МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ  
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования   
**«Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского» (ННГУ)**

**Институт информационных технологий, математики и механики**

Направление подготовки: «Фундаментальная информатика и информационные технологии»

**ОТЧЕТ**

по лабораторной работе

**ДИНАМИЧЕСКАЯ СТРУКТУРА ХРАНЕНИЯ ДАННЫХ «ИЕРАРХИЧЕСКИЙ СПИСОК»**

**Выполнил:** студент группы

382006-2

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_В.Н. Грачёв

Подпись

**Проверил:** к.ф.-м.н., доц.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ К.А. Баркалов

Подпись

Нижний Новгород  
2022

Оглавление

[1. Введение 2](#_Toc101192869)

[2. Постановка задачи 2](#_Toc101192870)

[3. Описание алгоритмов 3](#_Toc101192871)

[Описание алгоритма работы динамической структуры данных «Иерархический список». 3](#_Toc101192872)

[Структура звена списка 3](#_Toc101192873)

[Общая структура иерархического списка 3](#_Toc101192874)

[Вставка новой строки 4](#_Toc101192875)

[Удаление записи 5](#_Toc101192876)

[Навигация по списку 6](#_Toc101192877)

[Работа с файлами 7](#_Toc101192878)

[Вывод текста на консоль 8](#_Toc101192879)

[Копирование текста 8](#_Toc101192880)

[Описание алгоритма работы сборщика мусора для класса TText. 9](#_Toc101192881)

[Структура TMemory 9](#_Toc101192882)

[Модификация звена иерархического списка – TTextNode 9](#_Toc101192883)

[4. Описание программы 11](#_Toc101192884)

[При запуске TText\_Test.cpp 11](#_Toc101192885)

[5. Результаты 17](#_Toc101192886)

[6. Заключение 17](#_Toc101192887)

[7. Литература 17](#_Toc101192888)

[8. Приложение 18](#_Toc101192889)

[TTextNode.h 18](#_Toc101192890)

[TText.h 20](#_Toc101192891)

[Control.h 26](#_Toc101192892)

[TText\_Test.cpp 27](#_Toc101192893)

# Введение

В программах часто встречается такая структура, при которой некоторые элементы-потомки подчинены одному элементу-предку. Если представить такую систему в виде текста, то получится, что в нем будут заголовки, подзаголовки, строки и т.д. Применение такого подхода можно встретить в модульном и структурном программировании. В этой работе будет представлено как реализовать такой иерархический текст и на его примере можно будет продемонстрировать возможности структуры, предполагающей подчинение одних элементов другим.

# Постановка задачи

Необходимо разработать класс TText, который будет содержать в себе реализацию структуры «Иерархический список». Она должна включать в себя следующие возможности: вставка и удаление элементов с переподчинением других частей, навигация по списку, запись текста в файл и его чтение из файла.

Также необходимо реализовать собственный сборщик мусора, который будет использоваться при обработке данных, хранящихся в списке. Нужно будет продемонстрировать состояние памяти до сборки мусора и после.

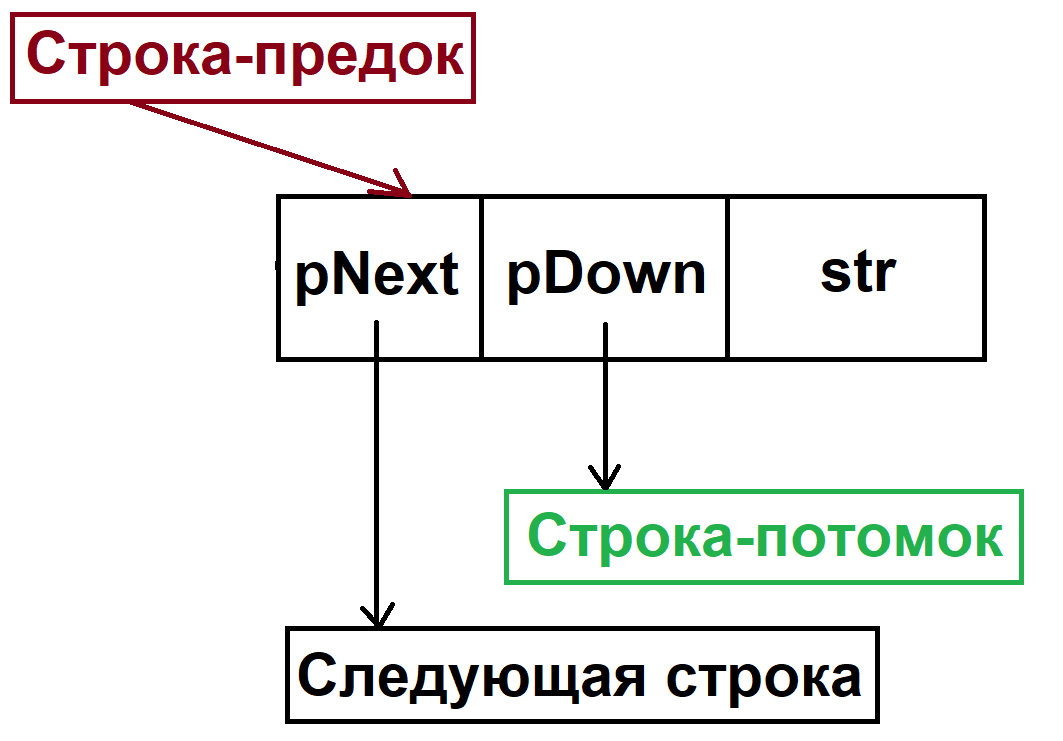
Организовать программу в виде консольного приложения, в котором демонстрируются реализованные функции.

