

This repository has been archived by the owner. It is now read-only.

 HBreithaupt / DigiPenCode

Archived

Code

Issues

Pull requests

Actions

Projects

Security

Insights

 master ▾

...

DigiPenCode / CS280 / Assignment4 / BSTree.cpp



HBreithaupt CS280



 0 contributors

Raw

Blame



712 lines (568 sloc) | 15.5 KB

```
1  /*****/
2  /*!
3  \file BSTree.cpp
4  \author      Haven Breithaupt
5  \par DP email: h.breithaupt\@digipen.edu
6  \par Course: CS280
7  \par Assignment 4
8  \date 10/31/15
9
10 \brief
11  Implementation of binary search tree.
12
13
14  Hours spent on assignment: 2 (on this file)
15
16  Specific portions that gave you the most trouble:
17  - Not much here, mostly copy paste from the web site. Assignment and copy
18  probably the most difficult.
19  */
20 /*****/
21
22
23 /*****/
24 /*!
25
26 \brief
27  Constructor for BST
28
```

```
29 \param OA
30 ObjectAllocator to use with the tree (either provided or created).
31
32 \param ShareOA
33 Flag to indicate sharing of the object allocator with copies
34 of the object.
35
36 */
37 /*****
38 template <typename T>
39 BSTree<T>::BSTree(ObjectAllocator *OA, bool ShareOA) : Root(0), Height(-1), NumNodes(0)
40 {
41     // if allocator provided use it
42     // else make our own
43     if(OA)
44     {
45         // flag that we do NOT own the allocator
46         // and will not delete it (client will handle that)
47         OwnOA = false;
48         allocator = OA;
49     }
50     else
51     {
52         // make our own allocator
53         OAConfig config(true);
54         allocator = new ObjectAllocator(sizeof(BinTreeNode), config);
55         OwnOA = true;
56     }
57
58     // flag whether or not to share this allocator with copies of this object
59     if(ShareOA)
60         ShareAlloc = true;
61     else
62         ShareAlloc = false;
63 }
64
65 /*****
66 /*!
67
68 \brief
69 Copy constructor
70
71 \param rhs
72 Object being copied
73
74 */
75 /*****
76 template <typename T>
77 BSTree<T>::BSTree(const BSTree& rhs)
78 {
79     // if rhs is sharing their allocator, use it
80     if(rhs.ShareAlloc)
```

```

81     {
82         allocator = rhs.allocator;
83
84         // we dont own the allocator
85         OwnOA = false;
86
87         // continue sharing the allocator if more copies requested
88         ShareAlloc = true;
89     }
90     else // make our own allocatr
91     {
92         OAConfig config(true);
93         allocator = new ObjectAllocator(sizeof(BinTreeNode), config);
94
95         // we own this, are responsible for deleting it
96         OwnOA = true;
97
98         // do not share this allocator
99         ShareAlloc = false;
100    }
101
102    // call helper function to copy rhs tree
103    CopyHelper(Root, rhs.Root);
104
105    // copy stats
106    NumNodes = rhs.NumNodes;
107    Height = rhs.Height;
108 }
109
110 /*****
111  *!
112
113  \brief
114  Destructor
115
116  */
117 /*****
118  template <typename T>
119  BSTree<T>::~~BSTree()
120  {
121      // call clear to destroy the tree
122      clear();
123
124
125      // if we own the allocator, delete it
126      if(OwnOA)
127          delete allocator;
128
129  }
130
131 /*****
132  *!

