HBreithaupt CS280

0 contributors

Raw Blame

559 lines (455 sloc) 12.6 KB

```
1  /******************************************************************************/
2  /*!
3   \file AVLTree.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   Implmentation of AVL tree funciontality.
12
13
14  Hours spent on assignment: 10
15
16  Specific portions that gave you the most trouble:
17  - using templates here gave me a lot of trouble, not sure why. I had
18    some trouble getting to use functions and data inherited from BSTree.
19
20  - deleting nodes was troublesome took me awhile to get all cases correct, even
21    with pseudocode posted on the site.
22
23  - stree test will fail. At some point along the way pointers were being dropped.
24    StressSmall works correctly and I implemented example from website to make
25    sure I was handling multiple rotations in one deletion and it was correct.
26    I just couldnt figure out where things went wrong.
27
28  */
```

```
/******************************************************************************/


/******************************************************************************/
/*!

 \brief
  Constructor for AVLTree

 \param OA
  Object Allocator to be used with the tree. (either provided or object makes
  its own).

 \param ShareOA
  Flag to indicate sharing of an allocator between copies of objects

 */
/******************************************************************************/
template <typename T>
AVLTree<T>::AVLTree(ObjectAllocator *OA, bool ShareOA) : BSTree<T>(OA, ShareOA)
{

}

/******************************************************************************/
/*!

 \brief
  Destructor, will call base class destructor and that will be sufficient

 */
/******************************************************************************/
template <typename T>
AVLTree<T>::~AVLTree()
{

}

/******************************************************************************/
/*!

 \brief
  Public insert function. Redirects to private recursive function to handle
  insertion.

 \param value
  Whate is being inserted into the tree


 */
/******************************************************************************/
template <typename T>
```

```
81  void AVLTree<T>::insert(const T& value)
82  {
83    try
84    {
85      // stack to keep track how we got to insertion place
86      std::stack<BinTree> nodes;
87      InsertItemAVL(this->Root, value, nodes, 0);
88
89      this->Height = this->tree_height(this->Root);
90    }
91    catch (const BSTException &e)
92    {
93      throw e;
94    }
95  }
96
97  /******************************************************************************/
98  /*!
99
100  \brief
101   Public remove function. Redirects to private recursive function.
102
103  \param value
104   What's being removed from the tree.
105
106  */
107 /******************************************************************************/
108 template <typename T>
109 void AVLTree<T>::remove(const T& value)
110 {
111     // use private function to recursively delete and balance the tree
112   std::stack<BinTree> nodes;
113   DeleteItemAVL(this->Root, value, nodes);
114
115     // re calculate height after removal
116   this->Height = this->tree_height(this->Root);
117
118   //unsigned int count = count_nodes(this->Root);
119   //std::cout << "real count is : " << count << std::endl;
120 }
121
122 /******************************************************************************/
123 /*!
124
125  \brief
126   Private recursive insert function to properly insert an item
127
128  \param tree
129   Reference to a pointer of the root of the tree
130
131  \param value
132   What's being added to the tree
```

```
133
134   \param nodes
135    Reference to a stack to keep track of where we went to balance the tree
136    after inserting.
137
138   \param depth
139    How deep down the tree we went. Used to update height
140
141   */
142  /******************************************************************************/
143  template <typename T>
144  void AVLTree<T>::InsertItemAVL(BinTree &tree, const T& value, stack &nodes, int depth)
145  {
146    try
147    {
148      // found our spot to be
149      if(!tree)
150      {
151        // insert
152        tree = this->make_node(value);
153
154        //balance
155        Balance(nodes);
156
157        // increment node counter
158        ++(this->NumNodes);
159
160        //if(depth > this->Height)
161          //++(this->Height);
162      }
163      else if(value < tree->data)
164      {
165        // push address of visited node onto the stack
166        // move on to left subtree
167        nodes.push(tree);
168        InsertItemAVL(tree->left, value, nodes, ++depth);
169      }
170      else // value > tree->data
171      {
172        // push address of visited node onto the stack
173        // move on to right subtree
174        nodes.push(tree);
175        InsertItemAVL(tree->right, value, nodes, ++depth);
176      }
177    }
178    catch (const BSTException &e)
179    {
180      throw e;
181    }
182  }
183
184  /******************************************************************************/
```

```
/*!

 \brief
  Balances an AVL Tree.

