cs280s21-b.sg

Dashboard / My courses / cs280s21-b.sg / General / Assignment 3: AVL Trees

Description Submission Edit Submission view

Submitted on Sunday, March 14, 2021, 1:00 AM (Download)

Automatic evaluation[-]

Proposed grade: 100 / 100

Compilation[-]

Running code ....

Executable successfully created

Comments[-]

[+]**Summary of tests**

AVLTree.h

```
/******************************************************************************/
/*!
\file:      AVLTree.h
\author:    Goh Wei Zhe, weizhe.goh, 440000119
\par email: weizhe.goh\@digipen.edu
\date:      March 12, 2021
\brief      This file contains the declarations needed to implement the simple
            API for AVL Trees with recursive algorithms.

Copyright (C) 2021 DigiPen Institute of Technology.
Reproduction or disclosure of this file or its contents without the
prior written consent of DigiPen Institute of Technology is prohibited.
*/
/******************************************************************************/

//----------------------------------------------------------------------------
#ifndef AVLTREE_H
#define AVLTREE_H
//----------------------------------------------------------------------------
#include <stack>
#include "BSTree.h"

/*!
  Definition for the AVL Tree
*/
template <typename T>
class AVLTree : public BSTree<T>
{
  public:
    AVLTree(ObjectAllocator *OA = 0, bool ShareOA = false);
    virtual ~AVLTree() = default; // DO NOT IMPLEMENT
    virtual void insert(const T& value) override;
    virtual void remove(const T& value) override;

      // Returns true if efficiency implemented
    static bool ImplementedBalanceFactor(void);

  private:
    // private stuff

    unsigned int node_count(typename BSTree<T>::BinTree& tree) const;

    void insert_start(typename BSTree<T>::BinTree& tree, const T& value);
    void insert_node(typename BSTree<T>::BinTree& node, const T& value,
                     std::stack<typename BSTree<T>::BinTree*>& nodes);

    void remove_start(typename BSTree<T>::BinTree& tree, const T& value);
    void remove_node(typename BSTree<T>::BinTree &tree, const T& value,
                     std::stack<typename BSTree<T>::BinTree*>& nodes);


    void RotateLeft(typename BSTree<T>::BinTree& tree);
    void RotateRight(typename BSTree<T>::BinTree& tree);

    void BalanceAVLTree(std::stack<typename BSTree<T>::BinTree*>& nodes);
};

#include "AVLTree.cpp"

#endif
//----------------------------------------------------------------------------

