SLL

Generated by Doxygen 1.10.0

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 Node< T > Class Template Reference

A class representing node.

```
#include <classes.h>
```

Public Member Functions

- **Node** ()

  A default constructor.
- Node (T data)

  A constructor.
- ∼**Node** ()

  Destructor.

Public Attributes

- T **data**

  Variable which holds data stored in the node.
- std::shared_ptr< Node< T > > **next**

  Pointer to another node.

3.1.1 Detailed Description

template<typename T>
class Node< T >

A class representing node.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 Node()

```
template<typename T >
Node< T >::Node (
            T data )  [inline]
```

A constructor.

Parameters

data	information to be stored in the node.

The documentation for this class was generated from the following file:

- C:/source/repos/3831c825-gr02-repo/Project/SLL_proj/classes.h

3.2 Person Struct Reference

Public Attributes

- std::string **name**
- int **age**

The documentation for this struct was generated from the following file:

- C:/source/repos/3831c825-gr02-repo/Project/SLL_proj/SLL_proj.cpp

3.3 SinglyLinkedlist< T > Class Template Reference

A class representing a SLL.

```
#include <classes.h>
```

Public Member Functions

- **SinglyLinkedlist** ()

  Default constructor.
- **SinglyLinkedlist** (SinglyLinkedlist &SLL)

  Copy constructor.
- **SinglyLinkedlist** (SinglyLinkedlist &&SLL)

  Move constructor.
- ∼**SinglyLinkedlist** ()

  Destructor.
- void insertNode (T data)

  A function which adds a new element to the container.
- void **print** ()

  A function which prints the whole list.
- void **operator=** (const SinglyLinkedlist &SLL)

  An overloaded assignment operator.
- SinglyLinkedlist & **operator=** (SinglyLinkedlist &&SLL)

  An overloaded move operator.
- std::shared_ptr< Node< T > > get (T val)

  A function searches for a specific element in the container.
- void **sort** ()

  A function sorting the list in ascending order, bubble sort algorithm is used.
- bool SaveToFile (std::string fname)

  A function which saves the current state of the structure to a file.
- bool ReadFromFile (std::string fname)

  A function which loades state of the structure from a file.

3.3.1 Detailed Description

template<typename T>
class SinglyLinkedlist< T >

A class representing a SLL.

3.3.2 Member Function Documentation

3.3.2.1 get()

```
template<typename T >
std::shared_ptr< Node< T > > SinglyLinkedlist< T >::get (
             T val )  [inline]
```

A function searches for a specific element in the container.

Parameters

val	value of the searched element.

Returns

the searched element, if not presented the head is returned.

3.3.2.2 insertNode()

```
template<typename T >
void SinglyLinkedlist< T >::insertNode (
             T data )  [inline]
```

A function which adds a new element to the container.

Parameters

data	information to be stored in the new node.

3.3.2.3 ReadFromFile()

```
template<typename T >
bool SinglyLinkedlist< T >::ReadFromFile (
             std::string fname )  [inline]
```

A function which loades state of the structure from a file.

Parameters

fname	name of the file, where data is stored.

Returns

true if succesfuul, false if no.

3.3.2.4 SaveToFile()

```
template<typename T >
bool SinglyLinkedlist< T >::SaveToFile (
             std::string fname )  [inline]
```

A function which saves the current state of the structure to a file.

Parameters

fname	name of the file, where data will be stored.

Returns

true if succesfuul, false if no.

The documentation for this class was generated from the following file:

