

Project Report

Implementation of a custom single
linked list class

Author: Youssef AL BALI.

Teacher: Dr. Inż. Agnieszka Danek

Date: 12 January 2022

Analysis of the problem:

The aim of the microproject is to implement a custom single linked list in the form of a class with a convenient programming interface.

The class should be implemented as a template, and an iterator that will allow the use of STL should be implemented as well.

Requirements:

- usage of smart pointers,
- default constructor,
- copy / move constructors,
- destructor,
- assignment / move operators,
- adding container elements,
 - searching for an element,
- sorting content (different criteria),
- serialization / deserialization (reading and writing from byte stream).

External Specification:

The class can be declared and initialized with different data types as the following:

```
SingleLinkedList<int> s_int{1,2,5,4,80,100};  
SingleLinkedList<std::string> s_names{"Youssef", "Viktor", "Andrii"};  
SingleLinkedList<float> s_float{0.02, 12.1, 100.001, 25.5};  
SingleLinkedList<char> s_chars{'a', 'b', 'C', 'F', 'g'};
```

The class supports the following methods:

the push_back and push_front functions.

```
///Adding a new node at the the end of the list.  
s_int.push_back( data: 3);  
s_int.push_back( data: 5);  
  
///Adding a new node at the the back of the list.  
s_int.push_front( data: 100);  
s_int.push_front( data: 258);
```

As well as the pop_back and pop_front methods:

```
//Deleting the node at the beginning of the list.  
s_names.pop_front();  
  
//Deleting the node at the end of the list.  
s_names.pop_back();
```

Insert function:

```
//inserts the node at a certain index.  
s_int.insert( data: 5, pos: 1);
```

Two delete functions (First one is deleting node based on Data so it's more like filtering and the second takes the position of the node as a parameter):

```
//filters the list based on some data  
s_int.deleteByData( data: 7);  
  
//Deletes a node at a certain index  
s_int.deleteByNode( position: 2);
```

Function that prints all the elements of the list and the overloaded output operator:

```
//Printing the elements using the print function  
s_int1.print();  
  
//Printing using the output overloaded operator  
std::cout << s_int1;
```

Input operator used to read data from a file:

```
SingleLinkedList<std::string> s_string_2;  
  
std::fstream inFile( s: "input.txt");  
  
inFile >> s_string_2;
```

Search function:

```
//Searching for an element in the list  
  
std::cout << "Is Youssef in the list? " << " " << s_names.search( data: "Youssef") << std::endl;
```

Clear function:

```
//deletes all the elements of the list.  
  
s_int.clear();
```

We can also sort the elements of the container in ascending order:

```
//Sorting the integers in ascending order  
s_int.sort();  
  
//Sorting the strings in ascending order based on their lengths  
s_names.sort();
```

The container has also been tested using the “Student” class in order to sort this type by grade.

```
struct Student {  
    int age;  
    std::string name;  
    double grade;  
  
    Student() = default;  
  
    Student(int _age, std::string _name, double _grade) {  
        age = _age;  
        name = _name;  
        grade = _grade;  
    }  
};
```

```
SingleLinkedList<Student> s_students{  
    Student( age: 19, name: "Youssef", grade: 5.0),  
    Student( age: 18, name: "Andrii", grade: 4.5),  
    Student( age: 30, name: "Asser", grade: 3.5),  
    Student( age: 17, name: "Viktor", grade: 5)  
};
```

```
s_students.sort();
```

And finally, a function that get the size of the list:

```
//Prints the size of the container  
std::cout<<s_int.size()<<"\n";
```

Internal Specification:

Everything concerning the internal Specification is included in the **Doxygen file** at the end of the report.

Testing:

The program was tested with many types of variables including Student class.

All the methods were tested and worked properly as expected.