 \param nodes
  stack of where we went for insertion/deletion

 \param Inserting
  Bool to indicate whether we are balancing because of an insertion or deletion.
  If inserting we only need one rotation to fix the balance, if deleting
  we have to back track all the way up the tree.

 */
/******************************************************************************/
template <typename T>
void AVLTree<T>::Balance(stack &nodes, bool Inserting)
{
  while(!nodes.empty())
  {
    BinTree y = nodes.top();
    nodes.pop();

      // find the node that is now out of balance
    int LeftHeight = this->tree_height(y->left);
    int RightHeight = this->tree_height(y->right);

    //compare heights
    // v and w are always the left and right children of u, respectively

      // left heavy
    if(LeftHeight > RightHeight + 1)
    {
        // call left havy balance function
      LeftHeavyBalance(y, nodes);

        // if inserting only do one rotation
        // else this was called from deleting
        // so we must continue until the stack is empty
      if(Inserting)
        return;
    }
      // right heavy
    else if(RightHeight > LeftHeight + 1)
    {
      RightHeavyBalance(y, nodes);
        // if this was called from insert
        // we are done, if called from delete
        // must keep going until stack is empty
      if(Inserting)
        return;
    }
```

```
237   }
238 }
239
240 /******************************************************************************/
241 /*!
242
243  \brief
244   Balance if we are right heavy.
245
246  \param y
247   The node that is out of balance.
248
249  \param nodes
250   Stack of nodes we traveled. Need to get the parent of y to properly
251   rotate.
252
253  */
254 /******************************************************************************/
255 template <typename T>
256 void AVLTree<T>::RightHeavyBalance(BinTree &y, stack &nodes)
257 {
258   BinTree u, v, w, NulParent = 0;
259
260   u = y->right;
261   v = u->left;
262   w = u->right;
263
264     // height of subtrees at v & w
265   int VHeight = this->tree_height(v);
266   int WHeight = this->tree_height(w);
267
268   /*  y
269        \
270         u
271        /
272       v */
273   // zig-zag
274   if(VHeight > WHeight)
275   {
276       // promote v twice
277     RotateRight(y->right);
278     RotateLeft(y);
279
280     /*      v
281           /   \
282          y     u  */
283
284     // attach rotation back to the tree
285     if(nodes.empty())
286       AttachRotation(NulParent, v);
287     else
288       AttachRotation(nodes.top(), v);
```

```
289   }
290
291   /*    y
292          \
293           u
294            \
295             w  */
296
297   // zig-zig
298   else if(WHeight > VHeight || VHeight == WHeight)
299   {
300     //  rotate left about y
301
302     /*      u
303           /   \
304          y     w  */
305     RotateLeft(y);
306
307     // attach rotation back to the tree
308     if(nodes.empty())
309       AttachRotation(NulParent, u);
310     else
311       AttachRotation(nodes.top(), u);
312   }
313 }
314
315 /******************************************************************************/
316 /*!
317
318  \brief
319   Balance if offending node is left heavy.
320
321  \param y
322   Offending node
323
324  \param nodes
325   stack of nodes traveled. needed to get parent of y to rotate
326
327  */
328 /******************************************************************************/
329 template <typename T>
330 void AVLTree<T>::LeftHeavyBalance(BinTree &y, std::stack<BinTree>& nodes)
331 {
332   BinTree u, v, w, NulParent = 0;
333
334   u = y->left;
335   v = u->left;
336   w = u->right;
337
338   // height of subtrees at v & w
339   int VHeight = this->tree_height(v);
340   int WHeight = this->tree_height(w);
```

```
341
342
343     /*      y
344            /
345           u
346          /
347         v     */
348     // zig-zig
349     if(VHeight > WHeight || WHeight == VHeight)
350     {
351       // rotate right about u
352
353       /*      u
354             /   \
355            v     y */
356       RotateRight(y);
357
358       // attach rotation done back to tree
359       if(nodes.empty())
360         AttachRotation(NulParent, u);
361       else
362         AttachRotation(nodes.top(), u);
363     }
364
365     /*       y
366             /
367            u
368             \
369              w     */
370
371     // zig zag
372     else if (WHeight > VHeight)
373     {
374       // rotate about u, promote w
375
376       /*      y
377              /
378             w
379            /
380           u        */
381       RotateLeft(y->left);
382
383       /* rotate about y, promote w again
384               w
385             /   \
386            u     y  */
387       RotateRight(y);
388
389       // reattach rotations
390       if(nodes.empty())
391         AttachRotation(NulParent, w);
392       else
```

```
393         AttachRotation(nodes.top(), w);
394       }
395     }
396
397     /******************************************************************************/
398     /*!