```

AVLTree.cpp

```
/******************************************************************************/
/*!
\file:      AVLTree.cpp
\author:    Goh Wei Zhe, weizhe.goh, 440000119
\par email: weizhe.goh\@digipen.edu
\date:      March 12, 2021
\brief      This file contains the definitions needed to implement the simple
            API for AVL Trees with recursive algorithms.

Copyright (C) 2021 DigiPen Institute of Technology.
Reproduction or disclosure of this file or its contents without the
prior written consent of DigiPen Institute of Technology is prohibited.
*/
/******************************************************************************/

#include "AVLTree.h"

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

\brief  Constructor of AVLTree

\param  OA - Object allocator for the BSTree nodes

\param  ShareOA - boolean for sharing object allocator
*/
/******************************************************************************/
template<typename T>
AVLTree<T>::AVLTree(ObjectAllocator *OA, bool ShareOA):BSTree<T>(OA, ShareOA){}

/******************************************************************************/
/*!
\fn     template<typename T>
        void AVLTree<T>::insert(const T& value)

\brief  Insert a node into the tree

\param  value - The value of the node to be inserted
*/
/******************************************************************************/
template<typename T>
void AVLTree<T>::insert(const T& value)
{
    insert_start(BSTree<T>::get_root(), value);
    node_count(BSTree<T>::get_root());
}

/******************************************************************************/
/*!
\fn     template <typename T>
        void AVLTree<T>::insert_start(typename BSTree<T>::BinTree& tree,
        const T& value)

\brief  Function to start the recursion to insert a node into the tree

\param  tree - The tree that the node to be inserted

\param  value - The value of the node to be inserted
*/
/******************************************************************************/
template <typename T>
void AVLTree<T>::insert_start(typename BSTree<T>::BinTree& tree, const T& value)
{
    std::stack<typename BSTree<T>::BinTree*> stack_;

    insert_node(tree, value, stack_);
}

/******************************************************************************/
/*!
\fn     template <typename T>
        void AVLTree<T>::insert_node(typename BSTree<T>::BinTree & node,
        const T& value, std::stack<typename BSTree<T>::BinTree*> & nodes)

\brief  Helper function to insert a node into the tree by recursion

\param  node - The tree or subtree for node to be inserted

\param  value - The value of the node to be inserted

\param  nodes - The stack to push or pop the nodes
*/
/******************************************************************************/
template <typename T>
void AVLTree<T>::insert_node(typename BSTree<T>::BinTree & node, const T& value,
std::stack<typename BSTree<T>::BinTree*> & nodes)
{
    if(node == 0)
    {
        node = BSTree<T>::make_node(value);
        BalanceAVLTree(nodes);
    }
    else if (value < node->data)
    {
        nodes.push(&node);
        insert_node(node->left, value, nodes);
    }
    else if (value > node->data)
    {
        nodes.push(&node);
        insert_node(node->right, value, nodes);
    }
    else
        std::cout << "Error, duplicate item" << std::endl;
```

```
}

/******************************************************************************/
/*!
\fn     template <typename T>
        unsigned int AVLTree<T>::node_count(typename BSTree<T>::BinTree& tree)
        const

\brief  Count the number of nodes in a tree or subtree

\param  tree - The tree to count the number of total nodes in that tree.

\return Return the total node count of a tree
*/
/******************************************************************************/
template <typename T>
unsigned int AVLTree<T>::node_count(typename BSTree<T>::BinTree& tree) const
{
    if(tree == NULL)
        return 0;

    tree->count = 1 + node_count(tree->left) + node_count(tree->right);

    return tree->count;
}
/******************************************************************************/
/*!
\fn     template <typename T>
        void AVLTree<T>::RotateLeft(typename BSTree<T>::BinTree& tree)

\brief  Rotate the tree around the node to the left

\param  tree - The node to rotate about
*/
/******************************************************************************/
template <typename T>
void AVLTree<T>::RotateLeft(typename BSTree<T>::BinTree& tree)
{
    typename BSTree<T>::BinTree temp = tree;
    tree = tree->right;
    temp->right = tree->left;
    tree->left = temp;

    tree->count = temp->count;

    //recount node count
    unsigned leftCount = (temp->left) ? temp->left->count : 0;
    unsigned rightCount = (temp->right) ? temp->right->count : 0;

    temp->count = leftCount + rightCount + 1;
}

/******************************************************************************/
/*!
\fn     template <typename T>
        void AVLTree<T>::RotateRight(typename BSTree<T>::BinTree& tree)

\brief  Rotate the tree around the node to the right

\param  tree - The node to rotate about
*/
/******************************************************************************/
template <typename T>
void AVLTree<T>::RotateRight(typename BSTree<T>::BinTree& tree)
{
