СУ “Св. Климент Охридски”,
ФМИ – Софтуерно инженерство
Курсов проект по Обектно-ориентирано програмиране

Двусвързан списък

Костадин Тодоров Хамънов, Факултетен № 61855

Съдържание

1. Въведение

Целта на този курсов проект е имплементиране на основните функционалности, които се поддържат от структурата от данни **„двусвързан списък“**. Настоящата документация представя кратко описание на програмния код, основните алгоритми и използвани техники за създаване на проекта.

На места в текста е пропуснато подробно описание на същността на фунциите, предполагайки се, че читателят е достатъчно запознат с основните принципи на процедурното и обектно-ориентирано програмиране.

За въпроси, предложения, препоръки или допуснати грешки или неточности в текста на изложението на проекта, авторът на настоящата документация очаква вашето мнение по въпроса.

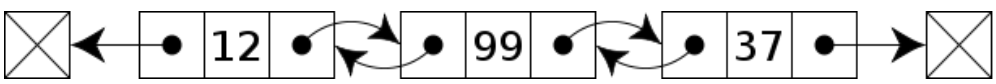
Кодът на проекта можете да на: https://github.com/KostadinHamanov/Doubly-Linked-List

- ## Общо описание

Двусвързаният списък представлява **линейна структура** от свързани еднотипни компоненти. Компонентите на двусвързания списък са динамични променливи от тип запис с **три** полета:

- **информационно поле** - обикновено е от тип запис с полета, определени от конкретното предназначение на списъка.
- **две свързващи полета** - указващи предходната и следващата компоненти в двусвързан списък

За удобство, при реализирането на операциите включване, изключване и обхождане, се въвеждат тройни кутии, с едно информационно и две свързващи полета, съдържащи текущия елемент и адресите на предшестващия и следващия го елементи на свързания списък



2. Описание на програмния код

- ### Общо описание на класът List

Програмният код се състои от **.h файл**, в които се декларира шаблонен интерфейс на класа List и **.cpp** файл, в който са дефинициите на функциите.

Класът, представящ двусвързания списък е шаблонен и носи името List. Стуктурата му е представена по долу:

```
template <class T>
class List
{
public:

	struct Node;

	class iterator;

	List();
	~List();
	List(const List&);
	List& operator=(const List&);

	void push_front(const T&);
	void pop_front();
	void push_back(const T&);
	void pop_back();

	T& front();
	T& back();

	iterator begin();
	iterator end();

	void insert(iterator, const T&);
	void erase(iterator);
	int size();
	void clear();
	bool empty();

private:
	Node* head;
	Node* tail;
	Node* current;
	int length;

	//Помощни функции
	void copyList(const List<T>&);
	void deleteList();

};
```

- ## Struct Node

В шаблонния клас List е имплементирана структура за елемент в свързания списък **(Node).**

В структурата има конструктор и член-променливи: **m_data(int), m_next(Node*), m_prev(Node*).**

```
struct Node
{
	T m_data;
	Node* m_next;
	Node* m_prev;

	Node(T data, Node *p_next = nullptr, Node *p_prev = nullptr) :
	m_data(data), m_next(p_next), m_prev(p_prev)
	{

	}
};
```

m_data пази информация за съдържанието на елемента, останалите две член-променливи са променливи от тип указател към Node, представляващи връзка с предходния и следващия елемент.

Инициализацията на член-променливите се извършва чрез initialization list в констуктора Node.

- ## Член-променливи на List

Член променливите на класа са няколко.

```
Node* head – представлява указател към първия елемент в списъкът.
Node* tail – указател към последния елемент.
Node* current – указател към текущ елемент.
int length – дължина на списъкът.
```

- ## Канонично представяне

```
List();
~List();
List(const List&);
List& operator=(const List&);
```

а) Констуктор

```
template <class T>
List<T>::List() : head(nullptr), tail(nullptr), current(nullptr), length(0)
{
}
```

b) Деструктор

```
template <class T>
List<T>::~List()
{
	deleteList();
}
```

c) Конструктор за присвояване

```
template <class T>
List<T>::List(const List<T>& r)
{
	copyList(r);
}
```

d) Операторна функция за присвояване

```
template <class T>
List<T>& List<T>::operator=(const List<T>& right)
{
	if (this != &right)
	{
		deleteList();
		copyList(right);
	}
	return *this;
}
```