There was a small issue concerning the insert function, when the index passed as parameter was out of range the program crashes so I decided to handle this by throwing an Exception that I made which derived from `std::exception` and it outputs the following message when the program throws the [OutOfRange](#) exception:

```
terminate called after throwing an instance of 'OutOfRange'
what():  The Index 100 Is out of range
```

Single Linked List

Generated by Doxygen 1.9.3

1 Class Index	1
1.1 Class List	1
2 File Index	3
2.1 File List	3
3 Class Documentation	5
3.1 Node< T > Struct Template Reference	5
3.1.1 Detailed Description	6
3.1.2 Constructor & Destructor Documentation	6
3.1.2.1 Node()	6
3.2 S_Iterator< T > Class Template Reference	6
3.2.1 Detailed Description	8
3.2.2 Constructor & Destructor Documentation	8
3.2.2.1 S_Iterator()	8
3.2.3 Member Function Documentation	8
3.2.3.1 operator!=(())	8
3.2.3.2 operator*()	9
3.2.3.3 operator+()	9
3.2.3.4 operator++() [1/2]	9
3.2.3.5 operator++() [2/2]	10
3.2.3.6 operator->()	10
3.2.3.7 operator==(())	10
3.2.3.8 operator[]()	11
3.3 SingleLinkedList< T > Class Template Reference	11
3.3.1 Detailed Description	14
3.3.2 Member Typedef Documentation	14
3.3.2.1 Iterator	14
3.3.3 Constructor & Destructor Documentation	14
3.3.3.1 SingleLinkedList() [1/3]	14
3.3.3.2 SingleLinkedList() [2/3]	14
3.3.3.3 SingleLinkedList() [3/3]	15
3.3.4 Member Function Documentation	15
3.3.4.1 deleteByData()	15
3.3.4.2 deleteByNode()	15
3.3.4.3 deserialize()	16
3.3.4.4 insert()	16
3.3.4.5 operator=(()) [1/2]	16
3.3.4.6 operator=(()) [2/2]	17
3.3.4.7 operator[]()	17
3.3.4.8 pop_back()	17
3.3.4.9 pop_front()	18
3.3.4.10 push_back()	18

3.3.4.11 push_front()	18
3.3.4.12 search()	18
3.3.4.13 serialize()	19
3.3.4.14 size()	19
3.3.4.15 sort()	19
3.3.5 Friends And Related Function Documentation	20
3.3.5.1 operator<<	20
3.3.5.2 operator>>	20
3.4 Student Struct Reference	21
4 File Documentation	23
4.1 SingleLinkedList.h	23
Index	29

Chapter 1

Class Index

1.1 Class List

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

Node< T >	
Struct Node	5
S_Iterator< T >	
Single linked list iterator	6
SingleLinkedList< T >	
Single Linked list class	11
Student	21

Chapter 2

File Index

2.1 File List

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

SingleLinkedList.h	??
------------------------------------	-------	----

Chapter 3

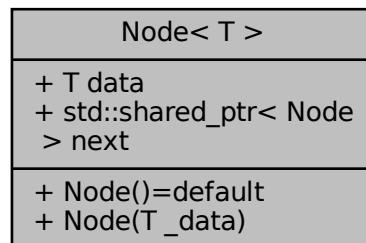
Class Documentation

3.1 Node< T > Struct Template Reference

struct [Node](#).

```
#include <SingleLinkedList.h>
```

Collaboration diagram for Node< T >:



Public Member Functions

- **Node** ()=default
default constructor
- [Node](#) (T _data)
One Argument Constructor that assigns the [Node](#) data.

Public Attributes

- **T data**
data of the [Node](#).
- std::shared_ptr< [Node](#) > **next** {}
pointer to the next node.

3.1.1 Detailed Description

```
template<class T>
struct Node< T >
```

struct [Node](#).

Represent a node of the single linked list.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 Node()

```
template<class T >
Node< T >::Node (
    T _data ) [inline]
```

One Argument Constructor that assigns the [Node](#) data.

Parameters

<code>_data</code>	Node data.
--------------------	----------------------------

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

- [SingleLinkedList.h](#)

3.2 S_Iterator< T > Class Template Reference

Single linked list iterator.

```
#include <SingleLinkedList.h>
```

Collaboration diagram for S_Iterator< T >:

S_Iterator< T >
- std::shared_ptr< Node < T > > Current_Node
+ S_Iterator()=default + S_Iterator(std::shared_ptr< Node< T > > ptr) + S_Iterator & operator++() + S_Iterator operator ++(int) + S_Iterator operator +(int i) + bool operator!=(const S_Iterator< T > &iterator) + bool operator==(const S_Iterator< T > &iterator) + T & operator[](int index) + T & operator*() + T * operator->()

Public Member Functions

- **S_Iterator** ()=default
Default constructor.
- **S_Iterator** (std::shared_ptr< **Node**< T > > ptr)
*Constructor that assigns the current **Node**.*
- **S_Iterator** & **operator++** ()
Pre-increment operator.
- **S_Iterator** **operator++** (int)
Post-increment operator.
- **S_Iterator** **operator+** (int i)
operator +
- bool **operator!=** (const **S_Iterator**< T > &iterator)
comparison operator.
- bool **operator==** (const **S_Iterator**< T > &iterator)
comparison operator.
- T & **operator[]** (int index)
Square bracket operator.
- T & **operator*** ()
Dereference operator.
- T * **operator->** ()
Pointer operator.