```

```
133
134 \brief
135 Assignment operator.
136
137 \param rhs
138 Object being assigned from.
139
140 \return
141 Reference to lhs object.
142 */
143 /*****
144 template <typename T>
145 BSTree<T>& BSTree<T>::operator=(const BSTree& rhs)
146 {
147     // self assignment check
148     if(this == &rhs)
149         return *this;
150
151     // use rhs allocator if needed
152     if(rhs.ShareAlloc)
153     {
154         // however if we own our own allocate
155         // delete it first
156         if(OwnOA)
157         {
158             // delete underlying nodes first
159             clear();
160
161             // delete the allocator
162             delete allocator;
163         }
164
165         // use rhs allocator and set flags
166         allocator = rhs.allocator;
167         OwnOA = false;
168         ShareAlloc = true;
169     }
170
171     // clear out the left list
172     clear();
173
174     // copy the right hand list
175     CopyHelper(Root, rhs.Root);
176
177     // copy stats
178     NumNodes = rhs.NumNodes;
179     Height = rhs.Height;
180
181     return *this;
182 }
183
184 /*****/
```

```
185  /*!
186
187  \brief
188    Indexing operator (not implemented).
189
190  \param index
191    Node requested from the tree
192
193  \return
194    Pointer to node requested
195  */
196  /*****
197  template <typename T>
198  const typename BSTree<T>::BinTreeNode* BSTree<T>::operator[](int index) const
199  {
200    // not implemented, DO NOT USE
201
202    // 'use' index
203    if(index)
204        return 0;
205    else
206        return 0;
207  }
208  *****/
209  /*****
210  /*!
211
212  \brief
213    Public insert function. Redirects to private recursive function
214    to handle insertion.
215
216  \param value
217    Value being inserted.
218
219  */
220  /*****
221  template <typename T>
222  void BSTree<T>::insert(const T& value)
223  {
224      try
225      {
226          // call recursive function to insert
227          // catch exception from InsertItem and throw back to client
228          InsertItem(Root, value, 0);
229      }
230      catch(const BSTException &e)
231      {
232          throw ;
233      }
234  }
235
236  *****/
```

```

237  /*!
238
239  \brief
240    Public removal function. Redirects to private recursive function.
241
242  \param value
243    Value being removed from the tree.
244
245  */
246  /*****
247  template <typename T>
248  void BSTree<T>::remove(const T& value)
249  {
250      // call private delete to recursively delete this value
251      // not entirely sure how value lost it's constness
252      // tried calling it by value of the parameter given
253      // and ocmplier complained about lost qualifiers
254      DeleteItem(Root, const_cast<T&>(value));
255
256      // recalculate the height of the tree
257      Height = tree_height(Root);
258  }
259
260  *****/
261  /*!
262
263  \brief
264    Removes all nodes from the tree. Calls recursive function
265
266  */
267  /*****
268  template <typename T>
269  void BSTree<T>::clear(void)
270  {
271      // if there are nodes in the tree
272      // call ClearRec to delete them
273      if(Root)
274      {
275          // delete all the nodes in the tree
276          ClearRec(Root);
277
278          // reset counters and pointer
279          Root = 0;
280          Height = -1;
281          NumNodes = 0;
282      }
283  }
284
285  *****/
286  /*!
287
288  \brief

```

```

289     Recursive functio to delete all nodes in the tree.
290
291     \param tree
292         node being removed
293
294     */
295     /*****
296     template <typename T>
297     void BSTree<T>::ClearRec(BinTree tree)
298     {
299         // delete all nodes in the tree using post order traversal
300         if(!tree)
301             return;
302
303         ClearRec(tree->left);
304         ClearRec(tree->right);
305
306         FreeNode(tree);
307     }
308
309     /*****
310     /*!
311
312     \brief
313         Finds an item in the tree. Calls recursive function to do the work.
314
315     \param value
316         Value looking for.
317
318     \param compares
319         How many comparisons it took to find (or not) the item.
320
321     \return
322         True if the item was found. False if it was node
323     */
324     /*****
325     template <typename T>
326     bool BSTree<T>::find(const T& value, unsigned &compares) const
327     {
328         // call recursive function to find our value
329         return FindItem(Root, value, compares);
330     }
331
332     /*****
333     /*!
334
335     \brief
336         Checks if the tree is empty.
337
338     \return
339         True if it's empty, false if there are nodes in it.
340     */

