МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

**«Нижегородский государственный университет им. Н.И. Лобачевского»**

**Национальный исследовательский университет**

**Институт информационных технологий, математики и механики**

**Кафедра математического обеспечения и суперкомпьютерных технологий**

**Отчет по учебной практике**

**«Структура хранения данных: Односвязный линейный список с использованием указателей»**

**Выполнил:** студент группы 381903-3

Стеценко Данил Михайлович

**Научный руководитель:**

ассистент каф. МОСТ ИИТММ Лебедев И.Г

Нижний Новгород

2020

**Оглавление**

[1. Введение. 2](#_Toc711842)

[2. Постановка задачи. 3](#_Toc711843)

[3. Руководство пользователя. 4](#_Toc711844)

[4. Руководство программиста. 5](#_Toc711845)

[4.1. Описание структуры программы. 5](#_Toc711846)

[4.2. Описание структур данных. 5](#_Toc711847)

[4.3. Описание алгоритмов. 6](#_Toc711848)

[5. Эксперименты. 8](#_Toc711849)

[6. Заключение. 9](#_Toc711850)

[7. Литература. 10](#_Toc711851)

**8.** Приложения……………………………………………………………………….11

# Введение.

**Список** — это абстрактный тип данных, представляющий собой упорядоченный набор значений, в котором некоторое значение может встречаться более одного раза. Экземпляр списка является компьютерной реализацией математического понятия конечной последовательности. Экземпляры значений, находящихся в списке, называются **элементами**; если значение встречается несколько раз, каждое вхождение считается отдельным элементом.

**Связный список** — базовая динамическая структура данных в информатике, состоящая из узлов, каждый из которых содержит как собственно данные, так и ссылку на следующий и/или предыдущий узел списка. Принципиальным преимуществом перед массивом является структурная гибкость: порядок элементов связного списка может не совпадать с порядком расположения элементов данных в памяти компьютера, а порядок обхода списка всегда явно задаётся его внутренними связями.

# Постановка задачи.

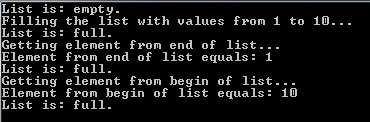
В данной лабораторной работе нужно разработать эффективную библиотеку для хранения и работы с такой структурой данных, как список на указателях.

Для этого нам нужно:

* Описать и реализовать класс элемента списка **TElem**.
* Описать и реализовать класс списка **TList**.
* Протестировать класс **TList** с помощью Google Test.
* Продемонстрировать работу класса **TList**.

# Руководство пользователя.

Пример работы класса **TList**:

****

*Рис 1. Пример работы программы*

Программа работает следующим образом:

* Создается список.
* Проверяем список на пустоту.
* Заполняем список значениями от 1 до 10.
* Проверяем список на полноту.
* Забираем элемент из конца списка.
* Проверяем список на полноту и пустоту.
* Забираем элемент из начала списка.
* Проверяем список на полноту и пустоту.

# Руководство программиста.

# Описание структуры программы.

Программа состоит из модулей:

* **List** – содержит в себе файл **list\_main.cpp** с реализацией примера использования класса **TList.**
* **ListLib –** содержит в себе файлы **List.h** и **Elem.h**, в которых описаны и реализованы классы **TList** и **TElem** соответственно.
* **ListTest** – содержит в себе файл **test\_list.cpp**, в котором находится набор тестов, для проверки работоспособности класса **TList**.

# Описание структур данных.

**Класс TElem.**

Класс **TElem** является шаблонным классом и содержит два поля со спецификатором **protected**:

* **T elem** – переменная, для хранения элемента списка.
* **TElem<T>\* next** – указатель на следующий элемент списка.

Далее идут методы класса со спецификатором доступа **public**:

* **TElem(T \_elem = 0, TElem<T>\* \_next = 0)** – конструктор инициализатор.
* **TElem(TElem<T> &obj)** – конструктор копирования.
* **T GetElem()** – метод для получения элемента списка.
* **TElem<T>\* GetNext()** – метод для получения указателя на следующий элемент списка.
* **void SetElem(T \_elem)** – метод для добавления элемента в список.
* **void SetNext(TElem<T>\* \_next)** – метод для добавления указателя на следующий элемент списка.

**Класс TList.**

Класс **TList** является шаблонным классом и содержит одно поля со спецификатором **protected**:

* **TElem<T>\* begin** – указатель на начало списка.

Далее идут методы класса со спецификатором доступа **public**:

* **TList()** – конструктор по умолчанию.
* **TList(TList<T> &obj)** – конструктор копирования.
* **void PutBegin(T \_elem)** – метод для добавления элемента в начало списка.
* **void PutEnd(T \_elem)** – метод для добавления элемента в конец списка.
* **T GetBegin()** – метод для получения элемента из начала списка.
* **T GetEnd()** – метод для получения элемента из конца списка.
* **bool IsEmpty()** – метод для проверки списка на пустоту.
* **bool IsFull() –** метод для проверки списка на полноту.