# Описание алгоритмов

## Описание алгоритма работы динамической структуры данных «Иерархический список».

### Структура звена списка

Иерархический список должен состоять из звеньев, которые содержат в себе строку, указатель на следующий элемент и указатель на элемент-потомок:



### Общая структура иерархического списка

Иерархический список – это множество вышеопределенных звеньев, которые связаны между собой через указатели pNext или pDown. Для хранения этих звеньев, необходимо хранить указатель на первый элемент списка. Также для навигации по списку нам понадобится стек – в нем будем хранить особым образом путь к текущему элементу. Для навигации понадобится указатель на текущий элемент. Также необходимо реализовать следующие методы класса TText:

* 4 способа вставки записи:
  + вставить как следующую строку,
  + вставить как подчиненную строку,
  + вставить как следующую строку, которая переподчинит себе следующие строки,
  + вставить как подчиненную строку, которая переподчинит себе подчиненные строки.
* 2 способа удаления записи:
  + удалить следующую строку,
  + удалить подчиненную строку.
* Навигация по списку:
  + перейти к следующему элементу,
  + перейти к подчиненному элементу,
  + вернуться к предыдущему элементу,
  + полный обход списка.
* Работа с файлами:
  + печать текста в файл,
  + загрузка текста из файла.
* Вывод текста на консоль.
* Копирование текста.

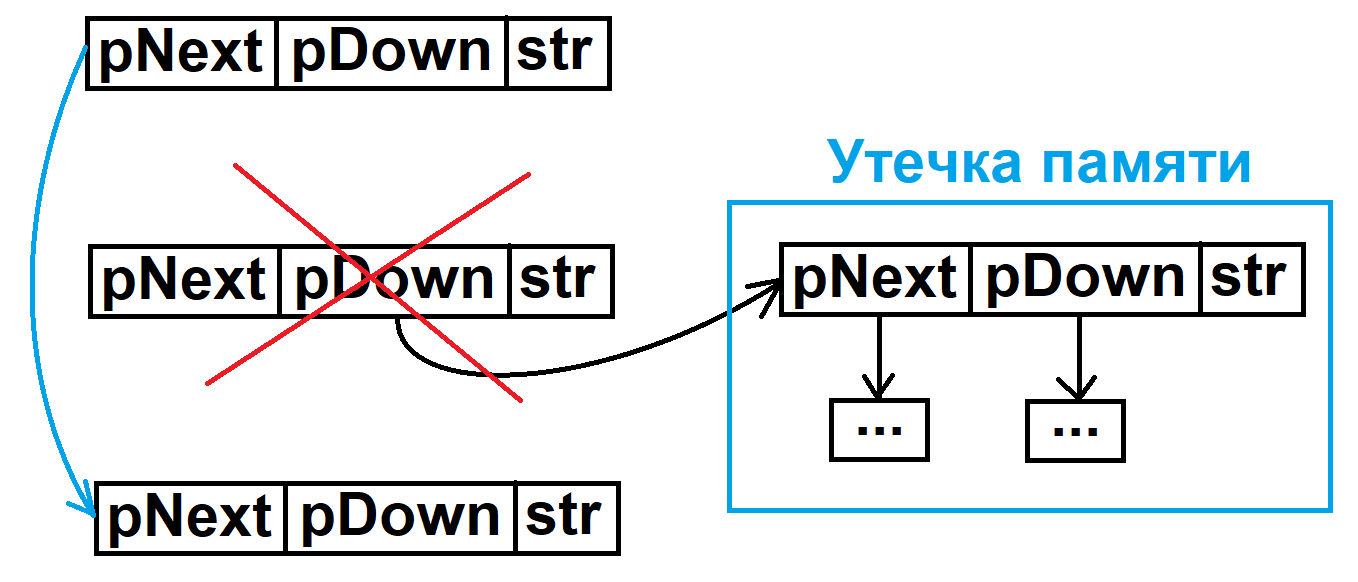
### Вставка новой строки

Во время вставки новой строки необходимо выделить память под новое звено списка. Затем, в зависимости от того, что требуется сделать, произвести следующие действия:

* *Вставить как следующую строку*:   
  Указатель pNext нового элемента ставим на pNext текущего элемента, затем указатель pNext текущего элемента ставим на новый элемент.
* *Вставить как подчиненную строку*:  
  Указатель pNext нового элемента ставим на pDown текущего элемента, затем указатель pDown текущего элемента ставим на новый элемент.
* *вставить как следующую строку, которая переподчинит себе следующие строки:*Указатель pDown нового элемента ставим на pNext текущего элемента, затем указатель pNext текущего элемента ставим на новый элемент.
* *вставить как подчиненную строку, которая переподчинит себе подчиненные строки:*Указатель pDown нового элемента ставим на pDown текущего элемента, затем указатель pDown текущего элемента ставим на новый элемент.

### Удаление записи

Рассмотрим реализацию удаления с потенциальной утечкой памяти. То есть, если нужно удалить следующий элемент – перебрасываем указатель pNext текущего элемента на pNext следующего элемента, а затем следующий элемент, будучи уже не связан с текстом, удаляем операцией delete. При таком подходе мы не сможем очистить память, которая была выделена под подчиненные удаленному элементу звенья. Аналогичная ситуация произойдет и с удалением подчиненного элемента.



Избавиться от утечки памяти можно при помощи создания алгоритма сборки мусора – его реализация будет описана чуть позже.

### Навигация по списку

Для того, чтобы можно было получить доступ к любой строке в иерархическом списке необходимо реализовать систему навигации по нему. Для этого достаточно иметь возможность переводить указатель на текущий элемент pCurrent к следующему, подчиненному, предыдущему элементам.

Переход к следующему элементу осуществляется установкой pCurrent на его указатель pNext, но перед этим следует в стек, который хранит указатели на элементы, составляющие путь до текущего элемента, добавить pCurrent.

Переход к подчиненному элементу осуществляется точно также, только pCurrent ставится на pDown текущего элемента.

Чтобы перейти к предыдущему элементу необходимо pCurrent перевести на извлеченный из стека указатель.

Для того, чтобы произвести полный обход списка, понадобятся дополнительные методы: Reset() – переход на начало списка, GoNext() – переход к следующему по обходу элементу списка, IsEnd() – проведен ли полный обход списка. Для того, чтобы навигация никак не влияла на полный обход списка, нужно завести 2 отдельных стека, а также 2 отдельных указателя на текущий элемент для полного обхода и навигации.

