cs380su21-meta.sg

Dashboard / My courses / cs380su21-meta.sg / 24 May - 30 May / Assignment 3 (Tree + Flood-Fill)

Description Submission </> Edit Submission view

Grade

Reviewed on Tuesday, 1 June 2021, 2:55 AM by Automatic grade
grade: 100.00 / 100.00

Assessment report [-]
[+]**Summary of tests**

Submitted on Tuesday, 1 June 2021, 2:55 AM (Download)

functions.cpp

```
/*!****************************************************************************
\file functions.cpp
\author Vadim Surov, Goh Wei Zhe
\par DP email: vsurov\@digipen.edu, weizhe.goh\@digipen.edu
\par Course: CS380
\par Section: B
\par Programming Assignment 3
\date 05-31-2021
\brief
This file has declarations and definitions that are required for submission
******************************************************************************/
#include "functions.h"

namespace AI
{


}
```

functions.h

```cpp
/*!*****************************************************************************
\file functions.h
\author Vadim Surov, Goh Wei Zhe
\par DP email: vsurov\@digipen.edu, weizhe.goh\@digipen.edu
\par Course: CS380
\par Section: B
\par Programming Assignment 3
\date 05-31-2021
\brief
This file has declarations and definitions that are required for submission
*******************************************************************************/
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#include <iostream>
#include <vector>
#include <list>
#include <stack>
#include <string>
#include <algorithm>

#include "data.h"

#define UNUSED(x) (void)x;

namespace AI
{
    // A simple graph node definition with serialization functions
    template<typename T>
    struct Node
    {
        // Member data
        T value;
        Node* parent;
        std::list<Node*> children;

        Node(T value = {}, Node* parent = nullptr,
            const std::list<Node*>& children = {})
            : value{ value }, parent{ parent }, children{ children }{}

        ~Node()
        {
            for (auto child : children)
                delete child;
        }

        /*!*********************************************************************
        \brief
        An overloading insertion operator function that takes and return a
        stream object.

        \param os
        Output stream to perform output.

        \param rhs
        Right hand side object.

        \return
        Returns the output through ostream.
        ***********************************************************************/
        friend std::ostream& operator<<(std::ostream& os, Node const& rhs)
        {
            Serialization(os, &rhs);
            return os;
        }

        /*!*********************************************************************
        \brief
        Serialization. A recursive function to print output.

        \param os
        Output stream to perform output.

        \param rhs
        Right hand side object.

        \return
        Returns the output through ostream.
        ***********************************************************************/
        static void Serialization(std::ostream& os, const Node* rhs)
        {
            os <<rhs->value + " {" + std::to_string(rhs->children.size()) + " ";

            //loop through each node in children's list
            for (Node* n : rhs->children)
                Serialization(os, n);

            os << "} ";
        }

        /*!*********************************************************************
        \brief
        An operator overloading function to handle input streams and return an
        istream object.

        \param is
        Input stream to read inputs.

        \param rhs
        Right hand side object.

        \return
        Returns the input through istream.
        ***********************************************************************/
        friend std::istream& operator>>(std::istream& is, Node& rhs)
        {
            is >> rhs.value;
            Deserialization(is, &rhs);
    
            return is;
        }

        /*!*************************************************************************
        \brief
        Deserialization. A recursive function to read input.

        \param is
        Input stream to read inputs.

        \param rhs
        Right hand side object.

        \return
        None.
        *************************************************************************/
        static void Deserialization(std::istream& is, Node* rhs)
        {
            std::string s;
            while (is >> s)
            {
                if (s.find("{") != std::string::npos)
                {
                    const char* stringToInt = &s[1];
                    int numChild = std::atoi(stringToInt);

                    //For each children, check if children has a child
                    for (int i = 0; i < numChild; ++i)
                    {
                        Node* child = new Node;
                        is >> s;
                        child->parent = rhs;
                        child->value = s;

                        rhs->children.push_back(child);
                        Deserialization(is, child);