   typename BSTree<T>::BinTree temp = tree;
   tree = tree->left;
   temp->left = tree->right;
   tree->right = temp;

   tree->count = temp->count;

   //recount node count
   unsigned leftCount = (temp->left) ? temp->left->count : 0;
   unsigned rightCount = (temp->right) ? temp->right->count : 0;

   temp->count = leftCount + rightCount + 1;
}

/******************************************************************************/
/*!
\fn     template <typename T>
        void AVLTree<T>::BalanceAVLTree
        (std::stack<typename BSTree<T>::BinTree*>& nodes)

\brief  Balance the AVL tree

\param  nodes - The stack of nodes used to balance the tree
*/
/******************************************************************************/
template <typename T>
void AVLTree<T>::BalanceAVLTree(std::stack<typename BSTree<T>::BinTree*>& nodes)
{
    while(!nodes.empty())
    {
        typename BSTree<T>::BinTree* topnode = nodes.top();
        nodes.pop();

        typename BSTree<T>::BinTree& y = *topnode;

        int RH = BSTree<T>::tree_height(y->right);
        int LH = BSTree<T>::tree_height(y->left);

        if(abs(LH - RH) > 1)
        {
            if(RH > (LH + 1))
            {
                //promote twice
                if(BSTree<T>::tree_height(y->right->left) >
                BSTree<T>::tree_height(y->right->right))
                {
                    RotateRight(y->right);
                    RotateLeft(y);
                }
                else
                {
                    //promote once
                    RotateLeft(y);
                }
            }
            else if ((RH + 1) < LH)
            {
                //promote once
                if(BSTree<T>::tree_height(y->left->left) >
                BSTree<T>::tree_height(y->left->right))
                {
                    RotateRight(y);
                }
                else
                {
                   //promote twice
                   RotateLeft(y->left);
                   RotateRight(y);
                }
            }
        }
    }
}
/******************************************************************************/
/*!
\fn     template <typename T>
        void AVLTree<T>::remove(const T& value)

\brief  Remove a node in the tree

\param  value - The value of the node to be removed
*/
/******************************************************************************/
template <typename T>
void AVLTree<T>::remove(const T& value)
{
    remove_start(BSTree<T>::get_root(), value);
    node_count(BSTree<T>::get_root());
}

/******************************************************************************/
/*!
\fn     template <typename T>
        void AVLTree<T>::remove_start(typename BSTree<T>::BinTree& tree,
        const T& value)

\brief  Function to begin the recusion to remove node from the tree

\param  tree - The tree to remove the node from

\param  value - The value of the node to be removed
*/
/******************************************************************************/
template <typename T>
void AVLTree<T>::remove_start(typename BSTree<T>::BinTree& tree, const T& value)
{
    std::stack <typename BSTree<T>::BinTree*> stack_;
    remove_node(tree, value, stack_);
}

/******************************************************************************/
/*!
\fn     template<typename T>
        void AVLTree<T>::remove_node(typename BSTree<T>::BinTree &tree,
        const T& value, std::stack<typename BSTree<T>::BinTree*>& nodes)

\brief  Helper function to remove the node to delete by recursion

\param  tree - The tree to remove the node from

\param  value - The value of the node to be removed

\param  nodes - The stack to pop out the nodes from
*/
/******************************************************************************/
template<typename T>
void AVLTree<T>::remove_node(typename BSTree<T>::BinTree &tree, const T& value,
    std::stack<typename BSTree<T>::BinTree*>& nodes)
{
    if(tree == 0)
        return ;
    else if (value < tree->data)
    {
        nodes.push(&tree);
        remove_node(tree->left, value, nodes);
    }
    else if (value > tree->data)
    {
        nodes.push(&tree);
        remove_node(tree->right, value, nodes);
    }
    else
    {
        if(tree->left == 0)
        {
            typename BSTree<T>::BinTree temp = tree;
            tree = tree->right;
            BSTree<T>::free_node(temp);
            BalanceAVLTree(nodes);
        }
        else if (tree->right == 0)
        {
            typename BSTree<T>::BinTree temp = tree;
            tree = tree->left;
            BSTree<T>::free_node(temp);
            BalanceAVLTree(nodes);
        }
        else
        {
            //two child
            typename BSTree<T>::BinTree pred = 0;
            BSTree<T>::find_predecessor(tree, pred);
            tree->data = pred->data;
            nodes.push(&tree);
            remove_node(tree->left, tree->data, nodes);
        }
    }
}

/******************************************************************************/
/*!
\fn     template<typename T>
        bool AVLTree<T>::ImplementedBalanceFactor(void)

\brief  Function for efficient balancing

\return Returns true if implemented efficient balancing, else return false
*/
/******************************************************************************/
template<typename T>
bool AVLTree<T>::ImplementedBalanceFactor(void)
```

