Учреждение образования

«Белорусский государственный технологический университет»

Факультет информационных технологий

ТЕСТИРОВАНИЕ КОДА

  Выполнил:

Студент 1 курса 10 группы

Жамойдо Артём Игоревич

Преподаватель:

Доцент, кандидат технических наук

Белодед Николай Иванович

2024, Минск

**Содержание**

[Тестирование №1 2](#_Toc165421716)

[Тестирование №2 6](#_Toc165421717)

[Тестирование №3 10](#_Toc165421718)

[Тестирование №4 13](#_Toc165421719)

[Тестирование №5 24](#_Toc165421720)

[Тестирование №6 30](#_Toc165421721)

# **Тестирование №1**

|  |  |  |  |
| --- | --- | --- | --- |
| #include <iostream> // Подключение библиотеки для ввода/вывода  using namespace std; // Использование стандартного пространства имен  // Определение структуры узла бинарного дерева  struct node {  int Key; // Ключ узла  int Count; // Количество повторений ключа (для подсчета дубликатов)  node\* Left; // Указатель на левое поддерево  node\* Right; // Указатель на правое поддерево  };  // Определение класса для работы с бинарным деревом  class TREE {  private:  node\* Tree; // Указатель на корень дерева  void Search(int, node\*&); // Поиск вершины с ключом и вставка новой вершины  public:  TREE() { Tree = NULL; } // Конструктор класса TREE  node\* GetTree() { return Tree; } // Метод для получения указателя на корень дерева  void BuildTree(); // Метод для построения дерева  void CleanTree(node\*); // Метод для очистки дерева  void ObhodEnd(node\*); // Метод для концевого обхода дерева  void ObhodLeft(node\*); // Метод для левостороннего обхода дерева  void ObhodBack(node\*); // Метод для обратного обхода дерева  void Vyvod(node\*, int); // Метод для вывода дерева на экран  int Height(node\*); // Метод для определения высоты дерева  };  // Основная функция программы  int main() {  setlocale(LC\_ALL, "Rus"); // Установка локали для корректного отображения русского текста  TREE A; // Создаем объект класса TREE для работы с деревом  A.BuildTree(); // Построение дерева  cout << "\nВывод дерева:\n"; | | | |
| A.Vyvod(A.GetTree(), 0); // Вывод дерева на экран  cout << "\nВысота дерева:" << A.Height(A.GetTree()) << endl; // Определение и вывод высоты дерева  cout << "\nЛевосторонний обход дерева: ";  A.ObhodLeft(A.GetTree()); // Левосторонний обход дерева  cout << "\nКонцевой обход дерева: ";  A.ObhodEnd(A.GetTree()); // Концевой обход дерева  cout << "\nОбратный обход дерева: ";  A.ObhodBack(A.GetTree()); // Обратный обход дерева  A.CleanTree(A.GetTree()); // Очистка дерева  cout << "\n";  system("PAUSE"); // Приостановка выполнения программы до нажатия клавиши пользователем  return 0; // Возвращаем 0, чтобы сообщить операционной системе об успешном завершении программы  }  // Метод для построения бинарного дерева  void TREE::BuildTree() {  int el; // Переменная для ввода ключей вершин дерева  cout << "Вводите ключи вершин дерева (для завершения введите 0):\n";  cin >> el;  // Цикл для ввода ключей и построения дерева  while (el != 0) {  // Вызов метода Search для поиска и вставки новой вершины  Search(el, Tree);  cin >> el;  }  }  // Метод для поиска вершины с ключом и вставки новой вершины  void TREE::Search(int x, node\*& p) {  if (p == NULL) {  // Создание новой вершины, если текущая вершина пуста  p = new node;  // Присвоение значения ключу новой вершины  p->Key = x;  // Установка начального значения счетчика дубликатов  p->Count = 1;  // Установка указателей на поддеревья в NULL  p->Left = NULL;  p->Right = NULL;  }  else if (x < p->Key) {  // Рекурсивный поиск и вставка в левое поддерево  Search(x, p->Left);  }  else if (x > p->Key) {  // Рекурсивный поиск и вставка в правое поддерево  Search(x, p->Right);  }  else {  // Увеличение счетчика дубликатов, если ключ уже существует  p->Count++;  }  }  // Метод для левостороннего обхода дерева  void TREE::ObhodLeft(node\* w) {  if (w != NULL) {  // Вывод ключа текущей вершины  cout << w->Key << " "; | | | |
| // Рекурсивный обход левого поддерева  ObhodLeft(w->Left);  // Рекурсивный обход правого поддерева  ObhodLeft(w->Right);  }  }  // Метод для концевого обхода дерева  void TREE::ObhodEnd(node\* w) {  if (w != NULL) {  // Рекурсивный обход левого поддерева  ObhodEnd(w->Left);  // Рекурсивный обход правого поддерева  ObhodEnd(w->Right);  // Вывод ключа текущей вершины  cout << w->Key << " ";  }  }  // Метод для обратного обхода дерева  void TREE::ObhodBack(node\* w) {  if (w != NULL) {  // Рекурсивный обход левого поддерева  ObhodBack(w->Left);  // Вывод ключа текущей вершины  cout << w->Key << " ";  // Рекурсивный обход правого поддерева  ObhodBack(w->Right);  }  }  // Метод для очистки дерева  void TREE::CleanTree(node\* w) {  if (w != NULL) {  // Рекурсивная очистка левого поддерева  CleanTree(w->Left);  // Рекурсивная очистка правого поддерева  CleanTree(w->Right);  // Удаление текущей вершины  delete w;  }  }  // Метод для вывода дерева на экран  void TREE::Vyvod(node\* w, int l) {  int i; // Переменная для цикла форматирования вывода  if (w != NULL) {  // Рекурсивный вызов для правого поддерева  Vyvod(w->Right, l + 1);  // Вывод пробелов для форматирования  for (i = 1; i <= l; i++) cout << " ";  // Вывод ключа текущей вершины  cout << w->Key << endl;  // Рекурсивный вызов для левого поддерева  Vyvod(w->Left, l + 1);  }  }  // Метод для определения высоты бинарного дерева  int TREE::Height(node\* w) {  int h1, h2; // Переменные для хранения высот левого и правого поддеревьев  // Пустое дерево имеет высоту -1 | | | |
| if (w == NULL) return (-1);  else {  // Вызов метода для левого поддерева  h1 = Height(w->Left);  // Вызов метода для правого поддерева  h2 = Height(w->Right);  // Возвращение высоты левого поддерева, если она больше  if (h1 > h2) return (1 + h1);  // Возвращение высоты правого поддерева в противном случае  else return (1 + h2);  }  } | | | |

|  |
| --- |
|  |

В исходном коде была ошибка, связанная с передачей аргументов в методы класса **TREE**. Вместо передачи указателя на узел (**node\***), передавался указатель на указатель на узел (**node\*\***). Это приводило к несовместимости типов данных. Для исправления этой ошибки были внесены соответствующие изменения в определении методов класса **TREE** и их реализации, чтобы передавать указатель на узел (**node\***) вместо указателя на указатель на узел (**node\*\***).

