fantasy19 / Derp

Code

Issues

Pull requests

Actions

Projects


Security


Insights


 e681c03cbb ▾



Derp / 280 / assignment04-BinaryTree / assignment04-BinaryTree / BSTree.cpp




fantasy19 No commit message



 1 contributor

Raw

Blame



653 lines (523 sloc) | 15.3 KB

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

```
1  /*****
2  /*!
3  \file   BSTree.cpp
4  \author Ang Cheng Yong
5  \par    email: a.chengyong\@digipen.edu
6  \par    DigiPen login: a.chengyong
7  \par    Course: CS280
8  \par    Programming Assignment #4
9  \date   8/11/2016
10 \brief
11 This file contains the implementation for BST.
12 */
13 /*****
14
15 /*****
16 /*!
17 \fn template <typename T> BSTree<T>::BSTree(ObjectAllocator *OA, bool ShareOA)
18 \brief
19 Constructor of the BSTree
20
21 \param OA
22 object allocator for the tree's nodes
23
24 \param ShareOA
25 boolean for sharing object allcoator among trees
26
27 \return
28 none
29
30 */
```

```

31  /*****
32
33  template <typename T>
34  BSTree<T>::BSTree(ObjectAllocator *OA, bool ShareOA) : oa(OA), share(ShareOA), root_(0){
35
36      try {
37          if (!share)
38              oa = new ObjectAllocator(sizeof(BinTreeNode), OAConfig());
39      }
40      catch (const OAException &e) {
41          throw(BSTException(BSTException::E_NO_MEMORY, e.what()));
42      }
43
44  }
45
46  /*****
47  /*!
48  \fn template <typename T> BSTree<T>::BSTree(const BSTree& rhs)
49  \brief
50  Copy constructor of BSTree
51
52  \param rhs
53  the BSTree that is to be copied
54
55  \return
56  none
57  */
58  /*****
59
60  template <typename T>
61  BSTree<T>::BSTree(const BSTree& rhs) {
62
63      if (rhs.oa){ // share object allocator if available
64          oa = rhs.oa;
65          share = true;
66      }
67      else { // create own if not shared
68          OAConfig config(true);
69          oa = new ObjectAllocator(sizeof(BinTreeNode), config);
70          share = false;
71      }
72
73      if (rhs.root()){
74          BinTree tmp = make_node(rhs.root()->data);
75          tmp->count = rhs.root()->count;
76          copy_tree(tmp, rhs.root());
77          root_ = tmp;;
78      }
79
80  }
81
82  /*****

```

```

83  /*!
84  \fn template <typename T> BSTree<T>::~~BSTree()
85  \brief
86  Destructor of BSTree
87  \return
88  none
89  */
90  /*****
91
92  template <typename T>
93  BSTree<T>::~~BSTree() {
94      clear();
95      if (!share)
96          delete oa;
97  }
98
99
100 /*****
101 /*!
102 \fn template <typename T> BSTree<T>& BSTree<T>::operator=(const BSTree& rhs)
103 \brief
104 assignment operator of BSTree
105
106 \param rhs
107 the BSTree that is to be based on
108
109 \return
110 The assigned tree itself
111 */
112 /*****
113
114 template <typename T>
115 BSTree<T>& BSTree<T>::operator=(const BSTree& rhs) {
116
117     if (this != &rhs) {
118         if (rhs.share) { // share object allocator if available
119             oa = rhs.oa;
120             share = true;
121         }
122         else { // create own if not shared
123             OAConfig config(true);
124             oa = new ObjectAllocator(sizeof(BinTreeNode), config);
125             share = false;
126         }
127
128         if (rhs.root()) {
129             BinTree tmp = make_node(rhs.root()->data);
130             tmp->count = rhs.root()->count;
131
132             copy_tree(tmp, rhs.root());
133             clear();
134             root_ = tmp;;
135         }
136     }