399
400      \brief
401       Reataches a rotation since we dont inheritanly have a parent pointer.
402
403      \param parent
404       Parent of the offending node that was rotated.
405
406      \param rotation
407       The 'top' node (see diagram a few lines above, w is the 'top')
408       of a rotation to hook back up.
409
410      */
411     /******************************************************************************/
412     template <typename T>
413     void AVLTree<T>::AttachRotation(BinTree &parent, BinTree &rotation)
414     {
415         // if parent is null then we rotated around the root
416         // else attach accordginly
417       if(!parent)
418         this->Root = rotation;
419       else
420       {
421         if(rotation->data < parent->data)
422           parent->left = rotation;
423         else
424           parent->right = rotation;
425       }
426     }
427
428     template <typename T>
429     unsigned int AVLTree<T>::count_nodes(BinTree &Root)
430     {
431       unsigned int count = 0;
432       if (Root != NULL)
433       {
434         count = 1 + count_nodes(Root->left) + count_nodes(Root->right);
435       }
436
437       return count;
438     }
439
440     /******************************************************************************/
441     /*!
442
443      \brief
444       Rotate a node left
```

```
445
446   \param tree
447    Offending node
448
449   */
450  /******************************************************************************/
451  template <typename T>
452  void AVLTree<T>::RotateLeft(BinTree &tree)
453  {
454    BinTree temp = tree;
455    tree = tree->right;
456    temp->right = tree->left;
457    tree->left = temp;
458  }
459
460  /******************************************************************************/
461  /*!
462
463   \brief
464    Rotate a node right
465
466   \param tree
467    the offending node
468
469   */
470  /******************************************************************************/
471  template <typename T>
472  void AVLTree<T>::RotateRight(BinTree &tree)
473  {
474    BinTree temp = tree;
475    tree = tree->left;
476    temp->left = tree->right;
477    tree->right = temp;
478  }
479
480  /******************************************************************************/
481  /*!
482
483   \brief
484    Deleting an item in avl tree.
485
486   \param tree
487    node to start at (in first call it is the root).
488
489   \param value
490    Value being removed from the tree
491
492   \param nodes
493    Stack of nodes we traversed to balance the tree after deletion
494
495   */
496  /******************************************************************************/
```

```
497 template <typename T>
498 void AVLTree<T>::DeleteItemAVL(BinTree &tree, const T& value, stack &nodes)
499 {
500     // didnt find item where it should be
501     // does not exist in the tree
502   if(tree == 0)
503     return;
504 
505   else if(value < tree->data)
506   {
507       // push visited node onto the stack
508       // and move on to left subtree
509     nodes.push(tree);
510     DeleteItemAVL(tree->left, value, nodes);
511   }
512   else if(value > tree->data)
513   {
514       // push visited node onto the stack
515       // move on to right subtree
516     nodes.push(tree);
517     DeleteItemAVL(tree->right, value, nodes);
518   }
519   else //value == tree->data
520   {
521       // leaf node deletion, base case
522     if(tree->left == 0)
523     {
524       BinTree temp = tree;
525       tree = tree->right;
526       this->FreeNode(temp);
527       --(this->NumNodes);
528 
529         // call balance with flag to indicate
530         // a deletion happened
531         // so must exhaust stack looking for iffending nodes
532       Balance(nodes, false);
533     }
534       // leaf node deletion, base case
535     else if (tree->right == 0)
536     {
537       BinTree temp = tree;
538       tree = tree->left;
539       this->FreeNode(temp);
540       --(this->NumNodes);
541 
542         // call balance with flag to indicate
543         // a deletion happened
544         // so must exhaust stack looking for iffending nodes
545       Balance(nodes, false);
546     }
547     else // node has two children
548     {
```

```
549         // find predecessor
550         // and then delete source of predecessor
551       BinTree pred = 0;
552       this->FindPredecessor(tree, pred);
553       tree->data = pred->data;
554       nodes.push(tree);
555       DeleteItemAVL(tree->left, tree->data, nodes);
556     }
557   }
558 }
559
```