```

```
341  /*****
342  template <typename T>
343  bool BSTree<T>::empty(void) const
344  {
345      // if height is -1 there are no nodes in the tree,
346      // therefore empty
347      if(Height == -1)
348          return true;
349      else
350          return false;
351  }
352
353  /*****
354  /*!
355
356  \brief
357      Returns number of nodes in the tree.
358
359
360  \return
361      Number of nodes in the tree.
362  */
363  /*****
364  template <typename T>
365  unsigned int BSTree<T>::size(void) const
366  {
367      // return count member of the head (total nodes in tree)
368      // else if root is null, return 0
369      return NumNodes;
370  }
371
372  /*****
373  /*!
374
375  \brief
376      Checks height of the tree.
377
378  \return
379      Height of the tree.
380  */
381  /*****
382  template <typename T>
383  int BSTree<T>::height(void) const
384  {
385      // call tree_height with root of the tree
386      // to find total height
387      return Height;
388  }
389
390  /*****
391  /*!
392
```



```

393  \brief
394  Returns root of the tree.
395
396  \return
397  Root of the tree.
398  */
399  /*****
400  template <typename T>
401  typename BSTree<T>::BinTree BSTree<T>::root(void) const
402  {
403      return Root;
404  }
405
406  /*****
407  /*!
408
409  \brief
410  Checks for extra credit implementation.
411
412  \return
413  True if extra credit was implementated. False if not.
414  */
415  /*****
416  template <typename T>
417  bool BSTree<T>::ImplementedIndexing(void)
418  {
419      // index not implemented
420      return false;
421  }
422
423  /*****
424  /*!
425
426  \brief
427  Makes a node to put in the tree.
428
429  \param value
430  Value to store increated node
431
432  \return
433  Pointer to node created.
434  */
435  /*****
436  template <typename T>
437  typename BSTree<T>::BinTree BSTree<T>::make_node(const T& value)
438  {
439      try
440      {
441          // use objectallocaltor to make memory for node
442          BinTree mem = reinterpret_cast<BinTree>(allocator->Allocate()); // Allocate memory for the
443
444          // put node in memoery allocated above

```

```
445     BinTree node = new (mem) BinTreeNode(value);
446
447     // return node created
448     return node;
449
450 }
451 // catch any exception from objectallocator and throw our own exception
452 catch (const OAException &e)
453 {
454     throw(BSTException(BSTException::E_NO_MEMORY, e.what()));
455 }
456 }
457
458 /*****
459  *!
460
461  \brief
462  First deleted underlying node in the allocator and then
463  gives back thta memory to the allocator.
464
465  \param node
466  node being freed.
467
468  */
469 /*****
470  template <typename T>
471  void BSTree<T>::FreeNode(BinTree node)
472  {
473      // manually call destructor on the node inside the OA
474      node->~BinTreeNode();
475
476      // give back the memory used to the OA
477      allocator->Free(node);
478  }
479
480 /*****
481  *!
482
483  \brief
484  Calculate height of the tree.
485
486  \param tree
487  Root of the tree
488
489  \return
490  Height of the tree.
491  */
492 /*****
493  template <typename T>
494  int BSTree<T>::tree_height(BinTree tree) const
495  {
496      if (tree == 0)
```

```

497     return -1;
498 else
499     return (1 + std::max(tree_height(tree->left), tree_height(tree->right)));
500 }
501
502 /*****
503  *!
504
505  \brief
506   Finds predecessor of a node.
507
508  \param tree
509   Starting node (left of the node being removed).
510
511  \param predecessor
512   Reference to a pointer that will be used in deleting after
513   predecessor is found.
514
515  */
516 /*****
517  template <typename T>
518  void BSTree<T>::FindPredecessor(BinTree tree, BinTree &predecessor) const
519  {
520      // start looking for the RIGHTMOST node in the LEFT sub-tree
521      predecessor = tree->left;
522
523      while (predecessor->right != 0)
524          predecessor = predecessor->right;
525  }
526
527 /*****
528  *!
529
530  \brief
531   Recursive copy function used in copy constructor and assignment
532
533  \param destination
534   lhs object
535
536  \param source
537   rhs object
538
539  */
540 /*****
541  template <typename T>
542  void BSTree<T>::CopyHelper(BinTree &destination, const BinTree &source)
543  {
544      // base case
545      if(!source)
546          destination = 0;
547      // copy the source
548      else

