|  |  |
| --- | --- |
| ДИСЦИПЛИНА | Алгоритмы и структуры данных |
| ИНСТИТУТ | Институт перспективных технологий и индустриального программирования |
| КАФЕДРА | Кафедра индустриального программирования |
| ВИД УЧЕБНОГО МАТЕРИАЛА | Текущий контроль |
| ПРЕПОДАВАТЕЛЬ | Преснецова Виктория Юрьевна, Яковлев Михаил Сергеевич, Дворецкий Артур Геннадьевич, Гиматдинов Дамир Маратович |
| СЕМЕСТР | 2 семестр, 2024-2025 гг. |

**Рабочая тетрадь 1.**

**Связные списки. Двоичные деревья поиска. Дерево отрезков**

|  |
| --- |
| **Требования** |
| 1. Язык программирования - C++. 2. Код должен быть оптимизирован для производительности и использования ресурсов. 3. Необходимо избегать избыточных вычислений и памяти. 4. Комментарии должны объяснять сложные участки кода и логику работы программы. |

**1. Двусвязный список. Кольца**

|  |  |
| --- | --- |
| **Теоретический материал** | |
| Двусвязные списки (doubly linked lists) - это динамическая структура данных, состоящая из узлов, каждый из которых содержит три компонента:  • Значение (данные).  • Указатель на предыдущий узел.  • Указатель на следующий узел.  Эта структура данных обеспечивает удобную вставку и удаление элементов как с начала, так и с конца списка за постоянное время O(1) , что делает её полезной для задач, где требуется частое обновление структуры.    Кольцевой список (circular linked list) - это разновидность связного списка, в которой последний узел связан с первым, образуя цикл. В случае двусвязного кольцевого списка указатели «предыдущий» и «следующий» тоже замыкаются на узлы.    Каждый узел списка в C++ обычно представлен структурой или классом.  *Пример структуры узла:*  struct Node {  int data; // Данные узла  Node\* next; // Указатель на следующий узел  Node\* prev; // Указатель на предыдущий узел  } | |
| **Пример 1** | |
| ***Задача:*** | |
|  | 1. Определите структуру Node, которая будет представлять узел двусвязного списка. Узел должен содержать:   1. Целочисленное поле data для хранения данных. 2. Указатель next на следующий узел. 3. Указатель prev на предыдущий узел.   2. Реализуйте функцию append(), которая добавляет новый узел в конец двусвязного списка.  3. Реализуйте функцию display(), которая выводит элементы списка в прямом порядке. |
| ***Решение:*** | |
|  |  |
| ***Ответ:*** | |
|  |  |
| **Задание 1 (1 балл)** | |
| ***Задача:*** | |
|  | Представьте, что в списке некоторые узлы случайно «ссылаются» не только на следующий, но и пересекаются (указывают на узлы из середины списка). Нужно проверить, что таких «лишних» ссылок в структуре нет, то есть каждый узел ссылается ровно на один следующий элемент (или на NULL).  Как обнаружить «проскок» или «перескок», не используя хеш-таблицу или другие явные структуры хранения посещённых узлов? |
| ***Решение:*** | |
|  | #include <iostream>  struct Node {      int data;      Node\* next;      Node\* prev;  };  void append(Node\*\* head, int new\_data) {      Node\* new\_node = new Node();      Node\* last = \*head;      new\_node->data = new\_data;      new\_node->next = nullptr;      // Если список пустой      if (\*head == nullptr) {          new\_node->prev = nullptr;          \*head = new\_node;          return;      }      // Находим последний узел в списке, чтобы после него добавить новый элемент      while (last->next != nullptr) {          last = last->next;      }      last->next = new\_node;      new\_node->prev = last;  }  bool is\_break(Node\* list) {      if (list == nullptr || list->next == nullptr) {return false;}      // Создадим два указателя, которые указывают на два идущих подряд элемента      Node\* element1 = list;      Node\* element2 = list->next;      while (element2 != nullptr && element2->next != nullptr) {          // Если случится так, что предыдущий от element2 элемент указывает не на          // element1, значит, произошел перескок          if (element1 != element2->prev) {              return true;          }          element1 = element1->next;          element2 = element2->next;      }      return false;  }  void display\_list(Node\* list) {      while (list != nullptr) {          std::cout << list->data;          list = list->next;          if (list != nullptr) {              std::cout << " -> ";          }      }  }  int main() {      Node\* break\_list = NULL;      for (int i = 1; i < 6; i++) {          append(&break\_list, i);      }      std::cout << "Список:\n";      display\_list(break\_list);      // Проверим на "сломанность":      std::cout << "\nИзначальный список сломан?: " << is\_break(break\_list) << "\n\n";      // Сделаем так, чтобы 3 элемент указывал вперед не на 4-ый, а на 1-ый      Node\* broke\_element = break\_list->next->next->next;      broke\_element->next = break\_list;      std::cout << "Измененный список сломан?: " << is\_break(break\_list);      return 0;  } |