BSTree.h

```
/******************************************************************************/
/*!
\file:      BSTree.h
\author:    Goh Wei Zhe, weizhe.goh, 440000119
\par email: weizhe.goh\@digipen.edu
\date:      March 12, 2021
\brief      This file contains the declarations needed to implement the simple
            API for Binary Search Tree with recursive algorithms.

Copyright (C) 2021 DigiPen Institute of Technology.
Reproduction or disclosure of this file or its contents without the
prior written consent of DigiPen Institute of Technology is prohibited.
*/
/******************************************************************************/

//---------------------------------------------------------------------------
#ifndef BSTREE_H
#define BSTREE_H
//---------------------------------------------------------------------------
#include <string>    // std::string
#include <stdexcept> // std::exception

#include "ObjectAllocator.h"

/*!
  The exception class for the AVL/BST classes
*/
class BSTException : public std::exception
{
  public:
    /*!
      Non-default constructor

      \param ErrCode
        The kind of exception (only one currently)

      \param Message
        The human-readable reason for the exception.
    */
    BSTException(int ErrCode, const std::string& Message) :
      error_code_(ErrCode), message_(Message) {
    };

    /*!
      Retrieve the exception code.

      \return
        E_NO_MEMORY
    */
    virtual int code() const {
      return error_code_;
    }

    /*!
      Retrieve the message string

      \return
        The human-readable message.
    */
    virtual const char *what() const throw() {
      return message_.c_str();
    }

    //! Destructor
    virtual ~BSTException() {}

    //! The kinds of exceptions (only one currently)
    enum BST_EXCEPTION{E_NO_MEMORY};

  private:
    int error_code_;      //!< The code of the exception
    std::string message_; //!< Readable message text
};

/*!
  The definition of the BST
*/
template <typename T>
class BSTree
{
  public:
    //! The node structure
    struct BinTreeNode
    {
      BinTreeNode *left;  //!< The left child
      BinTreeNode *right; //!< The right child
      T data;             //!< The data
      int balance_factor; //!< optional for efficient balancing
      unsigned count;     //!< nodes in this subtree for efficient indexing
      //! Default constructor
      BinTreeNode() : left(0), right(0), data(0), balance_factor(0), count(1){};

      //! Conversion constructor
      BinTreeNode(const T& value) :
      left(0), right(0), data(value), balance_factor(0), count(1) {};
    };

    //! shorthand
    typedef BinTreeNode* BinTree;

    BSTree(ObjectAllocator *OA = 0, bool ShareOA = false);
    BSTree(const BSTree& rhs);
    virtual ~BSTree();
    BSTree& operator=(const BSTree& rhs);
    const BinTreeNode* operator[](int index) const;//for r-values (Extra Credit)
    virtual void insert(const T& value);
    virtual void remove(const T& value);
```

```cpp
    void clear();
    bool find(const T& value, unsigned &compares) const;
    bool empty() const;
    unsigned int size() const;
    int height() const;
    BinTree root() const;

  protected:
    BinTree& get_root();
    BinTree make_node(const T& value) const;
    void free_node(BinTree node);
    int tree_height(BinTree tree) const;
    void find_predecessor(BinTree tree, BinTree &predecessor) const;

  private:
    // private stuff...

    ObjectAllocator* oa;

    bool Custom_OA;
    bool share;
    BinTree root_;

    BinTree copy_tree(BinTree& destination, const BinTree& source);
    void free_tree(BinTree& root);
    void delete_node(BinTree& tree, const T& value);
    const BinTreeNode* sub_node(BinTree tree, int compares) const;
    void insert_node(BinTree& tree, const T& value);
    bool find_node (BinTree tree, const T& value, unsigned& compares) const;
};

#include "BSTree.cpp"

#endif
//---------------------------------------------------------------------------
```

