Санкт-Петербургский политехнический университет Петра Великого

Высшая школа интеллектуальных систем и суперкомпьютерных технологий

**Отчёт по лабораторной работе №5**

**Дисциплина**: Низкоуровневое программирование

**Тема**: Программирование на языке C

Выполнил студент гр. 3530901/90003 А.А. Ундольский

(подпись)

Преподаватель А. О. Алексюк

(подпись)

“ ” 2021 г.

Санкт-Петербург

2021

Оглавление

[1. Техническое задание 3](#_Toc71541752)

[2. Метод решения 3](#_Toc71541753)

[3. Решение 3](#_Toc71541754)

[3.1. myapp.c 3](#_Toc71541754)

[3.2. bst.c 6](#_Toc71541754)

[3.3. bst.h 10](#_Toc71541754)

[4. Работа программы 11](#_Toc71541755)

[5. Тестирование программы 21](#_Toc71541756)

[5.1. mytests.c 21](#_Toc71541754)

[5.1. mytests.h 22](#_Toc71541754)

[6. GitHub 22](#_Toc71541757)

# Техническое задание

Реализовать на языке C двоичное дерево.

# Метод решения

Binary search tree (оно же двоичное дерево поиска) – это конечное множество элементов, которое либо пусто, либо содержит элемент (корень), связанный с двумя различными бинарными деревьями, называемыми левым и правым поддеревьями. Каждый элемент бинарного дерева называется узлом. Связи между узлами дерева называются его ветвями. Реализация осуществлена с помощью узлов дерева, каждый из которых хранит ссылки на левого и правого потомка, а также содержимое (собственное значение узла). Само дерево содержит указатель на корень (корневой узел) и размер дерева (количество слов). Функции дерева (добавление, удаление, поиск, печать) основаны на поэтапном спуске от узла к узлу.