| ***Ответ:*** | |
|  |  |
| **Задание 2 (0.5 балла)** | |
| ***Задача:*** | |
|  | Напишите функцию, которая принимает односвязный список и разделяет его на два новых списка по следующему критерию:  Первый список должен содержать узлы со значениями меньше заданного числа X.  Второй список - узлы со значениями больше или равными X.  X вводится с клавиатуры. Изначальный порядок элементов в каждом из новых списков должен быть сохранён. Создание новых узлов не требуется, используются только существующие |
| ***Решение:*** | |
|  | #include <iostream>  struct Node {      int data;      Node\* next;  };  void append(Node\*\* head, int new\_data) {      Node\* new\_node = new Node();      Node\* last = \*head;      new\_node->data = new\_data;      new\_node->next = nullptr;      if (\*head == nullptr) {          \*head = new\_node;          return;      }      // Находим последний узел в списке, чтобы после него добавить новый элемент      while (last->next != nullptr) {          last = last->next;      }      last->next = new\_node;  }  void display\_list(Node\* list) {      while (list != nullptr) {          std::cout << list->data;          list = list->next;          if (list != nullptr) {              std::cout << " -> ";          }      }  }  std::pair<Node\*, Node\*> list\_split(Node\* list, int sep) {      // Есть 3 указателя: первый (list) указывает на элемент исходного списка, который мы рассматриваем      // Второй будет указывать на начало списка, в котором все числа меньше sep      // Второй будет указывать на начало списка, в котором все числа больше либо равны sep      Node\* new\_list1 = nullptr;      Node\* new\_list2 = nullptr;      while (list != nullptr) {          // Каждый раз, "рассматривая" новый элемент исходного списка, будем создавать указатель          // на этот элемент для того, чтобы после копирования значения исходного элемента в          // один из новых списков, удалить этот элемент в исходном списке.          Node\* temp\_element = list;          if (list->data < sep) {              append(&new\_list1, list->data);          }          else {              append(&new\_list2, list->data);          }          // Переходим "рассматривать" следующий элемент          list = list->next;          // и удаляем прошлый          delete temp\_element;      }      return {new\_list1, new\_list2};  }  int main() {      Node\* list = NULL;      for (int i = 0; i < 20; i++) {          append(&list, i);      }      std::cout << "Список: " << "\n";      display\_list(list);      int X;      std::cout << "\nВведите разделитель: ";      std::cin >> X;      std::pair<Node\*, Node\*> pair\_of\_lists = list\_split(list, X);      display\_list(pair\_of\_lists.first);      std::cout << std::endl;      display\_list(pair\_of\_lists.second);      return 0;  } |
| ***Ответ:*** | |
|  |  |
| **Задание 3 (0.5 балла)** | |
| ***Задача:*** | |
|  | Реализуйте кольцевой двусвязный список с использованием структуры Node и набора функций. Реализация должна поддерживать следующие операции:  1. Добавление узла в конец кольцевого списка.  2. Удаление узла по значению.  3. Вывод элементов кольца  4. Проверка, является ли список кольцевым.  *Дополнительные условия*   1. Кольцо должно оставаться замкнутым после добавления и удаления узлов. 2. Если удалён последний элемент, список должен корректно обнулиться. |
| ***Решение:*** | |
