

Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Программирование на языках высокого уровня

Тема: В-дерево – удаление узла

Студент группы 3331506/80401

Л. Н. Леонтьев

Преподаватель

Е.М. Кузнецова

«__» _____ 2021 г.

Санкт-Петербург

2021 г

Оглавление

Оглавление	2
Введение.....	3
Принцип работы	5
Оценка скорости и памяти	7
Список литературы	8
Приложение – Код	9

Введение

В-дерево (по-русски произносится как Би-дерево) — структура данных, дерево поиска. С точки зрения внешнего логического представления, сбалансированное, сильно ветвистое дерево. Часто используется для хранения данных во внешней памяти.

Использование В-деревьев впервые было предложено Р. Бэйером (англ. R. Bayer) и Э. МакКрейтом (англ. E. McCreight) в 1970 году.

Сбалансированность означает, что длина любых двух путей от корня до листьев различается не более, чем на единицу.

Ветвистость дерева — это свойство каждого узла дерева ссылаться на большое число узлов-потомков.

С точки зрения физической организации В-дерево представляется как мультисписочная структура страниц памяти, то есть каждому узлу дерева соответствует блок памяти (страница). Внутренние и листовые страницы обычно имеют разную структуру.

Структура В-дерева применяется для организации индексов во многих современных СУБД (системах управления базами данных).

В-дерево может применяться для структурирования (индексирования) информации на жёстком диске (как правило, метаданных). Время доступа к произвольному блоку на жёстком диске очень велико (порядка миллисекунд), поскольку оно определяется скоростью вращения диска и перемещения головок. Поэтому важно уменьшить количество узлов, просматриваемых при каждой операции. Использование поиска по списку каждый раз для нахождения случайного блока могло бы привести к чрезмерному количеству обращений к диску вследствие необходимости последовательного прохода по всем его элементам, предшествующим заданному, тогда как поиск в В-дереве, благодаря свойствам сбалансированности и высокой ветвистости, позволяет значительно сократить количество таких операций.

Относительно простая реализация алгоритмов и существование готовых библиотек (в том числе для C) для работы со структурой B-дерева обеспечивают популярность применения такой организации памяти в самых разнообразных программах, работающих с большими объёмами данных.

Принцип работы

Если корень одновременно является листом, то есть в дереве всего один узел, мы просто удаляем ключ из этого узла. В противном случае сначала находим узел, содержащий ключ, запоминая путь к нему. Пусть этот узел — x .

Если x — лист, удаляем оттуда ключ. Если в узле x осталось не меньше $t-1$ ключей, мы на этом останавливаемся. Иначе мы смотрим на количество ключей в следующем, а потом в предыдущем узле. Если следующий узел есть, и в нём не менее t ключей, мы добавляем в x ключ-разделитель между ним и следующим узлом, а на его место ставим первый ключ следующего узла, после чего останавливаемся. Если это не так, но есть предыдущий узел, и в нём не менее t ключей, мы добавляем в x ключ-разделитель между ним и предыдущим узлом, а на его место ставим последний ключ предыдущего узла, после чего останавливаемся. Наконец, если и с предыдущим ключом не получилось, мы объединяем узел x со следующим или предыдущим узлом, и в объединённый узел перемещаем ключ, разделяющий два узла. При этом в родительском узле может остаться только $t-2$ ключей. Тогда, если это не корень, мы выполняем аналогичную процедуру с ним. Если мы в результате дошли до корня, и в нём осталось от 1 до $t-1$ ключей, делать ничего не надо, потому что корень может иметь и меньше $t-1$ ключей. Если же в корне не осталось ни одного ключа, исключаем корневой узел, а его единственный потомок делаем новым корнем дерева.