e) Помощни функции

Отделно са дефиниране и **две помощни функции** за копиране и изтриване на списък.

```
template <class T>
void List<T>::copyList(const List<T>& original)
{
	Node *ptr_new;
	Node *current;

	if (head != nullptr)
	{
		deleteList();
	}

	if (original.head == nullptr)
	{
		head = nullptr;
		tail = nullptr;
		current = nullptr;
		length = 0;
		return;
	}

	//Създаване(копиране) на първия възел
	current = original.head;
```

```
            head = new Node(current->m_data);
            tail = tail;

            current = current->m_next;
            while (current != nullptr)
            {
                  //Създава се нов възел
                  ptr_new = new Node(current->m_data);
                  tail->m_next = ptr_new; //Връзка на предходния с новия възел
                  ptr_new->m_prev = tail; //Връзка на новия с предходния възел
                  tail = ptr_new;

                  current = current->m_next;
            }

            length = original.length;

      }


      template <class T>
      void List<T>::deleteList()
      {
            Node *p;

            while (head != nullptr)
            {
                  p = head;
                  head = head->m_next;
                  delete p;
            }
            tail = nullptr;
            length = 0;
      }
```

- ## Описание на основните функции на класа List

- **void push_front(const T& value)** - добавя елемент в началото на списъка

```
template <class T>
void List<T>::push_front(const T& value)
{
      if (head == nullptr)
      {
            head = new Node(value);
            tail = head;
      }
      else
      {
            Node* ptr_head = head;
            head = new Node(value, head);
            ptr_head->m_prev = head;
      }

      length++;
}
```

- **void pop_front()** - премахва елемент от началото на списъка

```
template <class T>
void List<T>::pop_front()
{
	if (head == nullptr)
	{
		return;
	}
	else if (head->m_next == nullptr)
	{
		//Списъкът не е празен, но има само един възел
		head = nullptr;
		tail = nullptr;
		delete head;
	}
	else
	{
		Node* ptr_node = head;
		head = head->m_next;
		delete ptr_node;
	}

	length--;
}
```

- **void push_back(const T& value)** - добавя елемент в края на списъка

```
template <class T>
void List<T>::push_back(const T& value)
{
	if (head == nullptr)
	{
		head = new Node(value);
		tail = head;
	}
	else
	{
		//Добавяне на нов възел след текущия последен
		Node* tail_node = tail;
		tail_node->m_next = new Node(value, nullptr, tail);
		tail = tail_node->m_next;
	}
	length++;
}
```

- **void pop_back()** - премахва елемент от края на списъка

```
template <class T>
void List<T>::pop_back()
{
	if (head == nullptr)
	{
		return;
	}
	else if (head->m_next == nullptr)
	{
		delete head;
```

```
                head = nullptr;
                tail = nullptr; //Списъкът не е празен, но има само един възел
        }
        else
        {
                Node* ptr_last = tail;
                Node* ptr_prev = tail->m_prev; //Адрес на предпоследния възел
                tail = ptr_prev;
                tail->m_next = nullptr;
                delete ptr_last;
        }

        length--;
}
```

- **T& front()** - връща стойността на елемента в началото на списъка

```
template <class T>
T& List<T>::front()
{
        return head->m_data;
}
```

- **T& back()** - връща стойността на елемента в края на списъка

```
template <class T>
T& List<T>::back()
{
        return tail->m_data;
}
```

- ## Клас iterator

Итераторът е абстракция на означението **указател** към елемент на редица или по-точно може да се смята за указател към елемент на контейнер (стекът, опашката, свързаният списък са контейнери). Всеки конкретен итератор е обект (в широкия смисъл на думата) от някакъв тип. Разнообразието на типове води до разнообразие на итераторите. В някои случаи итераторите са почти обикновени указатели към обекти, в други – са указател, снабден с индекс и т.н. В случая на свързан списък итераторът е **указател към двойна или тройна кутия**. Общото на всички итератори е тяхната семантика и имената на техните операции.

Обикновено операциите са:
++ - приложена към итератор, намира итератор, който сочи към следващия елемент;

-- - приложена към итератор, намира итератор, който сочи към предшестващия елемент;

***** - намира елемента, към който сочи итераторът.

В случая структурана на нашия итератор има вида:

```
class iterator
{
public:
	iterator();
	iterator(Node*);
	T& operator*();
	iterator operator++();
	iterator operator++(int);
	bool operator!=(iterator&);
	bool operator==(iterator&);
private:
	Node* pNode;

friend class List<T>;

};
```

Член-променливата на този клас е **указател към Node**. Естествено не трябва да забравяме, че този клас **е приятелски клас** на класа List<T>.

- ## Описание на основните функции на класа iterator

- **Конструктор по подразбиране**

```
template <class T>
List<T>::iterator::iterator() : pNode(nullptr)
{

}
```

В случая данните се инициализират чрез **initialization list**.