# Описание алгоритмов.

**Добавление элемента в начало списка.**

При добавлении элемента в начало списка, мы создаем указатель на объект класса **TElem**. После этого выделяем память под объект этого класса и, передав туда значение, которое необходимо положить в список, и указатель на текущее начало, создаем очередное элемент списка. Указатель на начало списка переопределяем на только что добавленный элемент.

**Добавление элемента в конец списка.**

При добавлении элемента списка в конец проверяем, есть ли элементы в списке. Если они есть, то создаем указатель на объект класса **TElem** и записываем в него значение начала списка. В цикле ищем текущий последний элемент, путем изменения значения указателя на объект класса **TElem***.* Как только конец списка будет найден, выделяем память под новой элемент списка и создаем его. Указатель для текущего последнего элемента переопределяем на следующий – только что созданный.

**Удаление элемента списка из начала.**

Для удаления элемента списка из начала выполняем проверку на пустоту списка. Если список пуст, то бросаем исключение. Иначе создаем указатель на объект класса **TElem**, которому присваиваем значение текущего начала списка. Создаем временную переменную*,* в которую записываем значение, хранящееся в первом элементе списка. Начало списка устанавливаем на следующий за удаляемый элемент. Удаляем указатель на объект класса **TElem** для того, чтобы очистить память, занимаемую бывшим первым элементом.

**Удаление элемента списка из конца.**

Для удаления элемента списка из конца выполняем проверку на пустоту списка. Если список пуст, то бросаем исключение. Иначе необходимо проверить: в списке больше одного элемента или ровно один. Для этого смотрим на следующий за первым элемент. Если указатель на него равен нулю, то мы возвращаем только данные из первого элемента списка, начало списка обнуляем.

Ели элементов больше нуля, то создаем указатель **\*tmp1** на объект класса **TElem**. Ищем в цикле предпоследнее звено списка. Далее, оздаем еще один указатель **\*tmp2** на объект класса **TElem**. В него записываем указатель на последний элемент списка и получаем данные из этого элемента. Затем удаляем указатель **\*tmp2**. Для **\*tmp1***,* устанавливаем в качестве следующего за ним 0, т.к. он теперь стал последним.

# Эксперименты.

|  |  |  |
| --- | --- | --- |
| Количество элементов в списке | Время работы метода **GetBegin()** (в млс) | Время работы метода **GetEnd()** (в млс) |
| 10000 | 0 | 1 |
| 100000 | 0 | 1 |
| 1000000 | 0 | 3 |

*Таблица 1. Результаты экспериментов.*

Таким образом, можно увидеть, что сложность работы метода **GetBegin()** – O(1). А сложность работы метода **GetEnd()** – O(n).

# Заключение.

В данной лабораторной работе мне удалось реализовать библиотеку для хранения и работы со списком на указателях, а именно:

* Удалось реализовать вспомогательный класс элемента списка **TElem**.
* Удалось реализовать класс списка **TList**.
* Удалось протестировать методы класса **TList**, а также обеспечить их работоспособность.

Таким образом, я смог реализовать структуру данных под названием список на указателях, каждый элемент которого указывает на следующий за ним элемент.

# Литература.

1. Ссылка из Википедии про список:

<https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B8%D1%81%D0%BE%D0%BA_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)>

1. Ссылка из Википедии про связный список:

<https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%8F%D0%B7%D0%BD%D1%8B%D0%B9_%D1%81%D0%BF%D0%B8%D1%81%D0%BE%D0%BA>

1. Гергель В.П. Методические материалы по курсу «Методы программирования 2», 2015.