# **Тестирование №2**

|  |  |  |  |
| --- | --- | --- | --- |
| #include <iostream> // Подключение библиотеки для ввода/вывода  using namespace std; // Использование стандартного пространства имен  // Структура узла дерева  struct node  {  int Key; // Ключ узла  int Count; // Количество узлов с данным ключом  node\* Left; // Указатель на левое поддерево  node\* Right; // Указатель на правое поддерево  };  // Структура звена стека  struct no  {  node\* elem; // Указатель на узел дерева  int ch; // Уровень вершины  no\* sled; // Указатель на следующий элемент стека  };  // Класс дерева  class TREE  {  private:  node\* Tree; // Указатель на корень дерева  void PushStack(no\*&, node\*, int\*); // Помещение элемента в стек  void PopStack(no\*&, node\*&, int\*); // Извлечение элемента из стека  void VyvodStack(no\*); // Вывод содержимого стека на экран  public:  // Конструктор класса TREE  TREE() { Tree = new node; (\*Tree).Right = NULL; } // Создание пустого дерева  node\* GetTreeRight() { return (\*Tree).Right; } // Получение указателя на правое поддерево  void TreeSearch(int); // Поиск узла в дереве  void VyvodTree(node\*); // Построение дерева на экране  };  // Главная функция программы  int main()  {  setlocale(LC\_ALL, "Rus"); // Установка русской локали для вывода на кириллице  TREE A; // Создание экземпляра класса дерева  int el; // Переменная для ввода значений узлов дерева  cout << "Вводите значения информационных полей вершин: " << endl; // Подсказка для пользователя  cin >> el; // Ввод первого значения  while (el != 0) // Цикл ввода значений узлов дерева (0 - конец ввода)  {  A.TreeSearch(el); // Вызов метода для поиска или добавления узла  cin >> el; // Ввод следующего значения  }  A.VyvodTree(A.GetTreeRight()); // Вывод дерева на экран  cout << "\n"; // Переход на новую строку  system("PAUSE"); // Ожидание нажатия клавиши пользователем  return 0; // Возврат нуля - успешное завершение программы  } | | | |
| // Метод для поиска узла в дереве и добавления нового, если узел не найден  void TREE::TreeSearch(int el)  {  node\* p1, \* p2; // Указатели на узлы дерева  int d; // Переменная для определения направления движения  p2 = Tree; // Указатель на корень дерева  p1 = (\*p2).Right; // Указатель на корень поддерева  d = 1; // Начальное направление движения  while (p1 != NULL && d != 0) // Цикл поиска узла в дереве  {  p2 = p1; // Перемещение указателя на текущий узел  if (el < (\*p1).Key) { p1 = (\*p1).Left; d = -1; } // Перемещение влево  else if (el > (\*p1).Key) { p1 = (\*p1).Right; d = 1; } // Перемещение вправо  else d = 0; // Узел найден  }  if (d == 0) (\*p1).Count = (\*p1).Count + 1; // Увеличение счетчика для найденного узла  else // Создание нового узла в дереве  {  p1 = new node; // Выделение памяти под новый узел  (\*p1).Key = el; // Присвоение ключа новому узлу  (\*p1).Left = NULL; // Начальная инициализация указателя на левое поддерево  (\*p1).Right = NULL; // Начальная инициализация указателя на правое поддерево  (\*p1).Count = 1; // Установка счетчика равным 1 для нового узла  if (d < 0) (\*p2).Left = p1; // Привязка нового узла к левому поддереву  else (\*p2).Right = p1; // Привязка нового узла к правому поддереву  }  }  // Метод для построения дерева на экране  void TREE::VyvodTree(node\* t)  {  no\* stk, \* stk1; // Указатели на стеки  node\* u; // Указатель на текущий узел  int i, n; // Переменные для уровня узла и итерации  stk = stk1 = NULL; // Инициализация указателей на стеки  n = 0; // Начальное значение уровня узла  while (t != NULL) // Цикл построения дерева на экране  {  PushStack(stk1, t, &n); // Помещение узла в стек 1  if ((\*t).Right != NULL) // Проверка наличия правого поддерева  {  if ((\*t).Left != NULL) PushStack(stk, (\*t).Left, &n); // Помещение левого поддерева в стек  t = (\*t).Right; // Перемещение к правому поддереву  }  else // Если правого поддерева нет  {  if ((\*t).Left != NULL) // Проверка наличия левого поддерева  {  if (stk1 != NULL) // Проверка наличия элементов в стеке 1  {  PopStack(stk1, u, &n); // Извлечение узла из стека 1  for (i = 0; i <= n; i++) cout << " "; // Вывод пробелов в зависимости от уровня узла  cout << (\*u).Key << endl; // Вывод ключа узла  }  t = (\*t).Left; // Перемещение к левому поддереву  } | | | |
| else if (stk == NULL) t = NULL; // Если стек пуст, завершаем процесс  else // Если в стеке есть элементы  {  while ((\*stk).elem != (\*((\*stk1).elem)).Left) // Цикл вывода элементов из стека 1  {  PopStack(stk1, u, &n); // Извлечение узла из стека 1  for (i = 0; i <= n; i++) cout << " "; // Вывод пробелов в зависимости от уровня узла  cout << (\*u).Key << endl; // Вывод ключа узла  }  PopStack(stk1, u, &n); // Извлечение узла из стека 1  for (i = 0; i <= n; i++) cout << " "; // Вывод пробелов в зависимости от уровня узла  cout << (\*u).Key << endl; // Вывод ключа узла  PopStack(stk, t, &n); // Извлечение узла из стека 2  }  }  n = n + 1; // Увеличение уровня узла  }  VyvodStack(stk1); // Вывод оставшихся узлов из стека 1  }  // Метод для помещения узла в стек  void TREE::PushStack(no\*& stk, node\* el, int\* n)  {  no\* q; // Указатель на новое звено стека  q = new no; // Выделение памяти под новое звено стека  (\*q).elem = el; // Присвоение указателю на узел значение элемента  (\*q).ch = \*n; // Присвоение уровня узла  (\*q).sled = stk; // Привязка нового звена к вершине стека  stk = q; // Обновление вершины стека  }  // Метод для извлечения узла из стека  void TREE::PopStack(no\*& stk, node\*& t, int\* n)  {  no\* q; // Указатель на звено стека  if (stk != NULL) // Проверка наличия элементов в стеке  {  t = (\*stk).elem; // Присвоение указателю на узел значение элемента стека  \*n = (\*stk).ch; // Присвоение уровня узла  q = stk; // Присвоение указателю на звено значение вершины стека  stk = (\*stk).sled; // Обновление вершины стека  delete q; // Освобождение памяти, занятой звеном стека  }  }  // Метод для вывода содержимого стека на экран  void TREE::VyvodStack(no\* stk)  {  node\* k; // Указатель на узел дерева  int i, n; // Переменные для уровня узла и итерации  while (stk != NULL) // Цикл вывода элементов стека на экран  {  k = (\*stk).elem; // Присвоение указателю на узел значение элемента стека  n = (\*stk).ch; // Присвоение уровня узла  for (i = 0; i <= n; i++) cout << " "; // Вывод пробелов в зависимости от уровня узла  cout << (\*k).Key << endl; // Вывод ключа узла  stk = (\*stk).sled; // Переход к следующему элементу стека | | | | |
| }  } | | |

|  |
| --- |
|  |

Код был исправлен и улучшен следующим образом: были исправлены ошибки в типах указателей в методах класса **TREE**, в частности, в методах **PushStack** и **PopStack**. В методе **PushStack** была исправлена ошибка при объявлении переменной **q** типа **no**, а также была исправлена ошибка при передаче аргумента методу **PushStack** в методе **VyvodTree**. Теперь методы корректно работают с указателями на структуру **no**, что устраняет ошибки компиляции.

# **Тестирование №3**

|  |
| --- |
| #include <time.h> // Подключение заголовочного файла для работы с временем  #include <iostream> // Подключение заголовочного файла для ввода/вывода в консоль  using namespace std; // Использование стандартного пространства имен  #define N 10 // Определение константы N равной 10, количество элементов массива.  // Структура узла дерева  struct node  {  int Key; // Ключ узла  int Count; // Количество узлов с данным ключом  node\* Left; // Указатель на левое поддерево  node\* Right; // Указатель на правое поддерево  };  // Класс хэш-списка  class Spisok {  private:  node\* UkStr[N]; // Массив указателей на корни деревьев  void Search(int, node\*\*); // Метод поиска узла в дереве  void PrintTree(node\*, int); // Метод вывода дерева на экран  void U\_d(node\*\*, node\*\*); // Вспомогательный метод для удаления узла  public:  Spisok(); // Конструктор класса  void BuildTree(); // Метод построения хэш-списка  void Sodergimoe(); // Метод вывода содержимого хэш-списка  node\*\* GetTree(unsigned i) { return &(UkStr[i]); } // Метод получения указателя на дерево по индексу  void Udaldr(node\*\*, int); // Метод удаления узла из дерева  };  // Конструктор класса Spisok  Spisok::Spisok()  {  // Инициализация хэш-списка.  for (int i = 0; i < N; i++) UkStr[i] = nullptr;  }  // Метод построения хэш-списка  void Spisok::BuildTree()  {  int klutch; // Переменная для хранения ключа  unsigned hash; // Хэш-значение ключа  srand(time(0)); // Инициализация генератора случайных чисел  cout << "\nВведите значение ключа..."; // Просьба ввести значение ключа  klutch = rand() % 31 + 0; // Генерация случайного ключа в диапазоне от 0 до 31 включительно  cout << klutch; // Вывод сгенерированного ключа  while (klutch != 0) // Пока вводимый ключ не равен 0  {  hash = klutch % 10; // Вычисление значения хэш-функции.  Search(klutch, &UkStr[hash]); // Вызов метода поиска узла в дереве  cout << "\nВведите значение ключа..."; // Просьба ввести значение ключа  klutch = rand() % 31 + 0; // Генерация нового случайного ключа  cout << klutch; // Вывод сгенерированного ключа  }  } |