Private Attributes

- `std::shared_ptr< Node< T > > Current_Node`
Pointer to the Current [Node](#).

3.2.1 Detailed Description

```
template<class T>
class S_Iterator< T >
```

Single linked list iterator.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 S_Iterator()

```
template<class T >
S\_Iterator< T >::S_Iterator (
    std::shared_ptr< Node< T > > ptr ) [inline]
```

Constructor that assigns the current [Node](#).

Parameters

<i>ptr</i>	Node pointer.
------------	-------------------------------

3.2.3 Member Function Documentation

3.2.3.1 operator"!="()

```
template<class T >
bool S\_Iterator< T >::operator!= (
    const S\_Iterator< T > & iterator ) [inline]
```

comparison operator.

Parameters

<i>iterator</i>	the iterator to compare with.
-----------------	-------------------------------

Returns

True id the iterators are different, otherwise false

3.2.3.2 operator*()

```
template<class T >
T & S_Iterator< T >::operator* ( ) [inline]
```

Dereference operator.

Returns

The data of the element at which the iterator is pointing to.

3.2.3.3 operator+()

```
template<class T >
S_Iterator S_Iterator< T >::operator+ (
    int i ) [inline]
```

operator +

Advances the operator and returns the reference of the result

Returns

reference to the result

3.2.3.4 operator++() [1/2]

```
template<class T >
S_Iterator & S_Iterator< T >::operator++ ( ) [inline]
```

Pre-increment operator.

Increments the operator and returns the reference to the result

Returns

reference to the result

3.2.3.5 operator++() [2/2]

```
template<class T >
S_Iterator S_Iterator< T >::operator++ (
    int ) [inline]
```

Post-increment operator.

Increments the operator and returns the iterator before the incrementation

Returns

old iterator

3.2.3.6 operator->()

```
template<class T >
T * S_Iterator< T >::operator-> ( ) [inline]
```

Pointer operator.

Returns

Pointer to T

3.2.3.7 operator==()

```
template<class T >
bool S_Iterator< T >::operator== (
    const S_Iterator< T > & iterator ) [inline]
```

comparison operator.

Parameters

<i>iterator</i>	the iterator to compare with.
-----------------	-------------------------------

Returns

True id the iterators are different, otherwise false

3.2.3.8 operator[]()

```
template<class T >
T & S_Iterator< T >::operator[] (
    int index ) [inline]
```

Square bracket operator.

Parameters

<i>index</i>	index of the element starting from where the iterator is pointing to.
--------------	---

Returns

reference to the data

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

- SingleLinkedList.h

3.3 SingleLinkedList< T > Class Template Reference

Single Linked list class.

```
#include <SingleLinkedList.h>
```

Collaboration diagram for `SingleLinkedList< T >`:

SingleLinkedList< T >
- <code>std::shared_ptr< Node < T > > head</code> - <code>int SIZE</code>
+ <code>Iterator begin()</code> + <code>Iterator end()</code> + <code>SingleLinkedList()</code> + <code>SingleLinkedList(std ::initializer_list< T > init)</code> + <code>SingleLinkedList(Single LinkedList< T > &)</code> + <code>SingleLinkedList(Single LinkedList< T > &&) noexcept</code> + <code>SingleLinkedList & operator=(const SingleLinked List< T > &)</code> + <code>SingleLinkedList & operator=(SingleLinkedList < T > &&)</code> + <code>T & operator[](int index)</code> + <code>int size()</code> and 14 more...

Public Types

- typedef `S_Iterator< T > Iterator`
Size of the list.

Public Member Functions

- `Iterator begin ()`
End Iterator.
- `Iterator end ()`
- `SingleLinkedList ()`
Default constructor.
- `SingleLinkedList (std::initializer_list< T > init)`
- `SingleLinkedList (SingleLinkedList< T > &)`
Copy constructor.
- `SingleLinkedList (SingleLinkedList< T > &&) noexcept`
Move constructor.
- `SingleLinkedList & operator= (const SingleLinkedList< T > &)`
Copy operator.

