|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | МИНОБРНАУКИ РОССИИ | | | | Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | |   Институт Информационных технологий | |
|  | |
| Кафедра Математического обеспечения и стандартизации информационных технологий | |
|  | |
|  | |

|  |  |
| --- | --- |
| **ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 6** | |
| **по дисциплине** | |
| **«**Структуры и алгоритмы обработки данных**»**  **Тема: «Рекурсивные алгоритмы и их реализация»** | |
|  | |
| Выполнил студент группы ИКБО-08-22 | Сенькевич Г.Д. |
| Принял преподаватель | Филатов А.С. |

|  |  |  |
| --- | --- | --- |
| Лабораторная работа выполнена | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись студента)* |
|  |  |  |
| «Зачтено» | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись руководителя)* |

Москва 2023

# **Цель работы**

Получить знания и практические навыки по разработке и реализации рекурсивных процессов.

# **Постановка задачи**

Разработать и протестировать рекурсивные функции в соответствии с задачами варианта. Составить отчет.

Требования к выполнению первой задачи варианта:

* приведите итерационный алгоритм решения задачи
* реализуйте алгоритм в виде функции и отладьте его
* определите теоретическую сложность алгоритма
* реализуйте и отладьте рекурсивную функцию решения задачи
* определите глубину рекурсии, изменяя исходные данные
* определите сложность рекурсивного алгоритма, используя метод подстановки и дерево рекурсии
* приведите для одного из значений схему рекурсивных вызовов
* разработайте программу, демонстрирующую выполнение обеих функций, и покажите результаты тестирования.

Требования к выполнению второй задачи варианта:

* рекурсивную функцию для обработки списковой структуры согласно варианту. Информационная часть узла – простого типа – целого;
* определите глубину рекурсии
* определите теоретическую сложность алгоритма
* разработайте программу, демонстрирующую работу функций и покажите результаты тестов.

Вариант №8. Условие задания:

|  |  |
| --- | --- |
| Упражнение 1 | Перевести число из 10-системы счисления в систему с основанием В(1<В≤10) |
| Упражнение 2 | Удаление очереди, реализованной на однонаправленном списке |

# **Решение**

Рекурсия — это процесс, при котором функция вызывает саму себя в своем теле. Она может применяться для решения задач, которые могут быть разбиты на более простые версии той же задачи. Рекурсия часто используется в программировании для написания алгоритмов и обработки данных, таких как обход деревьев и списков. Когда функция вызывает саму себя, это приводит к созданию нового экземпляра функции, который выполняет свою работу независимо от оригинальной функции, пока не будет достигнуто условие выхода из рекурсии.

Очередь (queue) — это структура данных, которая представляет собой список элементов, где новые элементы добавляются в конец очереди, а элементы извлекаются из начала очереди. Такая структура данных используется для хранения элементов, которые нужно обрабатывать в порядке их поступления.

Операции, которые можно выполнять с очередью:

* pust(item) - добавляет элемент item в конец очереди.
* pop() - извлекает и удаляет первый элемент из очереди.
* top() - возвращает первый элемент очереди без его удаления.
* is\_empty() - проверяет, пуста ли очередь.

Очередь может быть реализована разными способами, например, на массиве или на связном списке. Если очередь реализована на массиве, то при добавлении элемента в конец очереди может потребоваться увеличение размера массива, если он заполнен. При извлечении элемента из начала очереди необходимо сдвинуть все элементы в массиве на одну позицию влево.

Очередь поддерживает принцип FIFO (First In First Out) - первым в очередь пришел, первым и вышел. Этот принцип является основой для решения многих задач, например, обработки задач в операционных системах, моделировании событий в науке и технике, и т.д.