|  |  |
| --- | --- |
| // Метод поиска узла в дереве  void Spisok::Search(int X, node\*\* p)  {  if (\*p == nullptr)  { // Узла нет в дереве; включить его.  \*p = new node;  (\*\*p).Key = X;  (\*\*p).Count = 1;  (\*\*p).Left = (\*\*p).Right = nullptr;  }  else if (X < (\*\*p).Key) // Если значение ключа меньше ключа текущего узла, идем влево  Search(X, &((\*\*p).Left));  else if (X > (\*\*p).Key) // Если значение ключа больше ключа текущего узла, идем вправо  Search(X, &((\*\*p).Right));  else // Если значение ключа равно ключу текущего узла  (\*\*p).Count += 1; // Увеличение счетчика, если узел уже существует  }  // Метод вывода содержимого хэш-списка  void Spisok::Sodergimoe()  {  for (int i = 0; i < N; i++)  {  cout << " " << i << "... "; // Вывод номера списка  if (UkStr[i] == nullptr) // Если дерево пусто  cout << "Дерево пусто...\n"; // Вывод сообщения о пустом дереве  else  {  cout << endl;  PrintTree(UkStr[i], 0); // Вывод содержимого дерева  }  cout << "------------------------------------------" << endl; // Разделитель  }  }  // Метод вывода дерева на экран  void Spisok::PrintTree(node\* w, int l)  {  if (w != nullptr)  {  PrintTree((\*w).Right, l + 1); // Рекурсивный вызов для правого поддерева  cout << " ";  for (int i = 1; i <= l; i++) cout << " "; // Отступы в зависимости от уровня  cout << (\*w).Key << endl; // Вывод ключа узла  PrintTree((\*w).Left, l + 1); // Рекурсивный вызов для левого поддерева  }  }  // Метод удаления узла из дерева  void Spisok::Udaldr(node\*\* d, int k)  { // Удаление узла с ключом k из дерева d.  node\*\* q; // Указатель на указатель на узел  if (\*d == nullptr) // Если дерево пусто  cout << "Узел с заданным ключом в дереве не найден...\n"; // Вывод сообщения о неудаче  else  if (k < (\*\*d).Key) // Если ключ для удаления меньше ключа текущего узла, идем влево  Udaldr(&((\*\*d).Left), k);  else if (k > (\*\*d).Key) // Если ключ для удаления больше ключа текущего узла, идем вправо | |
| Udaldr(&((\*\*d).Right), k);  else  {  q = d;  if ((\*\*q).Right == nullptr) // Если правого поддерева нет  \*d = (\*\*q).Left;  else if ((\*\*q).Left == nullptr) // Если левого поддерева нет  \*d = (\*\*q).Right;  else // Если есть оба поддерева  U\_d(&((\*\*q).Left), &(\*q));  }  }  // Вспомогательный метод для удаления узла  void Spisok::U\_d(node\*\* r, node\*\* q)  {  if ((\*\*r).Right == nullptr) // Если у правого поддерева нет узлов  {  (\*\*q).Key = (\*\*r).Key;  (\*\*q).Count = (\*\*r).Count;  \*q = \*r; \*r = (\*\*r).Left; delete\* q; // Удаление узла  }  else // Если у правого поддерева есть узлы  U\_d(&((\*\*r).Right), &(\*q)); // Рекурсивный вызов для поиска узла  }  // Главная функция  int main()  {  setlocale(LC\_ALL, "Rus"); // Установка локали для корректного отображения кириллицы  Spisok A; // Создание объекта класса Spisok  int klutch; // Переменная для хранения ключа  unsigned hash; // Хэш-значение ключа  A.BuildTree(); // Вызов метода построения хэш-списка  cout << "\n Содержимое хэш-списка...";  cout << "\n -----------------------------------\n";  A.Sodergimoe(); // Вызов метода вывода содержимого хэш-списка  // Удаление элемента из хэш-списка.  for (int i = 0; i < 4; i++) // Будем удалять всего 4 раза!  {  cout << "\nВведите значение удаляемого ключа..."; // Просьба ввести значение ключа для удаления  cin >> klutch; // Ввод ключа для удаления  hash = klutch % 10; // Вычисление хэш-значения ключа  A.Udaldr(A.GetTree(hash), klutch); // Вызов метода удаления узла  cout << " Содержимое хэш-списка...\n";  cout << " ----------------------------------\n";  A.Sodergimoe(); // Вызов метода вывода содержимого хэш-списка  }  cout << "\n";  system("PAUSE"); // Ожидание нажатия клавиши  return 0; // Возврат значения 0  } |

|  |
| --- |
|  |

В изначальном коде были исправлены ошибки: в методе **Udaldr** добавлена разыменовка указателя **d**, чтобы обратиться к узлу и добавлена корректная замена удаляемого узла при наличии двух поддеревьев, также добавлена проверка наличия левого поддерева при удалении узла в методе **U\_d**, исправлен порядок вывода узлов в методе **Spisok::PrintTree**.

# **Тестирование №4**

|  |  |  |
| --- | --- | --- |
| #include <conio.h> // Подключение заголовочного файла для использования \_getch()  #include <iostream> // Подключение заголовочного файла для использования стандартных потоков ввода/вывода  using namespace std; // Использование стандартного пространства имён std  // Структура узла дерева  struct node  {  int key; // Ключ узла  int count; // Счётчик для повторяющихся ключей  node\* Left; // Указатель на левого потомка  node\* Right; // Указатель на правого потомка  };  // Класс дерева  class Tree  { | | |
| private:  node\* root; // Указатель на корень дерева  // Прототипы приватных методов  void DisposeTree(node\*); // Рекурсивный метод для освобождения памяти, занятой деревом  void printTree(node\*, int); // Рекурсивный метод для вывода дерева на экран  void Delete(node\*\*, int); // Рекурсивный метод для удаления узла с заданным ключом из дерева  void del(node\*\*, node\*); // Вспомогательный метод для удаления узла с двумя потомками из дерева  public:  Tree() { root = NULL; }; // Конструктор по умолчанию  ~Tree(); // Деструктор  // Публичные методы класса Tree  void creat\_Tree(); // Метод для построения дерева  void look\_Tree(); // Метод для вывода содержимого дерева  void add\_Tree(); // Метод для добавления узла в дерево  void delete\_Tree(); // Метод для удаления узла из дерева  void search(int, node\*\*); // Метод для поиска узла с заданным ключом в дереве  node\* getTree() { return root; }; // Метод для получения указателя на корень дерева  };  // Структура звена кольца  struct zveno  {  int element; // Элемент звена  Tree ukTree; // Дерево, связанное с звеном  zveno\* sled; // Указатель на следующее звено в кольце  };  // Класс кольца  class ring  {  private:  zveno\* ukring; // Указатель на первое звено кольца  public:  ring() { ukring = NULL; }; // Конструктор по умолчанию  ~ring(); // Деструктор  // Публичные методы класса ring  void create(); // Метод для построения кольца  void look(); // Метод для вывода содержимого кольца  void add\_after(int, zveno\*); // Метод для добавления звена после указанного звена  void add\_befor(int, zveno\*); // Метод для добавления звена перед указанным звеном  void Delete(zveno\*); // Метод для удаления указанного звена из кольца  void delete\_next(zveno\*); // Метод для удаления следующего звена после указанного звена  int poisk(int, zveno\*\*); // Метод для поиска звена с заданным элементом в кольце  zveno\*\* getring() { return &ukring; }; // Метод для получения указателя на первое звено кольца  };  // Метод для построения кольца  void ring::create()  {  zveno\* ukzv; // Указатель на текущее звено |