```

```
135     }
136     return *this;
137 }
138
139 /*****
140  *!
141  \fn template <typename T> void BSTree<T>::clear(void)
142  \brief
143  clear the tree's nodes
144
145  \return
146  none
147  */
148 /*****/
149
150 template<typename T>
151 void BSTree<T>::clear(void){
152     free_tree(root_);
153 }
154
155 /*****
156  *!
157  \fn template <typename T> void BSTree<T>::free_tree(BinTree& tree)
158  \brief
159  helper function to clear tree of nodes by recursion
160
161  \param tree
162  the tree to clear
163
164  \return
165  none
166  */
167 /*****/
168
169 template<typename T>
170 void BSTree<T>::free_tree(BinTree& tree){
171
172     if (!tree)
173         return;
174
175     free_tree(tree->left);
176     free_tree(tree->right);
177
178     delete_node(tree, tree->data);
179 }
180
181 /*****
182  *!
183  \fn template <typename T> void BSTree<T>::copy_tree(BinTree &lhs, BinTree rhs)throw(BSTException)
184  \brief
185  helper function to construct tree based on another tree by recursion.
186
```

```

187 \param lhs
188 input tree to copy data on
189
190 \param rhs
191 input tree to copy data off
192
193 \return
194 none
195 */
196 /*****
197
198 template<typename T>
199 void BSTree<T>::copy_tree(BinTree &lhs, BinTree rhs) throw(BSTException){
200     try{
201
202         if (rhs){
203             // fill data and count of both nodes
204             if (rhs->left){
205                 lhs->left = make_node(rhs->left->data);
206                 lhs->left->count = rhs->left->count;
207                 copy_tree(lhs->left, rhs->left); // before copying recursively
208             }
209
210             if (rhs->right){
211                 lhs->right = make_node(rhs->right->data);
212                 lhs->right->count = rhs->right->count;
213                 copy_tree(lhs->right, rhs->right);
214             }
215         }
216     }
217     catch (const OException &e)
218     {
219         throw(BSTException(BSTException::E_NO_MEMORY, e.what()));
220     }
221 }
222
223 /****
224 /*!
225 \fn template <typename T> void BSTree<T>::remove(const T& value)
226 \brief
227 remove a node based on input
228
229 \param value
230 the value to match for removal of node
231
232 \return
233 none
234 */
235 /****
236
237 template <typename T>
238 void BSTree<T>::remove(const T& value) {

```

```
239     delete_node(root_, value);
240 }
241
242 /*****
243  *!
244  \fn template <typename T> void BSTree<T>::delete_node(BinTree & Tree, const T& value)
245  \brief
246  helper function to search for the correct node to delete by recursion.
247
248  \param Tree
249  the tree to remove node from
250
251  \param value
252  the value to match
253
254  \return
255  none
256  */
257 /*****
258
259  template<typename T>
260  void BSTree<T>::delete_node(BinTree & Tree, const T& value){
261
262      if (Tree == 0)
263          return;
264      else if (value < Tree->data){
265          --Tree->count;
266          delete_node(Tree->left, value);
267      }
268      else if (value > Tree->data){
269          --Tree->count;
270          delete_node(Tree->right, value);
271      }
272      else{ // match found
273          --Tree->count;
274
275          if (Tree->left == 0){
276              BinTree temp = Tree;
277              Tree = Tree->right;
278              free_node(temp);
279          }
280          else if (Tree->right == 0){
281              BinTree temp = Tree;
282              Tree = Tree->left;
283              free_node(temp);
284          }
285          else { // if node has 2 children
286              BinTree pred = 0;
287              find_predecessor(Tree, pred);
288              Tree->data = pred->data;
289              delete_node(Tree->left, Tree->data);
290          }
```

```

291     }
292 }
293
294 /*****
295  *!
296  \fn template <typename T> const typename BSTree<T>::BinTreeNode* BSTree<T>::operator[](int index) const
297  \brief
298  subscript operator to search in the tree
299
300  \param index
301  the value to match
302
303  \return
304  the matching node
305  */
306 /*****/
307
308 template <typename T>
309 const typename BSTree<T>::BinTreeNode* BSTree<T>::operator[](int index) const {
310
311     return sub_node(root_, index);
312 }
313
314 /*****
315  *!
316  \fn template <typename T> const typename BSTree<T>::BinTreeNode* BSTree<T>::sub_node(BinTreeNode* tree, int compares) const
317  \brief
318  helper function to find the correct node by recursion.
319
320  \param tree
321  the tree to find the match
322
323  \param compares
324  the value to match
325
326  \return
327  the matching node
328  */
329 /*****/
330
331 template<typename T>
332 const typename BSTree<T>::BinTreeNode* BSTree<T>::sub_node(BinTreeNode* tree, int compares) const {
333     if (!tree)
334         return NULL;
335
336     unsigned tmp = (tree->left) ? tree->left->count : 0;
337
338     if (tmp > static_cast<unsigned>(compares))
339         return sub_node(tree->left, compares);
340     else if (tmp < static_cast<unsigned>(compares))
341         return sub_node(tree->right, compares - tmp - 1);
342     else