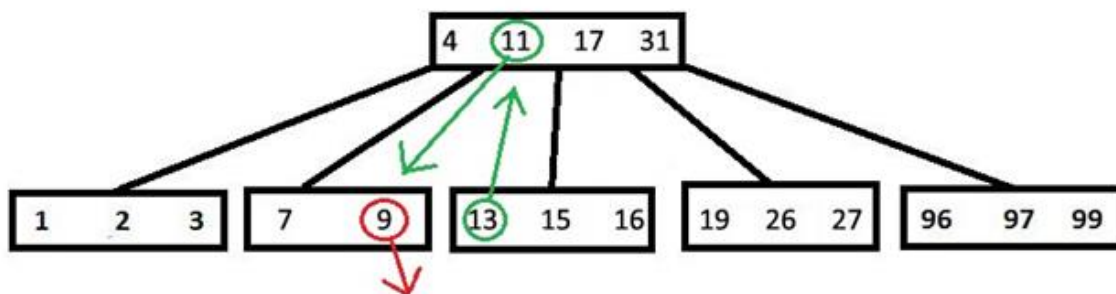


Рисунок 1 – Удаление из листа

Если x — не лист, а K — его i -й ключ, удаляем самый правый ключ из поддерева потомков i -го потомка x , или, наоборот, самый левый ключ из поддерева потомков $i+1$ -го потомка x . После этого заменяем ключ K удалённым ключом. Удаление ключа происходит так, как описано в предыдущем абзаце.

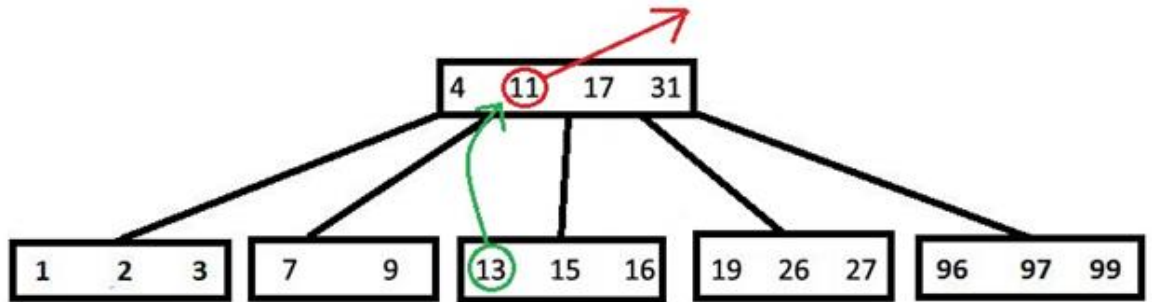


Рисунок 2 – Удаление из внутреннего узла

Оценка скорости и памяти

Операция удаления происходит за время $O(t \log t n)$. Дисковых операций требуется всего лишь $O(h)$, где h – высота дерева.

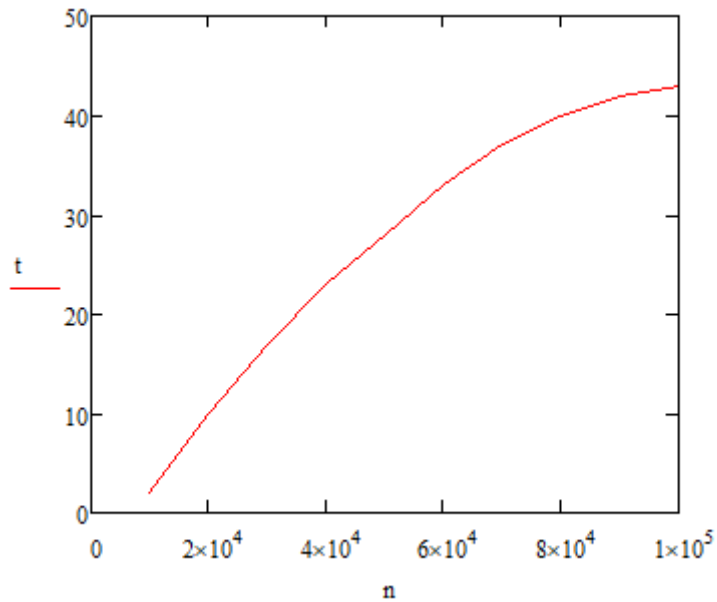


Рисунок 3 – График зависимости времени выполнения алгоритма от кол-ва входных данных

Список литературы

1. Т. Кормен «Алгоритмы: построение и анализ» второе издание, глава 18
2. Левитин А. В. Глава 7. Пространственно-временной компромисс: В-деревья // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006. — С. 331—339. — 576 с.
3. <https://habr.com/ru/post/114154/>