# Решение

* 1. myapp.c

|  |
| --- |
| #include **<stdio.h>** #include **<stdlib.h>** #include **<unistd.h>** *// Для getopt()* #include **"bst.h"** *// Демонстрационная программа, использующая статическую библиотеку двоичного дерева поиска* **int** main(**int** argc, **char** \*\*argv) {  *// Подсказка об использовании программы* **const char** \*sUsage = **"myapp (binary search tree test program) usage:\nmyapp -a inputFileWithNumbersToAdd -d inputFileWithNumbersToDelete -o outputFile\n"**;   *// Разбираем командную строку* **if**(argc < 7) { *// Eсли аргументов недостаточно, выводим справку* printf(**"%s"**, sUsage);  **return** -1;  }  **char** \*opts = **"a:d:o:"**; *// Опции, каждая имеет аргумент* **char** \*inputFilenameAdd = 0;  **char** \*inputFilenameDel = 0;  **char** \*outputFilename = 0;  **int** opt; *// Прочитанная опция* **while** ((opt = getopt(argc, argv, opts)) != -1 ) { *// Вызываем getopt(), пока она не вернет -1* **switch**(opt) {  **case 'a'**:  inputFilenameAdd = optarg;  **break**;  **case 'd'**:  inputFilenameDel = optarg;  **break**;  **case 'o'**:  outputFilename = optarg;  **break**;  }  }  **if** (inputFilenameAdd == NULL || inputFilenameDel == NULL || outputFilename == NULL) { *// В командной строке не заданы или неверно заданы имена файлов* printf(**"%s"**, sUsage);  **return** -1;  }   *// Контрольный вывод прочитанных из командной строки имен файлов, открытие файлов* printf(**"File with numbers to add: %s\nFile with numbers to delete: %s\nOutput file: %s\n"**, inputFilenameAdd, inputFilenameDel, outputFilename);  FILE \*inputFileAdd, \*inputFileDel, \*outputFile;  inputFileAdd = fopen(inputFilenameAdd, **"r"**);  inputFileDel = fopen(inputFilenameDel, **"r"**);  outputFile = fopen(outputFilename, **"w"**);  **if** (inputFileAdd == NULL) {  printf(**"Input file with numbers to add opening error!\n"**);  **return** -1;  }  **if** (inputFileDel == NULL) {  printf(**"Input file with numbers to delete opening error!\n"**);  **return** -1;  }  **if** (outputFile == NULL) {  printf(**"Output file opening error!\n"**);  **return** -1;  }   *// Создаем дерево* tree \*test\_tree = createTree();   *// Добавляем в дерево узлы, заданные во входном файле* **int** value;  **while** (fscanf(inputFileAdd, **"%d"**, &value) != EOF) {  addNode(test\_tree, value);  };  *// outputFile = stdout; // Для отладки (вывод не в файл, а в консоль)   // Выводим сформированное дерево (тремя способами обхода), его размер, а также минимальный и максимальный узлы* fprintf(outputFile, **"Binary Search Tree\n"**);  fprintf(outputFile, **"==================\n"**);  fprintf(outputFile, **"PREORDER TRAVERSAL:\n"**);  fprintf(outputFile, **"-------------------\n"**);  traverseTree(test\_tree, outputFile, PREORDER);  fprintf(outputFile, **"--------------------\n"**);  fprintf(outputFile, **"POSTORDER TRAVERSAL:\n"**);  fprintf(outputFile, **"--------------------\n"**);  traverseTree(test\_tree, outputFile, POSTORDER);  fprintf(outputFile, **"------------------\n"**);  fprintf(outputFile, **"INORDER TRAVERSAL:\n"**);  fprintf(outputFile, **"------------------\n"**);  traverseTree(test\_tree, outputFile, INORDER);  fprintf(outputFile, **"-------------\n"**);  fprintf(outputFile, **"Tree size: %d\n"**, treeSize(test\_tree));  node \*tree\_min = minNode(test\_tree);  node \*tree\_max = maxNode(test\_tree);  fprintf(outputFile, **"Min node: %d\n"**, nodeValue(tree\_min));  fprintf(outputFile, **"Max node: %d\n"**, nodeValue(tree\_max));  fprintf(outputFile, **"-------------\n"**);   *// По очереди удаляем из дерева узлы, заданные во входном файле, и выводим дерево, его размер, минимальный и максимальный узлы,  // а также проверяем наличие в дереве двух произвольно заданных узлов* **while** (fscanf(inputFileDel, **"%d"**, &value) != EOF) {  removeNode(test\_tree, value);  fprintf(outputFile, **"\nTree after deleting %d:\n"**, value);  fprintf(outputFile, **"-------------------\n"**);  fprintf(outputFile, **"PREORDER TRAVERSAL:\n"**);  fprintf(outputFile, **"-------------------\n"**);  traverseTree(test\_tree, outputFile, PREORDER);  fprintf(outputFile, **"--------------------\n"**);  fprintf(outputFile, **"POSTORDER TRAVERSAL:\n"**);  fprintf(outputFile, **"--------------------\n"**);  traverseTree(test\_tree, outputFile, POSTORDER);  fprintf(outputFile, **"------------------\n"**);  fprintf(outputFile, **"INORDER TRAVERSAL:\n"**);  fprintf(outputFile, **"------------------\n"**);  traverseTree(test\_tree, outputFile, INORDER);  fprintf(outputFile, **"-------------\n"**);  fprintf(outputFile, **"Tree size: %d\n"**, treeSize(test\_tree));  tree\_min = minNode(test\_tree);  tree\_max = maxNode(test\_tree);  fprintf(outputFile, **"Min node: %d\n"**, nodeValue(tree\_min));  fprintf(outputFile, **"Max node: %d\n"**, nodeValue(tree\_max));  value=4;  fprintf(outputFile, **"Value=%d -> node is %s\n"**, value, containsNode(test\_tree, value)? **"present"** : **"not present"**);  value=12;  fprintf(outputFile, **"Value=%d -> node is %s\n"**, value, containsNode(test\_tree, value)? **"present"** : **"not present"**);  fprintf(outputFile, **"-------------\n"**);  };   *// Закрываем файлы* fclose(inputFileAdd);  fclose(inputFileDel);  fclose(outputFile);   *// Освобождаем память, занятую деревом* destroyTree(test\_tree);   **return** 0;  } |

