cs380su21-meta.sg

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

```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);
```

```
109
110         return is;
111     }
112
113     /*!*************************************************************************
114     \brief
115     Deserialization. A recursive function to read input.
116
117     \param is
118     Input stream to read inputs.
119
120     \param rhs
121     Right hand side object.
122
123     \return
124     None.
125     *************************************************************************/
126     static void Deserialization(std::istream& is, Node* rhs)
127     {
128         std::string s;
129         while (is >> s)
130         {
131             if (s.find("{") != std::string::npos)
132             {
133                 const char* stringToInt = &s[1];
134                 int numChild = std::atoi(stringToInt);
135
136                 //For each children, check if children has a child
137                 for (int i = 0; i < numChild; ++i)
138                 {
139                     Node* child = new Node;
140                     is >> s;
141                     child->parent = rhs;
142                     child->value = s;
143
144                     rhs->children.push_back(child);
145                     Deserialization(is, child);
146                 }
147             }
148             else if (s.find("}") != std::string::npos)
149             {
150                 return;
151             }
152         }
153     }
154
155     /*!*************************************************************************
156     \brief
157     Function to get path from tree root to current node.
158
159     \param
160     None.
161
162     \return
163     Returns values from root to this node as an array.
164     *************************************************************************/
165     std::vector<T> getPath() const
166     {
167         std::vector<T> r;
168
169         Node* node = this->parent;
170         while (node)
171         {
172             r.push_back(node->value);
173             node = node->parent;
174         }
175         std::reverse(r.begin(), r.end());
176         return r;
177     }
178 };
179
180 // The actual node type for this assignment
181 using TreeNode = Node<std::string>;
182
183 // Abstract base class for domain specific functors that return adjacent
184 //nodes
185 class GetAdjacents
186 {
187 public:
188
189     virtual ~GetAdjacents(){}
190
191     //virtual std::vector<Node*> operator()(Key key) = 0;
192     virtual std::vector<TreeNode*> operator()(TreeNode* pNode) = 0;
193
194     /*!*************************************************************************
195     \brief
196     Set value of a tree node.
197
198     \param pNode
199     The tree node to be assigned.
200
201     \param value
202     value of tree node.
203
204     \return
205     None.
206     *************************************************************************/
207     void setValue(TreeNode* pNode, std::string value)
208     {
209         pNode->value = value;
210     }
211 };
212
213 // Domain specific functor that returns adjacent nodes
214 class GetTreeAdjacents : public GetAdjacents
215 {
216 public:
```

```
217
218         GetTreeAdjacents()
219             : GetAdjacents(){}
220
221         /*!*************************************************************************
222         \brief
223         An Operator Overloading function that finds all adjcent nodes that has
224         specific value and insert into an array vector of nodes and return it.
225
226         \param pNode
227         Tree node to search from.
228
229         \return
230         Returns an array vector of tree nodes.
231         *************************************************************************/
232         std::vector<TreeNode*> operator()(TreeNode* pNode)
233         {
234             std::vector<AI::TreeNode*> list = {};
235
236             // Push to the list all children of pNode equal to value "x"
237             for (auto x : pNode->children)
238             {
239                 if (x->value.find("x") != std::string::npos)
240                     list.push_back(x);
241             }
242
243             return list;
244         }
245     };
246
247     // Domain specific functor that returns shuffled adjacent nodes
248     class GetTreeStochasticAdjacents : public GetTreeAdjacents
249     {
250     public:
251
252         GetTreeStochasticAdjacents()
253             : GetTreeAdjacents{}{}
254
255         /*!*************************************************************************
256         \brief
257         An Operator Overloading function that finds all adjcent nodes that has
258         specific value and insert into an array vector of nodes, then shuffles
259         the result and return it.
260
261         \param pNode
262         The tree node to search from.
263
264         \return
265         Returns an array vector of tree nodes.
266         *************************************************************************/
267         std::vector<TreeNode*> operator()(TreeNode* pNode)
268         {
269             UNUSED(pNode)
270
271             std::vector<TreeNode*> adjacents;
272
273             // Use the base class operator() and then shuffle the result
274             adjacents = GetTreeAdjacents::operator()(pNode);
275             std::random_shuffle(adjacents.begin(), adjacents.end());
276
277             return adjacents;
278         }
279     };
280
281     // Wrappers that provide same interface for queue and stack
282     struct Interface
283     {
284         virtual void clear() = 0;
285         virtual void push(TreeNode* pNode) = 0;
286         virtual TreeNode* pop() = 0;
287     };
288
289     struct Queue : Interface
290     {
291         std::vector<TreeNode*> Q;
292         int count = 0;
293
294         /*!*************************************************************************
295         \brief
296         Clears all tree nodes in the vector array.
297
298         \param
299         None.
300
301         \return
302         None.
303         *************************************************************************/
304         void clear()
305         {
306             Q.clear();
307             count = 0;
308         }
309
310         /*!*************************************************************************
311         \brief
312         Add tree nodes by pushing back to the vector array.
313
314         \param
315         None.
316
317         \return
318         None.
319         *************************************************************************/
320         void push(TreeNode* pNode)
321         {
322             Q.push_back(pNode);
323             ++count;
324         }
```

```
325
326     /*!*************************************************************************
327     \brief
328     Remove tree nodes by popping it from the back of the vector array.
329
330     \param
331     None.
332
333     \return
334     None.
335     *************************************************************************/
336     TreeNode* pop()
337     {
338         TreeNode* pNode = nullptr;
339
340         pNode = Q.back();
341         Q.pop_back();
342         --count;
343
344         return pNode;
345     }
346
347     /*!*************************************************************************
348     \brief
349     Check if the vector array is empty.
350
351     \param
352     None.
353
354     \return
355     Returns true if vector is empty, else returns false.
356     *************************************************************************/
357     bool empty()
358     {
359         return (count == 0) ? true : false;
360     }
361 };
362
363 struct Stack : Interface
364 {
365     std::vector<TreeNode*> Stack;
366     int count = 0;
367
368     /*!*************************************************************************
369     \brief
370     Clears all tree nodes in the vector array.
371
372     \param
373     None.
374
375     \return
376     None.
377     *************************************************************************/
378     void clear()
379     {
380         Stack.clear();
381         count = 0;
382     }
383
384     /*!*************************************************************************
385     \brief
386     Add nodes by pushing back to the vector array.
387
388     \param
389     None.
390
391     \return
392     None.
393     *************************************************************************/
394     void push(TreeNode* pNode)
395     {
396         Stack.push_back(pNode);
397         ++count;
398     }
399
400     /*!*************************************************************************
401     \brief
402     Remove tree nodes by popping it from the back of the vector array.
403
404     \param
405     None.
406
407     \return
408     None.
409     *************************************************************************/
410     TreeNode* pop()
411     {
412         TreeNode* pNode = nullptr;
413
414         pNode = Stack.back();
415         Stack.pop_back();
416         --count;
417
418         return pNode;
419     }
420
421     /*!*************************************************************************
422     \brief
423     Check if the vector array is empty.
424
425     \param
426     None.
427
428     \return
429     Returns true if vector is empty, else returns false.
430     *************************************************************************/
431     bool empty()
432     {
```

```
        {
            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 ►