- `SingleLinkedList & operator= (SingleLinkedList< T > &&)`

Move operator.

- `T & operator[] (int index)`

Square Bracket operator.

- `int size ()`

Function that return the size of a the list.

- `void push_back (T data)`

Push_back.

- `void push_front (T data)`

Push_front.

- `void pop_front ()`

Pop_Front.

- `void pop_back ()`

Pop_Front.

- `void deleteByNode (int position)`

Function that deletes a node at a certain index.

- `void insert (T data, int pos)`

Function that inserts a node at a certain index.

- `void deleteByData (T _data)`

Function the filters the list by data.

- `void print ()`

Function that prints the elements of the list.

- `bool search (T _data)`

Search function.

- `void sort ()`

Sort function.

- `void clear ()`

Function that deletes all the elements of the list.

- `void serialize (const std::string &filename)`

Serialize function.

- `void deserialize (const std::string &filename)`

Deserialize function.

- `void sort ()`

Private Attributes

- `std::shared_ptr< Node< T > > head {}`

Pointer to the head of the list.

- `int SIZE {}`

Friends

- `template<class s_type > std::ostream & operator<< (std::ostream &os, const SingleLinkedList< s_type > &s)`

Output stream operator.

- `template<class s_type > std::istream & operator>> (std::istream &input, SingleLinkedList< s_type > &s)`

Input stream operator.

3.3.1 Detailed Description

```
template<class T>
class SingleLinkedList< T >
```

Single Linked list class.

3.3.2 Member Typedef Documentation

3.3.2.1 Iterator

```
template<class T >
typedef S\_Iterator<T> SingleLinkedList< T >::Iterator
```

Size of the list.

Begin Iterator.

3.3.3 Constructor & Destructor Documentation

3.3.3.1 [SingleLinkedList\(\)](#) [1/3]

```
template<class T >
SingleLinkedList< T >::SingleLinkedList
```

Default constructor.

Setting the head to a Null-pointer When the list is created.

3.3.3.2 [SingleLinkedList\(\)](#) [2/3]

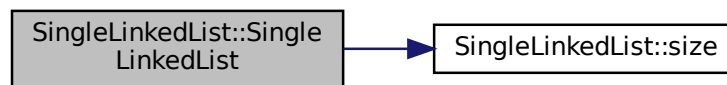
```
template<class T >
SingleLinkedList< T >::SingleLinkedList (
    SingleLinkedList< T > & s )
```

Copy constructor.

Parameters

s	A single linked list to copy from.
-------------------	------------------------------------

Here is the call graph for this function:



3.3.3.3 SingleLinkedList() [3/3]

```
template<class T >
SingleLinkedList< T >::SingleLinkedList (
    SingleLinkedList< T > && s ) [noexcept]
```

Move constructor.

Parameters

<code>s</code>	A single linked list to Move from.
----------------	------------------------------------

3.3.4 Member Function Documentation

3.3.4.1 deleteByData()

```
template<class T >
void SingleLinkedList< T >::deleteByData (
    T _data )
```

Function the filters the list by data.

Parameters

<code>_data</code>	Element to be deleted.
--------------------	------------------------

3.3.4.2 deleteByNode()

```
template<class T >
```

```
void SingleLinkedList< T >::deleteByNode (
    int position )
```

Function that deletes a node at a certain index.

Parameters

<i>position</i>	Index of the element to be deleted.
-----------------	-------------------------------------

3.3.4.3 deserialize()

```
template<class T >
void SingleLinkedList< T >::deserialize (
    const std::string & filename )
```

Deserialize function.

reads data in binary format

Parameters

<i>filename</i>	
-----------------	--

3.3.4.4 insert()

```
template<class T >
void SingleLinkedList< T >::insert (
    T data,
    int pos )
```

Function that inserts a node at a certain index.

Parameters

<i>position</i>	The Position where to add the element.
<i>data</i>	The element to be added.

3.3.4.5 operator=() [1/2]

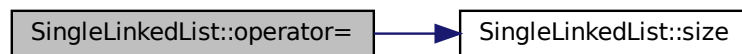
```
template<class T >
SingleLinkedList< T > & SingleLinkedList< T >::operator= (
    const SingleLinkedList< T > & other )
```

Copy operator.