|  |
| --- |
| int elem; // Переменная для ввода элементов кольца  cout << "\nПостроение кольца ..." << endl;  cout << "Вводите элементы кольца (ввод окончите 0): \n";  cout << "-->";  cin >> elem; // Ввод элемента  if (elem != 0)  {  ukzv = ukring = new (zveno); // Выделение памяти под первое звено  (\*ukzv).element = elem; // Присвоение элементу значения  (\*ukzv).ukTree.creat\_Tree(); // Создание дерева для звена  cout << "\n-->";  cin >> elem;  while (elem != 0)  {  (\*ukzv).sled = new (zveno); // Выделение памяти под следующее звено  ukzv = (\*ukzv).sled; // Переход к следующему звену  (\*ukzv).element = elem; // Присвоение элементу значения  (\*ukzv).ukTree.creat\_Tree(); // Создание дерева для звена  cout << "\n-->";  cin >> elem; // Ввод элемента  }  ukzv->sled = ukring; // Замыкание кольца  }  }  // Деструктор класса ring  ring::~ring()  {  zveno\* ukzv; // Указатель на текущее звено  ukzv = ukring; // Начинаем с первого звена  while (ukring != NULL)  if (ukzv->sled == ukring)  {  ukring = NULL; // Очистка указателя на первое звено  ukzv->ukTree.~Tree(); // Удаление дерева, связанного с звеном  delete ukzv; // Освобождение памяти, занятой звеном  }  else  {  while (ukzv->sled->sled != ukring) ukzv = (\*ukzv).sled;  (\*ukzv).sled->ukTree.~Tree(); // Удаление дерева, связанного с звеном  delete (\*ukzv).sled; // Освобождение памяти, занятой звеном  ukzv->sled = ukring; // Обновление связей между звеньями  ukzv = ukring; // Переход к следующему звену  }  }  // Метод для вывода содержимого кольца  void ring::look()  {  zveno\* ukzv; // Указатель на текущее звено  cout << "\nВывод содержимого кольца ...";  ukzv = ukring; // Начинаем с первого звена  do {  cout << "\n-->" << (\*ukzv).element << endl;  ukzv->ukTree.look\_Tree(); // Вывод содержимого дерева, связанного с звеном  ukzv = ukzv->sled; // Переход к следующему звену  \_getch(); // Ожидание нажатия клавиши  } while (ukzv != ukring); // Повторяем, пока не вернёмся к первому звену  cout << endl;  } |
| // Метод для добавления звена перед указанным звеном  void ring::add\_befor(int elem, zveno\* zv)  {  zveno\* ukzv; // Указатель на новое звено  Tree temp; // Временное дерево для обмена данными  ukzv = new (zveno); // Выделение памяти под новое звено  temp = ukzv->ukTree; // Сохранение дерева нового звена  ukzv->element = zv->element; // Копирование элемента из указанного звена  ukzv->ukTree = zv->ukTree; // Копирование дерева из указанного звена  ukzv->sled = zv->sled; // Копирование указателя на следующее звено из указанного звена  zv->element = elem; // Присвоение указанному звену нового элемента  zv->ukTree = temp; // Присвоение указанному звену временного дерева  zv->ukTree.creat\_Tree(); // Перестроение дерева указанного звена  zv->sled = ukzv; // Присвоение указанному звену указателя на новое звено  }  // Метод для добавления звена после указанного звена  void ring::add\_after(int elem, zveno\* zv)  {  zveno\* ukzv; // Указатель на новое звено  ukzv = new (zveno); // Выделение памяти под новое звено  ukzv->element = elem; // Присвоение новому звену указанного элемента  ukzv->ukTree.creat\_Tree(); // Создание дерева для нового звена  ukzv->sled = zv->sled; // Присвоение новому звену указателя на следующее звено  zv->sled = ukzv; // Присвоение указанному звену указателя на новое звено  }  // Метод для удаления указанного звена из кольца  void ring::Delete(zveno\* zv)  {  zveno\* ukzv1, \* ukzv2; // Указатели на звенья для перемещения по кольцу  zveno\* time; // Временный указатель для удаления звена  if (zv->sled != ukring) // Проверка, не последнее ли звено в кольце  {  time = zv->sled; // Сохранение указателя на удаляемое звено  zv->ukTree.~Tree(); // Удаление дерева, связанного с звеном  (\*zv) = \*((\*zv).sled); // Копирование данных из следующего звена в текущее  delete time; // Освобождение памяти, занятой удаляемым звеном  }  else  if (zv->sled == zv) // Проверка, последнее ли звено в кольце  {  zv->ukTree.~Tree(); // Удаление дерева, связанного с звеном  delete ukring; // Освобождение памяти, занятой последним звеном  ukring = NULL; // Обнуление указателя на первое звено (кольцо пусто)  cout << "Список пуст...\n"; // Вывод сообщения о пустом кольце  }  else  {  ukzv2 = ukring; // Начало с первого звена  ukzv1 = ukring->sled; // Начало со второго звена  while (ukzv1 != zv) // Поиск указанного звена в кольце  {  ukzv2 = ukzv1; ukzv1 = ukzv1->sled; // Переход к следующему звену  }  time = ukzv2->sled; // Сохранение указателя на удаляемое звено | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ukzv2->sled->ukTree.~Tree(); // Удаление дерева, связанного с удаляемым звеном  ukzv2->sled = ukzv2->sled->sled; // Удаление удаляемого звена из кольца  delete time; // Освобождение памяти, занятой удаляемым звеном  }  }  // Метод для удаления следующего звена после указанного звена  void ring::delete\_next(zveno\* zv)  {  zveno\* time; // Временный указатель для удаления звена  if (zv->sled != ukring) // Проверка, не последнее ли звено в кольце  {  time = zv->sled; // Сохранение указателя на удаляемое звено  zv->sled = zv->sled->sled; // Удаление следующего звена  time->ukTree.~Tree(); // Удаление дерева, связанного с удаляемым звеном  delete time; // Освобождение памяти, занятой удаляемым звеном  }  else  if (zv->sled == zv) // Проверка, последнее ли звено в кольце  cout << "В кольце только один элемент!\n"; // Вывод сообщения об одном элементе в кольце  else  {  time = ukring->sled; // Сохранение указателя на удаляемое звено  \*((\*zv).sled) = (\*(\*(\*zv).sled).sled); // Удаление следующего звена  time->ukTree.~Tree(); // Удаление дерева, связанного с удаляемым звеном  delete time; // Освобождение памяти, занятой удаляемым звеном  }  }  // Метод для поиска звена с заданным элементом в кольце  int ring::poisk(int elem, zveno\*\* Res)  {  zveno\* ukzv; // Указатель на текущее звено  int vozvr = 0; // Переменная для возврата значения  if (\*(getring()) == NULL) cout << "Кольцо не существует...\n"; // Проверка, существует ли кольцо  else  {  ukzv = ukring; // Начинаем с первого звена  while (ukzv->sled != ukring && (\*Res) == NULL) // Поиск элемента в кольце  {  if (ukzv->element == elem) // Если элемент найден  {  vozvr = 1; \*Res = ukzv; // Устанавливаем флаг и сохраняем указатель на звено  }  ukzv = ukzv->sled; // Переход к следующему звену  }  if ((\*Res) == NULL) // Если элемент не найден до конца кольца  if (ukzv->element == elem) // Проверка последнего звена на соответствие элементу  {  vozvr = 1; \*Res = ukzv; // Устанавливаем флаг и сохраняем указатель на звено  }  }  return vozvr; // Возвращаем результат поиска  }  // Деструктор класса Tree  Tree::~Tree()  { | | | | |
| DisposeTree(root); // Вызов рекурсивной функции удаления дерева  root = NULL; // Обнуление указателя на корень (дерево удалено)  }  // Рекурсивная функция для удаления дерева  void Tree::DisposeTree(node\* p)  {  if (p != NULL) // Проверка, существует ли текущий узел  {  DisposeTree(p->Left); // Рекурсивный вызов для левого поддерева  DisposeTree(p->Right); // Рекурсивный вызов для правого поддерева  delete p; // Освобождение памяти, занятой текущим узлом  }  }  // Метод для добавления узла в дерево  void Tree::search(int x, node\*\* p)  {  if (\*p == NULL) // Проверка, пусто ли дерево  {  \*p = new (node); // Выделение памяти под новый узел  (\*\*p).key = x; (\*\*p).count = 1; // Присвоение ключа и установка счетчика  (\*\*p).Left = (\*\*p).Right = NULL; // Обнуление указателей на левое и правое поддеревья  }  else if (x < (\*\*p).key) search(x, &((\*\*p).Left)); // Рекурсивный вызов для левого поддерева  else  if (x > (\*\*p).key) search(x, &((\*\*p).Right)); // Рекурсивный вызов для правого поддерева  else (\*\*p).count += 1; // Увеличение счетчика, если ключ уже существует  }  // Метод для создания дерева  void Tree::creat\_Tree()  {  int elem; // Переменная для ввода ключей узлов  cout << "Вводите ключи узлов дерева (ввод окончите 0):\n";  cin >> elem; // Ввод первого ключа  while (elem != 0) // Цикл для ввода всех ключей  {  search(elem, &root); // Добавление ключа в дерево  cin >> elem; // Ввод следующего ключа  }  }  // Метод для просмотра дерева  void Tree::look\_Tree()  {  if (root == NULL) cout << "Дерево пусто ...\n"; // Проверка, пусто ли дерево  else printTree(root, 0); // Вызов рекурсивной функции для вывода дерева  }  // Рекурсивная функция для вывода дерева на экран  void Tree::printTree(node\* w, int L)  {  if (w != NULL) // Проверка, существует ли текущий узел  {  printTree(w->Left, L + 1); // Рекурсивный вызов для левого поддерева  for (int i = 1; i <= L; i++) cout << " "; // Отступ для текущего уровня  cout << w->key << endl; // Вывод ключа текущего узла  printTree(w->Right, L + 1); // Рекурсивный вызов для правого поддерева  } | | |
| }  // Метод для добавления узлов в дерево  void Tree::add\_Tree()  {  int k; // Переменная для ввода ключей добавляемых узлов  cout << "\nВводите ключи добавляемых узлов (ввод окончите 0):\n";  cin >> k; // Ввод первого ключа  cout << " ";  while (k != 0) // Цикл для ввода всех ключей  {  search(k, &(root)); // Добавление ключа в дерево  cin >> k; // Ввод следующего ключа  cout << " ";  }  }  // Метод для удаления узла из дерева  void Tree::delete\_Tree()  {  int elem; // Переменная для ввода ключа удаляемого узла  if (root == NULL) cout << "Дерево пусто ...\n"; // Проверка, пусто ли дерево  else  {  cout << "Введите ключ удаляемого узла : ";  cin >> elem; // Ввод ключа удаляемого узла  cout << endl;  Delete(&root, elem); // Вызов функции удаления узла  }  }  // Рекурсивная функция для удаления узла из дерева  void Tree::Delete(node\*\* d, int k)  {  node\* q; // Вспомогательный указатель для сохранения удаляемого узла  node\* s; // Вспомогательный указатель для сохранения удаляемого узла  if (\*d == NULL) cout << "Узел с заданным ключом в дереве не найден ...\n"; // Проверка, существует ли узел с заданным ключом  else  if (k < (\*\*d).key) Delete(&((\*\*d).Left), k); // Рекурсивный вызов для левого поддерева  else  if (k > (\*\*d).key) Delete(&((\*\*d).Right), k); // Рекурсивный вызов для правого поддерева  else  {  q = \*d; s = \*d; // Сохранение указателей на удаляемый узел  if ((\*q).Right == NULL) // Проверка, существует ли правое поддерево  {  \*d = (\*q).Left; // Перемещение левого поддерева на место удаляемого узла  delete s; // Освобождение памяти, занятой удаляемым узлом  }  else  if ((\*q).Left == NULL) // Проверка, существует ли левое поддерево  {  \*d = (\*q).Right; // Перемещение правого поддерева на место удаляемого узла  delete s; // Освобождение памяти, занятой удаляемым узлом |

