插入数字。如果根节点未满，则递归地插入数字。

InsertNonFull

```
void B_Tree::InsertNonFull(Node* node, int num) {
    int i = node->keys.size() - 1;
    if (node->Is_Leaf) {
        // 如果节点是叶子节点，直接插入数据
        while (i >= 0 && num < node->keys[i]) {
            node->keys[i + 1] = node->keys[i];
            i--;
        }
        if (i == -1)
            node->keys.insert(node->keys.begin(), num);
        else if (i == node->keys.size() - 1)
            node->keys.push_back(num);
        else
            node->keys[i + 1] = num;
    } else {
        // 如果节点是内部节点，找到正确的子节点并递归插入
        while (i >= 0 && num < node->keys[i]){
            i--;
        }
        i++;
        if (node->Child[i]->keys.size() == M - 1) {
            // 如果子节点已满，先进行分裂
            splitChild(node, i);
            if (num > node->keys[i]) {
                i++;
            }
        }
        InsertNonFull(node->Child[i], num);
    }
}
```

这个方法负责向节点中插入一个数字。如果该节点是叶子节点，则直接插入数字。如果该节点是内部节点，则找到正确的子节点并递归插入，如果子节点满了则先分裂再递归插入。

splitChild

```
void B_Tree::splitChild(Node* parent, int index) {
    Node* child = parent->Child[index];
    Node* newNode = new Node;
    newNode->Is_Leaf = child->Is_Leaf;

    parent->keys.insert(parent->keys.begin() + index, child->keys[M - 1]);
     //孩子子树弹出中间值 进入父节点

    std::move(child->keys.begin() + M, child->keys.end(),
std::back_inserter(newNode->keys));
    child->keys.erase(child->keys.begin() + M - 1, child->keys.end());

    if (!child->Is_Leaf) {      //将子树右半部分插入到新生成的节点
        std::move(child->Child.begin() + M, child->Child.end(),
std::back_inserter(newNode->Child));
        child->Child.erase(child->Child.begin() + M, child->Child.end());
    }

    parent->Child.insert(parent->Child.begin() + index + 1, newNode);
}
```

这个方法负责将一个子节点分裂成两个节点。它首先从子节点中弹出中间的键值，然后将其添加到父节点的键列表中。然后，它将子节点的后半部分移动到一个新的节点中，并更新父节点的子节点列表。

Find

```

bool B_Tree::Find(int num, Node* node) {
    int i = 0;
    while (i < node->keys.size() && num > node->keys[i]) {
        i++;
    }
    if (i < node->keys.size() && num == node->keys[i]) {
        // 如果找到了目标数据
        return true;
    } else if (node->Is_Leaf) {
        // 如果是叶子节点且未找到目标数据
        return false;
    } else {
        // 在合适的子节点继续查找
        return Find(num, node->Child[i]);
    }
}
```

如果找到目标节点返回true，反之递归进入合适的子节点继续查找。

DeleteNode

```
void B_Tree::DeleteNode(Node *node, int num) {
    int index = 0;
    while (index < node->keys.size() && num > node->keys[index]) {
        index++;
    }

```

```
    if (index < node->keys.size() && num == node->keys[index]) {
        if (node->Is_Leaf) {
            // 如果目标数据在叶子节点中，直接删除
            node->keys.erase(node->keys.begin() + index);
        } else {
            Node* predecessor = node->Child[index];
            Node* successor = node->Child[index + 1];

            if (predecessor->keys.size() >= (M + 1) / 2) {
                // 如果前驱节点有足够的数据，则找到前驱数据并递归删除
                int predKey = getPredecessor(predecessor);
                node->keys[index] = predKey;
                DeleteNode(predecessor, predKey);
            } else if (successor->keys.size() >= (M + 1) / 2) {
                // 如果后继节点有足够的数据，则找到后继数据并递归删除
                int succKey = getSuccessor(successor);
                node->keys[index] = succKey;
                DeleteNode(successor, succKey);
            } else {
                // 如果前驱和后继节点都只有(M + 1) / 2 - 1个数据，则合并两个节点并删除目标数据
                mergeNodes(node, predecessor, successor, index);
                DeleteNode(predecessor, num);
            }
        }
    } else {
        if (node->Is_Leaf) {
            return;
        }

        bool flag = (index == node->keys.size());
        Node* child = node->Child[index];

        if (child->keys.size() < (M + 1) / 2) {
            // 如果子节点只有(M + 1) / 2 - 1个数据，则进行修复操作
            fillChild(node, index);
        }

        if (flag && index > node->keys.size()) {
            child = node->Child[index - 1];
        }

        DeleteNode(child, num);
    }
}
```

