**数据结构实验报告**

|  |  |  |  |
| --- | --- | --- | --- |
| 学号-姓名 | 桑龙龙-20030540015 | 实验时间 | 2020年 12月 12日 |
| 诚信声明 | 本实验及实验报告所写内容为本人所作 | | |
| 实验题目 | 实验五  查找方法  题目一 顺序查找、折半查找  题目二 二叉排序树的建立、查找、插入和删除运算  题目三 哈希表的设计和应用 | | |
| 实验过程中遇到的主要问题 | 无 | | |
| 实验小结 | 本次试验进行了二分查找、二叉排序树、哈希表的设计，在进行二叉排序树的设计时，实践了书中讲的二叉平衡树，了解为什么要进行LL,RR,LR,RL旋转。在哈希表的设计时，考虑到一般的哈希表都是要有遍历操作的，实现了哈希表的迭代器。 | | |
| 数据结构  （自定义数据类型） | 二叉树类似的节点 | | |
| 主要算法  （或算法说明） | 1. //1、二分查找 2. #include <stdio.h> 3. #include <iostream> 4. #include <stdlib.h> 5. **using** **namespace** std; 6. **namespace** recursion{ 7. **template** <**typename** T> 8. T\* binary\_search(T\* begin,T\* end,T tar,**bool**(\*cmp)(**const** T&,**const** T&)){ 9. //递归二分查找 10. //功能:对以cmp为排序方式的数组数组下标范围[begin,end)内二分查找tar 11. //如果查找到，返回一个T\*类型地址，如果未查找到，返回NULL 12. **if**(begin==end) **return** NULL; 13. T\* mid=begin+(end-begin)/2; 14. **if**(\*mid==tar) **return** mid; 15. **else** **if**(cmp(tar,\*mid)) **return** recursion::binary\_search(begin,mid,tar,cmp); 16. **else** **return** recursion::binary\_search(mid+1,end,tar,cmp); 17. } 18. }; 19. **namespace** non\_recursion{ 20. **template** <**typename** T> 21. T\* binary\_search(T\* begin,T\* end,T tar,**bool**(\*cmp)(**const** T&a,**const** T&b)){ 22. //非递归二分查找 23. T\* mid; 24. **while**(begin<end){ 25. mid=(end-begin)/2+begin; 26. **if**(\*mid==tar) **return** mid; 27. **else** **if**(cmp(tar,\*mid)) end=mid; 28. **else** begin=mid+1; 29. } 30. **return** NULL; 31. } 32. }; 33. **bool** cmp1(**const** **int**&a,**const** **int**&b){ 34. **return** a<b; 35. } 36. **bool** cmp2(**const** **int**& a,**const** **int**&b){ 37. **return** a>b; 38. } 39. #define N 100 40. **int** main(){ 41. **int** arr[N]; 42. **for**(**int** i=0;i<N;i++) arr[i]=i; 43. **int** target=20; 44. **int**\* pos1=recursion::binary\_search(arr,arr+N,target,cmp1); 45. **int**\* pos2=non\_recursion::binary\_search(arr,arr+N,target,cmp1); 46. printf("%d %d\n",pos1?pos1-arr:-1,pos2?pos2-arr:-1); 47. } 48. //2、二叉平衡树 49. #include <stdio.h> 50. #include <iostream> 51. #include <stdlib.h> 52. #include <time.h> 53. #include <unordered\_set> 54. #include <vector> 55. **using** **namespace** std; 57. **template** <**class** T> 58. **class** AVL{ 59. **private**: 60. **struct** node{ 61. T val; 62. node\* left; 63. node\* right; 64. **int** height; 65. node(){}; 66. node(T v):val(v),height(1),left(NULL),right(NULL){} 67. }; 68. node\* root; 69. **bool** (\*cmp)(T&,T&);//排序比较函数 70. **bool** (\*eq)(T&,T&);//相等比较函数 71. **int** capacity; 72. **bool** isdel; 73. /\* 74. 假如我们遍历到一个节点root，出现了第一次不平衡 75. 本质：本质height(root->left)-height(root->right)>=2 76. 1、LL类型不平衡： 77. height(root->left->left)>=height(root->left->right); 78. 2、LR类型不平衡 79. height(root->left->right)>=height(root->left->left); 80. 3、RR类型不平衡 81. height(root->right->right)>=height(root->right->left); 82. 