- C:/source/repos/3831c825-gr02-repo/Project/SLL_proj/classes.h

Chapter 4

File Documentation

4.1 C:/source/repos/3831c825-gr02-repo/Project/SLL_proj/classes.h

```
#pragma once
#ifndef CLASSES_H
#define CLASSES_H
#include <iostream>
#include <fstream>
#include <string>

    template <typename T>
    class Node
    {
    public:

        T data;
        std::shared_ptr< Node<T>> next;

        Node()
        {
            data = 0;
            next = nullptr;
        }


        Node(T data)
        {
            this->data = data;
            this->next = nullptr;
        }

        ~Node() //destructor
        {

        };


    };

    template <typename T>
    class SinglyLinkedlist
    {
        std::shared_ptr<Node<T>> head;

    public:

        SinglyLinkedlist() { head = nullptr; };

        SinglyLinkedlist(SinglyLinkedlist& SLL)
        {
            if (SLL.head != nullptr)
            {
                this->head = std::shared_ptr<Node<T>>(new Node<T>);
                this->head->data = SLL.head->data;
                std::shared_ptr<Node<T>> t(SLL.head->next);
                std::shared_ptr<Node<T>> t1(this->head);
                while (t != nullptr)
                {
                    std::shared_ptr<Node<T>> NewNode(new Node<T>);
                    NewNode->data = t->data;
                    t1->next = NewNode;
                    t1 = t1->next;
                    t = t->next;
                }
            }
            else
            {
                head = nullptr;
            }
        };

        SinglyLinkedlist(SinglyLinkedlist&& SLL) //move constructor
        {
            std::shared_ptr< Node<T>> t = SLL.head;

            if (SLL.head != nullptr)
            {
                this->head = std::shared_ptr<Node<T>>(new Node<T>);
                this->head->data = SLL.head->data;
                std::shared_ptr<Node<T>> t(SLL.head->next);
                std::shared_ptr<Node<T>> t1(this->head);
                while (t != nullptr)
                {
                    std::shared_ptr<Node<T>> NewNode(new Node<T>);
                    NewNode->data = t->data;
                    t1->next = NewNode;
                    t1 = t1->next;
                    t = t->next;
                }
                SLL.head = nullptr;
            }
            else
            {
                head = nullptr;
            }

        };

        ~SinglyLinkedlist() //destructor
        {

        };


        void insertNode(T data)
        {
            std::shared_ptr<Node<T>> NewNode(new Node<T>(data));

            if (head == nullptr)
            {
                head = NewNode;
                return;
            }

            std::shared_ptr<Node<T>> t(head);
            while (t->next != nullptr)
            {
                t = t->next;
            }
            t->next = NewNode;
        };

        void print()
        {
            std::shared_ptr<Node<T>> t(head);

            if (head == nullptr)
            {
                std::cout << "List is empty" << std::endl;
                return;
            }

            while (t != nullptr)
            {
                std::cout << t->data << " ";
                t = t->next;
            }
            std::cout << std::endl;
        };

        void operator=(const SinglyLinkedlist& SLL)
        {
            if (SLL.head != nullptr) {
                this->head = std::shared_ptr<Node<T>>(new Node<T>);
                this->head->data = SLL.head->data;
                std::shared_ptr<Node<T>> t(SLL.head->next);
                std::shared_ptr<Node<T>> t1(this->head);
                while (t != nullptr) {
                    std::shared_ptr< Node<T>> NewNode(new Node<T>);
                    NewNode->data = t->data;
                    t1->next = NewNode;
                    t1 = t1->next;
                    t = t->next;
                }
            }
            else
            {
                head = nullptr;
            }
        };

        SinglyLinkedlist& operator=(SinglyLinkedlist&& SLL)
        {
            std::shared_ptr<Node<T>> t(SLL.head);
            if (SLL.head != nullptr)
            {
                this->head = std::shared_ptr<Node<T>>(new Node<T>);
                this->head->data = SLL.head->data;
                std::shared_ptr<Node<T>> t(SLL.head->next);
                std::shared_ptr<Node<T>> t1(this->head);
                while (t != nullptr)
                {
                    std::shared_ptr<Node<T>> NewNode(new Node<T>);
                    NewNode->data = t->data;
                    t1->next = NewNode;
                    t1 = t1->next;
                    t = t->next;
                }

                SLL.head = nullptr;
            }
            else
            {
                head = nullptr;
            }

            return *this;
        };


        std::shared_ptr<Node<T>> get(T val)
        {
            std::shared_ptr<Node<T>> t(head);

            if (head == nullptr)
            {
                std::cout « "List is empty" « std::endl;
                return nullptr;
            }

            while (t != nullptr)
            {
                if (t->data == val)
                {
                    return t;
                }
                t = t->next;
            }
            std::cout « "Element is not presented" « std::endl;
            return head;
        };


        void sort()
        {
            std::shared_ptr<Node<T>> t(head);
            std::shared_ptr<Node<T>> c1(head);
            std::shared_ptr<Node<T>> c2(head);
            int l = 0;

            while (t != nullptr)
            {
                t = t->next;
                l++;
            }

            for (int i = 0; i < l; i++)
            {
                for (int j = 0; j < l - 1; j++)
                {
                    if (c1->data < c2->data)
                    {
                        std::swap(c1->data, c2->data);
                    }

                    c2 = c2->next;
                }

                c2 = head;
                c1 = c2->next;
                for (int k = 0; k < i; k++)
                {
                    c1 = c1->next;
                }
            }

        };


        bool SaveToFile(std::string fname)
        {
            std::ofstream fout(fname, std::ios::out | std::ios::binary);

            std::shared_ptr<Node<T» t(head);
            int l = 0;
            while (t != nullptr)
            {
                t = t->next;
                l++;
            }

            t = head;

            if (!fout)
            {
                return false;
            }

            fout.write((char*)&l, sizeof(l));

            while (t != nullptr)
            {
                fout.write((char*)&t->data, sizeof(t->data));
                t = t->next;
            }

            fout.close();
            return true;
        };


        bool ReadFromFile(std::string fname)
        {
            std::ifstream fin(fname, std::ios::in | std::ios::binary);
            int l;
            T buffer;

            if (!fin)
            {
                return false;
            }

            head = nullptr;

            fin.read((char*)&l, sizeof(int));

            for (int i = 0; i < l; i++)
            {
                fin.read((char*)&buffer, sizeof(T));
                insertNode(buffer);

            }

            fin.close();

            return true;
        };
    };

#endif
```

Index