- **Предефиниран констуктор по подразбиране**

```
template <class T>
List<T>::iterator::iterator(Node* data) : pNode(data)
{

}
```

- **T& operator*()** - връща стойността на даден Node (data)

```
template <class T>
inline T& List<T>::iterator::operator*()
{
	return pNode->m_data;
}
```

- **iterator operator++()** – префиксен оператор за инкрементиране (it = ++v.begin())

```
iterator& operator++()
{
	pNode = pNode->m_next;
	iterator temp(pNode);
	return temp;
}
```

- **iterator operator++(int)** – постфиксен оператор за инкрементиране (it = v.begin()++)

```
iterator operator++(int)
{
	iterator temp(pNode);
	pNode = pNode->m_next;
	return temp;
}
```

- **bool operator!=()** - проверява дали адресите на два Node-а са различни

```
template <class T>
bool List<T>::iterator::operator!=(iterator& secondIterator)
{
	return pNode != secondIterator.pNode;
}
```

- **bool operator==()** - проверява дали адресите на два Node-а са еднакви

```
template <class T>
bool List<T>::iterator::operator==(iterator& otherIterator)
{
	return pNode == otherIterator.pNode;
}
```

- ## Допълнителни фунции в класа List и функции, свързани и използването на итератора

- **iterator begin()** - връща iterator към началото на списъка

```
iterator begin()
{
	iterator it(head);
	return it;
}
```

- **iterator end()** - връща iterator към края на списъка (един елемент след края на списъка)

```
iterator end()
{
```

```
	iterator it(tail->m_next);
	return it;
}
```

- **void insert(iterator it, const T& value**) - вмъква елемент със стойност value на позиция iterator

Вмъкването се извършва на позицията**, предхождаща** итератора. Извършени са няколко **проверки** за установяване на **местоположението** на итератора, пряко свързани с логиката **за добавяне** на елемент със стойност value

```
template <class T>
void List<T>::insert(iterator it, const T& value)
{
	if (it.pNode == nullptr)
	{
		push_back(value);
		length++;
		return;
	}
	else if (it.pNode == head)
	{
		Node *ptr_new = new Node(value);
		ptr_new->m_prev = nullptr;
		ptr_new->m_next = head;
		head->m_prev = ptr_new;
		head = ptr_new;
		length++;
		return;
	}
	else
	{
		Node* ptr_new = new Node(value);
		assert(it.pNode != nullptr);

		ptr_new->m_next = it.pNode;
		ptr_new->m_prev = it.pNode->m_prev;

		ptr_new->m_next->m_prev = ptr_new->m_prev;
		ptr_new->m_prev->m_next = ptr_new;

		length++;
	}
}
```

- **void erase(iterator it)** - изтрива елемент на позиция iterator

Отново са извършени няколко **проверки** за установяване на **местоположението** на итератора, пряко свързани с логиката **за изтриване** на елемент със стойност value.

```
template <class T>
void List<T>::erase(iterator it)
{
	if (it.pNode == head)
	{
		head = head->m_next;
		head->m_prev = nullptr;
		delete it.pNode;
		length--;
		return;
	}
```

```
        else if (it.pNode == tail)
        {
                tail = tail->m_prev;
                tail->m_prev = nullptr;
                delete it.pNode;
                length--;
                return;
        }
        else
        {
                it.pNode->m_prev->m_next = it.pNode->m_next;
                it.pNode->m_next->m_prev = it.pNode->m_prev;
                delete it.pNode;
                length--;
        }
}
```

- **int size()** - връща броя елементи в списъка

```
template <class T>
int List<T>::size()
{
        return length;
}
```