```

```
549 {
550     // copy the node
551     destination = make_node(source->data);
552
553     // copy the left
554     CopyHelper(destination->left, source->left);
555     //copy the right
556     CopyHelper(destination->right, source->right);
557
558 }
559 }
560
561 /*****
562 */
563
564 \brief
565 Recursive function to remove an item from the tree
566
567 \param tree
568 Starts at the root of the tree
569
570 \param Data
571 Data being removed from the tree.
572
573 */
574 /*****
575
576 template <typename T>
577 void BSTree<T>::DeleteItem(BinTree& tree, const T& Data)
578 {
579     // didnt find value where it should be
580     // does not exist in this tree
581     if (tree == 0)
582         return;
583
584     else if (Data < tree->data)
585         DeleteItem(tree->left, Data);
586     else if (Data > tree->data)
587         DeleteItem(tree->right, Data);
588     else // (Data == tree->data)
589     {
590         // leaf node deletion
591         // base case
592         if (tree->left == 0)
593         {
594             BinTree temp = tree;
595             tree = tree->right;
596             FreeNode(temp);
597             --NumNodes;
598         }
599         // leaf node deletion
600         // base case
601         else if (tree->right == 0)
```

```

601     {
602         BinTree temp = tree;
603         tree = tree->left;
604         FreeNode(temp);
605         --NumNodes;
606     }
607     else
608     {
609         // find predecessor
610         // and call again to get to base case
611         BinTree pred = 0;
612         FindPredecessor(tree, pred);
613         tree->data = pred->data;
614         DeleteItem(tree->left, tree->data);
615     }
616 }
617 }
618
619 /*****
620  *!
621
622  \brief
623  Recursively inserts an item into the tree.
624
625  \param tree
626  Starts at the root of the tree
627
628  \param value
629  value being inserted into the tree
630
631  \param depth
632  how deep down the tree we went. Used to update height
633
634  */
635 /*****
636  template <typename T>
637  void BSTree<T>::InsertItem(BinTree &tree, const T& value, int depth)
638  {
639      try
640      {
641          // found out spot to be, insert
642          if (tree == 0)
643          {
644              // if we gone further down than height
645              // increment height
646              if(depth > Height)
647                  ++Height;
648
649              tree = make_node(value);
650
651              // increment node counter
652              ++NumNodes;

```

```
653
654     return;
655 }
656
657 if (value < tree->data)
658     InsertItem(tree->left, value, depth + 1);
659 else
660     InsertItem(tree->right, value, depth + 1);
661 }
662 catch (const BSTException &e)
663 {
664     throw e;
665 }
666 }
667
668
669 /*****
670  *!
671
672  \brief
673   Finds an item in the tree.
674
675  \param tree
676   Starts at root of the tree
677
678  \param Data
679   Data looking for
680
681  \param compares
682   Number of comparisons it took to find (or not find) the item.
683
684  \return
685   True if item was found, False if it wasn't;
686  */
687 /*****
688  template <typename T>
689  bool BSTree<T>::FindItem(BinTree tree, const T& Data, unsigned &compares) const
690  {
691      // increment comparison counter
692      ++compares;
693
694      // found where value would have been but it's empty
695      // value is not in the tree
696      if(tree == 0)
697          return false;
698      else if(Data == tree->data)
699          return true;
700      else if(Data < tree->data)
701          return FindItem(tree->left, Data, compares);
702      else
703          return FindItem(tree->right, Data, compares);
704  }
```