Parameters

<i>other</i>	A single linked list to copy from.
--------------	------------------------------------

Here is the call graph for this function:



3.3.4.6 operator=() [2/2]

```
template<class T >
SingleLinkedList< T > & SingleLinkedList< T >::operator= (
    SingleLinkedList< T > && other )
```

Move operator.

Parameters

<i>other</i>	A single linked list to Move from.
--------------	------------------------------------

3.3.4.7 operator[]()

```
template<class T >
T & SingleLinkedList< T >::operator[] (
    int index ) [inline]
```

Square Bracket operator.

Parameters

<i>index</i>	The index of the element.
--------------	---------------------------

3.3.4.8 pop_back()

```
template<class T >
void SingleLinkedList< T >::pop_back
```

Pop_Front.

Deleting the node at the the end of the list.

3.3.4.9 pop_front()

```
template<class T >
void SingleLinkedList< T >::pop_front
```

Pop_Front.

Deleting the node at the beginning of the list.

3.3.4.10 push_back()

```
template<class T >
void SingleLinkedList< T >::push_back (
    T data )
```

Push_back.

Adding a new node at the the end of the list.

Parameters

<i>data</i>	The element to add.
-------------	---------------------

3.3.4.11 push_front()

```
template<class T >
void SingleLinkedList< T >::push_front (
    T data )
```

Push_front.

Adding a new node at the the beginning of the list.

Parameters

<i>data</i>	The element to add.
-------------	---------------------

3.3.4.12 search()

```
template<class T >
bool SingleLinkedList< T >::search (
    T _data )
```

Search function.

Parameters

<code>_data</code>	The element to search.
--------------------	------------------------

Returns

Returns true if the element exists, otherwise false.

3.3.4.13 `serialize()`

```
template<class T >
void SingleLinkedList< T >::serialize (
    const std::string & filename )
```

Serialize function.

stores data in binary format

Parameters

<code>filename</code>	
-----------------------	--

3.3.4.14 `size()`

```
template<class T >
int SingleLinkedList< T >::size ( ) [inline]
```

Function that return the size of a the list.

Returns

The size of the list.

3.3.4.15 `sort()`

```
template<class T >
void SingleLinkedList< T >::sort
```

Sort function.

Sorts the elements of the list using Bubble sort algorithm.

3.3.5 Friends And Related Function Documentation

3.3.5.1 operator<<

```
template<class T >
template<class s_type >
std::ostream & operator<< (
    std::ostream & os,
    const SingleLinkedList< s_type > & s ) [friend]
```

Output stream operator.

The operator is used to output the elements of the either to Console or to an output file

Parameters

<i>os</i>	std::ostream
<i>s</i>	Single linked list Object

Returns

std::ostream

3.3.5.2 operator>>

```
template<class T >
template<class s_type >
std::istream & operator>> (
    std::istream & input,
    SingleLinkedList< s_type > & s ) [friend]
```

Input stream operator.

The operator is used to read the elements of the list either to Console or to an output file.

Parameters

<i>is</i>	std::istream.
<i>s</i>	Single linked list Object.

Returns

std::istream.

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

- SingleLinkedList.h

3.4 Student Struct Reference

Collaboration diagram for Student:

Student
+ int age + std::string name + double grade
+ Student()=default + Student(int _Age, std::string _name, double _grade) + ~Student()=default + bool operator>(Student &st) const + bool operator<(Student &st) const

Public Member Functions

- **Student** (int _Age, std::string _name, double _grade)
- bool **operator>** ([Student](#) &st) const
- bool **operator<** ([Student](#) &st) const