|  |  |  |
| --- | --- | --- |
| }  else del(&((\*q).Left), &(\*q)); // Вызов функции для удаления узла с двумя потомками  }  }  // Вспомогательная рекурсивная функция для удаления узла с двумя потомками  void Tree::del(node\*\* r, node\* q)  {  node\* s; // Вспомогательный указатель для сохранения удаляемого узла  if ((\*\*r).Right == NULL) // Проверка, существует ли правое поддерево у узла r  {  (\*q).key = (\*\*r).key; (\*q).count = (\*\*r).count; // Копирование ключа и счетчика  q = s = \*r; \*r = (\*\*r).Left; // Перемещение левого поддерева на место удаляемого узла  delete s; // Освобождение памяти, занятой удаляемым узлом  }  else del(&((\*\*r).Right), &(\*q)); // Рекурсивный вызов для правого поддерева  }  // Главная функция программы  int main()  {  setlocale(LC\_ALL, "Rus"); // Установка локали для корректного отображения текста на русском языке  int menu1 = 1, choice, elem1, elem2, menu2; // Переменные для управления меню и ввода данных  ring A; // Создание объекта класса ring  cout << "<------------- Структура --------------->\n";  cout << "<---------\"кольцо с деревьями\"---------->\n\n";  while (menu1) // Главный цикл программы  {  cout << endl;  cout << "<---------- Главное меню 1.0 : --------->\n";  cout << "1. Построение структуры.................. \n";  cout << "2. Просмотр структуры.................... \n";  cout << "3. Добавление элемента после указанного.. \n";  cout << "4. Добавление элемента перед указанным... \n";  cout << "5. Удаление элемента..................... \n";  cout << "6. Удаление элемента после указанного.... \n";  cout << "7. Преобразование дерева заданного эл-та. \n";  cout << "8. Удаление структуры.................... \n";  cout << "9. Выход................................. \n";  cout << "Введите номер режима и нажмите <Enter> : ";  cin >> choice; cout << endl; // Ввод выбранного режима работы  switch (choice) // Обработка выбранного режима  {  case 1:  if (\*(A.getring()) == NULL) A.create(); // Построение структуры, если кольцо пусто  else cout << "Кольцо уже существует...\n"; // Сообщение, если кольцо уже существует  break;  case 2:  if (\*(A.getring()) == NULL) cout << "Кольцо пусто...\n"; // Проверка, пусто ли кольцо  else A.look(); // Вывод содержимого кольца  break;  case 3:  if (\*(A.getring()) == NULL) cout << "Кольцо пусто...\n"; // Проверка, пусто ли кольцо | | |
| else // Добавление элемента после указанного  {  zveno\* Res = NULL; // Указатель на результат поиска элемента  cout << "Введите элемент, после которого ";  cout << " хотите добавить звено: ";  cin >> elem1; cout << endl;  if (A.poisk(elem1, &Res)) // Поиск элемента в кольце  {  cout << "Введите элемент, который ";  cout << "хотите добавить: ";  cin >> elem2;  cout << endl;  A.add\_after(elem2, Res); // Добавление элемента после указанного  }  else  cout << "Элемент " << elem1 << " не найден.\n"; // Сообщение об ошибке, если элемент не найден  }  break;  case 4:  if (\*(A.getring()) == NULL) cout << "Кольцо пусто...\n"; // Проверка, пусто ли кольцо  else // Добавление элемента перед указанным  {  zveno\* Res = NULL; // Указатель на результат поиска элемента  cout << "Введите элемент, перед которым ";  cout << " хотите добавить звено: ";  cin >> elem1; cout << endl;  if (A.poisk(elem1, &Res)) // Поиск элемента в кольце  {  cout << "Введите элемент, который ";  cout << "хотите добавить: ";  cin >> elem2;  cout << endl;  A.add\_befor(elem2, Res); // Добавление элемента перед указанным  }  else  cout << "Элемент " << elem1 << " не найден.\n"; // Сообщение об ошибке, если элемент не найден  }  break;  case 5:  if (\*(A.getring()) == NULL) cout << "Кольцо пусто...\n"; // Проверка, пусто ли кольцо  else // Удаление элемента  {  zveno\* Res = NULL; // Указатель на результат поиска элемента  cout << "Введите элемент, который";  cout << " хотите удалить: ";  cin >> elem1; cout << endl;  if (A.poisk(elem1, &Res)) A.Delete(Res); // Поиск элемента в кольце и удаление  else cout << "Элемент отсутствует...\n"; // Сообщение об ошибке, если элемент не найден  }  break;  case 6:  if (\*(A.getring()) == NULL) cout << "Кольцо пусто...\n"; // Проверка, пусто ли кольцо  else // Удаление элемента после указанного  {  zveno\* Res = NULL; // Указатель на результат поиска элемента  cout << "Введите элемент, после которого";  cout << " хотите удалить: "; | | | |
| cin >> elem1; cout << endl;  if (A.poisk(elem1, &Res)) A.delete\_next(Res); // Поиск элемента в кольце и удаление  else cout << "Элемент отсутствует...\n"; // Сообщение об ошибке, если элемент не найден  }  break;  case 7:  if (\*(A.getring()) == NULL) cout << "Кольцо пусто...\n"; // Проверка, пусто ли кольцо  else // Преобразование дерева заданного элемента  {  zveno\* Res = NULL; // Указатель на результат поиска элемента  cout << "Введите элемент кольца: ";  cin >> elem1; cout << endl;  if (A.poisk(elem1, &Res)) // Поиск элемента в кольце  {  menu2 = 1; // Переменная для управления подменю  while (menu2) // Цикл работы подменю  {  cout << endl;  cout << "<---------- Mеню 1.1 : --------->\n";  cout << "1. Построение дерева.............\n";  cout << "2. Просмотр дерева...............\n";  cout << "3. Добавление элемента в дерево..\n";  cout << "4. Удаление элемента из дерева...\n";  cout << "5. Удаление дерева...............\n";  cout << "6. Выход в главное меню..........\n";  cout << "Введите номер режима и нажмите <Enter>: ";  cin >> choice; cout << endl; // Ввод выбранного режима работы в подменю  switch (choice) // Обработка выбранного режима в подменю  {  case 1:  if ((\*Res).ukTree.getTree() == NULL) // Проверка, существует ли дерево  (\*Res).ukTree.creat\_Tree(); // Создание дерева, если оно не существует  else cout << "Дерево существует...\n"; // Сообщение, если дерево уже существует  break;  case 2:  (\*Res).ukTree.look\_Tree(); // Вывод содержимого дерева  break;  case 3:  (\*Res).ukTree.add\_Tree(); // Добавление элемента в дерево  break;  case 4:  (\*Res).ukTree.delete\_Tree(); // Удаление элемента из дерева  break;  case 5:  if ((\*Res).ukTree.getTree() == NULL) // Проверка, существует ли дерево  cout << "Дерево не существует...\n"; // Сообщение, если дерево не существует  else (\*Res).ukTree.~Tree(); // Удаление дерева  break;  case 6:  menu2 = 0; // Выход в главное меню  break;  } | |

|  |
| --- |
| }  }  else cout << "Элемент " << elem1 << " не найден.\n"; // Сообщение об ошибке, если элемент не найден  }  break;  case 8:  if (\*(A.getring()) == NULL) cout << "Кольцо пусто...\n"; // Проверка, пусто ли кольцо  else A.~ring(); // Удаление кольца  break;  case 9:  A.~ring(); // Удаление кольца  menu1 = 0; // Выход из главного цикла программы  break;  } //End Case  } //End while  cout << "\n";  system("PAUSE"); // Пауза перед завершением программы  } |

|  |
| --- |
|  |

Я заменил функцию **getch()** на **\_getch()** в этой программе, чтобы использовать безопасную и переносимую альтернативу. Функция **\_getch()** обеспечивает аналогичную функциональность, но является более предпочтительным выбором, так как **getch()** может быть неопределенным и зависеть от конкретной реализации или среды выполнения. Использование **\_getch()** может обеспечить более надежный код.