- 输入：一个节点指针`node`和一个整数`num`。
- 功能：递归地删除给定节点中的目标数字。
- 流程：
    - 初始化一个索引`index`为0。
    - 在节点中循环查找目标数字。
    - 如果找到了目标数字且节点不是叶子节点：
        - 获取前驱和后继节点。

- 根据前驱和后继节点的键的数量，决定如何删除目标数字（通过前驱、后继或合并节点）。

5.测试数据(规模,测试次数)

规模:小于1e4的素数以及小于1e3的偶数

测试次数:3次

6.运行结果

Tree1.txt

```
200 no
201 no
202 no
203 no
204 no
205 no
206 no
207 no
208 no
209 no
210 no
211 yes
212 no
213 no
214 no
215 no
216 no
217 no
218 no
219 no
220 no
221 no
222 no
223 yes
224 no
225 no
226 no
227 yes
228 no
229 yes
230 no
231 no
232 no
233 yes
234 no
235 no
236 no
237 no
238 no
239 yes
240 no
241 yes
242 no
243 no
```

```
244 no
245 no
246 no
247 no
248 no
249 no
250 no
251 yes
252 no
253 no
254 no
255 no
256 no
257 yes
258 no
259 no
260 no
261 no
262 no
263 yes
264 no
265 no
266 no
267 no
268 no
269 yes
270 no
271 yes
272 no
273 no
274 no
275 no
276 no
277 yes
278 no
279 no
280 no
281 yes
282 no
283 yes
284 no
285 no
286 no
287 no
288 no
289 no
290 no
291 no
292 no
293 yes
294 no
295 no
296 no
297 no
298 no
299 no
300 no
```

Tree2.txt

```
601 no
607 no
613 no
617 no
619 no
631 no
641 no
643 no
647 no
653 no
659 no
661 no
673 no
677 no
683 no
691 no

```

Tree3.txt

```
100 yes
102 yes
104 yes
106 yes
108 yes
110 yes
112 yes
114 yes
116 yes
118 yes
120 yes
122 yes
124 yes
126 yes
128 yes
130 yes
132 yes
134 yes
136 yes
138 yes
140 yes
142 yes
144 yes
146 yes
148 yes
150 yes
152 yes
154 yes
156 yes
158 yes
160 yes
162 yes
164 yes
```