                    }
                }
                else if (s.find("}") != std::string::npos)
                {
                    return;
                }
            }
        }

        /*!*************************************************************************
        \brief
        Function to get path from tree root to current node.

        \param
        None.

        \return
        Returns values from root to this node as an array.
        *************************************************************************/
        std::vector<T> getPath() const
        {
            std::vector<T> r;

            Node* node = this->parent;
            while (node)
            {
                r.push_back(node->value);
                node = node->parent;
            }
            std::reverse(r.begin(), r.end());
            return r;
        }
    };

    // The actual node type for this assignment
    using TreeNode = Node<std::string>;

    // Abstract base class for domain specific functors that return adjacent
    //nodes
    class GetAdjacents
    {
    public:

        virtual ~GetAdjacents(){}

        //virtual std::vector<Node*> operator()(Key key) = 0;
        virtual std::vector<TreeNode*> operator()(TreeNode* pNode) = 0;

        /*!*************************************************************************
        \brief
        Set value of a tree node.

        \param pNode
        The tree node to be assigned.

        \param value
        value of tree node.

        \return
        None.
        *************************************************************************/
        void setValue(TreeNode* pNode, std::string value)
        {
            pNode->value = value;
        }
    };

    // Domain specific functor that returns adjacent nodes
    class GetTreeAdjacents : public GetAdjacents
    {
    public:
```

```
public:

    GetTreeAdjacents()
        : GetAdjacents(){}

    /*!*********************************************************************
    \brief
    An Operator Overloading function that finds all adjcent nodes that has
    specific value and insert into an array vector of nodes and return it.

    \param pNode
    Tree node to search from.

    \return
    Returns an array vector of tree nodes.
    **********************************************************************/
    std::vector<TreeNode*> operator()(TreeNode* pNode)
    {
        std::vector<AI::TreeNode*> list = {};

        // Push to the list all children of pNode equal to value "x"
        for (auto x : pNode->children)
        {
            if (x->value.find("x") != std::string::npos)
                list.push_back(x);
        }

        return list;
    }
};

// Domain specific functor that returns shuffled adjacent nodes
class GetTreeStochasticAdjacents : public GetTreeAdjacents
{
public:

    GetTreeStochasticAdjacents()
        : GetTreeAdjacents{}{}

    /*!**********************************************************************
    \brief
    An Operator Overloading function that finds all adjcent nodes that has
    specific value and insert into an array vector of nodes, then shuffles
    the result and return it.

    \param pNode
    The tree node to search from.

    \return
    Returns an array vector of tree nodes.
    **********************************************************************/
    std::vector<TreeNode*> operator()(TreeNode* pNode)
    {
        UNUSED(pNode)

        std::vector<TreeNode*> adjacents;

        // Use the base class operator() and then shuffle the result
        adjacents = GetTreeAdjacents::operator()(pNode);
        std::random_shuffle(adjacents.begin(), adjacents.end());

        return adjacents;
    }
};

// Wrappers that provide same interface for queue and stack
struct Interface
{
    virtual void clear() = 0;
    virtual void push(TreeNode* pNode) = 0;
    virtual TreeNode* pop() = 0;
};

struct Queue : Interface
{
    std::vector<TreeNode*> Q;
    int count = 0;

    /*!*********************************************************************
    \brief
    Clears all tree nodes in the vector array.

    \param
    None.

    \return
    None.
    **********************************************************************/
    void clear()
    {
        Q.clear();
        count = 0;
    }

    /*!*********************************************************************
    \brief
    Add tree nodes by pushing back to the vector array.

    \param
    None.

    \return
    None.
    **********************************************************************/
    void push(TreeNode* pNode)
    {
        Q.push_back(pNode);
        ++count;
    }
```

```
        }

        /*!**********************************************************************
        \brief
        Remove tree nodes by popping it from the back of the vector array.

        \param
        None.

        \return
        None.
        ***********************************************************************/
        TreeNode* pop()
        {
            TreeNode* pNode = nullptr;

            pNode = Q.back();
            Q.pop_back();