Будем выполнять обход следующим образом:  
В начале производим Reset() – pFirst становится pCurrent`ом, стек очищается, затем в этот стек кладем pCurrent. Метод GoNext() будет производить извлечение указателя из стека, присваивать его pCurrent`у и проверять, указывают ли указатели pDown и pNext на ненулевые элементы. В том случае, если pDown не указывает на null – необходимо добавить стек указатель pDown, аналогично с pNext – если не указывает на null – то добавляем в стек указатель pNext. Метод IsEnd() будет возвращать true – если стек пуст, иначе – false.  
Таким образом, полный обход списка можно реализовать при помощи одного цикла: for (Reset(); !IsEnd(); GoNext()).

### Работа с файлами

Печатать текст в файл будем при помощи использования рекурсивной функции. Она будет выполнять следующие действия: печатать строку, взятую из переданного в качестве параметра элемента, затем проверять, не указывает ли указатель pDown на null. Если нет – тогда следует напечатать в файл открывающую скобку «{», затем вызвать рекурсивную функцию заново, передав ей в качестве параметра указатель pDown. После того, как будет напечатана вся подчиненная часть, следует добавить закрывающую скобку «}», а затем, если pNext не равен null, вызвать рекурсивную функцию, передав ей параметром указатель pNext. В результате в файле мы увидим напечатанные строки и открывающие/закрывающие скобки, которые означают начало/конец подчиненной заголовку части.

Считывание из файла также будет проводиться с помощью рекурсивной функции. Мы считываем строку из файла до тех пор, пока не дойдем до его конца. Получив строку в первый раз, мы должны записать ее в заранее созданный элемент pHead, а затем присвоить pTemp (также был создан заранее) указатель на pHead. Получая строки далее, мы будем присваивать pNext`у, взятому pTemp, новый элемент с полученной строкой. Если в качестве полученной строки придет открывающая скобка – это значит начало вложенной части, нужно в pDown, взятый у pTemp записать то значение, которое вернет вновь вызванная рекурсивная функция (она уже будет считывать из файла с того момента, на котором остановилась прошлая рекурсивная функция). Если же в качестве строки пришла закрывающая скобка, означающая конец вложенной части, то настала пора вернуть значение pHead. В случае, если файл закончился, также нужно вернуть значение pHead.

### Вывод текста на консоль

Вывод текста на консоль будет осуществляться также при помощи рекурсивной функции. Параметрами она будет принимать элемент для печати, поток вывода и целое значение глубины. В теле функции будет производиться печать принятого через параметр элемента, затем эта же функция будет вызвана для pNext и pDown (в случае, если они не равны null). Если мы вызываем рекурсивную функцию для pDown, то значение глубины для нее будет на единицу больше. Значением глубины можно регулировать количество табуляций перед печатью строки. Также, чтобы отобразить на консоли текущий элемент, следует сделать проверку указателей перед печатью – если указатель на печатаемый элемент равен pCurrent, то следует перед строкой напечатать стрелочку «=>». Данная рекурсивная функция начинает свою работу с момента ее вызова с передачей ей указателя pFirst.

### Копирование текста

Копирование одного текста в другой будет проводиться также с помощью рекурсивной функции, которая принимает параметром копируемый элемент и возвращает новый скопированный элемент. В теле функции производится создание нового элемента, в него копируется строка пришедшего через параметр элемента, затем, если у копируемого элемента указатели pNext и pDown не равны null, то нужно pNext и pDown нового элемента присвоить значение, которое вернет вызванная рекурсивная функция с передачей ей в качестве параметра pNext или pDown копируемого элемента.

Рекурсивная функция вызывается в нерекурсивной функции GetCopy(), которая возвращает указатель на новый скопированный текст. Рекурсия начинает работу с указателя pFirst.

## Описание алгоритма работы сборщика мусора для класса TText.

Выше мы столкнулись с проблемой утечки памяти при удалении элементов из иерархического списка. Это можно исправить, реализовав собственный алгоритм сборки мусора и переопределив операции выделения удаления памяти new и delete.

### Структура TMemory

В структуре TMemory будут храниться 3 указателя на звенья списка: pFirst – начало массива, pLast – конец массива, pFree – указатель на первый свободный элемент массива.

### Модификация звена иерархического списка – TTextNode

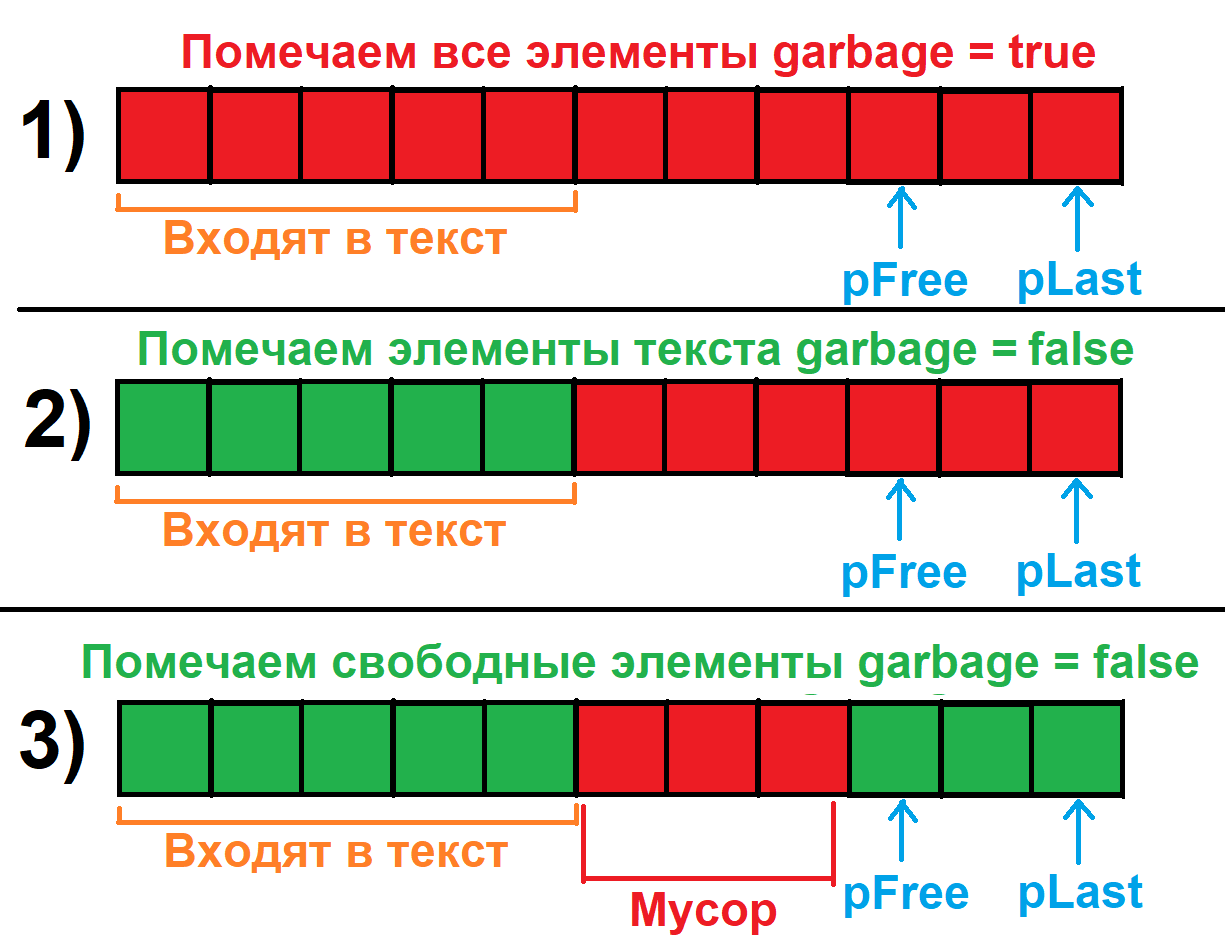
Объявим в структуре TTextNode статическое поле TMemory – это будет наша заранее подготовленная память с которой будут взаимодействовать элементы текста. Помимо этого, необходимо добавить поле – булевую переменную garbage, которое показывает, является ли данный элемент мусорным (с которым произошла утечка памяти).