```

```
343         return tree;
344
345     }
346
347     /*****
348     /*!
349     \fn template <typename T> void BSTree<T>::insert(const T& value) throw(BSTException)
350     \brief
351     insert a node into the tree
352
353     \param value
354     the value of node to be inserted
355
356     \return
357     none
358     */
359     /*****/
360
361     template <typename T>
362     void BSTree<T>::insert(const T& value) throw(BSTException) {
363         insert_node(root_, value);
364     }
365
366     /*****
367     /*!
368     \fn void BSTree<T>::insert_node(BinTree & tree, const T& value) throw(BSTException)
369     \brief
370     helper function insert a node into the tree by recursion
371
372     \param tree
373     the tree for node to be inserted
374
375     \param value
376     the value of node to be inserted
377
378     \return
379     none
380     */
381     /*****/
382
383     template<typename T>
384     void BSTree<T>::insert_node(BinTree & tree, const T& value) throw(BSTException){
385         try{
386             if (!tree){
387                 tree = make_node(value);
388                 ++tree->count;
389             }
390             else if (value < tree->data){
391                 ++tree->count;
392                 insert_node(tree->left, value);
393             }
394             else if (value > tree->data){
```



```

395         ++tree->count;
396         insert_node(tree->right, value);
397     }
398     else{
399         std::cout << "Error, duplicate item" << std::endl;
400     }
401 }
402 catch (const OAEException &e){
403     throw(BSTException(BSTException::E_NO_MEMORY, e.what()));
404 }
405 }
406
407 /*****
408  *!
409  \fn template <typename T> bool BSTree<T>::find(const T& value, unsigned &compares) const
410  \brief
411  find a node in the tree with a matching value
412
413  \param value
414  the value to be matched
415
416  \param compares
417  number of function calls used to find the matching node
418
419  \return
420  whether a node with the matching value exist
421  */
422 /*****
423
424  template <typename T>
425  bool BSTree<T>::find(const T& value, unsigned &compares) const {
426
427      return find_node(root_, value, compares);
428
429  }
430
431 /*****
432  *!
433  \fn template <typename T> bool BSTree<T>::find_node(BinTree tree, const T& value, unsigned &compares) const
434  \brief
435  helper function find a node in the tree with a matching value by recursion
436
437  \param tree
438  the tree to be searched
439
440  \param value
441  the value to be matched
442
443  \param compares
444  number of function calls used to find the matching node
445
446  \return

```

```

447 whether a node with the matching value exist
448 */
449 /*****
450
451 template<typename T>
452 bool BSTree<T>::find_node(BinTree tree, const T& value, unsigned &compares) const
453 {
454     ++compares;
455
456     if (tree == 0)
457         return false;
458     else if (value == tree->data) // match fit
459         return true;
460     else if (value < tree->data) // continue finding
461         return find_node(tree->left, value, compares);
462     else
463         return find_node(tree->right, value, compares);
464
465 }
466
467 /*****
468 /*!
469 \fn template <typename T> int BSTree<T>::height(void) const
470 \brief
471 count the height of tree
472
473 \return
474 height of tree
475 */
476 /*****
477
478 template <typename T>
479 int BSTree<T>::height(void) const {
480     return tree_height(root_);
481 }
482
483 /*****
484 /*!
485 \fn template <typename T> int BSTree<T>::tree_height(BinTree tree) const
486 \brief
487 helper function find the height of a tree by recursion
488
489 \param tree
490 the tree to be counted
491
492 \return
493 height of tree
494 */
495 /*****
496
497 template<typename T>
498 int BSTree<T>::tree_height(BinTree tree) const

