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

```
/***************************************************************************/
/*!
\file   BSTree.cpp
\author Ang Cheng Yong
\par    email: a.chengyong\@digipen.edu
\par    DigiPen login: a.chengyong
\par    Course: CS280
\par    Programming Assignment #4
\date   8/11/2016
\brief
This file contains the implementation for BST.
*/
/***************************************************************************/

/***************************************************************************/
/*!
\fn template <typename T> BSTree<T>::BSTree(ObjectAllocator *OA, bool ShareOA)
\brief
Constructor of the BSTree

\param OA
object allocator for the tree's nodes

\param ShareOA
boolean for sharing object allcoator among trees


\return
none

*/
```

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

template <typename T>
BSTree<T>::BSTree(ObjectAllocator *OA, bool ShareOA) : oa(OA), share(ShareOA), root_(0){

        try {
                if (!share)
                        oa = new ObjectAllocator(sizeof(BinTreeNode), OAConfig());
        }
        catch (const OAException &e) {
                throw(BSTException(BSTException::E_NO_MEMORY, e.what()));
        }

}

/*************************************************************************/
/*!
\fn template <typename T> BSTree<T>::BSTree(const BSTree& rhs)
\brief
Copy constructor of BSTree

\param rhs
the BSTree that is to be copied

\return
none
*/
/*************************************************************************/

template <typename T>
BSTree<T>::BSTree(const BSTree& rhs) {

        if (rhs.oa){ // share object allocator if available
                oa = rhs.oa;
                share = true;
        }
        else { // create own if not shared
                OAConfig config(true);
                oa = new ObjectAllocator(sizeof(BinTreeNode), config);
                share = false;
        }

        if (rhs.root()){
                BinTree tmp = make_node(rhs.root()->data);
                tmp->count = rhs.root()->count;
                copy_tree(tmp, rhs.root());
                root_ = tmp;;
        }

}

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

```
/*!
\fn template <typename T> BSTree<T>::~BSTree()
\brief
Destructor of BSTree
\return
none
*/
/***************************************************************************/

template <typename T>
BSTree<T>::~BSTree() {
        clear();
        if (!share)
                delete oa;

}

/***************************************************************************/
/*!
\fn template <typename T> BSTree<T>& BSTree<T>::operator=(const BSTree& rhs)
\brief
assignment operator of BSTree

\param rhs
the BSTree that is to be based on

\return
The assigned tree itself
*/
/***************************************************************************/

template <typename T>
BSTree<T>& BSTree<T>::operator=(const BSTree& rhs) {

        if (this != &rhs) {
                if (rhs.share) { // share object allocator if available
                        oa = rhs.oa;
                        share = true;
                }
                else { // create own if not shared
                        OAConfig config(true);
                        oa = new ObjectAllocator(sizeof(BinTreeNode), config);
                        share = false;
                }

                if (rhs.root()) {
                        BinTree tmp = make_node(rhs.root()->data);
                        tmp->count = rhs.root()->count;
                        copy_tree(tmp, rhs.root());
                        clear();
                        root_ = tmp;;
                }
```

```
        }
        return *this;
}

/*************************************************************************/
/*!
\fn template <typename T> void BSTree<T>::clear(void)
\brief
clear the tree's nodes

\return
none
*/
/*************************************************************************/

template<typename T>
void BSTree<T>::clear(void){
        free_tree(root_);
}

/*************************************************************************/
/*!
\fn template <typename T> void BSTree<T>::free_tree(BinTree& tree)
\brief
helper function to clear tree of nodes by recursion

\param tree
the tree to clear

\return
none
*/
/*************************************************************************/

template<typename T>
void BSTree<T>::free_tree(BinTree& tree){

        if (!tree)
                return;

        free_tree(tree->left);
        free_tree(tree->right);

        delete_node(tree, tree->data);
}

/*************************************************************************/
/*!
\fn template <typename T> void BSTree<T>::copy_tree(BinTree &lhs, BinTree rhs)throw(BSTExceptio
\brief
helper function to construct tree based on another tree by recursion.