4、RL类型不平衡 83. height(root->right->left)>=height(root->right->right); 84. \*/ 85. **void** LL(node\* &root){ 86. node\* temp=root->left; 87. root->left=temp->right; 88. temp->right=root; 89. root->height=max(height(root->left),height(root->right))+1; 90. root=temp; 91. root->height=max(height(root->left),height(root->right))+1; 92. } 93. **void** RR(node\*& root){ 94. node\* temp; 95. temp=root->right; 96. root->right=temp->left; 97. temp->left=root; 98. root->height=max(height(root->left),height(root->right))+1; 99. root=temp; 100. root->height=max(height(root->left),height(root->right))+1; 101. } 102. **void** LR(node\*& root){ 103. RR(root->left); 104. LL(root); 105. } 106. **void** RL(node\*& root){ 107. LL(root->right); 108. RR(root); 109. } 110. **int** height(node\*& root){ 111. **if**(!root) **return** 0; 112. **else** **return** root->height; 113. } 114. **void** adjust(node\*& root){ 115. //调整root节点 116. **int** dev=height(root->left)-height(root->right); 117. **if**(abs(dev)!=2) **return**; 118. **if**(dev==2){ 119. **if**(height(root->left->left)>=height(root->left->right)) LL(root); 120. **else** LR(root); 121. }**else**{ 122. **if**(height(root->right->right)>=height(root->right->left)) RR(root); 123. **else** RL(root); 124. } 125. } 126. **void** inner\_insert(node\* &root,T& val){ 127. //向root中插入val这个值 128. **if**(!root){ 129. //插入查询到空节点，表明原来的avl树中没有插入值val 130. root=**new** node(val); 131. **return**; 132. } 133. **if**(val==root->val){ 134. //存在该节点取消插入 135. capacity--; 136. **return**; 137. }**else** **if**(cmp(val,root->val)){ 138. //插到左子树 139. inner\_insert(root->left,val); 140. **if**(abs(height(root->left)-height(root->right))==2){ 141. //不平衡 142. /\* 143. 方法一进行LL，LR调整 144. if(cmp(val,root->left->val)){ 145. //插入到root的左子树的左子树导致不平衡 146. //进行LL调整 147. LL(root); 148. }else{ 149. //插入到root的左子树的右子树导致不平衡 150. //进行LR调整 151. //先对root->left进行RR调整 152. //再对root进行LL调整 153. LR(root); 154. } 155. \*/ 156. //方法二 157. adjust(root); 158. } 159. }**else**{ 160. //插到右子树 161. inner\_insert(root->right,val); 162. **if**(abs(height(root->right)-height(root->left))==2){ 163. //不平衡 164. /\* 165. 方法一进行LL，LR调整 166. if(cmp(root->right->val,val)){ 167. //插入到root的右子树的右子树导致不平衡 168. //进行RR调整 169. RR(root); 170. }else{ 171. //插入到root的右子树的左子树导致不平衡 172. //进行RL调整 173. //先对root->right进行LL调整 174. //再对root进行RR调整 175. RL(root); 176. } 177. \*/ 179. //方法二 180. adjust(root); 181. } 183. } 184. root->height=max(height(root->left),height(root->right))+1; 185. //更新root的高度 186. } 187. **bool** inner\_find(node\* root,T& val){ 188. //查询root为根的树中是否包含值为val的节点 189. **if**(!root) **return** **false**; 190. **if**(root->val==val) **return** **true**; 191. **else** **if**(cmp(val,root->val)) **return** inner\_find(root->left,val); 192. **else** **return** inner\_find(root->right,val); 193. } 194. **void** inner\_mid\_traverse(node\* root,vector<T>& ans,**int**&cnt){ 195. **if**(!root) **return**; 196. inner\_mid\_traverse(root->left,ans,cnt); 197. ans[cnt++]=root->val; 198. inner\_mid\_traverse(root->right,ans,cnt); 199. } 200. node\* find\_max(node\*& root){ 201. //找到以root为跟子树中的最大值的节点 202. **if**(!root || !root->right){ 203. **return** root; 204. }**else**{ 205. **return** find\_max(root->right); 206. } 207. } 208. node\* find\_min(node\*& root){ 209. //找到以root为跟子树中的最小值的节点 210. **if**(!root || !root->left){ 211. **return** root; 212. }**else**{ 213. **return** find\_min(root->left); 214. } 215. } 216. **void** inner\_erase(node\*& root,T& val){ 217. //在root中查找删除val 218. **if**(!root) **return**; 219. **if**(root->val==val){ 220. //查找到了要删除的值 221. isdel=**true**; 