Необходимо переопределить операции new и delete.   
Память под элементы будет выделяться из структуры TMemory – здесь возьмем первый свободный элемент по указателю pFree, а затем переместим pFree на его pNext. Если выйдет так, что нам вернется указатель на null – это будет значить, что свободная память закончилась и необходимо провести очистку памяти.  
При удалении элемента из иерархического списка необходимо сделать так, чтобы pFree стал указателем на удаленный элемент, а pNext удаленного элемента показывал на ранний pFree.

Для того, чтобы можно было пользоваться памятью, которая предоставляется структурой TMemory, нужно проинициализировать ее – выделить нужное количество памяти, используя стандартный new, поставить указатель pFree на pFirst, pFirst – на первый, pLast – на последний элемент в массиве. Функция инициализации памяти будет вызываться в конструкторе списка.

Также, в целях демонстрации, реализуем метод PrintFree(), который будет печатать строки всех свободных элементов. Для этого нужно пройтись циклом по массиву от pFree до pLast и вывести строки на экран.

Очистка памяти будет производиться в классе TText. Чтобы идентифицировать мусорные элементы, необходимо пройтись по всему массиву памяти и выставить garbage = true для каждого элемента. Затем, нужно произвести полный обход списка и всем элементам, которые встретились в текста выставить garbage = false. Аналогично пройдемся по массиву от pFree до pLast и тоже встретившимся элементам выставим garbage = false. После такой процедуры у нас под флагом garbage = true останутся только мусорные элементы, которые не входят в текст и не отнесены к свободной памяти. Обходом по массиву от pFirst до pLast выполняем переопределенную ранее операцию delete для тех элементов, у которых garbage = true.



# Описание программы

Программа содержит в себе 4 файла:

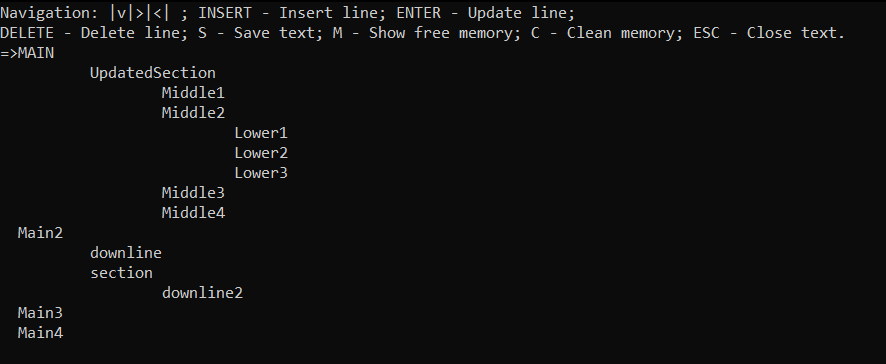
* TTextNode.h – содержит реализацию функционала звена иерархического текста, а также обеспечивает работу с TMemory.
* TText.h – содержит реализацию иерархического списка и методов класса.
* Control.h – содержит функции управления консолью приложения.
* TText\_Test.cpp – здесь находится алгоритм программы, позволяющий работать с текстом в консоли.

Содержание файлов можно найти в приложении.

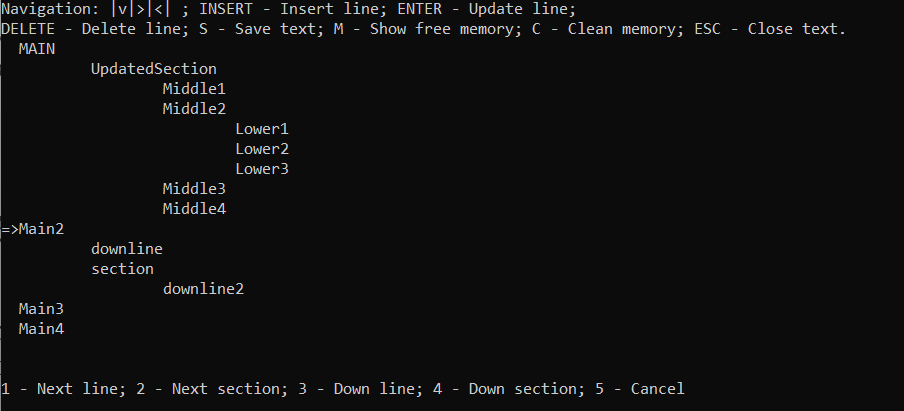
### При запуске TText\_Test.cpp

В начале необходимо указать название текстового файла, который содержит записанный в него иерархический текст.