```

```
\param lhs
input tree to copy data on

\param rhs
input tree to copy data off

\return
none
*/
/*************************************************************************/

template<typename T>
void BSTree<T>::copy_tree(BinTree &lhs, BinTree rhs)throw(BSTException){
        try{

                if (rhs){
                        // fill data and count of both nodes
                        if (rhs->left){
                                lhs->left = make_node(rhs->left->data);
                                lhs->left->count = rhs->left->count;
                                copy_tree(lhs->left, rhs->left); // before copying recursively
                        }

                        if (rhs->right){
                                lhs->right = make_node(rhs->right->data);
                                lhs->right->count = rhs->right->count;
                                copy_tree(lhs->right, rhs->right);
                        }
                }
        }
        catch (const OAException &e)
        {
                throw(BSTException(BSTException::E_NO_MEMORY, e.what()));
        }
}

/*************************************************************************/
/*!
\fn template <typename T> void BSTree<T>::remove(const T& value)
\brief
remove a node based on input

\param value
the value to match for removal of node

\return
none
*/
/*************************************************************************/

template <typename T>
void BSTree<T>::remove(const T& value) {
```

```
        delete_node(root_, value);
}

/*************************************************************************/
/*!
\fn template <typename T> void BSTree<T>::delete_node(BinTree & Tree, const T& value)
\brief
helper function to search for the correct node to delete by recursion.

\param Tree
the tree to remove node from

\param value
the value to match

\return
none
*/
/*************************************************************************/

template<typename T>
void BSTree<T>::delete_node(BinTree & Tree, const T& value){

        if (Tree == 0)
                return;
        else if (value < Tree->data){
                --Tree->count;
                delete_node(Tree->left, value);
        }
        else if (value > Tree->data){
                --Tree->count;
                delete_node(Tree->right, value);
        }
        else{ // match found
                --Tree->count;

                if (Tree->left == 0){
                        BinTree temp = Tree;
                        Tree = Tree->right;
                        free_node(temp);
                }
                else if (Tree->right == 0){
                        BinTree temp = Tree;
                        Tree = Tree->left;
                        free_node(temp);
                }
                else { // if node has 2 children
                        BinTree pred = 0;
                        find_predecessor(Tree, pred);
                        Tree->data = pred->data;
                        delete_node(Tree->left, Tree->data);
                }
```

```
        }
}

/*************************************************************************/
/*!
\fn template <typename T> const typename BSTree<T>::BinTreeNode* BSTree<T>::operator[](int inde
\brief
subscript operator to serach in the tree

\param index
the value to match

\return
the matching node
*/
/*************************************************************************/

template <typename T>
const typename BSTree<T>::BinTreeNode* BSTree<T>::operator[](int index) const {

        return sub_node(root_, index);
}

/*************************************************************************/
/*!
\fn template <typename T> const typename BSTree<T>::BinTreeNode* BSTree<T>::sub_node(BinTree tr
\brief
helper function to find the correct node by recursion.

\param tree
the tree to find the match

\param compares
the value to match

\return
the matching node
*/
/*************************************************************************/

template<typename T>
const typename BSTree<T>::BinTreeNode* BSTree<T>::sub_node(BinTree tree, int compares)const {
        if (!tree)
                return NULL;

        unsigned tmp = (tree->left) ? tree->left->count : 0;

        if (tmp > static_cast<unsigned>(compares))
                return sub_node(tree->left, compares);
        else if (tmp < static_cast<unsigned>(compares))
                return sub_node(tree->right, compares - tmp - 1);
        else
```

```
                return tree;

}

/*************************************************************************/
/*!
\fn template <typename T> void BSTree<T>::insert(const T& value) throw(BSTException)
\brief
insert a node into the tree

\param value
the value of node to be inserted

\return
none
*/
/*************************************************************************/

template <typename T>
void BSTree<T>::insert(const T& value) throw(BSTException) {
        insert_node(root_, value);
}

/*************************************************************************/
/*!
\fn void BSTree<T>::insert_node(BinTree & tree, const T& value) throw(BSTException)
\brief
helper function insert a node into the tree by recursion

\param tree
the tree for node to be inserted

\param value
the value of node to be inserted

\return
none
*/
/*************************************************************************/

template<typename T>
void BSTree<T>::insert_node(BinTree & tree, const T& value) throw(BSTException){
        try{
                if (!tree){
                        tree = make_node(value);
                        ++tree->count;
                }
                else if (value < tree->data){
                        ++tree->count;
                        insert_node(tree->left, value);
                }
                else if (value > tree->data){
```

```
                        ++tree->count;
                        insert_node(tree->right, value);
                }
                else{
                        std::cout << "Error, duplicate item" << std::endl;
                }
        }
        catch (const OAException &e){
                throw(BSTException(BSTException::E_NO_MEMORY, e.what()));
        }
}

/*************************************************************************/
/*!
\fn template <typename T> bool BSTree<T>::find(const T& value, unsigned &compares) const
\brief
find a node in the tree with a matching value

\param value
the value to be matched

\param compares
number of function calls used to find the matching node

\return
whether a node with the matching value exist
*/
/*************************************************************************/

template <typename T>
bool BSTree<T>::find(const T& value, unsigned &compares) const {

        return find_node(root_, value, compares);

}

/*************************************************************************/
/*!
\fn template <typename T> bool BSTree<T>::find_node(BinTree tree, const T& value, unsigned &cor
\brief
helper function find a node in the tree with a matching value by recursion

\param tree
the tree to be searched

\param value
the value to be matched

\param compares
number of function calls used to find the matching node

\return
```

```
whether a node with the matching value exist
*/
/***************************************************************************/

template<typename T>
bool BSTree<T>::find_node(BinTree tree, const T& value, unsigned &compares)const
{
        ++compares;

        if (tree == 0)
                return false;
        else if (value == tree->data) // match fit
                return true;
        else if (value < tree->data) // continue finding
                return find_node(tree->left, value, compares);
        else
                return find_node(tree->right, value, compares);

}

/***************************************************************************/
/*!
\fn template <typename T> int BSTree<T>::height(void) const
\brief
count the height of tree

\return
height of tree
*/
/***************************************************************************/

template <typename T>
int BSTree<T>::height(void) const {
        return tree_height(root_);
}

/***************************************************************************/
/*!
\fn template <typename T> int BSTree<T>::tree_height(BinTree tree) const
\brief
helper function find the height of a tree by recursion

\param tree
the tree to be counted

\return
height of tree
*/
/***************************************************************************/

template<typename T>
int BSTree<T>::tree_height(BinTree tree) const
```

```
{

        if (!tree)
                return -1;
        //recursively calculate height of subsequent subtrees
        int L = tree_height(tree->left);
        int R = tree_height(tree->right);

        if (L > R)
                return L + 1;
        else
                return R + 1;
}

/***************************************************************************/
/*!
\fn template <typename T> typename BSTree<T>::BinTree BSTree<T>::root(void) const
\brief
get the root of the tree

\return
root of tree
*/
/***************************************************************************/

template <typename T>
typename BSTree<T>::BinTree BSTree<T>::root(void) const {
        return root_;
}

/***************************************************************************/
/*!
\fn template <typename T> bool BSTree<T>::empty(void) const
\brief
whether a tree is empty

\return
root of tree
*/
/***************************************************************************/

template <typename T>
bool BSTree<T>::empty(void) const {
        return (root_ == 0);
}

/***************************************************************************/
/*!
\fn template <typename T> bool BSTree<T>::empty(void) const
\brief
whether a tree is empty