222. **if**(root->left && root->right){ 223. //左右子树均为非空 224. //将root->val的值与root->left为根的子树中的最大值进行替换 225. //然后再递归进行删除 226. //这样就能保证我们最后删除的值一定在叶子结点上 227. node\*temp=find\_max(root->left); 228. swap(temp->val,root->val); 229. inner\_erase(root->left,val); 230. }**else**{ 231. //左右子树至少有一个为空树 232. node\* temp=root; 233. root=root->left?root->left:root->right; 234. **delete** temp; 235. } 236. }**else** **if**(cmp(val,root->val)){ 237. inner\_erase(root->left,val); 238. }**else**{ 239. inner\_erase(root->right,val); 240. } 241. **if**(root){ 242. adjust(root); 243. root->height=max(height(root->left),height(root->right))+1; 244. } 245. } 246. **int** is\_avl\_tree(node\*root,**bool** &good){ 247. //判断是否是一颗高度平衡的树 248. **if**(!root) **return** 0; 249. **int** left=0,right=0; 250. **if**(good) left=is\_avl\_tree(root->left); 251. **if**(good) right=is\_avl\_tree(root->right); 252. **if**(abs(left-right)>1) good=**false**; 253. **return** max(left,right)+1; 254. } 255. **public**: 256. AVL(**bool** (\*cmp)(T&,T&), **bool** (\*eq)(T&,T&)){ 257. //构造函数2 258. root=NULL; 259. **this**->eq=eq; 260. **this**->cmp=cmp; 261. capacity=0; 262. } 263. AVL(**bool** (\*cmp)(T&,T&)){ 264. //构造函数2 265. root=NULL; 266. **this**->cmp=cmp; 267. capacity=0; 268. } 269. **void** insert(T val){ 270. //插入val这个值 271. capacity++; 272. inner\_insert(root,val); 273. } 274. **void** erase(T val){ 275. //删除val这个节点 276. isdel=**false**; 277. inner\_erase(root,val); 278. **if**(isdel) capacity--; 279. } 280. **bool** find(T val){ 281. //查询是否有val这个节点 282. **return** inner\_find(root,val); 283. } 284. **int** size(){ 285. //返回avl树中的节点数 286. **return** capacity; 287. } 288. **int** height(){ 289. //返回avl树的高度 290. **return** height(root); 291. } 292. **void** mid\_traverse(vector<**int**>& ans){ 293. //返回中序遍历 294. ans.resize(capacity); 295. **int** cnt=0; 296. inner\_mid\_traverse(root,ans,cnt); 297. } 298. **bool** is\_avl\_tree(){ 299. //是否是一颗高度平衡的树 300. **bool** good=**true**; 301. is\_avl\_tree(root,good); 302. **return** good; 303. } 304. };   308. **bool** cmp(**int**& a,**int** &b){ 309. **return** a<b; 310. } 311. **bool** is\_ok(vector<**int**>& num){ 312. //检查中序遍历数组是否为cmp规定的比较规则 313. **for**(**int** i=1,n=num.size();i<n;i++){ 314. **if**(!cmp(num[i-1],num[i])) **return** **false**; 315. } 316. **return** **true**; 317. } 318. **void** test1(){ 319. //该例子只简介int类型的用法 320. AVL<**int**> avl(cmp); 321. srand(time(0)); 322. **int** n=100000,dev=10000; 323. unordered\_set<**int**> have; 324. **int** del\_cnt=0; 325. **for**(**int** i=0;i<n;i++){ 326. **int** v=rand()%dev; 327. have.insert(v); 328. avl.insert(v); 329. v=rand()%dev; 330. avl.erase(v); 331. **if**(have.find(v)!=have.end()) have.erase(v),del\_cnt++; 332. } 333. vector<**int**> ans; 334. avl.mid\_traverse(ans); 335. cout<<"###after manipulate###"<<endl; 336. cout<<"avl.size() is: "<<avl.size()<<" avl.height()is: "<<avl.height()<<" unordered\_set<int>'s size() is: "<<have.size()<<endl; 337. cout<<"In this test totally delete "<<del\_cnt<<" times"<<endl; 338. **if**(is\_ok(ans)) cout<<"it is a avl(bst) tree"<<endl; 339. **else** cout<<"it is not a avl(bst) tree"<<endl; 340. } 342. **void** introduction(){ 343. /\* 344. 用法简介 345. 构造一颗中序遍历序列为cmp函数规定的比较规则的AVL 346. 