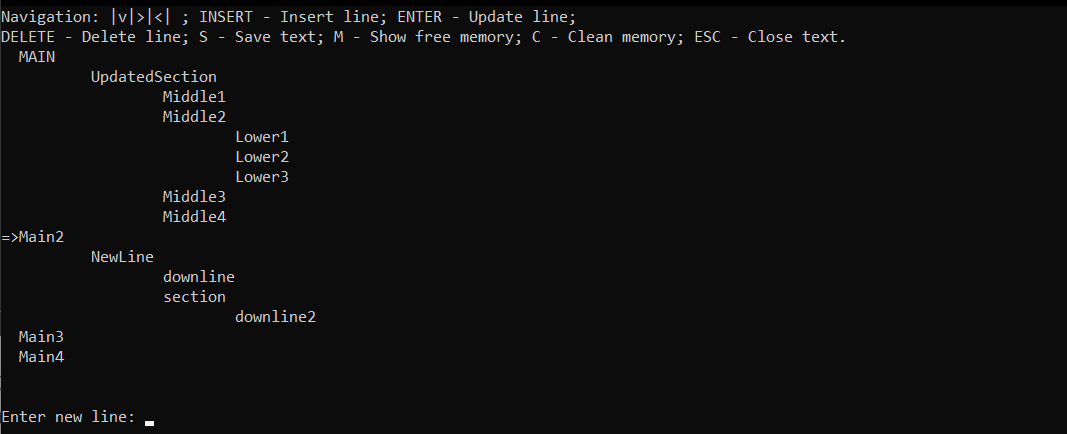
Далее появится следующая панель:



Сверху описаны элементы управления консолью: при помощи стрелок можно перемещать курсор по строкам. Ниже представлен иерархический список.

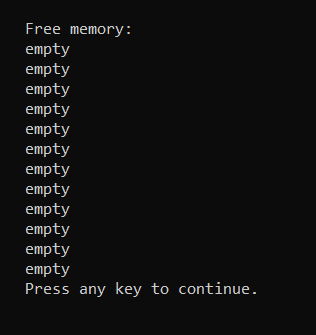
**Insert – вставить запись:**

После выбора нужного режима вставки, необходимо указать название новой строки, после чего произойдет вставка записи. Например, вставим переподчиняющую подстроку «NewLine» относительно строки «Main2».

**Enter – обновить запись:**

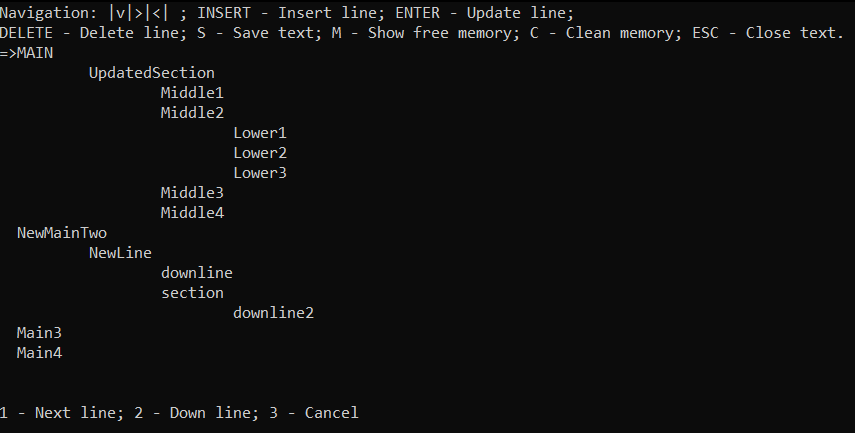
Заменим строку «Main2» на «NewMainTwo».

**Клавиша М – показать свободную память:**

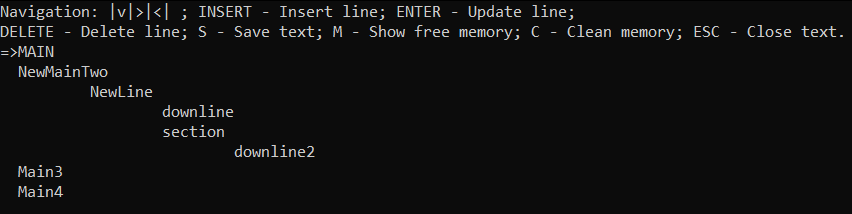


Как видим, в свободных ячейках памяти находятся незадействованные в тексте элементы.

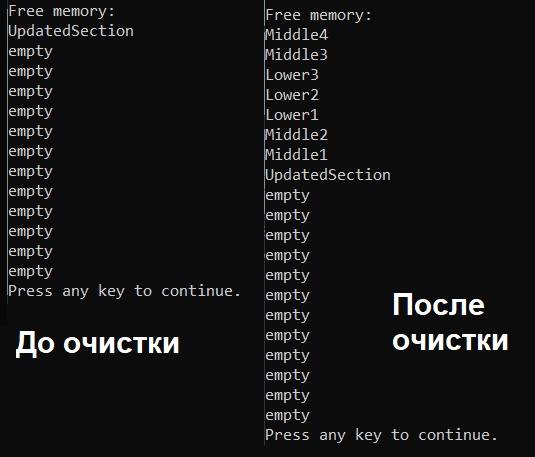
**Delete – удалить элемент:**

При нажатии клавиши необходимо выбрать режим удаления:

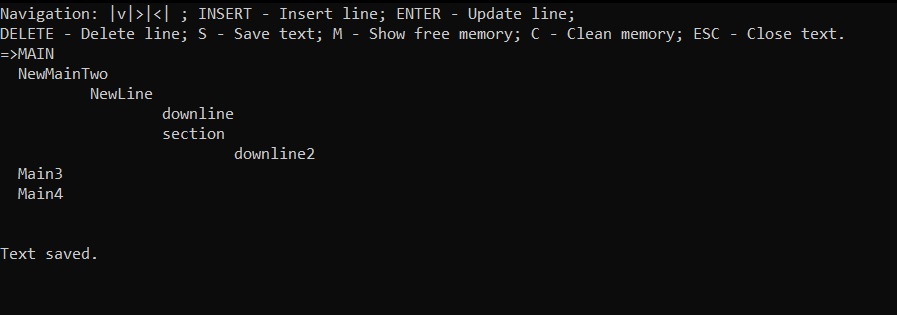
Удалим, к примеру, строку «Updated Section» – переместим курсор на строку «MAIN» и выберем Delete Down line.