            --count;

            return pNode;
        }

        /*!**********************************************************************
        \brief
        Check if the vector array is empty.

        \param
        None.

        \return
        Returns true if vector is empty, else returns false.
        ***********************************************************************/
        bool empty()
        {
            return (count == 0) ? true : false;
        }
    };

    struct Stack : Interface
    {
        std::vector<TreeNode*> Stack;
        int count = 0;

        /*!**********************************************************************
        \brief
        Clears all tree nodes in the vector array.

        \param
        None.

        \return
        None.
        ***********************************************************************/
        void clear()
        {
            Stack.clear();
            count = 0;
        }

        /*!**********************************************************************
        \brief
        Add nodes by pushing back to the vector array.

        \param
        None.

        \return
        None.
        ***********************************************************************/
        void push(TreeNode* pNode)
        {
            Stack.push_back(pNode);
            ++count;
        }

        /*!**********************************************************************
        \brief
        Remove tree nodes by popping it from the back of the vector array.

        \param
        None.

        \return
        None.
        ***********************************************************************/
        TreeNode* pop()
        {
            TreeNode* pNode = nullptr;

            pNode = Stack.back();
            Stack.pop_back();
            --count;

            return pNode;
        }

        /*!**********************************************************************
        \brief
        Check if the vector array is empty.

        \param
        None.

        \return
        Returns true if vector is empty, else returns false.
        ***********************************************************************/
        bool empty()
        {
```

```
        {
            return (count == 0) ? true : false;
        }
    };

    // Recursive Flood Fill
    class Flood_Fill_Recursive
    {
        GetTreeAdjacents* pGetAdjacents;

    public:
        Flood_Fill_Recursive(GetTreeAdjacents* pGetAdjacents)
            : pGetAdjacents{ pGetAdjacents }
        {
        }
        /*!*************************************************************************
        \brief
        Implement Recursive Flood Fill Algorithm.

        \param pNode
        The tree node to search from.

        \param value
        Value of tree node.

        \return
        None.
        *************************************************************************/
        void run(TreeNode* pNode, std::string value)
        {
            // Implement the flood fill
            std::vector<TreeNode*> adjcentlist
            = this->pGetAdjacents->operator()(pNode);

            for (auto& adj : adjcentlist)
            {
                GetTreeAdjacents* treeAjd =
                    dynamic_cast<GetTreeAdjacents*>(this->pGetAdjacents);

                treeAjd->setValue(adj, value);
                this->run(adj, value);
            }
        }
    };

    // Iterative Flood Fill
    // Type T defines is it depth- or breadth-first
    template<typename T>
    class Flood_Fill_Iterative
    {
        GetTreeAdjacents* pGetAdjacents;
        T openlist;

    public:
        Flood_Fill_Iterative(GetTreeAdjacents* pGetAdjacents)
            : pGetAdjacents{ pGetAdjacents }, openlist{}{}

        /*!*************************************************************************
        \brief
        Implement Iterative Flood Fill Algorithm, depth or breadth-first.

        \param pNode
        Tree Node to search from.

        \param value
        Value of Tree Node.

        \return
        None.
        *************************************************************************/
        void run(TreeNode* pNode, std::string value)
        {
            // Implement the flood fill
            openlist.clear();
            openlist.push(pNode);

            while (!openlist.empty())
            {
                TreeNode* current = openlist.pop();
                std::vector<TreeNode*> adjcentlist =
                    this->pGetAdjacents->operator()(current);

                for (auto& adj : adjcentlist)
                {
                    GetTreeAdjacents* treeAjd =
                        dynamic_cast<GetTreeAdjacents*>(this->pGetAdjacents);

                    treeAjd->setValue(adj, value);
                    this->run(adj, value);
                }
            }
         }
    };

} // end namespace

#endif
```

VPL

◄ Slides

Jump to...

Attendance cs380su21-meta.sg Friday 4/6/2021 9:00am-12:20pm ►

You are logged in as Wei Zhe GOH (Log out)
cs380su21-meta.sg
Data retention summary
Get the mobile app