Ответ на вопрос №2:

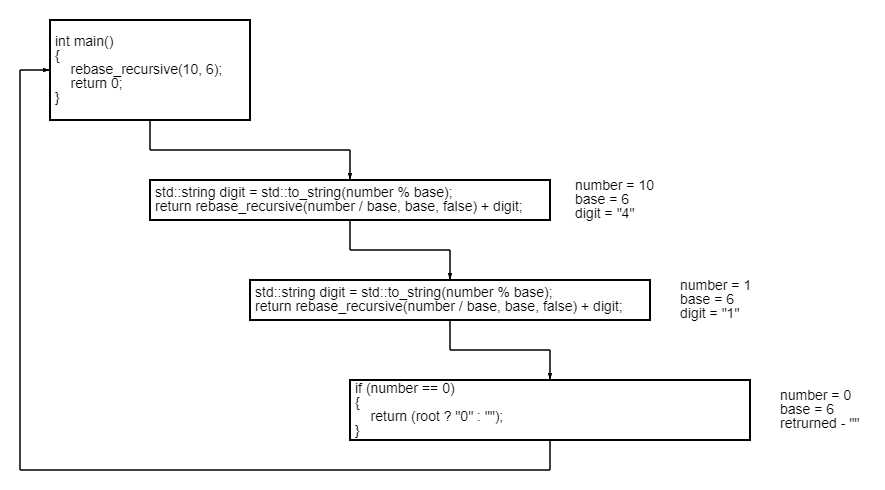
Шаг рекурсии — это один шаг выполнения рекурсивной функции, включающий вызов самой себя с определенными аргументами. Каждый шаг рекурсии уменьшает размер задачи, которую решает функция, и приводит к новому вызову функции с меньшими аргументами, до тех пор, пока размер задачи не станет настолько маленьким, чтобы решить ее можно было без дополнительных вызовов функции. Когда достигнут базовый случай, то есть когда размер задачи становится маленьким и простым, функция завершает свою работу и возвращает результаты выполнения базового случая, которые используются для решения более сложных задач, возникающих на более ранних шагах рекурсии.

Для решения первого упражнения были написаны функции rebase\_iterative и rebase\_recursive (первая решает задачу итеративно, а вторая - рекурсивно), выполняющие перевод числа из 10-ичной системы счисления в заданную (от 2 до 10). Функция принимает на вход две целочисленных переменных – число, которое нужно перевести и основание новой системы счисления и возвращает строку – результат перевода.

Теоретическая сложность итерационного алгоритма (rebase\_iterative) – O(logb(n)) = O(log(n) / log(b)), где n – число, b – основание системы счисления, в которую его нужно перевести.

Теоретическая сложность рекурсивного алгоритма (rebase\_recursivee) – O(logb(n)) = O(log(n) / log(b)), где n – число, b – основание системы счисления, в которую его нужно перевести. Глубина рекурсии, соответственно, также равна .

|  |
| --- |
| std::string rebase\_iterative(int number, int base)  {  std::string res\_reverse;  while (number > 0)  {  res\_reverse += std::to\_string(number % base);  number /= base;  }  std::string res;  for (int i = res\_reverse.length(); i >= 0; i--)  {  res += res\_reverse[i];  }  if (res.length() == 0)  {  res = "0";  }  return res;  }  std::string rebase\_recursive(int number, int base, bool root=true)  {  if (number == 0)  {  return (root ? "0" : "");  }  std::string digit = std::to\_string(number % base);  return rebase\_recursive(number / base, base, false) + digit;  } |

Схема рекурсивных вызовов функции rebase\_recursive, вызванной с аргументами 10 и 6 (результат работы - 14):

Для решения второго упражнения была написана функция delete\_queue, выполняющая рекурсивное удаление очереди. Функция принимает очередь (класс очереди реализован отдельно) и ничего не возвращает.

Теоретическая сложность алгоритма – O(n) где n – количество элементов в очереди. Глубина рекурсии, соответственно, также равна n.

|  |
| --- |
| // Заголовочный файл для очереди  #ifndef QUEUE\_H  #define QUEUE\_H  #include "list.h"  struct queue  {  list elems;  bool empty();  void push(int value);  int pop();  int size();  };  #endif  // Функция удаления очереди  void delete\_queue(queue& q)  {  if (!q.empty())  {  q.elems.print();  q.pop();  delete\_queue(q);  }  } |

При запуске программы пользователь видит пользовательское меню, позволяющее выбрать между двумя упражнениями варианта.

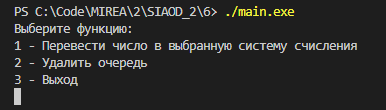


Рисунок 1. Интерфейс программы

# **Тестирование**

Протестируем выполнение программой первого упражнения. Для этого выберем в меню первую функцию, введём число 10 и новую систему счисления 6, поочерёдно выберем сначала итеративную реализацию решения, а потом рекурсивную. В обоих случаях программа должна дать ответ 14. На рисунке 2 видим, как программа вывела верный ответ.