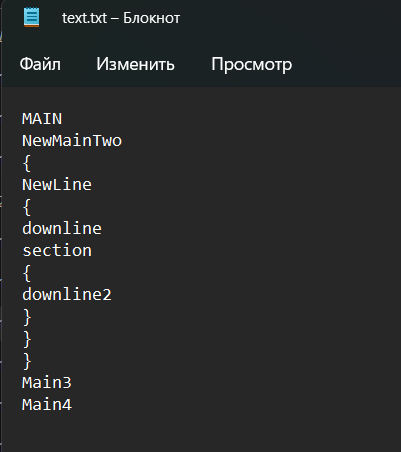
**Теперь посмотрим что находится в свободной памяти до и после ее чистки:**

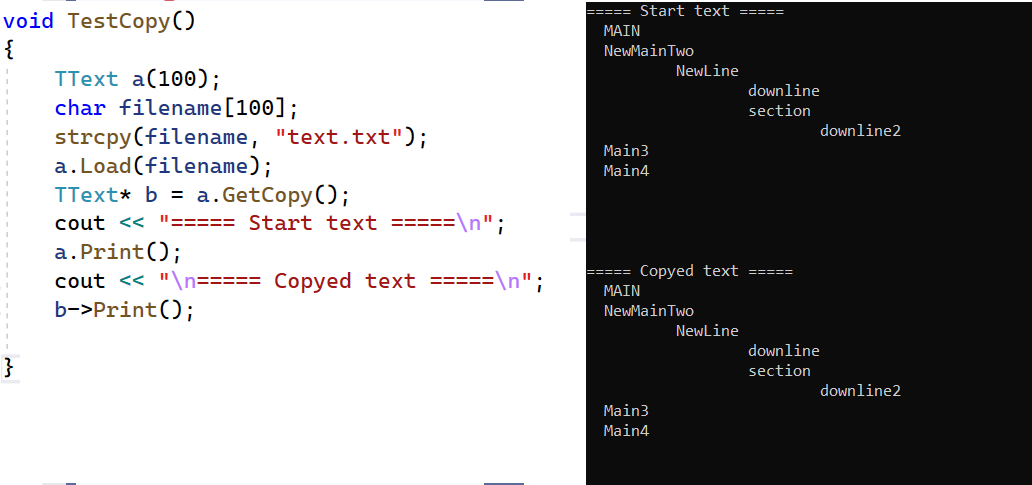


Как видим, после очистки в свободную память перешли все строки, которые были подчинены недавно удаленной строке.

**Клавиша S – сохранение текста в файл:**

Текст сохранен в файл «text.txt». Откроем его:



Проведем копирование текста. Для этого в теле функции main() вызовем функцию TestCopy() вместо TestConsole() – в функции TestCopy() происходит считывание информации в текст А из файла «text.txt», затем у текста А вызывается метод GetCopy() и возвращаемое значение передается тексту B:

# Результаты

Был разработан и протестирован класс для работы с иерархическим списком, произведена демонстрация его работоспособности. Также был реализован собственный сборщик мусора и применен для устранения проблемы с утечкой памяти при удалении элементов из текста.

# Заключение

Таким образом, мы увидели как работает динамическая структура хранения данных «Иерархический список» и успешно применили к нему собственный сборщик мусора. Данные наработки можно эффективно применять при разработке более масштабных программ.

# Литература

1. Microsoft [электронный ресурс]: // Документация по языку C++

URL: <https://docs.microsoft.com/ru-ru/cpp/cpp/>

1. Р. Лафоре Объектно-ориентированное программирование в C++ – Санкт-Петербург: “Питер”, 2018
2. <https://ru.wikipedia.org/wiki/Иерархическая_модель_данных>  
   ***Ссылка на GitHub на репозиторий с проектом:***<https://github.com/TechnoWolf96/Lab-Hierarchy-Text>

# Приложение

## TTextNode.h

#pragma once

#include <iostream>

#include <cstring>

using namespace std;

struct TTextNode;

struct TMemory

{

TTextNode\* pFirst;

TTextNode\* pFree;

TTextNode\* pLast;

};

struct TTextNode

{

static TMemory memory;

char str[81];

TTextNode\* pNext, \* pDown;

bool garbage;

TTextNode(char\* str = nullptr)

{

this->pNext = nullptr;

this->pDown = nullptr;

if (str == nullptr) this->str[0] = '\0';

else strcpy(this->str, str);

}

static void InitializeMemory(size\_t size)

{

memory.pFirst = (TTextNode\*) new char[size \* sizeof(TTextNode)];

memory.pFree = memory.pFirst;

memory.pLast = memory.pFirst + (size - 1);

TTextNode\* p = memory.pFirst;

for (int i = 0; i < size - 1; i++)

{

p->pNext = p + 1;

strcpy(p->str, "empty");

p->garbage = true;

p += 1;

}

memory.pLast->pNext = nullptr;

strcpy(memory.pLast->str, "empty");

}

static void PrintFree()

{

TTextNode\* p = memory.pFree;

while (p != nullptr)

{

cout << p->str << endl;

p = p->pNext;

}

}

void\* operator new(size\_t size)

{

TTextNode\* tmp = memory.pFree;

if (tmp == nullptr) throw "Have not memory";

memory.pFree = memory.pFree->pNext;

return tmp;

}

void operator delete(void\* p)

{

TTextNode\* pNode = (TTextNode\*)p;

pNode->pNext = memory.pFree;

memory.pFree = pNode;

}

~TTextNode() {}

};

## TText.h

#include "../TList/TStackList.h"

#include "TTextNode.h"

using namespace std;

class TText

{

private:

TTextNode\* pFirst, \* navCurrent, \*ptCurrent;

