СУ “Св. Климент Охридски”,

**ФМИ – Софтуерно инженерство**

**Курсов проект по Обектно-ориентирано програмиране**

**List**

**(Двусвързан списък)**

Славяна Бориславова Монкова, Фак. № 61784

Съдържание

[1. Въведение 3](#_Toc409876355)

[2. Описание на програмния код](#_Toc409876357) 3

* клас List………………………………………………………………………………………………………………………………3-4
* struct Node………………………………………………………………………………………………………………………...5-6
* член-променливите на List…………………………………………………………………………………………………..6
* канонично представяне ……………………………….………………………………………………………………….6-7
* помощни функции…………………………………………………………………………………………………………….7-8
* основните функции на класа List……………………………………………………………………………………8- 11
* клас Iterator……………………………………………………………………………………………………………….………11
* описание на основните функции на класа Iterator……………………………………………………12- 13
* допълнителни функции в клас List и функции,

свързани с използването на итератор………………………………………………………………………….13-16

3. Примерна употреба……………………………………………………………………………………………………………16-17

[4. Използвани технологии 17](#_Toc409876358)

# Въведение

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

**Двусвързаният** списък се реализира чрез шаблонен **клас List**, в който е написана **структура за даден елемент в свързания списък (Node)**. В структурата биват инициализирани различни член-функции, изпълняващи различни операции. Освен това е имплементиран **клас Iterator**  за обхождане на елементите на класа . Като член-променлива този клас има указател към Node и се предефинират някои оператори.

# Описание на програмния код

* Клас List

Програмният код се състои от **.h файл**, в които се декларира шаблонен интерфейс на класа List и след декларацията са описани и самите дефиниции на член-функциите. Има и един **.cpp файл** за главната функция с примерен вход.

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

template<class T>

class List

{

public:

struct Node

{

public:

Node(const T& m\_data, Node\* next = NULL, Node\* prev = NULL);

~Node();

T getData();

Node\* getNext();

Node\* getPrev();

void setNext(Node\* next);

void setPrev(Node\* prev);

private:

T m\_data;

Node\* m\_next;

Node\* m\_prev;

};

List();

~List();

List(const List<T>& ptr);

List& operator=(const List<T>& right);

//Помощни функции

void copyList(const List<T>&);

void deleteList();

void push\_front(const T& value);

void pop\_front();

void push\_back(const T& value);

void pop\_back();

T front();

T back();

class Iterator;

Iterator begin();

Iterator end();

void insert(Iterator& it, const T& value);

void erase(Iterator it);

int getSize();

void clear();

bool empty();

private:

unsigned int size; //брояч за елементите на списъка

Node\* start;

Node\* end;

};

* Struct Node

За представяне на елементите на двусвързания списък се използва структура Node, която съдържа полетата:

* m\_next от тип указател към структурата (Node\* ), което е насочено към следващият елемент;
* m\_prev (Node\* ), което е насочено към предходният елемент;
* m\_data, в което се съхраняват данните, асоциирани с даденият елемент на списъка;
* конструктор за инициализиране на тези полета;
* деструктор;
* някои помощни функции за достъп, тъй като член-променливите са private.

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

struct Node

{

public:

Node(const T& m\_data, Node\* next = NULL, Node\* prev = NULL) : m\_data(m\_data), m\_next(next), m\_prev(prev)

{

}

~Node()

{

}

//функция за достъп по член-променливата m\_data

T getData()

{

return this->m\_data;

}

//функция за достъп по член-променливата m\_next

Node\* getNext()

{

return this->m\_next;

}

//функция за достъп по член-променливата m\_prev

Node\* getPrev()

{

return this->m\_prev;

}

void setNext(Node\* next)

{

this->m\_next = next;

}

void setPrev(Node\* prev)

{

this->m\_prev = prev;

}

private:

T m\_data;

Node\* m\_next;

Node\* m\_prev;

};

* Член-променливите на List

Член променливите на класа са няколко.

unsigned int size; - дължина на списъка

Node\* start; - указател към първия елемент

Node\* end; - указател към последния елемент

**а) Конструктор**

template<class T>

List<T>::List() : size(0), start(NULL), end(NULL)

{

}

**б) Деструктор**

template<class T>

List<T>::~List()

{

deleteList();

}

**в)Конструктор за копиране**

template<class T>

List<T>::List(const List<T>& ptr)

{

copyList(ptr);

}

**г) Конструктор за присвояване**

template <class T>

List<T>& List<T>::operator=(const List<T>& right)

{

if (this != &right)

{

deleteList();

copyList(right);

}

return \*this;

}

* Помощни функции

**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 (start != NULL)

{

deleteList();

}

if (original.start == NULL)

{

start = NULL;

end = NULL;

size = 0;

return;

}

//Създаване(копиране) на първия възел

current = original.start;

start = new Node(current->m\_data);

end = start;

current = current->m\_next;

while (current != NULL)

{

//Създава се нов възел

ptr\_new = new Node(current->m\_data);

end->m\_next = ptr\_new; //Връзка на предходния с новия възел

ptr\_new->m\_prev = end; //Връзка на новия с предходния възел

end = ptr\_new;

current = current->m\_next;

}

size = original.size;

}

template<class T>

void List<T>::deleteList()