|  | #include <iostream>  struct Node {      int data;      Node\* next;      Node\* prev;      bool last; // Проверка на то, что элемент - последний  };  void append(Node\*\* head, int new\_data) {      Node\* new\_node = new Node();      new\_node->data = new\_data;      new\_node->last = 1;      if (\*head == nullptr) {          new\_node->prev = nullptr;          new\_node->next = nullptr;          \*head = new\_node;          return;      }      // Находим последний узел в списке, чтобы после него добавить новый элемент      Node\* element = \*head;      element->prev = new\_node;      new\_node->next = element;      while (element->last == 0) {          element = element->next;      }      element->next = new\_node;      element->last = 0;      new\_node->prev = element;  }  void display\_list(Node\* list) {      if (list->last == 1) {          std::cout << list->data;      }      else {          while (list->last == 0) {              std::cout << list->data;              list = list->next;              std::cout << " <-> ";          }          std::cout << list->data << "\n";      }  }  // Удаление узла по значению  void delete\_element(Node\* list, int value) {      // Доходим до нужного узла      Node\* element = list;      while (element->data != value) {          element = element->next;      }      // Меняем поле next предыдущего узла на следующий после найденного элемента узел      element->prev->next = element->next;      if (element->last == 1) {          element->prev->last = 1;      }      delete element;  }  bool is\_circular(Node\* list) {      Node\* first\_element = list;      Node\* last\_element = list;      while (last\_element->last == 0 && last\_element->next) {          last\_element = last\_element->next;      }      if (last\_element->next == first\_element && first\_element->prev == last\_element) {          return true;      }      return false;  }  int main() {      std::cout << "Список:\n";      Node\* list = nullptr;      append(&list,1);      append(&list,2);      append(&list,3);      append(&list,4);      display\_list(list);      std::cout << "Удалим элемент 2:\n";      delete\_element(list, 2);      display\_list(list);      std::cout << "Список цикличен?: ";      std::cout << is\_circular(list);      return 0;  } |
| ***Ответ:*** | |
|  |  |
| **Задание 4\*** | |
| ***Задача:*** | |
|  | Реализуйте кольцевой двусвязный список с расширенными возможностями для обработки данных. Напишите программу, которая выполняет следующие задачи:  1. Создаёт кольцевой двусвязный список.  2. Реализует следующие операции:   * Добавление узла с заданным значением в конец списка. * Вставка нового узла перед узлом с указанным значением. * Удаление узла с заданным значением. * Подсчёт количества узлов в списке. * Вывод списка в прямом и обратном порядке.   *Оформите каждую операцию в отдельную функцию.* |
| ***Решение:*** | |
|  |  |
| ***Ответ:*** | |
|  |  |
| **Задание 5\*** | |
| ***Задача:*** | |
|  | Даны два односвязных списка, где каждый узел содержит одну цифру числа. Цифры могут храниться как в прямом порядке (старшие разряды в начале), так и в обратном (младшие разряды в начале). Нужно найти сумму этих чисел и сохранить результат в виде нового списка (или, по желанию, дополнить один из старых).  Вход (при хранении в перевёрнутом виде):  2 4 3  5 6 4  Выход:  7 0 8  342 + 465 = 807 |
| ***Решение:*** | |
|  |  |
| ***Ответ:*** | |
|  |  |

**2. Дерево. Двоичное дерево поиска**

|  |  |
| --- | --- |
| **Теоретический материал** | |
| **Дерево**  Дерево состоит из элементов, называемых узлами (вершинами). Узлы соединены между собой направленными дугами. В случае X→Y вершина X называется родителем, а Y – потомком.    Дерево имеет единственный узел, не имеющий родителей (указателей на этот узел), который называется ***корнем***. Любой другой узел имеет ровно одного родителя, т.е. на каждый узел дерева имеется ровно один указатель.  Узел, не имеющий потомков, называется ***листом***.  ***Внутренний*** узел – это узел, не являющийся ни листом, ни корнем. ***Порядок узла*** равен количеству его узлов-потомков. ***Степень дерева*** – максимальный порядок его узлов. ***Высота (глубина) узла*** равна числу его родителей плюс один. ***Высота дерева*** – это наибольшая высота его узлов.  **Двоичное дерево**  Двоичное дерево – это иерархическая структура данных, в которой каждый узел имеет не более двух потомков (детей). Как правило, первый называется родительским узлом, а дети называются левым и правым наследниками    Дерево по своей организации является рекурсивной структурой данных, поскольку каждое его поддерево также является деревом. В связи с этим действия с такими структурами чаще всего описы­ваются с помощью рекурсивных алгоритмов.  Если дерево организовано таким образом, что для каждого узла все ключи (значения узлов) его ле­вого поддерева меньше ключа этого узла, а все ключи его правого поддерева – больше, оно называется **двоичным деревом поиска**. Одинаковые ключи здесь не допускаются.  **Двоичное дерево поиска (BST)**  Двоичное дерево поиска (англ. *binary search tree*, BST) — двоичное дерево, для которого выполняются следующие дополнительные условия (*свойства дерева поиска*):   * оба поддерева — левое и правое — являются двоичными деревьями поиска; * у всех узлов *левого* поддерева произвольного узла X значения ключей данных *меньше либо равны*, нежели значение ключа данных самого узла X; * у всех узлов *правого* поддерева произвольного узла X значения ключей данных *больше*, нежели значение ключа данных самого узла X.     Например, чтобы добавить узел со значением «5», процедура будет следующей:   1. Сравниваем 5 и 8. 5 меньше 8, поэтому идем в левое поддерево.      1. Сравниваем 5 и 3. 5 больше, чем 3, поэтому идем в правое поддерево.      1. Сравниваем 5 и 6. 5 меньше, чем 6, поэтому идем в левое поддерево      1. 4 – это лист. Сравниваем 5 и 4. 5 больше, чем 4. Поэтому 5 делаем правым потомком относительно 4. Итоговое дерево имеет вид:     Если в двоичное дерево поиска необходимо добавить данное, которое там уже есть, новый узел не добавляют, а увеличивают значение специальной служебной переменной – количество экземпляров узлов с определенным значением ключа (count)  Структура бинарного дерева построена из узлов. Как и в связанном списке эти узлы содержат поля данных и указатели на другие узлы в коллекции. Узел дерева содержит поле данных и два поля с указателями, которые называются левым и правым указателями. Значение nullptr является признаком пустого поддерева.  Дерево, по сути, является рекурсивной структурой данных. В результате множество операций будут реализованы через рекурсивные функции. У таких функций будет обязательный служебный параметр - адрес узла текущего уровня.  Структура данных, описывающих дерево, имеет вид:  **struct** Node  {  int value; //Значение узла (ключ), данные любого типа  intcount; //Количество экземпляров узла с данным значением (в дереве)  Node \* left; //  Node \* right; //  };  Перечислим основные рекурсивные функции для работы с деревом (названия являются условными):  • addNode() - добавление нового узла в дерево.  • printTree() - обход и печать данных дерева.  • depthTree() - вычисление глубины (высоты) дерева.  • searchNode() - поиск узла в дереве.  • delTree() - удаление дерева.  • delNode() - удаление определенного узла в дереве.  *Добавление нового узла в двоичное дерево поиска*  Наиболее ответственная операция: добавление в дерево нового узла. Суть алгоритма добавления в следующем: мы начинаем работу с корня всего дерева. Если дерево пустое (корень нулевой), то тогда сразу создается новый узел и его адрес возвращается в качестве корня дерева. Если корень не пустой, то по результату сравнения вставляемых данных и данных в узле дерева мы идем либо в левое, либо в правое поддерево. Далее, либо мы достигаем листа и добавляем новый узел в качестве его потомка, либо находим узел, значение данных в котором совпадает с добавляемым значением. В таком случае добавлять узел не надо, а только увеличить счетчик в найденном узле.  *Обход и печать данных дерева*  Существует несколько методов прохождения дерева для доступа к его элементам. К ним относятся **прямой, обратный и симметричный**. При прохождении дерева используется рекурсия, поскольку каждый узел является корнем своего поддерева. Каждый алгоритм выполняет в узле три действия: заходит в узел, рекурсивно спускается по левому и по правому поддереву. Спуск прекращается при достижении пустого поддерева (нулевой указатель).    *Поиск конкретного элемента в дереве*  Известно, что слева от узла располагается элемент, который меньше чем текущий узел. Из чего следует, что если у узла нет левого наследника, то он является минимумом в дереве. Таким образом, можно найти минимальный элемент дерева.  Поиск нужного узла по значению выполняется следующим образом. Если искомое значение больше узла, то продолжаем поиск в правом поддереве, если меньше, то продолжаем в левом. Если узлов уже нет, то элемент не содержится в дереве.  *Удаление узла с определенными данными*  Данная операция – наиболее сложная, поскольку при удалении произвольного узла должна сохраняться упорядоченность оставшихся элементов дерева. Существует 4 возможных ситуации при удалении узла дерева:   1. У удаляемого узла нет потомков. Тогда мы можем освободить память, занимаемую узлом, а у его родителя выставить nullptr в указателе на потомка. 