1、初始化该类型，必须传入一个比较函数 347. 2、你的类型T为自定义类型，你必须还要重载运算符== 349. AVL<T> avl(cmp); 350. bool cmp(T& a,T&b){ 351. //write your compare rule// 352. } 353. 可用函数 354. avl.size() 大小 355. avl.height() 树高 356. vector<int> ans=avl.mid\_traverse(); 357. avl.find(v) 查询是否有v这个值 358. avl.insert(v) 插入v 359. avl.erase(v) 删除v 360. \*/ 361. } 362. **int** main(){ 363. test1(); 364. **return** 0; 365. } 366. //3、hash表 367. #include <stdio.h> 368. #include <iostream> 369. #include <stdlib.h> 370. **using** **namespace** std; 371. **template** <**class** T,**class** V> 372. **class** myhash{ 373. **public**: 374. **struct** node{ 375. T first; 376. V second; 377. node\* next; 378. node(T& a,V& b):first(a),second(b),next(NULL){} 379. }; 380. **private**: 381. **int**(\*hash\_mapping)(T&); 382. **int** n,cap; 383. node\*\* head; 384. pair<node\*,node\*> find\_inner(T& key){ 385. **int** index=hash\_mapping(key)%n; 386. node\* cur=head[index]; 387. node\* last=NULL; 388. **bool** vis=**false**; 389. **while**(cur && !vis){ 390. **if**(cur->first==key){ 391. vis=**true**; 392. **break**; 393. } 394. last=cur; 395. cur=cur->next; 396. } 397. **if**(vis) **return** {last,cur}; 398. **else** **return** {NULL,NULL}; 399. } 400. **void** initialize(){ 401. cap=0; 402. head=**new** node\*[n]; 403. **for**(**int** i=0;i<n;i++) head[i]=NULL; 404. } 405. **public**: 407. myhash(**int**(\*hash\_mapping)(T&)){ 408. n=10000; 409. initialize(); 410. **this**->hash\_mapping=hash\_mapping; 411. } 412. myhash(**int**(\*hash\_mapping)(T&),**int** cap){ 413. n=cap; 414. initialize(); 415. **this**->hash\_mapping=hash\_mapping; 416. } 417. **void** insert(T key,V val){ 418. **int** index=hash\_mapping(key)%n; 419. node\* cur=head[index]; 420. **bool** vis=**false**; 421. **while**(cur && !vis){ 422. **if**(cur->first==key){ 423. vis=**true**; 424. **break**; 425. } 426. cur=cur->next; 427. } 428. **if**(!vis){ 429. node\* temp=**new** node(key,val); 430. temp->next=head[index]; 431. head[index]=temp; 432. cap++;//大小加一 433. }**else**{ 434. cur->second=val; 435. } 436. } 437. **void** erase(T key){ 438. **int** index=hash\_mapping(key)%n; 439. pair<node\*,node\*> temp=find\_inner(key); 440. **if**(temp.second && temp.first){ 441. temp.first=temp.second->next; 442. **delete** temp.second; 443. cap--;//大小减1 444. }**else** **if**(temp.second){ 445. head[index]=temp.second->next; 446. **delete** temp.second; 447. cap--;//大小减1 448. } 449. } 450. node\* find(T key){ 451. **return** find\_inner(key).second; 452. } 453. V& operator[](T key){ 454. V val; 455. **if**(find(key)==NULL) insert(key,val); 456. **return** find(key)->second; 457. } 458. **int** size(){ 459. **return** cap; 460. } 461. node\* end(){ 462. **return** NULL; 463. } 464. pair<node\*\*,**int**> begin(){ 465. **return** {head,n}; 466. } 468. **public**: 469. **class** iterator{ 470. **public**: 471. node\*\* arr; 472. node\* ptr; 473. **int** n,cur; 474. iterator(); 475. **bool** operator==(iterator& a); 476. **bool** operator!=(iterator& a); 477. **bool** operator!