Public Attributes

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

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

- main.cpp

Chapter 4

File Documentation

4.1 SingleLinkedList.h

```
1
2
3 #ifndef PROJECT_SINGLELINKEDLIST_H
4 #define PROJECT_SINGLELINKEDLIST_H
5
6 #include <iostream>
7 #include <memory>
8 #include <fstream>
9 #include "Exceptions.h"
10 #include <cstring>
11
12
13
14
15
16 template<class T>
17 struct Node {
18     T data;
19
20
21
22     std::shared_ptr<Node> next{};
23
24
25     Node() = default;
26
27
28     Node(T _data) {
29         data = _data;
30     }
31 };
32
33
34
35
36
37
38 template<class T>
39 class S_Iterator {
40 public:
41     S_Iterator() = default;
42
43
44
45
46     S_Iterator(std::shared_ptr<Node<T>> ptr) : Current_Node(ptr) {};
47
48
49
50
51     S_Iterator &operator++() {
52         this->Current_Node = Current_Node->next;
53         return *this;
54     }
55
56
57
58
59
60     S_Iterator operator++(int) {
61         return ++(*this);
62     }
63
64
65
66
67
68     S_Iterator operator+(int i) {
69         while (i != 0) { ;
70             (*this)++;
71             i--;
72         }
73         return *this;
74     }
75
76
77
78
```

```

80
83     bool operator!=(const S_Iterator<T> &iterator) {
84         return Current_Node != iterator.Current_Node;
85     }
86
87
88
91     bool operator==(const S_Iterator<T> &iterator) {
92         return Current_Node == iterator.Current_Node;
93     }
94
95
96
99     T &operator[](int index) {
100         auto tmp = Current_Node;
101         while (index != 0) {
102             tmp = tmp->m_nextNode;
103             index--;
104         }
105         return *tmp->m_data;
106     }
107
108
109
111     T &operator*() { return Current_Node->data; }
112
113
114
116     T *operator->() { return &Current_Node->data; } // returns the address to this data
117
118 private:
119     std::shared_ptr<Node<T>> Current_Node;
120 };
121
122
123 template<class T>
124 class SingleLinkedList {
125
126
127     std::shared_ptr<Node<T>> head{};
128     int SIZE{};
129 public:
130
131     typedef S_Iterator<T> Iterator;
132
133     Iterator begin() { return Iterator(head); }
134
135
136
137     Iterator end() { return Iterator(nullptr); }
138
139
140 public:
141
142
143
144
145
146
147
148     template<class s_type>
149     friend std::ostream &operator<<(std::ostream &os, const SingleLinkedList<s_type> &s);
150
151
152
153
154
155
156
157     template<class s_type>
158     friend std::istream &operator>>(std::istream &input, SingleLinkedList<s_type> &s);
159
160
161
162
163     SingleLinkedList();
164
165
166     SingleLinkedList(std::initializer_list<T> init) {
167         for (auto x: init)
168             push_back(x);
169     };
170
171
172
173     SingleLinkedList(SingleLinkedList<T> &);
174
175
176
177
178
179     SingleLinkedList(SingleLinkedList<T> &&) noexcept;
180
181
182
183
184
185     SingleLinkedList &operator=(const SingleLinkedList<T> &);
186
187
188
189
190
191     SingleLinkedList &operator=(SingleLinkedList<T> &&);
192
193
194
195
196     T &operator[](int index) {
197         auto tmp = head;
198         while (index) {
199             tmp = tmp->next;
200             index--;
201         }
202         return tmp->data;
203     }

```

```

204
206
208     int size() { return SIZE; }
209
211
214     void push_back(T data);
215
217
220     void push_front(T data);
221
222
224
226     void pop_front();
227
229
231     void pop_back();
232
234
236     void deleteByNode(int position);
237
239
242     void insert(T data, int pos);
243
245
247     void deleteByData(T _data);
248
250     void print();
251
253
256     bool search(T _data);
257
259
261     void sort();
262
264     void clear();
265
267
270
271
272     void serialize(const std::string &filename);
273
275
278     void deserialize(const std::string &filename);
279
280 };
281
282
283 template<class T>
284 SingleLinkedList<T>::SingleLinkedList() {
285     std::cout << "Single linked list is created\n";
286     head = nullptr;
287 }
288
289 template<class T>
290 SingleLinkedList<T>::SingleLinkedList(SingleLinkedList<T> &s) {
291     head = s.head;
292     SIZE = s.size();
293 }
294
295 template<class T>
296 SingleLinkedList<T>::SingleLinkedList(SingleLinkedList<T> &&s) noexcept {
297     head = std::move(s.head);
298     SIZE = std::move(s.SIZE);
299     s.SIZE = 0;
300 }
301
302
303 template<class T>
304 SingleLinkedList<T> &SingleLinkedList<T>::operator=(const SingleLinkedList<T> &other) {
305
306     if (this != &other) {
307         head = other.head;
308         SIZE = other.size();
309     }
310     return *this;
311 }
312
313 template<class T>
314
315 SingleLinkedList<T> &SingleLinkedList<T>::operator=(SingleLinkedList<T> &&other) {
316     if (this != &other) {
317         head = std::move(other.head);
318         SIZE = std::move(other.SIZE);
319         other.SIZE = 0;
320     }
321     return *this;
322 }