# **Тестирование №5**

|  |  |
| --- | --- |
| #include <iostream> // Включаем заголовочный файл для ввода-вывода через стандартные потоки.  #include <cstring> // Включаем заголовочный файл для работы с функциями строки из стандартной библиотеки.  using namespace std; // Используем пространство имен std.  // Определение структуры для элементов списка и дерева Хаффмана.  struct zveno  {  char Element; // Символ.  float Kol; // Количество повторений, частота повторений.  zveno\* Sled; // Указатель на следующий элемент списка.  zveno\* Left; // Указатель на левого потомка в дереве.  zveno\* Right; // Указатель на правого потомка в дереве.  zveno\* Father; // Указатель на родителя в дереве.  };  // Определение класса для работы с деревом Хаффмана.  class Tree  {  private:  zveno\* UkStr; // Указатель на список.  int Poisk1(zveno\*\*, float, zveno\*\*); // Прототип закрытой функции для поиска места в упорядоченном списке.  public:  Tree() { UkStr = new zveno; UkStr->Sled = nullptr; }; // Конструктор класса. Создает новый экземпляр списка.  ~Tree() { delete UkStr; } // Деструктор класса. Освобождает выделенную память для списка.  int Poisk(char, zveno\*\*); // Поиск элемента в списке.  int Kolich(char\*, char); // Подсчет количества повторений символа в тексте.  void Dobavlenie(char, float, zveno\*\*); // Добавление элемента в список.  void Redaktor(int); // Редактирование списка.  void Ukazateli(zveno\*\*, zveno\*\*); // Поиск указателей на предпоследний и предпредпоследний элементы.  void Vyvod(); // Вывод списка на экран.  void WstawkaSort(zveno\*); // Сортировка списка вставкой.  void PrintTree(zveno\*, int); // Вывод дерева на экран.  zveno\*\* GetTree() { return &UkStr; }; // Получение указателя на список.  zveno\* GetTree1() { return UkStr; }; // Получение списка.  };  // Реализация функции для поиска элемента в списке по символу.  int Tree::Poisk(char ENT, zveno\*\* Res)  {  zveno\* q; // Объявление указателя на элемент списка.  int vozvr = 0; // Инициализация переменной для возврата результата.  \*Res = nullptr; // Устанавливаем указатель на nullptr.  q = (\*UkStr).Sled; // Начинаем поиск с первого элемента списка.  while (q != nullptr && \*Res == nullptr)  {  if (q->Element == ENT)  {  vozvr = 1; // Элемент найден.  \*Res = q; // Устанавливаем указатель на найденный элемент.  return vozvr;  }  q = q->Sled; // Переходим к следующему элементу списка.  } | |
| return vozvr; // Возвращаем результат поиска.  }  // Реализация функции для поиска места в упорядоченном списке для добавления элемента.  int Tree::Poisk1(zveno\*\* st, float kol, zveno\*\* Res)  {  zveno\* q = (\*\*st).Sled, \* q1 = (\*st); // Объявление указателей на текущий и предыдущий элементы списка.  int vozvr = 0; // Инициализация переменной для возврата результата.  \*Res = nullptr; // Устанавливаем указатель на nullptr.  while (q != nullptr && \*Res == nullptr)  {  if (q->Kol < kol) { vozvr = 1; \*Res = q; } // Найдено место для вставки элемента.  q = q->Sled; q1 = q1->Sled; // Переходим к следующему элементу списка.  }  if (\*Res == nullptr) \*Res = q1; // Устанавливаем указатель на последний элемент списка.  return vozvr; // Возвращаем результат поиска.  }  // Реализация функции для подсчета количества повторений символа в тексте.  int Tree::Kolich(char\* F, char S)  {  int K = 0; // Инициализация счетчика повторений.  for (int i = 0; i < strlen(F); i++)  if (F[i] == S) K++; // Увеличиваем счетчик, если символ встречается в тексте.  return K; // Возвращаем количество повторений.  }  // Реализация функции для редактирования списка.  void Tree::Redaktor(int L)  {  zveno\* q = (\*UkStr).Sled; // Указатель на первый элемент списка.  while (q != nullptr)  {  q->Kol = q->Kol / L; // Вычисляем частоту повторений для каждого элемента списка.  q = q->Sled; // Переходим к следующему элементу списка.  }  }  // Реализация функции для добавления элемента в список.  void Tree::Dobavlenie(char bukva, float kol, zveno\*\* Sp)  {  zveno\* q, \* Res = nullptr, \* kladovaq; // Объявление указателей на новый элемент, найденный элемент и временный элемент.  q = new zveno; // Выделяем память под новый элемент списка.  q->Element = bukva; // Присваиваем символ новому элементу.  q->Kol = kol; // Присваиваем количество повторений новому элементу.  q->Left = q->Right = nullptr; // Устанавливаем указатели на потомков в nullptr.  q->Sled = q->Father = nullptr; // Устанавливаем указатели на следующий элемент и родителя в nullptr.  if ((\*\*Sp).Sled == nullptr) (\*\*Sp).Sled = q; // Если список пуст, устанавливаем новый элемент как первый.  else  if (Poisk1(&(\*Sp), kol, &Res)) | |