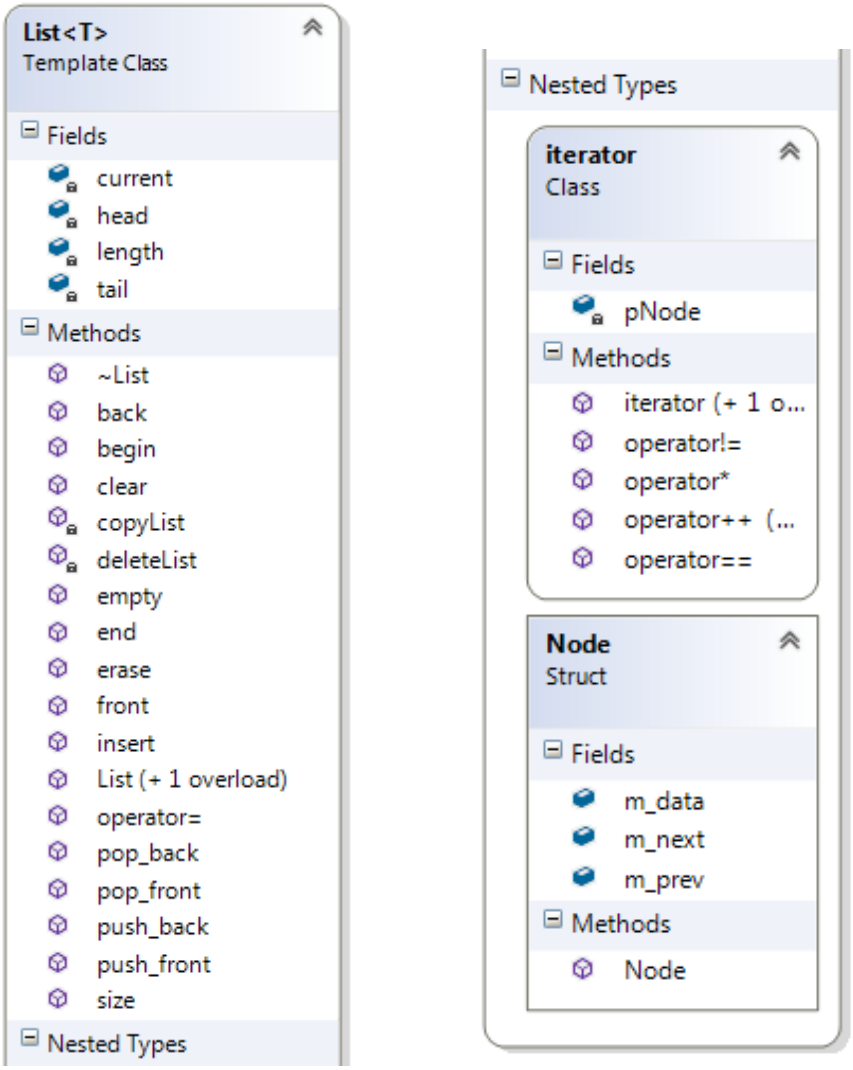
- **void clear()** - изтрива всички елементи на списъка

```
template <class T>
void List<T>::clear()
{
        deleteList();
}
```

- **bool empty()** - проверява дали списъкът е празен

```
template <class T>
bool List<T>::empty()
{
        return length == 0;
}
```

3. Class Diagram



4. Примерна употреба

```
#include <iostream>
#include <string>
#include "DoublyLinkedList.h"
#include "DoublyLinkedList.cpp"

using namespace std;

int main()
{
	List<int> list1;

	list1.push_front(100);
	list1.push_front(200);
	list1.push_front(300);
	list1.push_back(777);

	cout << list1.back() << endl; //777

	list1.pop_back(); //300 200 100

	cout << list1.back() << endl; //100
	cout << list1.front() << endl; //300
	list1.pop_front();
	cout << list1.front() << endl; //200

	list1.clear();

	cout << endl;

	List<int> list2;

	list2.push_back(616);
	list2.push_front(515);
	list2.push_front(313);
	list2.push_back(777);

	//Извежда 313 515 616 777
	for (List<int>::iterator it = list2.begin(); it != list2.end(); it++)
	{
		cout << *it << " ";
	}
	cout << endl;
	list2.clear();
	cout << endl;

	List<string> list3;

	list3.push_back("vidi");
	list3.push_back("vici");

	List<string>::iterator iter = list3.begin();

	list3.insert(iter, "Veni");

	//Извежда Veni vidi vici
	for (List<string>::iterator it = list3.begin(); it != list3.end(); it++)
	{
		cout << *it << " ";
	}
```

```
	list3.clear();

	cout << endl;

	List<string> list4;

	list4.push_back("Divide");
	list4.push_back("et");
	list4.push_back("impera");

	//Извежда Divide et impera
	for (List<string>::iterator it = list4.begin(); it != list4.end(); it++)
	{
		cout << *it << " ";
	}

	List<string>::iterator mid = ++list4.begin();
	list4.erase(mid);

	cout << endl;

	//Извежда Divide impera
	for (List<string>::iterator it = list4.begin(); it != list4.end(); it++)
	{
		cout << *it << " ";
	}
	cout << endl;

	list4.clear();

	cout << endl;

	return 0;
}
```

5. Използвани технологии

Езикът, използван за имплементирането на логиката на задачата е **C++**

Платформа: Microsoft .NET Framework 4.5.

Използваната **среда за разработка** на настоящия проект е: Microsoft Visual Studio 2013