Приложение – Код

```
1  #include<iostream>
2  #include <time.h>
3
4  using namespace std;
5
6  const int t = 2;
7  struct BNode {
8      int keys[2 * t];
9      BNode* children[2 * t + 1];
10     BNode* parent;
11     int count;
12     int countSons;
13     bool leaf;
14 };
15
16 class Tree {
17 private:
18     BNode* root;
19     void insert_to_node(int key, BNode* node);
20     void sort(BNode* node);
21     void restruct(BNode* node);
22     void deletenode(BNode* node);
23     bool searchKey(int key, BNode* node);
24     void remove(int key, BNode* node);
25     void removeFromNode(int key, BNode* node);
26     void removeLeaf(int key, BNode* node);
27     void lconnect(BNode* node, BNode* othernode);
28     void rconnect(BNode* node, BNode* othernode);
29     void repair(BNode* node);
30
31 public:
32     Tree();
33     ~Tree();
34     void insert(int key);
35     bool search(int key);
36     void remove(int key);
37 };
38
39 Tree::Tree() { root = nullptr; }
40
41 Tree::~Tree() { if (root != nullptr) deletenode(root); }
42
43 void Tree::deletenode(BNode* node) {
44     if (node != nullptr) {
45         for (int i = 0; i <= (2 * t - 1); i++) {
46             if (node->children[i] != nullptr) deletenode(node->children[i]);
47             else {
48                 delete(node);
49                 break;
50             }
51         }
52     }
53 }
54
55 void Tree::insert_to_node(int key, BNode* node) {
56     node->keys[node->count] = key;
57     node->count = node->count + 1;
58     sort(node);
59 }
60
```

```

61 void Tree::sort(BNode* node) {
62     int m;
63     for (int i = 0; i < (2 * t - 1); i++) {
64         for (int j = i + 1; j <= (2 * t - 1); j++) {
65             if ((node->keys[i] != 0) && (node->keys[j] != 0)) {
66                 if ((node->keys[i]) > (node->keys[j])) {
67                     m = node->keys[i];
68                     node->keys[i] = node->keys[j];
69                     node->keys[j] = m;
70                 }
71             }
72             else break;
73         }
74     }
75 }
76
77 void Tree::insert(int key) {
78     if (root == nullptr) {
79         BNode* newRoot = new BNode;
80         newRoot->keys[0] = key;
81         for (int j = 1; j <= (2 * t - 1); j++) newRoot->keys[j] = 0;
82         for (int i = 0; i <= (2 * t); i++) newRoot->children[i] = nullptr;
83         newRoot->count = 1;
84         newRoot->countSons = 0;
85         newRoot->leaf = true;
86         newRoot->parent = nullptr;
87         root = newRoot;
88     }
89     else {
90         BNode* ptr = root;
91         while (ptr->leaf == false) {
92             for (int i = 0; i <= (2 * t - 1); i++) {
93                 if (ptr->keys[i] != 0) {
94                     if (key < ptr->keys[i]) {
95                         ptr = ptr->children[i];
96                         break;
97                     }
98                     if ((ptr->keys[i + 1] == 0) && (key > ptr->keys[i])) {
99                         ptr = ptr->children[i + 1];
100                         break;
101                     }
102                 }
103             }
104             else break;
105         }
106         insert_to_node(key, ptr);
107
108         while (ptr->count == 2 * t) {
109             if (ptr == root) {
110                 restruct(ptr);
111                 break;
112             }
113             else {
114                 restruct(ptr);
115                 ptr = ptr->parent;
116             }
117         }
118     }
119 }
120