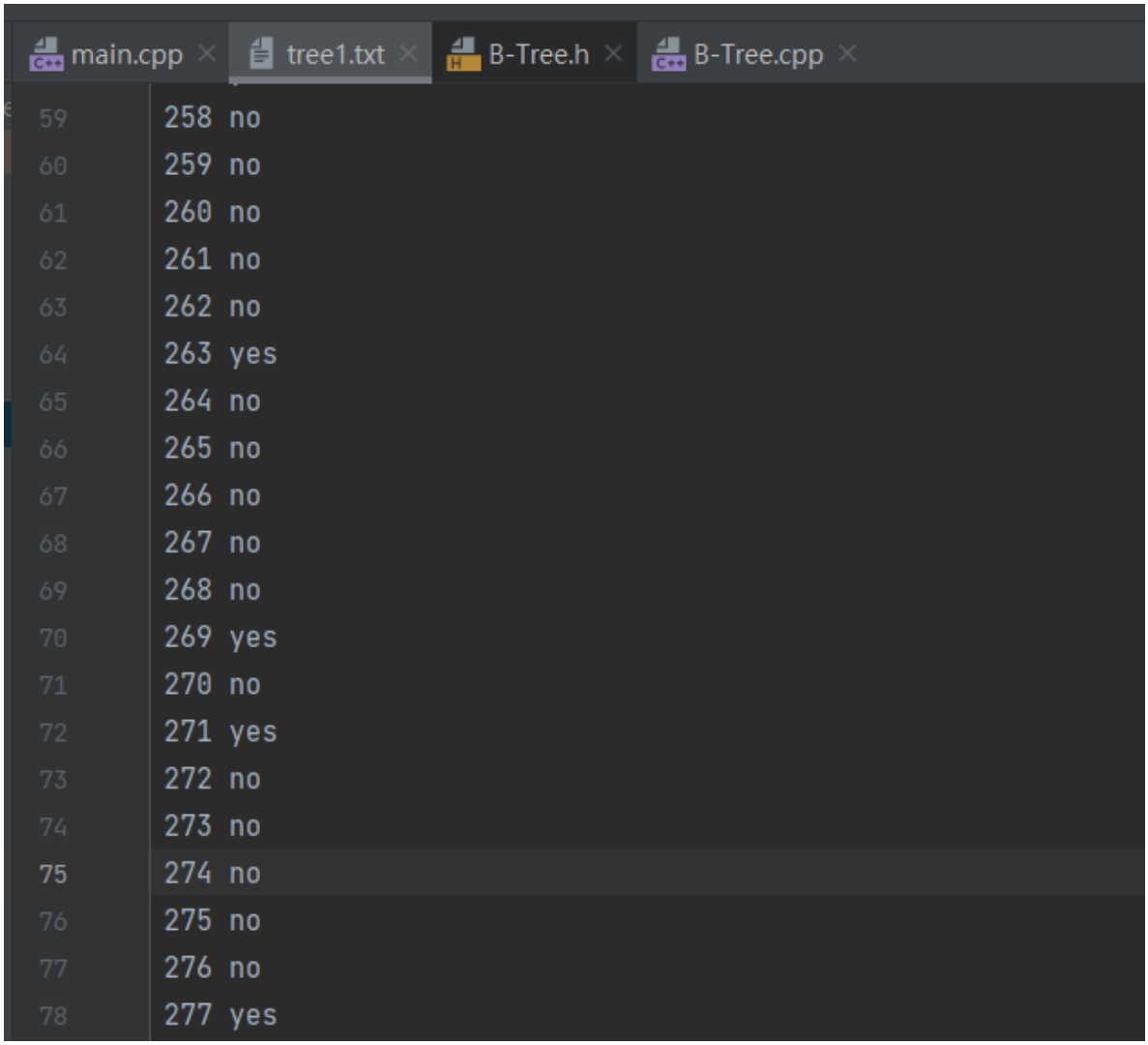
```
166 yes
168 yes
170 yes
172 yes
174 yes
176 yes
178 yes
180 yes
182 yes
184 yes
186 yes
188 yes
190 yes
192 yes
194 yes
196 yes
198 yes
200 yes
```

7.时间复杂度分析

1. `void B_Tree::CreateTree(int num)`：时间复杂度取决于 `Insert` 操作的时间复杂度，因此为 O(log n)，其中 n 是树中节点的数量。
2. `void B_Tree::Insert(int num)`：插入操作的时间复杂度为 O(log n)，其中 n 是树中节点的数量。在最坏情况下，需要进行树的分裂和节点的合并操作，但由于 B树的平衡性质，插入操作的平均时间复杂度仍然为 O(log n)。
3. `void B_Tree::InsertNonFull(Node* node, int num)`：时间复杂度为 O(log n)，其中 n 是节点中的键的数量。在最坏情况下，需要进行节点分裂操作，但节点的分裂复杂度是常数级的，因此不会影响整体的复杂度。
4. `void B_Tree::splitChild(Node* parent, int index)`：节点分裂操作的时间复杂度为 O(M)，其中 M 是 B树的阶数，即每个节点的最大键的数量。
5. `bool B_Tree::Find(int num, Node* node)`：查找操作的时间复杂度为 O(log n)，其中 n 是树中节点的数量。在最坏情况下，需要遍历到叶子节点或找到目标数据。
6. `void B_Tree::DeleteNode(Node *node, int num)`：删除节点操作的时间复杂度为 O(log n)，其中 n 是节点中的键的数量。在最坏情况下，需要进行节点的合并和修复操作，但节点的合并和修复复杂度是常数级的，因此不会影响整体的复杂度。
7. `void Select_Prime(){}`:素数筛时间复杂度为O(nloglogn)。

总体而言，时间复杂度为 O(nloglogn)。

8.结果截屏图片



9.心得体会

通过该题对素数筛有了更多的了解。通过实现B-树，对其的基本性质更加熟悉，同时对该树增删查的代码实现有了深入的了解。