# Приложения.

|  |
| --- |
| **List.h** |
| #ifndef \_LIST\_  #define \_LIST\_  #include <iostream>  using namespace std;  template <class T>  class TListElem  {  protected:  T data;  TListElem\* next = 0;  TListElem\* prev = 0;  public:  TListElem(T \_data);  ~TListElem();  T GetData();  TListElem\* GetNext();  TListElem\* GetPrev();  void SetData(T \_data);  void SetNext(TListElem\* \_next);  void SetPrev(TListElem\* \_prev);  template <class T1>  friend ostream& operator<< (ostream& ostr, const TListElem<T1>& A);  template <class T1>  friend istream& operator>> (istream& istr, TListElem<T1> A);  };  template <class T>  class TList  {  protected:  TListElem<T>\* root = 0;  TListElem<T>\* end = 0;  int count;  public:  TList();  TList(TList<T>& \_v);  ~TList();  TList<T>& operator =(TList<T>& \_v);  void InsFirst(T d);  void InsLast(T d);  void Ins(TListElem<T>\* e, T d);  bool IsFull(void) const;  bool IsEmpty(void) const;  TListElem<T>\* GetFirst();  TListElem<T>\* GetLast();  void DelFirst();  void DelLast();  void Del(TListElem<T>\* e);  int GetCount();  template <class T1>  friend ostream& operator<< (ostream& ostr, const TList<T1>& A);  template <class T1>  friend istream& operator >> (istream& istr, TList<T1>& A);  };  template<class T1>  inline ostream& operator<<(ostream& ostr, const TListElem<T1>& A)  {  return ostr << A.data;  }  template<class T1>  inline istream& operator>>(istream& istr, TListElem<T1> A)  {  return istr >> A.data;  }  template <class T1>  ostream& operator<< (ostream& ostr, const TList<T1>& A)  {  TListElem<T1>\* i = A.root;  while (i != 0)  {  ostr << \*i << endl;  i = i->GetNext();  }  return ostr;  }  template <class T1>  istream& operator >> (istream& istr, TList<T1>& A) {  int count;  istr >> count;  for (int i = 0; i < A.count; i++) {  T1 d;  istr >> d;  A.InsLast(d);  }  return istr;  }  template<class T>  inline TList<T>::TList()  {  root = 0;  end = 0;  count = 0;  }  template <class T>  TList<T>::TList(TList<T>& \_v)  {  count = \_v.count;  TListElem<T>\* i = \_v.root;  TListElem<T>\* j = this->root;  TListElem<T>\* p = 0;  while (i != 0)  {  j = new TListElem<T>(\*i);  j->SetNext(0);  if (p != 0)  {  p->SetNext(j);  j->SetPrev(p);  }  p = j;  if (root == 0)  {  root = j;  }  end = j;  i = i->GetNext();  }  }  template <class T>  TList<T>::~TList()  {  if (this->root != 0)  {  TListElem<T>\* i = this->root;  TListElem<T>\* p = 0;  while (i != 0)  {  p = i;  i = i->GetNext();  delete p;  }  this->root = 0;  this->end = 0;  count = 0;  }  }  template <class T>  TList<T>& TList<T>::operator =(TList<T>& \_v)  {  if (this == &\_v)  return \*this;  if (this->root != 0)  {  TListElem<T>\* i = this->root;  TListElem<T>\* p = 0;  while (i != 0)  {  p = i;  i = i->GetNext();  delete p;  }  this->root = 0;  this->end = 0;  count = 0;  }  count = \_v.count;  TListElem<T>\* i = \_v.root;  TListElem<T>\* j = this->root;  TListElem<T>\* p = 0;  while (i != 0)  {  j = new TListElem<T>(\*i);  j->SetNext(0);  if (p != 0)  {  p->SetNext(j);  j->SetPrev(p);  }  p = j;  if (root == 0)  {  root = j;  }  end = j;  i = i->GetNext();  }  return \*this;  }  template<class T>  inline void TList<T>::InsFirst(T d)  {  TListElem<T>\* temp = new TListElem<T>(d);  temp->SetNext(root);  root = temp;  count++;  }  template<class T>  inline void TList<T>::InsLast(T d)  {  TListElem<T>\* temp = new TListElem<T>(d);  temp->SetNext(end);  end = temp;  count++;  }  template<class T>  inline void TList<T>::Ins(TListElem<T>\* e, T d)  {  TListElem<T>\* temp = new TListElem<T>(d);  temp->SetNext(e->GetNext());  temp->SetPrev(e);  e->GetNext()->SetPrev(temp);  e->SetNext(temp);  count++;  }  template<class T>  inline bool TList<T>::IsFull(void) const  {  try  {  TListElem<T>\* temp = new TListElem<T>(1);  delete temp;  return false;  }  catch (...)  {  return true;  }  }  template<class T>  inline bool TList<T>::IsEmpty(void) const  {  return count == 0;  }  template<class T>  inline TListElem<T>\* TList<T>::GetFirst()  {  return root;  }  template<class T>  inline TListElem<T>\* TList<T>::GetLast()  {  return end;  }  template<class T>  inline void TList<T>::DelFirst()  {  TListElem<T>\* i = this->root;  root = root->GetNext();  count--;  delete i;  }  template<class T>  inline void TList<T>::DelLast()  {  TListElem<T>\* i = this->end;  end = root->GetPrev();  count--;  delete i;  }  template<class T>  inline void TList<T>::Del(TListElem<T>\* e)  {  e->GetPrev()->SetNext(e->GetNext());  e->GetNext()->SetPrev(e->GetPrev());  count--;  delete e;  }  template<class T>  inline int TList<T>::GetCount()  {  return count;  }  #endif  template<class T>  inline TListElem<T>::TListElem(T \_data)  {  data = \_data;  next = 0;  prev = 0;  }  template<class T>  inline TListElem<T>::~TListElem()  {  next = 0;  prev = 0;  }  template<class T>  inline T TListElem<T>::GetData()  {  return data;  }  template<class T>  inline TListElem<T>\* TListElem<T>::GetNext()  {  return next;  }  template<class T>  inline TListElem<T>\* TListElem<T>::GetPrev()  {  return prev;  }  template<class T>  inline void TListElem<T>::SetData(T \_data)  {  data = \_data;  }  template<class T>  inline void TListElem<T>::SetNext(TListElem\* \_next)  {  next = \_next;  }  template<class T>  inline void TListElem<T>::SetPrev(TListElem\* \_prev)  {  prev = \_prev;  } |