```

```

121 void Tree::restruct(BNode* node) {
122     if (node->count < (2 * t - 1)) return;
123
124     BNode* child1 = new BNode;
125     int j;
126     for (j = 0; j <= t - 2; j++) child1->keys[j] = node->keys[j];
127     for (j = t - 1; j <= (2 * t - 1); j++) child1->keys[j] = 0;
128     child1->count = t - 1;
129     if (node->countSons != 0) {
130         for (int i = 0; i <= (t - 1); i++) {
131             child1->children[i] = node->children[i];
132             child1->children[i]->parent = child1;
133         }
134         for (int i = t; i <= (2 * t); i++) child1->children[i] = nullptr;
135         child1->leaf = false;
136         child1->countSons = t - 1;
137     }
138     else {
139         child1->leaf = true;
140         child1->countSons = 0;
141         for (int i = 0; i <= (2 * t); i++) child1->children[i] = nullptr;
142     }
143
144     BNode* child2 = new BNode;
145     for (int j = 0; j <= (t - 1); j++) child2->keys[j] = node->keys[j + t];
146     for (j = t; j <= (2 * t - 1); j++) child2->keys[j] = 0;
147     child2->count = t;
148     if (node->countSons != 0) {
149         for (int i = 0; i <= (t); i++) {
150             child2->children[i] = node->children[i + t];
151             child2->children[i]->parent = child2;
152         }
153         for (int i = t + 1; i <= (2 * t); i++) child2->children[i] = nullptr;
154         child2->leaf = false;
155         child2->countSons = t;
156     }
157
158     else {
159         child2->leaf = true;
160         child2->countSons = 0;
161         for (int i = 0; i <= (2 * t); i++) child2->children[i] = nullptr;
162     }
163
164     if (node->parent == nullptr) {
165         node->keys[0] = node->keys[t - 1];
166         for (int j = 1; j <= (2 * t - 1); j++) node->keys[j] = 0;
167         node->children[0] = child1;
168         node->children[1] = child2;
169         for (int i = 2; i <= (2 * t); i++) node->children[i] = nullptr;
170         node->parent = nullptr;
171         node->leaf = false;
172         node->count = 1;
173         node->countSons = 2;
174         child1->parent = node;
175         child2->parent = node;
176     }
177     else {
178         insert_to_node(node->keys[t - 1], node->parent);
179         for (int i = 0; i <= (2 * t); i++) {
180             if (node->parent->children[i] == node) node->parent->children[i] = nullptr;
181         }
182         for (int i = 0; i <= (2 * t); i++) {
183             if (node->parent->children[i] == nullptr) {
184                 for (int j = (2 * t); j > (i + 1); j--) node->parent->children[j] = node->parent->children[j - 1];
185                 node->parent->children[i + 1] = child2;
186                 node->parent->children[i] = child1;
187                 break;
188             }
189         }
190         child1->parent = node->parent;
191         child2->parent = node->parent;
192         node->parent->leaf = false;
193         delete node;
194     }

```

```

195
196 bool Tree::search(int key) {
197     return searchKey(key, this->root);
198 }
199
200 bool Tree::searchKey(int key, BNode* node) {
201     if (node != nullptr) {
202         if (node->leaf == false) {
203             int i;
204             for (i = 0; i <= (2 * t - 1); i++) {
205                 if (node->keys[i] != 0) {
206                     if (key == node->keys[i]) return true;
207                     if ((key < node->keys[i])) {
208                         return searchKey(key, node->children[i]);
209                         break;
210                     }
211                 }
212                 else break;
213             }
214             return searchKey(key, node->children[i]);
215         }
216         else {
217             for (int j = 0; j <= (2 * t - 1); j++)
218                 if (key == node->keys[j]) return true;
219             return false;
220         }
221     }
222     else return false;
223 }
224
225 void Tree::removeFromNode(int key, BNode* node) {
226     for (int i = 0; i < node->count; i++) {
227         if (node->keys[i] == key) {
228             for (int j = i; j < node->count; j++) {
229                 node->keys[j] = node->keys[j + 1];
230                 node->children[j] = node->children[j + 1];
231             }
232             node->keys[node->count - 1] = 0;
233             node->children[node->count - 1] = node->children[node->count];
234             node->children[node->count] = nullptr;
235             break;
236         }
237     }
238     node->count--;
239 }
240

```

```

241 void Tree::removeLeaf(int key, BNode* node) {
242     if ((node == root) && (node->count == 1)) {
243         removeFromNode(key, node);
244         root->children[0] = nullptr;
245         delete root;
246         root = nullptr;
247         return;
248     }
249     if (node == root) {
250         removeFromNode(key, node);
251         return;
252     }
253     if (node->count > (t - 1)) {
254         removeFromNode(key, node);
255         return;
256     }
257     BNode* ptr = node;
258     int k1;
259     int k2;
260     int position;
261     int positionSon;
262     int i;
263     for (int i = 0; i <= node->count - 1; i++) {
264         if (key == node->keys[i]) {
265             position = i;
266             break;
267         }
268     }
269     BNode* parent = ptr->parent;
270     for (int j = 0; j <= parent->count; j++) {
271         if (parent->children[j] == ptr) {
272             positionSon = j;
273             break;
274         }
275     }