* 1. bst.c

|  |
| --- |
| *// Реализация двоичного дерева поиска на языке Си //* #include **<stdio.h>** #include **<stdlib.h>** #include **"bst.h"** *// Узел дерева* **typedef struct** node {  **int** value; *// Значение в узле* **struct** node \*left; *// Указатель на левого потомка* **struct** node \*right; *// Указатель на правого потомка* } node;  *// Дерево* **typedef struct** tree {  node \*root; *// Корень дерева* **int** count; *// Количество узлов в дереве* } tree;   *// Создание пустого дерева* tree \*createTree(**void**) {  tree \*new\_tree = malloc(**sizeof** \*new\_tree);  **if** (new\_tree == NULL) **return** NULL; *// Не удалось выделить память* new\_tree->root = NULL;  new\_tree->count = 0;  **return** new\_tree; *// Возвращаем указатель на дерево* }  *// Получение размера дерева // Возврат: количество узлов в дереве* **extern int** treeSize(tree \*bst) {  **return** bst->count; }  *// Получение значения из узла // Возврат: Значение в узле* **extern int** nodeValue(node \*n) {  **return** n->value; }   *// Проверка наличия в дереве узла с заданным значением // Возврат: 0 - такого узла нет, 1 - такой узел есть* **int** containsNode(**const** tree \*bst, **int** value) {  **const** node \*bst\_node = bst->root;  **while**(1) {  **if** (bst\_node == NULL) **return** 0;  **else if** (value == bst\_node->value) **return** 1;  **else if** (value > bst\_node->value) bst\_node = bst\_node->right;  **else** bst\_node = bst\_node->left;  } }  *// Вставка узла в дерево // Возврат: 0 - не удалось вставить узел, 1 - узел вставлен, 2 - такой узел уже есть в дереве* **int** addNode(tree \*bst, **int** value) {  node \*bst\_node = bst->root;  node \*\*new = &bst->root;   **while**(1) {  **if**(bst\_node == NULL) { *// Нашли место для вставки узла* bst\_node = \*new = malloc(**sizeof** \*bst\_node);  **if**(bst\_node != NULL) { *// Вставляем узел* bst\_node->value = value;  bst\_node->left = bst\_node->right = NULL;  bst->count++;  **return** 1;  }  **else return** 0; *// Не удалось добавить узел* }   **if** (value == bst\_node->value) **return** 2; *// Такой узел уже есть в дереве   // Ищем место для вставки узла* **if**(value > bst\_node->value) { *// Идем в правое поддерево* new = &bst\_node->right;  bst\_node = bst\_node->right;  }  **else** { *// Идем в левое поддерево* new = &bst\_node->left;  bst\_node = bst\_node->left;  }  } *// end of while* }   *// Удаление узла из дерева // Возврат: 0 - не удалось удалить узел, 1 - узел удален* **int** removeNode(tree \*bst, **int** value) {  node \*delNode = bst->root; *// Указатель на удаляемый узел* node \*\*aux = &bst->root;*// Вспомогательный указатель   // Ищем удаляемый узел в дереве* **while**(1) {  **if** (delNode == NULL) **return** 0; *// Не нашли удаляемый узел* **else if** (value == delNode->value) **break**; *// Нашли удаляемый узел - выходим из цикла поиска* **else if** (value > delNode->value) {  aux = &delNode->right;  delNode = delNode->right;  }  **else** {  aux = &delNode->left;  delNode = delNode->left;  }  }   *// Удаляем узел* **if** (delNode->right == NULL) { *// У удаляемого узла нет потомка справа* \*aux = delNode->left;  }  **else** { *// У удаляемого узла есть потомок справа* node \*y = delNode->right; *// Потомок справа* **if** (y->left == NULL) { *// У правого потомка удаляемого узла нет детей слева* y->left = delNode->left;  \*aux=y;  }  **else** { *// У правого потомка удаляемого узла есть левое поддерево* node \*x = y->left;  **while**(x->left != NULL) { *// Идем по левому поддереву до конца* y = x;  x = y->left;  }  *// Переносим узлы* y->left = x->right;  x->left = delNode->left;  x->right = delNode->right;  \*aux=x;  }  }  bst->count--; *// Удалили узел - уменьшаем количество узлов в дереве* free(delNode); *// Освобождаем память* **return** 1; }  **static void** fprintNode(node \*curNode, FILE \*f) {  fprintf(f, **"%d"**, curNode->value);  **if** (curNode->left != NULL) {  fprintf(f, **"-"**);  fprintf(f, **"%d(L)"**, curNode->left->value);  **if** (curNode->right != NULL) fprintf(f, **",%d(R)"**, curNode->right->value);  }  **else if** (curNode->right != NULL) {  fprintf(f, **"-"**);  fprintf(f, **"%d(R)"**, curNode->right->value);  }  fprintf(f, **"\n"**); }  *// Прямой обход дерева с выводом результата в заданный файл* **static void** preorder(node \*root, FILE \*f) {  **if** (root == NULL) **return**;  fprintNode(root, f);  preorder(root->left, f);  preorder(root->right, f); }  *// Обратный обход дерева с выводом результата в заданный файл* **static void** postorder(node \*root, FILE \*f) {  **if** (root == NULL) **return**;  postorder(root->left, f);  postorder(root->right, f);  fprintNode(root, f); }  *// Центрированный обход дерева с выводом результата в заданный файл* **static void** inorder(node \*root, FILE \*f) {  **if** (root == NULL) **return**;  inorder(root->left, f);  fprintNode(root, f);  inorder(root->right, f); }  *// Обход дерева в заданном порядке с выводом узлов в заданный файл* **void** traverseTree(tree \*bst, FILE \*f, **int** order) {  **switch**(order) {  **case** PREORDER:  preorder(bst->root, f);  **break**;  **case** POSTORDER:  postorder(bst->root, f);  **break**;  **case** INORDER:  inorder(bst->root, f);  } }   *// Освобождение памяти, занятой узлом и его потомками* **static void** destroySubtree(node \*bst\_node) {  **if**(bst\_node == NULL) **return**;  destroySubtree(bst\_node->left);  destroySubtree(bst\_node->right);  free(bst\_node); }   *// Освобождение памяти, занятой деревом* **void** destroyTree(tree \*bst) {  destroySubtree(bst->root);  free(bst); }  *// Поиск минимального узла в дереве* node \*minNode(tree \*bst) {  node \*bst\_node = bst->root;  **while** (bst\_node->left != NULL) bst\_node = bst\_node->left;  **return** bst\_node; }  *// Поиск максимального узла в дереве* node \*maxNode(tree \*bst) {  node \*bst\_node = bst->root;  **while** (bst\_node->right != NULL) bst\_node = bst\_node->right;  **return** bst\_node; } |