TStackList<TTextNode\*> navStack, ptStack;

TTextNode\* ReadRecursion(ifstream& is);

void PrintFileRecursion(TTextNode\* pNode, ofstream& os);

void PrintRecursion(TTextNode\* pNode, ostream& os, int level);

TTextNode\* CopyNode(TTextNode\* pCopyable);

public:

TText(int memorySize = 100)

{

pFirst = nullptr;

navCurrent = nullptr;

ptCurrent = nullptr;

TTextNode::InitializeMemory(memorySize);

}

TText\* GetCopy();

void CleanMemory();

char\* GetNavCurrent();

void SetNavCurrent(char\* str);

char\* GetPtCurrent();

void SetPtCurrent(char\* str);

void Reset();

void GoNext();

bool IsEnd();

// Navigation

void GoNextLine();

void GoBackLine();

void GoDownLine();

void GoFirstLine();

void InsertNextLine(char\* str); // Insert string after current

void InsertNextSection(char\* str); // Insert string to the same level with subjection next lines

void InsertDownLine(char\* str); // Insert string to the first position on the lower level

void InsertDownSection(char\* str); // Insert subjected header of the lower level

void DeleteNextLine();

void DeleteDownLine();

// Filestream

bool Load(char\* filename);

void Print(char\* filename);

// Console

void Print();

void ShowAll();

};

inline TTextNode\* TText::ReadRecursion(ifstream& is)

{

TTextNode\* pHead = nullptr, \* pTemp = nullptr;

char str[81];

while (!is.eof())

{

is.getline(str, 81, '\n');

if (str[0] == '{') pTemp->pDown = ReadRecursion(is);

else if (str[0] == '}') break;

else

{

TTextNode\* newNode = new TTextNode(str);

if (pHead == nullptr)

pTemp = pHead = newNode;

else

{

pTemp->pNext = newNode;

pTemp = newNode;

}

}

}

return pHead;

}

inline void TText::PrintFileRecursion(TTextNode\* pNode, ofstream& file)

{

if (pNode != nullptr)

{

file << pNode->str << endl;

if (pNode->pDown != nullptr)

{

file << '{' << endl;

PrintFileRecursion(pNode->pDown, file);

file << '}' << endl;

}

PrintFileRecursion(pNode->pNext, file);

}

}

inline void TText::PrintRecursion(TTextNode\* pNode, ostream& os, int level)

{

if (pNode != nullptr)

{

for (int i = 0; i < level; i++) os << "\t";

if (pNode == navCurrent) os << "=>" << pNode->str << endl;

else os << " " << pNode->str << endl;

if (pNode->pDown != nullptr)

{

PrintRecursion(pNode->pDown, os, level+1);

}

PrintRecursion(pNode->pNext, os, level);

}

}

inline TTextNode\* TText::CopyNode(TTextNode\* pCopyable)

{

TTextNode\* pCopyed = new TTextNode(pCopyable->str);

if (pCopyable->pDown != nullptr) pCopyed->pDown = CopyNode(pCopyable->pDown);

if (pCopyable->pNext != nullptr) pCopyed->pNext = CopyNode(pCopyable->pNext);

return pCopyed;

}

inline TText\* TText::GetCopy()

{

TText\* res = new TText();

res->pFirst = CopyNode(pFirst);

return res;

}

inline void TText::CleanMemory()

{

for (Reset(); !IsEnd(); GoNext()) ptCurrent->garbage = false;

TTextNode\* p = TTextNode::memory.pFree;

while (p != nullptr)

{

p->garbage = false;

p = p->pNext;

}

for (p = TTextNode::memory.pFirst; p <= TTextNode::memory.pLast; p++)

{

if (p->garbage) delete p;

p->garbage = true;

}

}

inline char\* TText::GetNavCurrent()

{

if (navCurrent == nullptr) throw "Pointer navCurrent is null";

return navCurrent->str;

}

inline void TText::SetNavCurrent(char\* str)

{

if (navCurrent == nullptr) throw "Pointer navCurrent is null";

strcpy(navCurrent->str, str);

}

inline char\* TText::GetPtCurrent()

{

if (ptCurrent == nullptr) throw "Pointer ptCurrent is null";

return ptCurrent->str;

}

inline void TText::SetPtCurrent(char\* str)

{

if (ptCurrent == nullptr) throw "Pointer ptCurrent is null";

strcpy(ptCurrent->str, str);

}

inline void TText::Reset()

{

ptStack.Clear();

if (pFirst == nullptr) return;

ptCurrent = pFirst;

ptStack.Push(ptCurrent);

if (ptCurrent->pNext != nullptr) ptStack.Push(ptCurrent->pNext);

if (ptCurrent->pDown != nullptr) ptStack.Push(ptCurrent->pDown);

}

inline void TText::GoNext()

{

ptCurrent = ptStack.Pop();

if (ptCurrent == pFirst) return;

if (ptCurrent->pNext != nullptr) ptStack.Push(ptCurrent->pNext);

if (ptCurrent->pDown != nullptr) ptStack.Push(ptCurrent->pDown);

}

inline bool TText::IsEnd()

{

return ptStack.IsEmpty();

}

inline void TText::GoNextLine()

{

if (navCurrent == nullptr || navCurrent->pNext == nullptr) throw "Current pointer don`t exist";

navStack.Push(navCurrent);

navCurrent = navCurrent->pNext;

}

inline void TText::GoBackLine()

{

if (navCurrent == nullptr || navStack.IsEmpty()) throw "Current pointer don`t exist";

TTextNode\* backNode = navStack.Pop();

navCurrent = backNode;

}

inline void TText::GoDownLine()

{

if (navCurrent == nullptr || navCurrent->pDown == nullptr) throw "Current pointer don`t exist";

navStack.Push(navCurrent);

navCurrent = navCurrent->pDown;

}

inline void TText::GoFirstLine()

{

navCurrent = pFirst;

navStack.Clear();

}

inline void TText::InsertNextLine(char\* str)

{

if (navCurrent == nullptr) throw "Current pointer don`t exist";

TTextNode\* newNode = new TTextNode(str);

newNode->pNext = navCurrent->pNext;

navCurrent->pNext = newNode;