```

```

276     if (positionSon == 0) {
277         if (parent->children[positionSon + 1]->count > (t - 1)) {
278             k1 = parent->children[positionSon + 1]->keys[0];
279             k2 = parent->keys[positionSon];
280             insert_to_node(k2, ptr);
281             removeFromNode(key, ptr);
282             parent->keys[positionSon] = k1;
283             removeFromNode(k1, parent->children[positionSon + 1]);
284         }
285         else {
286             removeFromNode(key, ptr);
287             if (ptr->count <= (t - 2)) repair(ptr);
288         }
289     }
290     else {
291         if (positionSon == parent->count) {
292             if (parent->children[positionSon - 1]->count > (t - 1)) {
293                 BNode* temp = parent->children[positionSon - 1];
294                 k1 = temp->keys[temp->count - 1];
295                 k2 = parent->keys[positionSon - 1];
296                 insert_to_node(k2, ptr);
297                 removeFromNode(key, ptr);
298                 parent->keys[positionSon - 1] = k1;
299                 removeFromNode(k1, temp);
300             }
301             else {
302                 removeFromNode(key, ptr);
303                 if (ptr->count <= (t - 2)) repair(ptr);
304             }
305         }
306         else {
307             if (parent->children[positionSon + 1]->count > (t - 1)) {
308                 k1 = parent->children[positionSon + 1]->keys[0];
309                 k2 = parent->keys[positionSon];
310                 insert_to_node(k2, ptr);
311                 removeFromNode(key, ptr);
312                 parent->keys[positionSon] = k1;
313                 removeFromNode(k1, parent->children[positionSon + 1]);
314             }
315             else {
316                 if (parent->children[positionSon - 1]->count > (t - 1)) {
317                     BNode* temp = parent->children[positionSon - 1];
318                     k1 = temp->keys[temp->count - 1];
319                     k2 = parent->keys[positionSon - 1];

```

```

320                     insert_to_node(k2, ptr);
321                     removeFromNode(key, ptr);
322                     parent->keys[positionSon - 1] = k1;
323                     removeFromNode(k1, temp);
324                 }
325             }
326             else {
327                 removeFromNode(key, ptr);
328                 if (ptr->count <= (t - 2)) repair(ptr);

```

```

329         }
330     }
331 }
332 }
333

```

```

334 void Tree::lconnect(BNode* node, BNode* othernode) {
335     if (node == nullptr) return;
336     for (int i = 0; i <= (othernode->count - 1); i++) {
337         node->keys[node->count] = othernode->keys[i];
338         node->children[node->count] = othernode->children[i];
339         node->count = node->count + 1;
340     }
341     node->children[node->count] = othernode->children[othernode->count];
342     for (int j = 0; j <= node->count; j++) {
343         if (node->children[j] == nullptr) break;
344         node->children[j]->parent = node;
345     }
346     delete othernode;
347 }
348
349 void Tree::rconnect(BNode* node, BNode* othernode) {
350     if (node == nullptr) return;
351     for (int i = 0; i <= (othernode->count - 1); i++) {
352         node->keys[node->count] = othernode->keys[i];
353         node->children[node->count + 1] = othernode->children[i + 1];
354         node->count = node->count + 1;
355     }
356     for (int j = 0; j <= node->count; j++) {
357         if (node->children[j] == nullptr) break;
358         node->children[j]->parent = node;
359     }
360     delete othernode;
361 }
362
363 void Tree::repair(BNode* node) {
364     if ((node == root) && (node->count == 0)) {
365         if (root->children[0] != nullptr) {
366             root->children[0]->parent = nullptr;
367             root = root->children[0];
368         }
369         else {
370             delete root;
371         }
372         return;
373     }
374     BNode* ptr = node;
375     int k1;
376     int k2;
377     int positionSon;
378     BNode* parent = ptr->parent;
379     for (int j = 0; j <= parent->count; j++) {
380         if (parent->children[j] == ptr) {
381             positionSon = j;
382             break;