|  |  |  |
| --- | --- | --- |
| {  kladovaq = new zveno; (\*kladovaq) = (\*Res); // Создаем копию найденного элемента.  (\*Res) = (\*q); Res->Sled = kladovaq; // Вставляем новый элемент в список.  }  else Res->Sled = q; // Вставляем новый элемент в конец списка.  }  // Реализация функции для поиска указателей на предпоследний и предпредпоследний элементы.  void Tree::Ukazateli(zveno\*\* zv, zveno\*\* zv\_p)  {  \*zv\_p = UkStr->Sled; \*zv = UkStr; // Установка указателей на первый элемент списка.  while ((\*zv\_p)->Sled->Sled != nullptr)  {  \*zv = \*zv\_p; \*zv\_p = (\*zv\_p)->Sled; // Переход к следующему элементу списка.  }  }  // Реализация функции для вывода списка на экран.  void Tree::Vyvod()  {  zveno\* q = UkStr->Sled; // Указатель на первый элемент списка.  while (q != nullptr)  {  cout << q->Element << " (" << q->Kol << ") --> "; // Вывод символа и его частоты повторений.  q = q->Sled; // Переходим к следующему элементу списка.  }  cout << endl; // Переход на новую строку.  }  // Реализация функции для сортировки списка вставкой.  void Tree::WstawkaSort(zveno\* zv)  {  zveno\* w1, \* w2; // Объявление указателей на текущий и предыдущий элементы списка.  w2 = UkStr; w1 = w2->Sled; // Установка указателей на первый и второй элементы списка.  while (w1 != nullptr && w1->Kol > zv->Kol)  {  w2 = w1; w1 = w2->Sled; // Переход к следующему элементу списка.  }  if (w1 == nullptr || w1->Kol <= zv->Kol)  {  w2->Sled = zv; zv->Sled = w1; // Вставка элемента в список.  }  }  // Реализация функции для вывода дерева на экран.  void Tree::PrintTree(zveno\* w, int l)  {  if (w != nullptr)  {  PrintTree(w->Right, l + 1); // Рекурсивный вызов для правого потомка.  for (int i = 1; i <= l; i++) cout << " "; // Отступ для наглядного отображения уровня вложенности.  cout << w->Element << " (" << w->Kol << ")\n"; // Вывод символа и его частоты повторений.  PrintTree(w->Left, l + 1); // Рекурсивный вызов для левого потомка.  } | | |
| }  // Главная функция программы.  int main()  {  setlocale(LC\_ALL, "Rus"); // Установка русской локали для корректного отображения текста.  Tree A; // Создание объекта класса Tree.  char T[255]; // Исходная строка.  int i, j; // Переменные для циклов.  zveno\* Res = nullptr; // Указатель на звено.  zveno\* Q[256]; // Массив указателей на элементы списка.  cout << "Введите текст, содержащий не менее двух символов...\n"; // Приглашение к вводу текста.  fgets(T, sizeof(T), stdin); // Считывание строки с клавиатуры.  // Формирование списка, содержащего символы текста.  for (i = 0; i < strlen(T); i++)  {  if (!A.Poisk(T[i], &Res)) // Если символ еще не встречался в списке.  A.Dobavlenie(T[i], A.Kolich(T, T[i]), A.GetTree()); // Добавляем его в список.  }  // ------------------------------- //  A.Redaktor(strlen(T)); // Редактирование списка.  cout << "Полученный список:\n"; // Вывод полученного списка.  A.Vyvod();  // Заполнение массива Q указателями на элементы списка.  zveno\* UkZv = A.GetTree1()->Sled, \* UkZv\_p = nullptr, \* Sli; // Указатели для формирования массива Q.  i = 0; // Инициализация счетчика.  while (UkZv != nullptr) // Пока указатель не достиг конца списка.  {  Q[i] = UkZv; // Присваиваем указатель на текущее звено массиву Q.  i++; // Переходим к следующему элементу массива Q.  UkZv = UkZv->Sled; // Переходим к следующему звену списка.  }  // Построение дерева Хаффмана.  while (A.GetTree1()->Sled->Sled != nullptr) // Пока в списке не останется одно звено.  {  A.Ukazateli(&UkZv, &UkZv\_p); // Находим указатели на последний и предпоследний элементы списка.  // Слияние последнего и предпоследнего звена.  Sli = new zveno; // Выделяем память под новое звено.  Sli->Element = '\*'; // Устанавливаем символ звена как '\*'.  Sli->Kol = UkZv\_p->Kol + UkZv\_p->Sled->Kol; // Вычисляем суммарную частоту повторений звена.  Sli->Left = UkZv\_p; // Устанавливаем левого потомка.  Sli->Right = UkZv\_p->Sled; // Устанавливаем правого потомка.  Sli->Father = Sli->Sled = nullptr; // Устанавливаем указатели на родителя и следующий элемент в nullptr.  UkZv\_p->Father = Sli; // Устанавливаем указатель на родителя для предпоследнего звена.  UkZv\_p->Sled->Father = Sli; // Устанавливаем указатель на родителя для последнего звена.  // Уничтожаем ссылки на последний и предпоследний элементы списка.  UkZv->Sled = nullptr; // Устанавливаем указатель на следующий элемент последнего звена в nullptr. |

|  |  |
| --- | --- |
| UkZv\_p->Sled = nullptr; // Устанавливаем указатель на следующий элемент предпоследнего звена в nullptr.  // Помещаем звено, заданное указателем Sli, в список.  if (A.GetTree1()->Sled == nullptr) A.GetTree1()->Sled = Sli; // Если список пуст, звено становится первым.  else A.WstawkaSort(Sli); // Иначе добавляем звено в упорядоченный список.  }  cout << "Построим дерево...\n"; // Вывод сообщения о построении дерева.  A.PrintTree(A.GetTree1()->Sled, 0); // Вывод дерева Хаффмана.  cout << "--------------------------------------------- " << endl; // Разделительная линия.  // Кодирование заданного текста.  cout << "Приступим к кодировке введенного текста...\n"; // Вывод сообщения о начале кодирования.  char Cod\_symbol[40]; // Буфер для хранения кода символа.  char Cod\_Haffmen[255]; // Код Хаффмана строки T.  char temp[255]; // Временный буфер для операций с кодом символа.  strcpy\_s(Cod\_symbol, sizeof(Cod\_symbol), ""); // Инициализация пустой строки.  strcpy\_s(Cod\_Haffmen, sizeof(Cod\_Haffmen), ""); // Инициализация пустой строки.  for (i = 0; i < strlen(T); i++) // Перебираем все символы в строке.  {  // Начнем поиск нужного указателя.  j = 0; // Инициализация счетчика.  while (Q[j]->Element != T[i]) j++; // Пока не найден нужный символ, переходим к следующему звену.  // А теперь начинаем "восхождение"...  UkZv = Q[j]; // Устанавливаем указатель на нужное звено.  while (UkZv->Father != nullptr) // Пока не достигнут корень дерева.  if (UkZv->Father->Left == UkZv) // Если текущее звено - левый потомок.  {  strcpy\_s(temp, sizeof(temp), "1"); // Добавляем к коду '1'.  strcat\_s(temp, sizeof(temp), Cod\_symbol); // Конкатенируем временный буфер с текущим кодом.  strcpy\_s(Cod\_symbol, sizeof(Cod\_symbol), temp); // Копируем получившийся код в основной буфер.  UkZv = UkZv->Father; // Переходим к родителю.  }  else // Если текущее звено - правый потомок.  {  strcpy\_s(temp, sizeof(temp), "0"); // Добавляем к коду '0'.  strcat\_s(temp, sizeof(temp), Cod\_symbol); // Конкатенируем временный буфер с текущим кодом.  strcpy\_s(Cod\_symbol, sizeof(Cod\_symbol), temp); // Копируем получившийся код в основной буфер.  UkZv = UkZv->Father; // Переходим к родителю.  }  strcat\_s(Cod\_Haffmen, sizeof(Cod\_Haffmen), Cod\_symbol); // Добавляем код символа к общему коду.  strcpy\_s(Cod\_symbol, sizeof(Cod\_symbol), ""); // Очищаем буфер кода символа.  }  cout << "Код перед Вами... " << Cod\_Haffmen << endl; // Вывод закодированного текста.  cout << "Коэффициент сжатия: " <<  100 \* strlen(Cod\_Haffmen) / 8.0 / strlen(T) << "%\n"; // Вывод коэффициента сжатия.  // Расшифровка закодированного сообщения.  cout << "Ранее было зашифровано... " << T; // Вывод оригинального текста.  strcpy\_s(T, sizeof(T), ""); // Очистка строки.  // Установим указатель на корень дерева. | |
| UkZv = A.GetTree1()->Sled;  i = 0; // Инициализация счетчика.  while (i < strlen(Cod\_Haffmen)) // Пока не расшифрован весь текст.  {  while (UkZv->Left != nullptr && UkZv->Right != nullptr) // Пока не достигнут лист дерева.  {  if (Cod\_Haffmen[i] == '1') UkZv = UkZv->Left; // Если код - '1', переходим к левому потомку.  else UkZv = UkZv->Right; // Если код - '0', переходим к правому потомку.  i++; // Переходим к следующему биту кода.  }  char s[2]; // Буфер для символа.  s[0] = UkZv->Element; s[1] = '\0'; // Записываем символ в буфер.  strcat\_s(T, sizeof(T), s); // Добавляем символ к расшифрованной строке.  UkZv = A.GetTree1()->Sled; // Устанавливаем указатель на корень дерева.  }  cout << "Расшифровано... " << T << endl; // Вывод расшифрованного текста.  cout << "\n"; // Переход на новую строку для читаемости.  system("PAUSE"); // Пауза перед завершением программы.  return 0; // Возвращаем ноль, чтобы сообщить об успешном завершении программы.  } | |

|  |
| --- |
|  |

Я добавил библиотеку **cstring**, чтобы использовать функции работы со строками, такие как **strlen**, **strcpy\_s** и **strcat\_s**.

Исправления: заменены вызовы функции **strcpy()** на безопасную функцию **strcpy\_s()** для предотвращения переполнения буфера; добавлен деструктор для класса **Tree** для освобождения памяти, выделенной для списка **UkStr**; заменен вызов устаревшей функции **gets()** на безопасную функцию **fgets()** для обеспечения безопасности при чтении ввода.