}

inline void TText::InsertNextSection(char\* str)

{

if (navCurrent == nullptr) throw "Current pointer don`t exist";

TTextNode\* newNode = new TTextNode(str);

newNode->pDown = navCurrent->pNext;

navCurrent->pNext = newNode;

}

inline void TText::InsertDownLine(char\* str)

{

if (navCurrent == nullptr) throw "Current pointer don`t exist";

TTextNode\* newNode = new TTextNode(str);

newNode->pNext = navCurrent->pDown;

navCurrent->pDown = newNode;

}

inline void TText::InsertDownSection(char\* str)

{

if (navCurrent == nullptr) throw "Current pointer don`t exist";

TTextNode\* newNode = new TTextNode(str);

newNode->pDown = navCurrent->pDown;

navCurrent->pDown = newNode;

}

inline void TText::DeleteNextLine()

{

if (navCurrent != nullptr)

{

TTextNode\* pDeletable = navCurrent->pNext;

if (pDeletable != nullptr)

{

navCurrent->pNext = pDeletable->pNext;

delete pDeletable;

}

}

}

inline void TText::DeleteDownLine()

{

if (navCurrent != nullptr)

{

TTextNode\* pDeletable = navCurrent->pDown;

if (pDeletable != nullptr)

{

navCurrent->pDown = pDeletable->pNext;

delete pDeletable;

}

}

}

bool TText::Load(char\* filename)

{

ifstream file(filename);

if (file.bad()) return false;

pFirst = ReadRecursion(file);

return true;

}

inline void TText::Print(char\* filename)

{

ofstream file(filename);

PrintFileRecursion(pFirst, file);

}

inline void TText::Print()

{

PrintRecursion(pFirst, cout, 0);

}

inline void TText::ShowAll()

{

for (Reset(); !IsEnd(); GoNext())

cout << GetPtCurrent() << endl;

}

## Control.h

#pragma once

#include <Windows.h>

void clrscr(void)

{

HANDLE Console = GetStdHandle(STD\_OUTPUT\_HANDLE);

if (!Console) return;

CONSOLE\_SCREEN\_BUFFER\_INFO buf;

GetConsoleScreenBufferInfo(Console, &buf);

DWORD Count;

COORD zpos;

zpos.X = 0;

zpos.Y = 0;

FillConsoleOutputCharacter(Console, ' ', buf.dwSize.X \* buf.dwSize.Y, zpos, &Count);

SetConsoleCursorPosition(Console, zpos);

}

void clreol(void)

{

HANDLE Console = GetStdHandle(STD\_OUTPUT\_HANDLE);

if (!Console)return;

CONSOLE\_SCREEN\_BUFFER\_INFO buf;

GetConsoleScreenBufferInfo(Console, &buf);

WORD Attr;

DWORD Count;

COORD zpos = buf.dwCursorPosition;

zpos.X = buf.dwCursorPosition.X + 1;

zpos.Y = buf.dwCursorPosition.Y;

FillConsoleOutputCharacter(Console, ' ', buf.dwSize.X - buf.dwCursorPosition.X - 1, zpos, &Count);

SetConsoleCursorPosition(Console, buf.dwCursorPosition);

}

void gotoxy(int x, int y)

{

HANDLE Console = GetStdHandle(STD\_OUTPUT\_HANDLE);

if (!Console)return;

COORD pos;

pos.X = x;

pos.Y = y;

SetConsoleCursorPosition(Console, pos);

}

## TText\_Test.cpp

#include "TText.h"

#include "Control.h"

#include <conio.h>

#define DOWN 80

#define NEXT 77

#define UP 72

#define ESC 27

#define INSERT 82

#define DELETE 83

#define ENTER 13

TMemory TTextNode::memory;

void TextConsole()

{

char filename[100];

bool saved = false;

begin:

while (true)

{

clrscr();

cout << "Enter filename: ";

cin >> filename;

TText text(30);

text.Load(filename);

text.GoFirstLine();

while (true)

{

clrscr();

cout << "Navigation: |v|>|<| ; INSERT - Insert line; ENTER - Update line;" << endl;

cout << "DELETE - Delete line; S - Save text; M - Show free memory; C - Clean memory; ESC - Close text." << endl;

text.Print();

if (saved) cout << "Text saved." << endl;

saved = false;

char control = \_getch();

switch (control)

{

case DOWN: try { text.GoNextLine(); }

catch (...) {} break;

case UP: try { text.GoBackLine(); }

catch (...) {} break;

case NEXT: try { text.GoDownLine(); }

catch (...) {} break;

case INSERT:

{

char str[81];

cout << "1 - Next line; 2 - Next section; 3 - Down line; 4 - Down section; 5 - Cancel" << endl;

char a = \_getch();

if (a == '5') break;

cout << "Enter line: ";

cin >> str;

switch (a)

{

case '1': text.InsertNextLine(str); break;

case '2': text.InsertNextSection(str); break;

case '3': text.InsertDownLine(str); break;

case '4': text.InsertDownSection(str); break;

}

break;

}

case ENTER:

char str[81];

cout << "Enter new line: ";

cin >> str;

text.SetNavCurrent(str);

break;

case 's':

text.Print(filename);

saved = true;

break;

case ESC:

goto begin;

break;

case DELETE:

{

cout << "1 - Next line; 2 - Down line; 3 - Cancel" << endl;

char a = \_getch();

if (a == '3') break;

switch (a)

{

case '1': text.DeleteNextLine(); break;

case '2': text.DeleteDownLine(); break;

case '3': break;

}

break;

}

case 'm':

cout << "Free memory:" << endl;

TTextNode::PrintFree();

cout << "Press any key to continue." << endl;

\_getch();

break;

case 'c':

text.CleanMemory();

cout << "Memory cleaned. Press any key to continue." << endl;

\_getch();

break;

}

}

}

}

void TestCopy()

{

TText a(100);

char filename[100];

strcpy(filename, "text.txt");

a.Load(filename);

TText\* b = a.GetCopy();

cout << "===== Start text =====\n";

a.Print();

cout << "\n===== Copyed text =====\n";

b->Print();

}

int main()

{

//TextConsole();

TestCopy();

}