=(node\* a); 478. **void** forward(); 479. **void** operator++(); 480. **void** operator++(**int**); 481. iterator& operator=(pair<node\*\*,**int**> temp); 482. iterator& operator=(iterator& temp); 483. node\* operator->(); 485. }; 486. }; 487. **template** <**class** T,**class** V> 488. **bool** myhash<T,V>::iterator::operator==(myhash<T,V>::iterator& a){ 489. **return** ptr==a.ptr; 490. } 491. **template** <**class** T,**class** V> 492. **bool** myhash<T,V>::iterator::operator!=(myhash<T,V>::iterator& a){ 493. **return** ptr!=a.ptr; 494. } 495. **template** <**class** T,**class** V> 496. **bool** myhash<T,V>::iterator::operator!=(node\* a){ 497. **return** ptr!=a; 498. } 499. **template** <**class** T,**class** V> 500. **void** myhash<T,V>::iterator::forward(){ 501. **if**(ptr && ptr->next){ 502. ptr=ptr->next; 503. **return**; 504. } 505. ptr=NULL; 506. **while**(cur!=n && ptr==NULL){ 507. ptr=arr[cur++]; 508. } 509. } 511. **template** <**class** T,**class** V> 512. **void** myhash<T,V>::iterator::operator++(){ 513. forward(); 514. } 515. **template** <**class** T,**class** V> 516. **void** myhash<T,V>::iterator::operator++(**int**){ 517. forward(); 518. } 520. **template** <**class** T,**class** V> 521. **typename** myhash<T,V>::iterator& myhash<T,V>::iterator::operator=(pair< myhash<T,V>::node \*\*,**int**> temp){ 522. **this**->arr=temp.first; 523. **this**->n=temp.second; 524. **this**->cur=0; 525. **this**->ptr=NULL; 526. forward(); 527. **return** \***this**; 528. } 529. **template** <**class** T,**class** V> 530. **typename** myhash<T,V>::iterator& myhash<T,V>::iterator::operator=(myhash<T,V>::iterator& temp){ 531. memcpy(**this**,&temp,**sizeof**(temp)); 532. **return** \***this**; 533. } 534. **template** <**class** T,**class** V> 535. myhash<T,V>::iterator::iterator(){ 536. arr=NULL,ptr=NULL; 537. n=0,cur=0; 538. } 539. **template** <**class** T,**class** V> 540. **typename** myhash<T,V>::node\* myhash<T,V>::iterator::operator->(){ 541. **return** **this**->ptr; 542. }   546. **struct** infor{ 547. **char** name[21]; 548. **char** phone[12]; 549. **char** add[51]; 550. }arr[10000]; 551. **int** n; 552. **int** mapping\_phone(**char**\*& phone){ 553. //hash映射函数 554. **int** ans=0; 555. **int** dev=1e9+7; 556. **for**(**int** i=0;phone[i];i++){ 557. ans=(ans\*10+phone[i]-'0')%dev; 558. } 559. **return** ans; 560. } 561. **void** test1(){ 562. //以电话号码作为键 563. myhash<**char**\*,pair<**char**\*,**char**\*>> dp(mapping\_phone); 564. **for**(**int** i=0;i<n;i++){ 565. dp[(**char**\*)arr[i].phone]={(**char**\*)arr[i].add,(**char**\*)arr[i].name}; 566. } 567. myhash<**char**\*,pair<**char**\*,**char**\*>>::iterator it; 568. **for**(it=dp.begin();it!=dp.end();it++){ 569. printf("phone num:%s address:%s name:%s\n",it->first,it->second.first,it->second.second); 570. } 571. dp.erase((**char**\*)arr[0].phone); 572. printf("after delete one items\n"); 573. **for**(it=dp.begin();it!=dp.end();it++){ 574. printf("phone num:%s address:%s name:%s\n",it->first,it->second.first,it->second.second); 575. } 576. } 577. **void** input(){ 578. scanf("%d",&n); 579. getchar(); 580. **for**(**int** i=0;i<n;i++){ 581. scanf("%s %s %s",arr[i].name,arr[i].phone,arr[i].add); 582. } 583. /\* 584. 3 585. sanglonglong 12345 xidian 586. wangyifa 110 xidainnan 587. lixiaofei 120 xidianbei 588. \*/ 589. } 590. **int** main(){ 591. input(); 592. test1(); 593. } | | |