# **Тестирование №6**

|  |  |  |  |
| --- | --- | --- | --- |
| #include <iostream>  #include <algorithm> // Для функции reverse  using namespace std; // Использование пространства имен std  // Объявление структуры для узла дерева  struct Uzel {  char Key; // Символ (ключ) в узле  Uzel\* Left; // Указатель на левое поддерево  Uzel\* Right; // Указатель на правое поддерево  };  // Объявление структуры для элемента стека  struct zveno {  Uzel\* Element; // Указатель на узел (элемент стека)  zveno\* Sled; // Указатель на следующий элемент стека  };  // Объявление класса для бинарного дерева  class Tree {  private:  Uzel\* Root; // Указатель на корень дерева  zveno\* Stack; // Указатель на вершину стека  public:  Tree(); // Конструктор класса  ~Tree(); // Деструктор класса  void Udalenie(Uzel\*\*); // Функция удаления элемента из стека  void V\_stack(Uzel\*); // Функция добавления элемента в стек  void PrintTree(Uzel\*, int); // Функция вывода дерева на экран  void Print\_Tree\_Left(Uzel\*, int); // Функция прямого обхода бинарного дерева  void Print\_Tree\_End(Uzel\*, int); // Функция обратного обхода бинарного дерева  void Print\_Tree\_Back(Uzel\*, int); // Функция симметричного обхода бинарного дерева  Uzel\* GetTree() { return Root; }; // Функция получения указателя на корень дерева  };  // Реализация методов класса Tree  // Конструктор класса  Tree::Tree() {  Stack = NULL; // Инициализация стека как пустого | | | |
| Root = new Uzel; // Выделение памяти под корневой узел дерева  Root->Right = NULL; // Инициализация правого поддерева как пустого  }  // Деструктор класса (необходим для освобождения памяти)  Tree::~Tree() {  // Деструктор не требует дополнительной реализации,  // так как память для узлов будет освобождена автоматически  }  // Функция добавления элемента в стек  void Tree::V\_stack(Uzel\* Elem) {  zveno\* q = new zveno; // Создание нового звена стека  q->Element = Elem; // Присваивание указателю на узел значение элемента  q->Sled = Stack; // Указание на следующий элемент стека  Stack = q; // Обновление вершины стека  }  // Функция удаления элемента из стека  void Tree::Udalenie(Uzel\*\* tmp) {  zveno\* q;  if (Stack != NULL) { // Если стек не пуст  (\*tmp) = Stack->Element; // Сохранение значения элемента  q = Stack; // Сохранение вершины стека  Stack = Stack->Sled; // Обновление вершины стека  delete q; // Освобождение памяти, выделенной для предыдущей вершины  }  }  // Функция вывода дерева на экран  void Tree::PrintTree(Uzel\* w, int l) {  if (w != NULL) { // Если узел не пуст  PrintTree(w->Right, l + 1); // Рекурсивный вызов для правого поддерева  for (int i = 1; i <= l; i++) // Вывод отступа для текущего уровня  cout << " ";  cout << w->Key << endl; // Вывод значения узла  PrintTree(w->Left, l + 1); // Рекурсивный вызов для левого поддерева  }  }  // Функция прямого обхода бинарного дерева  void Tree::Print\_Tree\_Left(Uzel\* w, int l) {  if (w != NULL) { // Если узел не пуст  cout << w->Key << " "; // Вывод значения узла  Print\_Tree\_Left(w->Left, l + 1); // Рекурсивный вызов для левого поддерева  Print\_Tree\_Left(w->Right, l + 1); // Рекурсивный вызов для правого поддерева  }  }  // Функция обратного обхода бинарного дерева  void Tree::Print\_Tree\_End(Uzel\* w, int l) {  if (w != NULL) { // Если узел не пуст  Print\_Tree\_End(w->Left, l + 1); // Рекурсивный вызов для левого поддерева  Print\_Tree\_End(w->Right, l + 1); // Рекурсивный вызов для правого поддерева  cout << w->Key << " "; // Вывод значения узла  }  }  // Функция симметричного обхода бинарного дерева  void Tree::Print\_Tree\_Back(Uzel\* w, int l) {  if (w != NULL) { // Если узел не пуст  cout << "("; // Вывод открывающей скобки перед левым поддеревом  Print\_Tree\_Back(w->Left, l + 1); // Рекурсивный вызов для левого поддерева  cout << w->Key << " "; // Вывод значения узла | | | |
| Print\_Tree\_Back(w->Right, l + 1); // Рекурсивный вызов для правого поддерева  cout << ")"; // Вывод закрывающей скобки после правого поддерева  }  }  // Основная функция программы  int main() {  setlocale(LC\_ALL, "Rus"); // Установка локали для корректного вывода на кириллице  char Formula\_Post[30]; // Объявление массива для хранения постфиксной формы  char k; // Объявление вспомогательной переменной  Uzel\* Ukazatel = NULL; // Объявление указателя  cout << "Введите формулу, записанную в постфиксной форме... \n"; // Вывод приглашения к вводу  cin.getline(Formula\_Post, 30); // Получение постфиксной формы  reverse(Formula\_Post, Formula\_Post + strlen(Formula\_Post)); // Реверс постфиксной формы для правильного построения дерева  cout << "Приступим к построению дерева-формулы...\n"; // Вывод подтверждения начала построения  Tree A; // Создание объекта класса Tree  Uzel\* Temp = A.GetTree(); // Получение указателя на корень дерева  // Построение дерева поиска и вывод на экран  for (int i = 0; i < strlen(Formula\_Post); i++) {  k = Formula\_Post[i]; // Получение текущего символа из строки формулы  // Проверяем, является ли символ операцией (+, -, \*, /, ^)  if (strchr("+-\*/^", k) != NULL) {  // Если у текущего узла отсутствует правое поддерево, добавляем операцию туда  if (Temp->Right == NULL) {  Temp->Right = new Uzel; // Выделяем память под новый узел  Temp = Temp->Right; // Перемещаем указатель на правое поддерево  Temp->Key = k; // Устанавливаем символ операции в узле  Temp->Left = Temp->Right = NULL; // Обнуляем указатели на потомков  A.V\_stack(Temp); // Добавляем узел в стек для последующего использования  }  else { // Если правое поддерево уже существует, добавляем операцию в левое поддерево  Temp->Left = new Uzel; // Выделяем память под новый узел  Temp = Temp->Left; // Перемещаем указатель на левое поддерево  Temp->Key = k; // Устанавливаем символ операции в узле  Temp->Left = Temp->Right = NULL; // Обнуляем указатели на потомков  A.V\_stack(Temp); // Добавляем узел в стек для последующего использования  }  }  else { // Если символ является операндом  // Если у текущего узла отсутствует правое поддерево, добавляем операнд туда  if (Temp->Right == NULL) {  Temp->Right = new Uzel; // Выделяем память под новый узел  Temp = Temp->Right; // Перемещаем указатель на правое поддерево  Temp->Key = k; // Устанавливаем символ операнда в узле  Temp->Left = Temp->Right = NULL; // Обнуляем указатели на потомков  // Возвращаем текущий указатель назад, так как правое поддерево уже занято  A.Udalenie(&Ukazatel);  Temp = Ukazatel;  } |

|  |
| --- |
| else { // Если правое поддерево уже существует, добавляем операнд в левое поддерево  Temp->Left = new Uzel; // Выделяем память под новый узел  Temp = Temp->Left; // Перемещаем указатель на левое поддерево  Temp->Key = k; // Устанавливаем символ операнда в узле  Temp->Left = Temp->Right = NULL; // Обнуляем указатели на потомков  // Возвращаем текущий указатель назад, так как правое поддерево уже занято  A.Udalenie(&Ukazatel);  Temp = Ukazatel;  }  }  }  // Вывод дерева в различных форматах  cout << "\nКонтрольный вывод дерева-формулы...\n"; // Вывод заголовка  A.PrintTree(A.GetTree()->Right, 0); // Вывод дерева на экран  cout << "Перед Вами формула, записанная в инфиксной форме...\n"; // Вывод заголовка  A.Print\_Tree\_Back(A.GetTree()->Right, 0); // Вывод инфиксной формы  cout << endl; // Пустая строка для разделения вывода  cout << "------------------------------------------ \n"; // Вывод разделительной строки  cout << "Перед Вами формула, записанная в префиксной форме...\n"; // Вывод заголовка  A.Print\_Tree\_Left(A.GetTree()->Right, 0); // Вывод префиксной формы  cout << endl; // Пустая строка для разделения вывода  cout << "------------------------------------------ \n"; // Вывод разделительной строки  cout << "Перед Вами формула, записанная в постфиксной форме...\n"; // Вывод заголовка  A.Print\_Tree\_End(A.GetTree()->Right, 0); // Вывод постфиксной формы  cout << "\n"; // Пустая строка для разделения вывода  system("PAUSE"); // Ожидание ввода пользователя перед завершением программы  return 0; // Возврат нулевого значения для обозначения успешного завершения программы  } |

|  |
| --- |
|  |

Изменения в программе: удалена библиотека **<conio.h>** и добавлена **<algorithm>**, функция **gets()** заменена на **cin.getline()**, функция **strrev()** заменена на **std::reverse()**, добавлен деструктор **~Tree()** для освобождения памяти. Все изменения направлены на повышение безопасности и читаемости кода, а также на использование стандартных средств ввода-вывода и обработки строк.