```

```

323
324
325 template<class T>
326 void SingleLinkedList<T>::push_back(T data) {
327
328     auto n = std::make_shared<Node<T>>(data);
329
330     if (head == NULL) {
331         head = n;
332         SIZE++;
333     } else {
334         std::shared_ptr<Node<T>> tmp_ptr = head;
335         while (tmp_ptr->next != NULL) {
336             tmp_ptr = tmp_ptr->next;
337         }
338         tmp_ptr->next = n;
339         SIZE++;
340     }
341 }
342 }
343
344 template<class T>
345 void SingleLinkedList<T>::push_front(T data) {
346
347     auto n = std::make_shared<Node<T>>(data);
348
349     if (head == NULL) {
350         head = n;
351         SIZE++;
352     } else {
353         n->next = head;
354         head = n;
355         SIZE++;
356     }
357 }
358
359
360 }
361
362
363 template<class T>
364 void SingleLinkedList<T>::print() {
365     auto tmp = head;
366     std::cout << "The elements of the list:\n";
367     while (tmp != NULL) {
368         std::cout << "Data: " << tmp->data << "\n";
369         tmp = tmp->next;
370     }
371 }
372
373 template<typename T>
374 std::ostream &operator<<(std::ostream &os, const SingleLinkedList<T> &s) {
375     {
376         auto tmp = s.head;
377         std::cout << "The elements of the list:\n";
378         while (tmp != NULL) {
379             os << "Data: " << tmp->data << "\n";
380             tmp = tmp->next;
381         }
382         return os;
383     }
384 }
385
386
387 template<class T>
388 void SingleLinkedList<T>::pop_front() {
389     if (head == NULL) {
390         std::cout << "The container is already empty\n";
391     } else {
392         head = head->next;
393         SIZE--;
394     }
395 }
396
397
398 template<class T>
399 void SingleLinkedList<T>::pop_back() {
400
401     if (head == NULL) {
402         std::cout << "The container is already empty\n";
403     } else {
404         auto tmp = head;
405         auto prev_node = head;
406         while (tmp->next != NULL) {
407             prev_node = tmp;
408             tmp = tmp->next;
409         }

```

```

410         prev_node->next = NULL;
411         SIZE--;
412     }
413 }
414 }
415
416
417 template<class T>
418 void SingleLinkedList<T>::deleteByNode(int position) {
419
420     if (head == NULL) {
421         std::cout << "The container is already empty (Nothing to delete)\n";
422     } else {
423         auto tmp = head;
424         auto prev_node = head;
425         while (position != 0) {
426             prev_node = tmp;
427             tmp = tmp->next;
428             position--;
429         }
430         prev_node->next = tmp->next;
431         SIZE--;
432     }
433 }
434
435 }
436
437 template<class T>
438 void SingleLinkedList<T>::deleteByData(T _data) {
439
440     auto tmp = head;
441     auto prev_node = head;
442     while (tmp != NULL) {
443         if (tmp->data == _data) {
444             prev_node->next = tmp->next;
445             tmp = tmp->next;
446             SIZE--;
447         } else {
448             prev_node = tmp;
449             tmp = tmp->next;
450         }
451     }
452 }
453 }
454
455 template<class T>
456 void SingleLinkedList<T>::insert(T data, int pos) {
457     auto tmp = head;
458     auto n = std::make_shared<Node<T>>(data);
459     if (pos < 0 || pos > SIZE) {
460         throw OutOfRange(pos);
461     } else {
462         while ((pos - 1) != 0) {
463             tmp = tmp->next;
464             pos--;
465         }
466         n->next = tmp->next;
467         tmp->next = n;
468         SIZE++;
469     }
470 }
471
472 template<class T>
473 std::istream &operator<>(std::istream &is, SingleLinkedList<T> &s) {
474     T tmp;
475     while (is >> tmp) {
476         s.push_back(tmp);
477         s.SIZE++;
478     }
479     return is;
480 }
481
482 template<class T>
483 void SingleLinkedList<T>::clear() {
484     head = NULL;
485     SIZE = 0;
486 }
487
488 template<class T>
489 void SingleLinkedList<T>::sort() {
490     if (head == NULL) {
491         return;
492     }
493     auto tmp = head;
494     bool swapped = true;
495     while (swapped) {
496         swapped = false;