2. Удаляемый узел имеет двух потомков, причем у левого потомка есть свое правое поддерево. В этом случае нужно найти в этом правом поддереве наибольший элемент и вставить его вместо удаляемого узла.      1. Удаляемый узел имеет двух потомков, причем у левого потомка нет правого поддерева. В этом случае элемент заменяется на корень левого поддерева:      1. Удаляемый узел имеет одного потомка (левого или правого). В этом случае мы присваиваем адрес потомка указателю нашего родителя, вместо адреса текущего узла: | |
| **Пример 2** | |
| ***Задача:*** | |
|  | Создать двоичное дерево поиска. Программа должна запрашивать количество элементов дерева, далее значения, хранящиеся в элементах. Для готового дерева реализовать операции:  а) добавление нового узла в дерево;  б) прямой обход дерева. |
| ***Решение:*** | |
|  |  |
| ***Ответ:*** | |
|  |  |
| **Задание 6 (1 балл)** | |
| ***Задача:*** | |
|  | Создать двоичное дерево поиска (в узлах хранятся целые положительные числа). Программа должна запрашивать количество элементов дерева, далее значения, хранящиеся в элементах, создаются генератором случайных чисел. Для готового дерева реализовать операции:  а) добавление нового узла в дерево;  б) обход дерева (прямой, обратный или симметричный – по выбору) и печать элементов дерева на экран;  в) вычисление глубины (высоты) дерева;  г) поиск конкретного элемента в дереве  д) удаление определенного узла в дереве. |
| ***Решение:*** | |
|  | #include <iostream>  // Структура узла двоичного дерева  struct Node {      int data;      Node\* left;      Node\* right;      int count;  };  Node\* createNode(int value) {      Node\* newNode = new Node;      newNode->left = nullptr;      newNode->right = nullptr;      newNode->data = value;      newNode->count = 1;      return newNode;  }  // Функция добавления узла в дерево  void insertNode (Node\*\* root, int value) {      if (!\*root) {          \*root = createNode(value);          return;      }      if (value < (\*root)->data) {          insertNode(&((\*root)->left), value);          return;      }      else if (value > (\*root)->data) {          insertNode(&((\*root)->right), value);          return;      }      else {(\*root)->count++;} // Если data проверяемого узла уже равна value, то увеличиваем count на 1  }  void preorder(Node\* root) {      if (root) {          std::cout << root->data << " ";          preorder(root->left);          preorder(root->right);      }  }  int height(Node\* tree, int c) {      // Если корень поддерева существует, то функция возвращает максимум из      // высоты левого и правого поддеревьев      if (tree) {          return std::max(height(tree->left, c + 1), height(tree->right, c + 1));      }      // Если поддерева нет, то функция возвращает высоту предыдущего поддерева      else {return c - 1;}  }  Node\* search(Node\* tree, int value) {      if (tree) {          // Если элемент больше данного узла, то проверяем его наличие той же функцией, но уже          // в правом поддереве          if (value > tree->data) {              return search(tree->right, value);          }          else if (value < tree->data) {              return search(tree->left, value);          }          else {              return tree;          }      }      // Если так вышло, что при таком обходе дерева мы дошли до несуществующего узла,      // значит, такого элемента нет      else {          return nullptr;      }  }  // Поиск максимального узла в дереве  Node\* max\_node(Node\* tree) {      if (!tree->left && !tree->right) {return tree;}      else if (tree->left && !tree->right) {return max\_node(tree->left);}      else if (!tree->left && tree->right) {return max\_node(tree->right);}      else {          Node\* max\_left = max\_node(tree->left);          Node\* max\_right = max\_node(tree->right);          if (max\_left->data > max\_right->data)  {              return max\_left;          }          else {              return max\_right;          }      }  }  void delete\_node(Node\* tree, int value) {      if (!tree) {          std::cout << "Пустое дерево";      }      else if (!search(tree, value)){          std::cout << "Элемента нет в дереве";      }      else {          // Дойдем до искомого узла:          