```

```
\return
root of tree
*/
/*************************************************************************/

template <typename T>
unsigned int BSTree<T>::size(void) const {
        return (root_) ? root_->count : 0;
}

/*************************************************************************/
/*!
\fn typename BSTree<T>::BinTree& BSTree<T>::get_root(void)
\brief
get the root of the tree

\return
root of tree
*/
/*************************************************************************/

template <typename T>
typename BSTree<T>::BinTree& BSTree<T>::get_root(void) {
        return root_;
}

/*************************************************************************/
/*!
\fn template <typename T> typename BSTree<T>::BinTree BSTree<T>::make_node(const T& value)
\brief
make a new node for the tree

\param value
the value for new node

\return
the new node
*/
/*************************************************************************/

template <typename T>
typename BSTree<T>::BinTree BSTree<T>::make_node(const T& value) {

        try{
                //placement new through using object allcoator
                BinTree mem = reinterpret_cast<BinTreeNode *>(oa->Allocate());
                BinTree node = new (mem) BinTreeNode(value);
                return node;
        }
        catch (const OAException &e){
                throw(BSTException(BSTException::E_NO_MEMORY, e.what()));
        }
```

```

}

/***************************************************************************/
/*!
\fn template <typename T> void BSTree<T>::free_node(BinTree node)
\brief
free a node from the tree

\param node
the node to free

\return
none
*/
/***************************************************************************/

template <typename T>
void BSTree<T>::free_node(BinTree node) {
        node->~BinTreeNode();
        oa->Free(node);
}

/***************************************************************************/
/*!
\fn template <typename T> void BSTree<T>::find_predecessor(BinTree tree, BinTree &predecessor)
\brief
find the parent of a node

\param tree
the node to be searched

\param predecessor
the node to fill as predecessor

\return
none
*/
/***************************************************************************/

template <typename T>
void BSTree<T>::find_predecessor(BinTree tree, BinTree &predecessor) const {
        predecessor = tree->left;
        while (predecessor->right != 0)
                predecessor = predecessor->right;
}
```