* 1. bst.h

|  |
| --- |
| #ifndef BST\_H #define BST\_H  *// Реализация двоичного дерева поиска на языке Си //  // Узел дерева* **struct** node; **typedef struct** node node;  *// Дерево* **struct** tree; **typedef struct** tree tree;  *// Создание пустого дерева // Возврат: указатель на дерево (NULL если дерево не создано)* **extern** tree \*createTree(**void**);  *// Получение размера дерева // Возврат: количество узлов в дереве* **extern int** treeSize(tree \*bst);  *// Проверка наличия в дереве узла с заданным значением // Возврат: 0 - такого узла нет, 1 - такой узел есть* **extern int** containsNode(**const** tree \*bst, **int** value);  *// Получение значения из узла // Возврат: Значение в узле* **extern int** nodeValue(node \*n);  *// Вставка узла в дерево // Возврат: 0 - не удалось вставить узел, 1 - узел вставлен, 2 - такой узел уже есть в дереве* **extern int** addNode(tree \*bst, **int** value);  *// Удаление узла из дерева // Возврат: 0 - не удалось удалить узел, 1 - узел удален* **extern int** removeNode(tree \*bst, **int** value);  *// Поиск минимального узла в дереве // Возврат: указатель на узел с минимальным значением в дереве* **extern** node \*minNode(tree \*bst);  *// Поиск максимального узла в дереве // Возврат: указатель на узел с максимальным значением в дереве* **extern** node \*maxNode(tree \*bst);  *// Обход дерева с выводом узлов в заданный файл* #define PREORDER 0 *// Прямой обход* #define POSTORDER 1 *// Обратный обход* #define INORDER 2 *// Центрированный обход* **extern void** traverseTree(tree \*bst, FILE \*f, **int** order);  *// Освобождение памяти, занятой деревом* **extern void** destroyTree(tree \*bst);   #endif |

# Работа программы

Изображение выглядит как текст

Автоматически созданное описание

add.txt

10

4

3

13

27

3

31

31

100

7

12

8

del.txt

31

100

12

3

4

5

out.txt

Binary Search Tree

==================

PREORDER TRAVERSAL:

-------------------

10-4(L),13(R)

4-3(L),7(R)

3

7-8(R)

8

13-12(L),27(R)

12

27-31(R)

31-100(R)

100

--------------------

POSTORDER TRAVERSAL:

--------------------

3

8

7-8(R)

4-3(L),7(R)

12

100

31-100(R)

27-31(R)

13-12(L),27(R)

10-4(L),13(R)