BSTree.cpp

```cpp
/******************************************************************************/
/*!
\file:      BSTree.cpp
\author:    Goh Wei Zhe, weizhe.goh, 440000119
\par email: weizhe.goh\@digipen.edu
\date:      March 12, 2021
\brief      This file contains the definitions needed to implement the simple
            API for Binary Search Tree with recursive algorithms.

Copyright (C) 2021 DigiPen Institute of Technology.
Reproduction or disclosure of this file or its contents without the
prior written consent of DigiPen Institute of Technology is prohibited.
*/
/******************************************************************************/

#include "BSTree.h"

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

\brief  Constructor of BSTree

\param  OA - Object allocator for the BSTree nodes

\param  ShareOA - boolean for sharing object allocator
*/
/******************************************************************************/
template<typename T>
BSTree<T>::BSTree(ObjectAllocator *OA, bool ShareOA): oa{OA}, share{ShareOA},
root_{0}
{
    if(OA)
        Custom_OA = false;
    else
    {
        OAConfig config(true);
        oa = new ObjectAllocator(sizeof(BinTreeNode), config);
        Custom_OA = true;
    }
}

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

\brief  Copy constructor of BSTree

\param  rhs - The BSTree to be copied from
*/
/******************************************************************************/
template<typename T>
BSTree<T>::BSTree(const BSTree& rhs)
{
    if(rhs.share)
    {
        oa = rhs.oa;
        Custom_OA = false;
        share = true;
    }
    else
    {
        OAConfig config(true);
        oa = new ObjectAllocator(sizeof(BinTreeNode), config);
        Custom_OA = true;
        share = false;
    }

    copy_tree(root_, rhs.root_);

    root_->count = rhs.root()->count;
}

/******************************************************************************/
/*!
\fn     template<typename T>
        BSTree<T>::~BSTree()

\brief  Destructor of BSTree
*/
/******************************************************************************/
template<typename T>
BSTree<T>::~BSTree()
{
    clear();

    //false
    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 to be copied from

\return Returns the assigned tree itself
*/
/******************************************************************************/
template<typename T>
BSTree<T>& BSTree<T>::operator=(const BSTree& rhs)
```

```
{

    if(this == &rhs)
        return *this;

     if(rhs.share)
    {
        oa = rhs.oa;
        Custom_OA = false;
        share = true;
    }
    else
    {
        OAConfig config(true);
        delete oa;
        oa = new ObjectAllocator(sizeof(BinTreeNode), config);
        Custom_OA = true;
        share = false;
    }

    if(rhs.root())
    {
        clear();
        copy_tree(root_, rhs.root_);
    }

    return *this;
}

/******************************************************************************/
/*!
\fn     template <typename T>
        typename BSTree<T>::BinTree BSTree<T>::copy_tree(BinTree& destination,
        const BinTree& source)

\brief  Helper function to construct copy tree by recursion

\param  destination - copied BSTree

\param  source - BSTree to be copied from
*/
/******************************************************************************/
template <typename T>
typename BSTree<T>::BinTree BSTree<T>::copy_tree(BinTree& destination,
const BinTree& source)
{

    if(!source)
        return nullptr;

    destination = make_node(source->data);

    destination->balance_factor = source->balance_factor;

    destination->count = source->count;

    destination->left = copy_tree(destination->left, source->left);
    destination->right = copy_tree(destination->right, source->right);

    return destination;
}

/******************************************************************************/
/*!
\fn     template<typename T>
        const typename BSTree<T>::BinTreeNode* BSTree<T>::operator[](int index)
        const

\brief  subscript operator to search in the tree

\param  index - the index to match

\return Returns 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 tree,
        int compares) const

\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 Returns 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 temp = (tree->left) ? tree->left->count : 0;

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

/******************************************************************************/
/*!
\fn     template <typename T>
        void BSTree<T>::insert(const T& value)

\brief  Insert a node into the tree

\param  value - the value of node to be inserted
*/
/******************************************************************************/
template <typename T>
void BSTree<T>::insert(const T& value)
{
    insert_node(root_, value);
}

/******************************************************************************/
/*!
\fn     template<typename T>
        void BSTree<T>::insert_node(BinTree& tree, const T& value)

\brief  Helper function to insert a node into tree by recusion

\param  tree - The tree for node to be inserted

\param  value - the value of node to be inserted
*/
/******************************************************************************/
template<typename T>
void BSTree<T>::insert_node(BinTree& tree, const T& value)
{
    try
    {
        if(!tree)
        {
            tree = make_node(value);
        }
        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, duplicated item" << std::endl;
        }
    }
    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 node based on input value

\param  value - The value of node to be removed
*/
/******************************************************************************/
template <typename T>
void BSTree<T>::remove(const T& value)
{
    delete_node(root_, value);
}

/******************************************************************************/
/*!
\fn     template<typename T>
        void BSTree<T>::clear()

\brief  Clear the tree's nodes
*/
/******************************************************************************/
template<typename T>
void BSTree<T>::clear()
{
    free_tree(root_);
}

/******************************************************************************/
/*!
\fn     template<typename T>