```

```

383     }
384 }
385 if (positionSon == 0) {
386     insert_to_node(parent->keys[positionSon], ptr);
387     lconnect(ptr, parent->children[positionSon + 1]);
388     parent->children[positionSon + 1] = ptr;
389     parent->children[positionSon] = nullptr;
390     removeFromNode(parent->keys[positionSon], parent);
391     if (ptr->count == 2 * t) {
392         while (ptr->count == 2 * t) {
393             if (ptr == root) {
394                 restruct(ptr);
395                 break;
396             }
397             else {
398                 restruct(ptr);
399                 ptr = ptr->parent;
400             }
401         }
402     }
403     else
404         if (parent->count <= (t - 2)) repair(parent);
405 }
406 else {
407     if (positionSon == parent->count) {
408         insert_to_node(parent->keys[positionSon - 1], parent->children[positionSon - 1]);
409         lconnect(parent->children[positionSon - 1], ptr);
410         parent->children[positionSon] = parent->children[positionSon - 1];
411         parent->children[positionSon - 1] = nullptr;
412         removeFromNode(parent->keys[positionSon - 1], parent);
413         BNode* temp = parent->children[positionSon];
414         if (ptr->count == 2 * t) {
415             while (temp->count == 2 * t) {
416                 if (temp == root) {
417                     restruct(temp);
418                     break;
419                 }
420                 else {
421                     restruct(temp);
422                     temp = temp->parent;
423                 }
424             }
425         }
426         else
427             if (parent->count <= (t - 2)) repair(parent);
428     }
429     else {
430         insert_to_node(parent->keys[positionSon], ptr);
431         lconnect(ptr, parent->children[positionSon + 1]);
432         parent->children[positionSon + 1] = ptr;
433         parent->children[positionSon] = nullptr;
434         removeFromNode(parent->keys[positionSon], parent);

```



```

435         if (ptr->count == 2 * t) {
436             while (ptr->count == 2 * t) {
437                 if (ptr == root) {
438                     restruct(ptr);
439                     break;
440                 }
441                 else {
442                     restruct(ptr);
443                     ptr = ptr->parent;
444                 }
445             }
446         }
447         else
448             if (parent->count <= (t - 2)) repair(parent);
449     }
450 }
451 }
452
453 void Tree::remove(int key, BNode* node) {
454     BNode* ptr = node;
455     int position;
456     int i;
457     for (int i = 0; i <= node->count - 1; i++) {
458         if (key == node->keys[i]) {
459             position = i;
460             break;
461         }
462     }
463     int positionSon;
464     if (ptr->parent != nullptr) {
465         for (int i = 0; i <= ptr->parent->count; i++) {
466             if (ptr->children[i] == ptr) {
467                 positionSon = i;
468                 break;
469             }
470         }
471     }
472     ptr = ptr->children[position + 1];
473     int newkey = ptr->keys[0];
474     while (ptr->leaf == false) ptr = ptr->children[0];
475     if (ptr->count > (t - 1)) {
476         newkey = ptr->keys[0];
477         removeFromNode(newkey, ptr);
478         node->keys[position] = newkey;
479     }
480     else {
481         ptr = node;
482         ptr = ptr->children[position];
483         newkey = ptr->keys[ptr->count - 1];
484         while (ptr->leaf == false) ptr = ptr->children[ptr->count];
485         newkey = ptr->keys[ptr->count - 1];
486         node->keys[position] = newkey;
487         if (ptr->count > (t - 1)) removeFromNode(newkey, ptr);

```

```

488     else {
489         removeLeaf(newkey, ptr);
490     }
491 }
492 }
493
494 void Tree::remove(int key) {
495     BNode* ptr = this->root;
496     int position;
497     int positionSon;
498     int i;
499     if (searchKey(key, ptr) == false) {
500         return;
501     }
502     else {
503         for (i = 0; i <= ptr->count - 1; i++) {
504             if (ptr->keys[i] != 0) {
505                 if (key == ptr->keys[i]) {
506                     position = i;
507                     break;
508                 }
509                 else {
510                     if ((key < ptr->keys[i])) {
511                         ptr = ptr->children[i];
512                         positionSon = i;
513                         i = -1;
514                     }
515                     else {
516                         if (i == (ptr->count - 1)) {
517                             ptr = ptr->children[i + 1];
518                             positionSon = i + 1;
519                             i = -1;
520                         }
521                     }
522                 }
523             }
524             else break;
525         }
526     }
527     if (ptr->leaf == true) {
528         if (ptr->count > (t - 1)) removeFromNode(key, ptr);
529         else removeLeaf(key, ptr);
530     }
531     else remove(key, ptr);
532 }

```

```

533
534 int main()
535 {
536     Tree tree;
537     int key;
538     for (int i = 0; i < 100000; i++)
539     {
540         tree.insert(i);
541     }
542     clock_t start_time = clock();
543     for (int i = 0; i < 100000; i++)
544     {
545         tree.remove(i);
546     }
547     clock_t end_time = clock();
548     double seconds = (double)(end_time - start_time) / CLOCKS_PER_SEC * 1000;
549     cout << "Time = " << seconds << endl;
550 }

```