------------------

INORDER TRAVERSAL:

------------------

3

4-3(L),7(R)

7-8(R)

8

10-4(L),13(R)

12

13-12(L),27(R)

27-31(R)

31-100(R)

100

-------------

Tree size: 10

Min node: 3

Max node: 100

-------------

Tree after deleting 31:

-------------------

PREORDER TRAVERSAL:

-------------------

10-4(L),13(R)

4-3(L),7(R)

3

7-8(R)

8

13-12(L),27(R)

12

27-100(R)

100

--------------------

POSTORDER TRAVERSAL:

--------------------

3

8

7-8(R)

4-3(L),7(R)

12

100

27-100(R)

13-12(L),27(R)

10-4(L),13(R)

------------------

INORDER TRAVERSAL:

------------------

3

4-3(L),7(R)

7-8(R)

8

10-4(L),13(R)

12

13-12(L),27(R)

27-100(R)

100

-------------

Tree size: 9

Min node: 3

Max node: 100

Value=4 -> node is present

Value=12 -> node is present

-------------

Tree after deleting 100:

-------------------

PREORDER TRAVERSAL:

-------------------

10-4(L),13(R)

4-3(L),7(R)

3

7-8(R)

8

13-12(L),27(R)

12

27

--------------------

POSTORDER TRAVERSAL:

--------------------

3

8

7-8(R)

4-3(L),7(R)

12

27

13-12(L),27(R)

10-4(L),13(R)

------------------

INORDER TRAVERSAL:

------------------

3

4-3(L),7(R)

7-8(R)

8

10-4(L),13(R)

12

13-12(L),27(R)

27

-------------

Tree size: 8

Min node: 3

Max node: 27

Value=4 -> node is present

Value=12 -> node is present

-------------

Tree after deleting 12:

-------------------

PREORDER TRAVERSAL:

-------------------

10-4(L),13(R)

4-3(L),7(R)

3

7-8(R)

8

13-27(R)

27

--------------------

POSTORDER TRAVERSAL:

--------------------

3

8

7-8(R)

4-3(L),7(R)

27

13-27(R)

10-4(L),13(R)

------------------

INORDER TRAVERSAL:

------------------

3

4-3(L),7(R)

7-8(R)

8

10-4(L),13(R)

13-27(R)

27

-------------

Tree size: 7

Min node: 3

Max node: 27

Value=4 -> node is present

Value=12 -> node is not present

-------------

Tree after deleting 3:

-------------------

PREORDER TRAVERSAL:

-------------------

10-4(L),13(R)

4-7(R)

7-8(R)

8

13-27(R)

27

--------------------

POSTORDER TRAVERSAL:

--------------------

8

7-8(R)

4-7(R)

27

13-27(R)

10-4(L),13(R)

------------------

INORDER TRAVERSAL:

------------------

4-7(R)

7-8(R)

8

10-4(L),13(R)

13-27(R)

27

-------------

Tree size: 6

Min node: 4

Max node: 27

Value=4 -> node is present

Value=12 -> node is not present

-------------

Tree after deleting 4:

-------------------

PREORDER TRAVERSAL:

-------------------

10-7(L),13(R)

7-8(R)

8

13-27(R)

27

--------------------

POSTORDER TRAVERSAL:

--------------------

8

7-8(R)

27

13-27(R)

10-7(L),13(R)

------------------

INORDER TRAVERSAL:

------------------

7-8(R)

8

10-7(L),13(R)

13-27(R)

27

-------------

Tree size: 5

Min node: 7

Max node: 27

Value=4 -> node is not present

Value=12 -> node is not present

-------------

Tree after deleting 5:

-------------------

PREORDER TRAVERSAL:

-------------------

10-7(L),13(R)

7-8(R)

8

13-27(R)

27

--------------------

POSTORDER TRAVERSAL:

--------------------

8

7-8(R)

27

13-27(R)

10-7(L),13(R)

------------------

INORDER TRAVERSAL:

------------------

7-8(R)

8

10-7(L),13(R)

13-27(R)

27

-------------

Tree size: 5

Min node: 7

Max node: 27

Value=4 -> node is not present

Value=12 -> node is not present

-------------

# Тестирование программы

Для проверки правильности работы программы были написаны автоматические тесты.