        void BSTree<T>::free_tree(BinTree& tree)

\brief  Helper function to clear tree nodes by recursion

\param  tree - The tree to be freed
*/
/******************************************************************************/
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>::delete_node(BinTree& tree, const T& value)

\brief  Helper function to delete node

\param  tree - the tree to remove node from

\param  value - The value of node to be removed
*/
/******************************************************************************/
template<typename T>
void BSTree<T>::delete_node(BinTree& tree, const T& value)
{
    if(tree == NULL)
        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
    {
        --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
        {
            BinTree pred = 0;
            find_predecessor(tree, pred);
            tree->data = pred->data;
            delete_node(tree->left, tree->data);
        }
    }
}

/******************************************************************************/
/*!
\fn     template<typename T>
        bool BSTree<T>::find(const T& value, unsigned& compares) const

\brief  Find the node in the tree with the matching value

\param  value - the value of node to be found

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

\return Return true if node with matching value exist, else return false
*/
/******************************************************************************/
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& compares) const

\brief  Helper function to find a node in the tree with a matching value by
        recursion

\param  tree - the tree to be searched

\param  value - the value of node to be found

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

\return Return true if node with matching value exist, else return false
*/
/******************************************************************************/
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)
        return true;
    else if(value < tree->data)
        return find_node(tree->left, value, compares);
    else
        return find_node(tree->right, value, compares);
}

/******************************************************************************/
/*!
\fn     template<typename T>
        bool BSTree<T>::empty() const

\brief  Check if the tree is empty

\return Return true if tree is empty, else return false
*/
/******************************************************************************/
template<typename T>
bool BSTree<T>::empty() const
{
    return (root_ == 0);
}

/******************************************************************************/
/*!
\fn     template<typename T>
        unsigned int BSTree<T>::size() const

\brief  Counts the number of nodes in the tree

\return Returns the number of nodes in the tree
*/
/******************************************************************************/
template<typename T>
unsigned int BSTree<T>::size() const
{
    return (root_) ? root_->count : 0;
}

/******************************************************************************/
/*!
\fn     template<typename T>
        int BSTree<T>::height() const

\brief  Counts the height of a tree

\return Returns the height of a tree
*/
/******************************************************************************/
template<typename T>
int BSTree<T>::height() const
{
    return tree_height(root_);
}

/******************************************************************************/
/*!
\fn     template<typename T>
        typename BSTree<T>::BinTree BSTree<T>::root() const

\brief  Get the root of the tree

\return Returns the root of the tree
*/
/******************************************************************************/
template<typename T>
typename BSTree<T>::BinTree BSTree<T>::root() const
{
    return root_;
}

/******************************************************************************/
/*!
\fn     template <typename T>
        typename BSTree<T>::BinTree& BSTree<T>::get_root()

\brief  Get the root of the tree

\return Returns the root of the tree
*/
/******************************************************************************/
template <typename T>
typename BSTree<T>::BinTree& BSTree<T>::get_root()
{
    return root_;
}

/******************************************************************************/
/*!
\fn     template <typename T>
        typename BSTree<T>::BinTree BSTree<T>::make_node(const T& value) const

\brief  Make a new node for the tree

\param  value - the value of the new node

\return Returns the new node
*/
/******************************************************************************/
template <typename T>
typename BSTree<T>::BinTree BSTree<T>::make_node(const T& value) const
{
    try
    {
        BinTree memory = reinterpret_cast<BinTreeNode*>(oa->Allocate());
        BinTree new_node = new (memory) BinTreeNode(value);
        return new_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 be freed
*/
/******************************************************************************/
template <typename T>
void BSTree<T>::free_node(BinTree node)
{
    node->~BinTreeNode();
    oa->Free(node);
}

/******************************************************************************/
/*!
\fn     template <typename T>
        int BSTree<T>::tree_height(BinTree tree) const

\brief  Helper function to find the height of a tree by recursion

\param  tree - The tree to be counted

\return Returns the height of the tree
*/
/******************************************************************************/
template <typename T>
int BSTree<T>::tree_height(BinTree tree) const
{
    if(tree == 0)
        return -1;

    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>
        void BSTree<T>::find_predecessor(BinTree tree, BinTree& predecessor)
        const

\brief  Finds the parent of a node

\param  tree - the node to be searched

\param  predecessor - the node to fill as predecessor
*/
/******************************************************************************/
template <typename T>
void BSTree<T>::find_predecessor(BinTree tree, BinTree& predecessor) const
{
```

VPL

◄ Bonus Assignment: Sudoku

Jump to...

Introduction to Data Structures ►

You are logged in as Wei Zhe GOH (Log out)
cs280s21-b.sg
Data retention summary
Get the mobile app