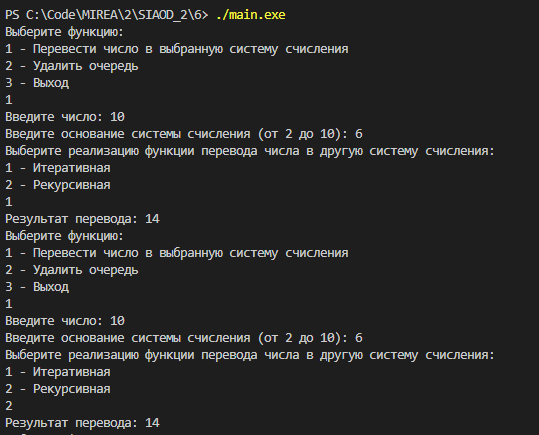


Рисунок 2. Тестирование программы

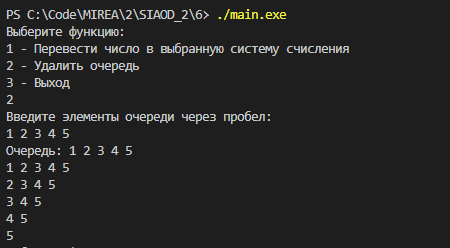
Протестируем выполнение программой второго упражнения. Для этого выберем в меню вторую функцию, введём элементы очереди в порядке их добавления в очередь. На рисунке 3 видим, как программа удалила элементы очереди в правильном порядке.

Рисунок 3. Тестирование программы

Из результатов выполнения программы видно, что программа работает корректно, решая все поставленные задачи.

# **Вывод**

В результате выполнения работы я:

1. Научился писать рекурсивные функции на языке C++
2. Научился реализовывать структуру данных очередь в C++.