mytests.c

|  |
| --- |
| #include **<stdio.h>** #include **<stdlib.h>** #include **"mytests.h"** #include **"bst.h"** *// Тестовая программа, реализующая модульные тесты для функций статической библиотеки двоичного дерева поиска* **int** main() {  **int** result;   *// Тест для createTree()* tree \*bst = NULL;  ASSERT\_NULL(bst);  bst = createTree();  ASSERT\_NOT\_NULL(bst);  **int** size = treeSize(bst);  ASSERT\_EQUALS(size,0); *// Дерево должно быть пустое   // Тесты для addNode(), containsNode() и treeSize()* **const int** iMin = 1; *// Зададим минимальный узел* **const int** iMax = 101; *// Зададим максимальный узел* **const int** iStep = 10;*// Шаг изменения в цикле* **int** cnt = 0;  printf(**"Adding nodes to tree:"**);  **for** (**int** i=iMin; i<=iMax; i+=iStep) { *// Добавляем узлы в дерево (от минимального до максимального) и проверяем как они добавились* result = addNode(bst, i);  ASSERT\_EQUALS(result, 1); *// Должен быть добавлен* cnt++;  **int** contains = containsNode(bst, i); *// Узел (i) добавили* ASSERT\_TRUE(contains); *// - проверим, что он есть в дереве* contains = containsNode(bst, i+1); *// Узел (i+1) не добавляли* ASSERT\_FALSE(contains); *// - проверим, что его нет в дереве* size = treeSize(bst); *// Проверим увеличение размера дерева* ASSERT\_EQUALS(size,cnt);  printf(**"\n"**);  traverseTree(bst, stdout, INORDER); *// Посмотрим на изменение дерева* }   *// Тест для addNode() - попытка вставить уже существующий узел* result = addNode(bst, iMin);  ASSERT\_EQUALS(result, 2);  *// Тест для minNode()* **int** min = nodeValue(minNode(bst));  ASSERT\_EQUALS(min, iMin);  *// Тест для maxNode()* **int** max = nodeValue(maxNode(bst));  ASSERT\_EQUALS(max, iMax);   *// Тест для removeNode() - попытка удалить несуществующий узел* result = removeNode(bst, iMin+1);  ASSERT\_EQUALS(result, 0);   printf(**"\n\nRemoving nodes from tree:"**);  *// Тесты для removeNode(), containsNode() и treeSize()* **for** (**int** i=iMin; i<=iMax; i+=iStep) {  result=removeNode(bst, i);  ASSERT\_EQUALS(result, 1); *// Должен быть удален* cnt--;  **int** contains = containsNode(bst, i); *// Узел (i) удалили* ASSERT\_FALSE(contains); *// - проверим, что его нет в дереве* size = treeSize(bst); *// Проверим уменьшение размера дерева* ASSERT\_EQUALS(size,cnt);  printf(**"\n"**);  traverseTree(bst, stdout, INORDER); *// Посмотрим на изменение дерева* }  ASSERT\_EQUALS(size,0); *// Проверим, что удалены все узлы* destroyTree(bst); *// Удаляем дерево - освобождаем память* **return** 0; } |

mytests.h

|  |
| --- |
| #ifndef MYTESTS\_H #define MYTESTS\_H  #define ASSERT\_EQUALS(a,b) **if** ((a)!=(b)) printf(**"\nLine %d: ASSERT\_EQUALS failed\n"**, \_\_LINE\_\_); #define ASSERT\_NOT\_EQUALS(a,b) **if** ((a)==(b)) printf(**"\nLine %d: ASSERT\_NOT\_EQUALS failed\n"**, \_\_LINE\_\_);  *// За false принимается 0, за true - любое значение, отличное от нуля* #define ASSERT\_TRUE(expression) **if** (!(expression)) printf(**"\nLine %d: ASSERT\_TRUE failed\n"**, \_\_LINE\_\_); #define ASSERT\_FALSE(expression) **if** ((expression)) printf(**"\nLine %d: ASSERT\_FALSE failed\n"**, \_\_LINE\_\_);  #define ASSERT\_NULL(pointer) **if** ((pointer)!=NULL) printf(**"\nLine %d: ASSERT\_NULL failed\n"**, \_\_LINE\_\_); #define ASSERT\_NOT\_NULL(pointer) **if** ((pointer)==NULL) printf(**"\nLine %d: ASSERT\_NOT\_NULL failed\n"**, \_\_LINE\_\_);  #endif |

# GitHub

<https://github.com/tengen-toppa-gurren-lagann/BST-C>