```

```
499 {
500
501     if (!tree)
502         return -1;
503     //recursively calculate height of subsequent subtrees
504     int L = tree_height(tree->left);
505     int R = tree_height(tree->right);
506
507     if (L > R)
508         return L + 1;
509     else
510         return R + 1;
511 }
512
513 /*****
514  *!
515  \fn template <typename T> typename BSTree<T>::BinTree BSTree<T>::root(void) const
516  \brief
517  get the root of the tree
518
519  \return
520  root of tree
521  */
522 /*****
523
524  template <typename T>
525  typename BSTree<T>::BinTree BSTree<T>::root(void) const {
526      return root_;
527  }
528
529 /*****
530  *!
531  \fn template <typename T> bool BSTree<T>::empty(void) const
532  \brief
533  whether a tree is empty
534
535  \return
536  root of tree
537  */
538 /*****
539
540  template <typename T>
541  bool BSTree<T>::empty(void) const {
542      return (root_ == 0);
543  }
544
545 /*****
546  *!
547  \fn template <typename T> bool BSTree<T>::empty(void) const
548  \brief
549  whether a tree is empty
550
```

```

551 \return
552 root of tree
553 */
554 /*****
555
556 template <typename T>
557 unsigned int BSTree<T>::size(void) const {
558     return (root_) ? root_->count : 0;
559 }
560
561 /*****
562 /*!
563 \fn typename BSTree<T>::BinTree& BSTree<T>::get_root(void)
564 \brief
565 get the root of the tree
566
567 \return
568 root of tree
569 */
570 /*****
571
572 template <typename T>
573 typename BSTree<T>::BinTree& BSTree<T>::get_root(void) {
574     return root_;
575 }
576
577 /*****
578 /*!
579 \fn template <typename T> typename BSTree<T>::BinTree BSTree<T>::make_node(const T& value)
580 \brief
581 make a new node for the tree
582
583 \param value
584 the value for new node
585
586 \return
587 the new node
588 */
589 /*****
590
591 template <typename T>
592 typename BSTree<T>::BinTree BSTree<T>::make_node(const T& value) {
593
594     try{
595         //placement new through using object allcoator
596         BinTree mem = reinterpret_cast<BinTreeNode*>(oa->Allocate());
597         BinTree node = new (mem) BinTreeNode(value);
598         return node;
599     }
600     catch (const OAEException &e){
601         throw(BSTException(BSTException::E_NO_MEMORY, e.what()));
602     }

```

```
603
604 }
605
606 /*****
607  *!
608  \fn template <typename T> void BSTree<T>::free_node(BinTree node)
609  \brief
610  free a node from the tree
611
612  \param node
613  the node to free
614
615  \return
616  none
617  */
618 /*****/
619
620 template <typename T>
621 void BSTree<T>::free_node(BinTree node) {
622     node->~BinTreeNode();
623     oa->Free(node);
624 }
625
626 /*****
627  *!
628  \fn template <typename T> void BSTree<T>::find_predecessor(BinTree tree, BinTree &predecessor)
629  \brief
630  find the parent of a node
631
632  \param tree
633  the node to be searched
634
635  \param predecessor
636  the node to fill as predecessor
637
638  \return
639  none
640  */
641 /*****/
642
643 template <typename T>
644 void BSTree<T>::find_predecessor(BinTree tree, BinTree &predecessor) const {
645     predecessor = tree->left;
646     while (predecessor->right != 0)
647         predecessor = predecessor->right;
648 }
649
650
651
652
653
```