# **Исходный код программы**

|  |
| --- |
| // ФАЙЛ main.cpp  #include <iostream>  #include <string>  #include "queue.h"  std::string rebase\_iterative(int number, int base)  {  std::string res\_reverse;  while (number > 0)  {  res\_reverse += std::to\_string(number % base);  number /= base;  }  std::string res;  for (int i = res\_reverse.length(); i >= 0; i--)  {  res += res\_reverse[i];  }  if (res.length() == 0)  {  res = "0";  }  return res;  }  std::string rebase\_recursive(int number, int base, bool root=true)  {  if (number == 0)  {  return (root ? "0" : "");  }  std::string digit = std::to\_string(number % base);  return rebase\_recursive(number / base, base, false) + digit;  }  void delete\_queue(queue& q)  {  if (!q.empty())  {  q.elems.print();  q.pop();  delete\_queue(q);  }  }  int main()  {  while (true)  {  int choice = 0;  std::cout << "Выберите функцию:\n"  << "1 - Перевести число в выбранную систему счисления\n"  << "2 - Удалить очередь\n"  << "3 - Выход\n";  std::cin >> choice;  if (choice == 1)  {  int number = 0, base = 0;  std::cout << "Введите число: ";  std::cin >> number;  std::cout << "Введите основание системы счисления (от 2 до 10): ";  std::cin >> base;  if (base >= 2 && base <= 10)  {  int way = 0;  std::cout << "Выберите реализацию функции перевода числа в другую систему счисления:\n"  << "1 - Итеративная\n"  << "2 - Рекурсивная\n";  std::cin >> way;  if (way == 1)  {  std::cout << "Результат перевода: "  << rebase\_iterative(number, base) << '\n';  }  else if (way == 2)  {  std::cout << "Результат перевода: "  << rebase\_recursive(number, base) << '\n';  }  else  {  std::cout << "Такого варианта нет!\n";  }  }  else  {  std::cout << "Некорректное основание!\n";  }  }  else if (choice == 2)  {  queue q;  std::cout << "Введите элементы очереди через пробел:\n";  std::cin.get();  char pilot = std::cin.get();  double elem = 0;  while (pilot != '\n')  {  std::cin.putback(pilot);  std::cin >> elem;  q.push(elem);  pilot = std::cin.get();  }  std::cout << "Очередь: ";  q.elems.print();  delete\_queue(q);  }  else if (choice == 3)  {  break;  }  else  {  std::cout << "Такого варианта нет!\n";  }  }  return 0;  }  // ФАЙЛ queue.h  #ifndef QUEUE\_H  #define QUEUE\_H  #include "list.h"  struct queue  {  list elems;  bool empty();  void push(int value);  int pop();  int size();  };  #endif  // ФАЙЛ queue.cpp  #include "queue.h"  bool queue::empty()  {  return (elems.size == 0);  }  void queue::push(int value)  {  elems.insert\_after(elems.last, value);  }  int queue::pop()  {  int ret = elems.first->value;  elems.remove(elems.first);  return ret;  }  int queue::size()  {  return elems.size;  }  // ФАЙЛ list.h  #ifndef LIST\_H  #define LIST\_H  #include <iostream>  struct list\_node  {  int value;  list\_node\* next;  list\_node();  list\_node(int value);  };  struct list  {  list\_node\* first;  list\_node\* last;  int size;  list();  ~list();  void insert\_before(list\_node\* node, int value);  void insert\_after(list\_node\* node, int value);  void append(int value);  void remove(list\_node\* node);  void remove\_after(list\_node\* node);  void print();  list\_node\* find\_max();  list\_node\* find\_first\_negative();  void insert\_two\_before\_last(int value1, int value2);  void delete\_first\_negative();  void move\_max\_to\_end();  };  #endif  // ФАЙЛ list.cpp  #include "list.h"  list\_node::list\_node()  {  value = 0;  next = nullptr;  }  list\_node::list\_node(int value)  {  this->value = value;  next = nullptr;  }  list::list()  {  first = nullptr;  last = nullptr;  size = 0;  }  list::~list()  {  list\_node\* current = first;  while (current != nullptr)  {  list\_node\* next\_node = current->next;  delete current;  current = next\_node;  }  }  void list::insert\_before(list\_node\* node, int value)  {  if (node == nullptr)  {  insert\_after(last, value);  }  else if (node == first)  {  list\_node\* new\_node = new list\_node(value);  new\_node->next = first;  first = new\_node;  size++;  }  else  {  list\_node\* current = first;  while (current != nullptr)  {  if (current->next == node)  {  insert\_after(current, value);  break;  }  current = current->next;  }  }  }  void list::insert\_after(list\_node\* node, int value)  {  if (node == nullptr)  {  if (first != nullptr)  {  insert\_before(first, value);  }  else  {  first = new list\_node(value);  last = first;  size++;  }  }  else  {  list\_node\* new\_node = new list\_node(value);  new\_node->next = node->next;  node->next = new\_node;  size++;  if (last == node)  {  last = new\_node;  }  }  }  void list::append(int value)  {  insert\_after(last, value);  }  void list::remove(list\_node\* node)  {  if (node == first)  {  first = node->next;  delete node;  size--;  }  else  {  list\_node\* current = first;  while (current != nullptr)  {  if (current->next == node)  {  remove\_after(current);  break;  }  current = current->next;  }  }  }  void list::remove\_after(list\_node\* node)  {  if (node != nullptr && node->next != nullptr)  {  list\_node\* to\_delete = node->next;  node->next = to\_delete->next;  if (last == to\_delete)  {  last = node;  }  delete to\_delete;  size--;  }  }  void list::print()  {  list\_node\* current = first;  while (current != nullptr)  {  std::cout << current->value << ' ';  current = current->next;  }  std::cout << '\n';  }  list\_node\* list::find\_max()  {  list\_node\* max = nullptr;  list\_node\* current = first;  while (current != nullptr)  {  if (max == nullptr || current->value > max->value)  {  max = current;  }  current = current->next;  }  return max;  }  list\_node\* list::find\_first\_negative()  {  list\_node\* current = first;  while (current != nullptr)  {  if (current->value < 0)  {  return current;  }  current = current->next;  }  return nullptr;  }  void list::insert\_two\_before\_last(int value1, int value2)  {  insert\_before(last, value1);  insert\_before(last, value2);  }  void list::delete\_first\_negative()  {  list\_node\* first\_negative = find\_first\_negative();  remove(first\_negative);  }  void list::move\_max\_to\_end()  {  list\_node\* max = find\_max();  int max\_value = max->value;  remove(max);  insert\_after(last, max\_value);  } |