```

```

497         while (tmp->next != NULL) {
498
499             if (tmp->data > tmp->next->data) {
500                 std::swap(tmp->data, tmp->next->data);
501                 swapped = true;
502             }
503             tmp = tmp->next;
504         }
505         tmp = head;
506     }
507 }
508 }
509
510 template<>
511 inline void SingleLinkedList<std::string>::sort() {
512     if (head == nullptr) {
513         return;
514     }
515     auto tmp = head;
516     bool swapped = true;
517     while (swapped) {
518         swapped = false;
519         while (tmp->next != nullptr) {
520
521             if (tmp->data.length() > tmp->next->data.length()) {
522                 std::swap(tmp->data, tmp->next->data);
523                 swapped = true;
524             }
525             tmp = tmp->next;
526         }
527         tmp = head;
528     }
529 }
530 }
531
532 template<class T>
533 bool SingleLinkedList<T>::search(T _data) {
534     auto tmp = head;
535     while (tmp != nullptr) {
536         if (tmp->data == _data) {
537             return true;
538         }
539         tmp = tmp->next;
540     }
541     return false;
542 }
543 }
544
545 template<class T>
546 void SingleLinkedList<T>::serialize(const std::string &filename) {
547     std::ofstream OutFile(filename,
548         std::ios::out | std::ios::binary); //flag that tells that you should write in a
549         binary form
550     if (OutFile) {
551         for (const auto &e: *this) {
552             OutFile.write((char *) &e, sizeof(e)); //1 param : ptr to the first byte
553         }
554         OutFile.close();
555     }
556 }
557
558 template<class T>
559 void SingleLinkedList<T>::deserialize(const std::string &filename) {
560     std::ifstream InFile(filename, std::ios::in | std::ios::binary);
561     if (InFile) {
562         T tmp;
563         while (InFile.read(const_cast<char *>((char *) &tmp), sizeof(tmp))) {
564             push_back(tmp);
565         }
566         InFile.close();
567     }
568 }
569 }
570
571
572 #endif //PROJECT_SINGLELINKEDLIST_H

```

Index

- deleteByData
 - SingleLinkedList< T >, 15
- deleteByNode
 - SingleLinkedList< T >, 15
- deserialize
 - SingleLinkedList< T >, 16
- insert
 - SingleLinkedList< T >, 16
- Iterator
 - SingleLinkedList< T >, 14
- Node
 - Node< T >, 6
- Node< T >, 5
 - Node, 6
- operator!=
 - S_Iterator< T >, 8
- operator<<
 - SingleLinkedList< T >, 20
- operator>>
 - SingleLinkedList< T >, 20
- operator*
 - S_Iterator< T >, 9
- operator+
 - S_Iterator< T >, 9
- operator++
 - S_Iterator< T >, 9
- operator->
 - S_Iterator< T >, 10
- operator=
 - SingleLinkedList< T >, 16, 17
- operator==
 - S_Iterator< T >, 10
- operator[]
 - S_Iterator< T >, 10
 - SingleLinkedList< T >, 17
- pop_back
 - SingleLinkedList< T >, 17
- pop_front
 - SingleLinkedList< T >, 18
- push_back
 - SingleLinkedList< T >, 18
- push_front
 - SingleLinkedList< T >, 18
- S_Iterator
 - S_Iterator< T >, 8
- S_Iterator< T >, 6
- operator!=, 8
- operator*, 9
- operator+, 9
- operator++, 9
- operator->, 10
- operator==, 10
- operator[], 10
- S_Iterator, 8
- search
 - SingleLinkedList< T >, 18
- serialize
 - SingleLinkedList< T >, 19
- SingleLinkedList
 - SingleLinkedList< T >, 14, 15
- SingleLinkedList< T >, 11
 - deleteByData, 15
 - deleteByNode, 15
 - deserialize, 16
 - insert, 16
 - Iterator, 14
 - operator<<, 20
 - operator>>, 20
 - operator=, 16, 17
 - operator[], 17
 - pop_back, 17
 - pop_front, 18
 - push_back, 18
 - push_front, 18
 - search, 18
 - serialize, 19
 - SingleLinkedList, 14, 15
 - size, 19
 - sort, 19
- size
 - SingleLinkedList< T >, 19
- sort
 - SingleLinkedList< T >, 19
- Student, 21