{

Node \*p;

while (start != end)

{

p = start;

start = start->getNext();

delete p;

p = NULL;

}

delete end;

end = NULL;

size = 0;

}

* Oсновните функции на класа List

1. **void push\_front(const T& value)** - добавя елемент в началото на списъка

template<class T>

void List<T>::push\_front(const T& value)

{

if (start == NULL)

{

start = new Node(value);

end = start;

}

else

{

Node\* ptr\_start = start;

start = new Node(value, start);

Node\* temp = ptr\_start->getPrev();

temp = start;

}

size++;

}

1. **void pop\_front()** - премахва елемент от началото на списъка

template<class T>

void List<T>::pop\_front()

{

if (start == NULL)

{

return;

}

else if (start->getNext() == NULL)

{

start = NULL;

end = NULL;

delete start;

}

else

{

Node\* temp = start;

start = start->getNext();

delete temp;

}

size--;

}

1. **void push\_back(const T& value)** - добавя елемент в края на списъка

template<class T>

void List<T>::push\_back(const T& value)

{

if (start == NULL)

{

start = new Node(value);

end = start;

}

else

{

Node\* temp = new Node(value, NULL, end);

end->setNext(temp);

end = temp;

}

this->size++;

}

1. **void pop\_back()** - премахва елемент от края на списъка

template<class T>

void List<T>::pop\_back()

{

if (start == NULL)

{

return;

}

else if (start->getNext() == NULL)

{

delete start;

start = NULL;

end = NULL;

}

else

{

Node\* ptr\_last = end;

Node\* ptr\_prev = end->getPrev();

end = ptr\_prev;

Node\* temp = end->getNext();

temp = NULL;

delete ptr\_last;

}

size--;

}

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

template<class T>

T List<T>::front()

{

return start->getData();

}

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

template<class T>

T List<T>::back()

{

return end->getData();

}

* 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 структурата на нашия итератор има вида:

class Iterator

{

public:

Iterator();

Iterator(Node\*);

T operator\*();

Iterator& operator++();

Iterator operator++(int);

bool operator!=(const Iterator& temp);

Node\* getCurrent();

private:

Node\* current; //за сегашен елемент

friend class List<T>;

};

**Член-променливата** на този клас e **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

1. Конструктор по подразбиране

template<class T>

List<T>::Iterator::Iterator() : current(NULL){}

1. Предефиниран констуктор по подразбиране

template<class T>

List<T>::Iterator::Iterator(Node\* data) : current(data){}

1. **T operator\*()** - връща стойността на даден Node (data)

template<class T>

T List<T>::Iterator::operator\*()

{

return current->getData();

}

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

Iterator& operator++()

{

current = current->m\_next;

Iterator temp(current);

return temp;

}

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

Iterator operator++(int)

{

Iterator temp = \*this;

++\*this;

return temp;

}

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

template<class T>

bool List<T>::Iterator::operator!=(const Iterator& temp)

{

return this->current != temp.current;

}

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

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

Iterator begin()

{

return Iterator(start);

}

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

Iterator end()

{

return Iterator(end);

}

1. **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)

{

Node\* insNode = new Node(value, NULL, NULL);

if (it.getCurrent() == NULL)

{

it.getCurrent()->setPrev(insNode);

insNode->setNext(it.getCurrent());

this->start = insNode;

this->size++;

}

else if (it.getCurrent()->getNext() == NULL)

{

it.getCurrent()->setNext(insNode);

insNode->setPrev(it.getCurrent());

this->end = insNode;

this->size++;

}

else

{

Node\* prev = it.getCurrent()->getPrev();

it.getCurrent()->setPrev(insNode);

prev->setNext(insNode);

insNode->setNext(it.getCurrent());

insNode->setPrev(prev);

this->size++;

}

it.setCurrent(insNode); //за да стои на една и съща позиция

}

1. **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)

{

// проверява се дали итератора е в началото на списъка

if (it.getCurrent()->getPrev() == NULL)

{

it.setCurrent(it.getCurrent()->getNext());

it.getCurrent()->setPrev(NULL);

this->start = it.getCurrent();

}

// проверява се дали итератора е в края на списъка

else if (it.getCurrent()->getNext() == NULL)

{

it.setCurrent(it.getCurrent()->getPrev());

it.getCurrent()->setNext(NULL);

this->end = it.getCurrent();

}

// ако итератора е в средата на списъка

else

{

it.getCurrent()->getPrev()->setNext(it.getCurrent()->getNext());

it.getCurrent()->getNext()->setPrev(it.getCurrent()->getPrev());

Node\* temp = it.getCurrent()->getNext();

delete it.getCurrent();

it.setCurrent(temp);

}

this->size--;

}

1. **int getSize()** - връща броя елементи в списъка

template<class T>

int List<T>::getSize()

{

return this->size;

}

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

template<class T>

void List<T>::clear()

{

delete this;

}

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

template<class T>

bool List<T>::empty()

{

return start == NULL;

}

1. Примерна употреба

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();

cout << list1.back() << endl; //100

cout << list1.front() << endl; //300

list1.pop\_front();

cout << list1.front() << endl; //200

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;

return 0;

}

# Използвани технологии

Език за програмиране: C++

Среда за разработка: Microsoft Visual Studio 2013