Node\* node = tree;          Node\* parent;          while (node->data != value) {              if (value > node->data) {                  parent = node;                  node = node->right;              }              else {                  parent = node;                  node = node->left;              }          }          // Проверим 3 случая:          // 1) У узла нет потомков          if (node->left == nullptr && node->right == nullptr) {              if (parent->right == node) {parent->right = nullptr;}              else {parent->left = nullptr;}              delete node;          }          // 2) У узла есть 2 потомка          else if (node->left != nullptr && node->right != nullptr) {              // 2.1) У левого потомка есть правое поддерево              if (node->left->right) {                  // В этом случае нужно найти в этом правом поддереве                  // наибольший элемент и вставить его вместо удаляемого узла.                  Node\* max = max\_node(node->left);                  node->data = max->data;                  delete max;              }              // 2.2) У левого потомка нет правого поддерева              else {                  // В этом случае элемент заменяется на корень левого поддерева:                  // Создаются локальные указатели на потомков (левый, чтобы                  // удалить узел, на который он указывает, после копирования                  // значения в "удаляемый" элемент; правый, чтобы в новое                  // скопированное значение вставить правого предка "удаляемого" элемента)                  Node\* newNode\_left = node->left;                  Node\* newNode\_right = node->right;                  \*node = \*(node->left);                  node->right = newNode\_right;                  delete newNode\_left;              }          }          // 3) У узла один потомок          else {              if (node->right) {                  Node\* newNode = node->right;                  \*node = \*(node->right);                  delete newNode;              }              else {                  Node\* newNode = node->left;                  \*node = \*(node->left);                  delete newNode;              }          }      }  }  int main() {      // Создадим дерево      std::cout << "Дерево:\n";      Node\* tree = nullptr;      insertNode(&tree, 4);      insertNode(&tree, 3);      insertNode(&tree, 2);      insertNode(&tree, 1);      insertNode(&tree, 6);      insertNode(&tree, 5);      insertNode(&tree, 7);      preorder(tree);      std::cout << std::endl;      std::cout << "Высота дерева: " << height(tree, 1) << "\n";      std::cout << "Узел с нормером 5: " << search(tree, 5) << " - " << search(tree, 5)->data << "\n";      std::cout << "Узел с нормером 7: " << search(tree, 5) << " - " << search(tree, 7)->data << "\n";      // Удалим элементы дерева      std::cout << "Удалим элемент 6:\n";      delete\_node(tree, 6);      preorder(tree);      delete\_node(tree, 3);      std::cout << "\n";      std::cout << "Удалим элемент 3:\n";      preorder(tree);      return 0;  } |
| ***Ответ:*** | |
|  |  |
| **Задание 7\*** | |
| ***Задача:*** | |
|  | На основе двоичного дерева поиска реализовать консольное приложение «Телефонная книга». Двоичное дерево поиска в данном случае – это хранилище записей (имя человека, его телефон) с операциями поиска и удаления записей по имени человека и операцией добавления новой записи. При этом у одного и того же человека может быть несколько номеров телефона. |
| ***Решение:*** | |
|  |  |
| ***Ответ:*** | |
|  |  |

**3. Дерево отрезков**

|  |  |
| --- | --- |
| **Теоретический материал** | |
| Операцию называют определённой на множестве если для любых элементов, входящих в него результат операции тоже принадлежит данному множеству. Например, операция сложения «+» определена на множестве N (натуральные числа), а операция деления «/» - нет, так как ее результат не будет элементом множества N.  Операцию (обозначим «\*») будем называть **ассоциативной** на множестве M, если для любых элементов a, b, c из множества M верно равенство **(a \* b) \* c = a \* (b \* c).**  **Нейтральным элементом** операции — элемент, который оставляет любой другой элемент неизменным при применении этой бинарной операции к этим двум элементам. Например, число 0 является нейтральным элементом для операции сложения на множестве целых чисел так, как **a+0=a**.  **Дерево отрезков (дерево интервалов, англ. Segment tree)** — это структура данных, которая позволяет за асимптотику O(log n) реализовать операции, которые определены и ассоциативны на данном множестве и для которых существует нейтральный элемент.  **Применимость**  Дерево отрезков позволяет решать достаточно широкий спектр задач, вот только несколько из них: суммирование на множестве натуральных чисел, поиск минимума на любом числовом множестве, перемножение матриц на множестве матриц размера N∗N, поиск максимальной последовательности возрастающих чисел, подсчет количества и суммы делителей, нахождение наибольшего общего делителя / наименьшего общего кратного, также в вычислительной геометрии дерево отрезков часто используется при решении задач о пересечении прямоугольников.  **Построение**  Дерево отрезков - это двоичное дерево. Для заданных целых чисел l и r таких, что l<r, дерево отрезков T(l, r) строится рекурсивно по следующим правилам: оно состоит из корня v с параметрами B[v]=l и E[v]=r (B и E мнемонически соответствуют словам "Beginning" (начало) и "End" (конец), а если r-l>1, то оно состоит из левого поддерева T(l, (B[v]+E[v])/2) и правого поддерева T((B[v]+E[v]) /2),r). Параметры B[v] и E[v] обозначают интервал [B[v], E[v]], включенный в [l, r], связанный с узлом v.  Пример дерева отрезков на интервале (1,6):    Интервалы, принадлежащие множеству**{[B[v],E[v]]: v - узел T(l, r) },** называются **стандартными интервалами** дерева **T(l, r).**  Стандартные интервалы, принадлежащие листьям T(l,r), называются **элементарными интервалами.** Строго говоря, интервал, связанный с v, это полуоткрытый интервал [B[v], E[v]), за исключением узлов самого правого пути в T(l, r), чьи интервалы замкнуты.  Структура данных, описывающих дерево отрезков, имеет вид:  struct node  {  int KeyMin; // Минимальный ключ вершины.  int KeyMax; // Максимальный ключ вершины.  node \*Left; // Указатель на левое поддерево.  node \*Right; // Указатель на правое поддерево.  };  Перечислим основные функции для работы с деревом отрезков (названия являются условными):  • buildTree() – построение дерева.  • printTree() - обход и печать данных дерева.  • searchX() - подсчет количества интервалов дерева содержащих точку X.  • cleanTree() - удаление дерева.  • searchSum() – поиск суммы элементов на отрезке (a,b).  • delNode() - удаление определенного узла в дереве. | |
| **3адание 8 (1 балл)** | |
| ***Задача:*** | |
|  | Создать дерево отрезков (в узлах хранятся интервалы). Программа должна запрашивать l и r начало и конец интервала соответственно. Для готового дерева реализовать операции:  а) рекурсивное построение дерева отрезков;  б) обход дерева (прямой, обратный или симметричный – по выбору) и печать элементов дерева на экран;  в) подсчет количества интервалов дерева, содержащих точку X. |
| ***Решение:*** | |
|  | #include <iostream>  struct Node {      int KeyMin; // Минимальный ключ вершины.      int KeyMax; // Максимальный ключ вершины.      Node \*Left; // Указатель на левое поддерево.      Node \*Right; // Указатель на правое поддерево.  };  Node\* buildTree(int B, int E) {      Node\* newNode = new Node;      newNode->KeyMax = std::max(B, E);      newNode->KeyMin = std::min(B, E);      if ((newNode->KeyMax - newNode->KeyMin) > 1) {          newNode->Left = buildTree(newNode->KeyMin, (B + E) / 2);          newNode->Right = buildTree((B + E) / 2, newNode->KeyMax);      }      else {          newNode->Left = nullptr;          newNode->Right = nullptr;      }      return newNode;  }  int count(Node\* tree, int X) {      int c = 0;      if (tree && tree->KeyMax > X && tree->KeyMin < X) {          c++;          c += count(tree->Left, X);          c += count(tree->Right, X);      }      return c;  }  void display(Node\* root) {      if (root) {          std::cout << "(" << root->KeyMin << ";" << root->KeyMax << ") ";          display(root->Left);          display(root->Right);      }  }  int main() {      int b, r, X;      std::cout << "Введите начало и конец интервала: ";      std::cin >> b >> r;      std::cout << "Построим дерево:\n";      Node\* tree = buildTree(b, r);      display(tree);      std::cout << "\nВведите точку X, которую нужно подсчитать: ";      std::cin >> X;      std::cout << "Кол-во точек X: ";      std::cout << count(tree, X);      return 0;  } |
| ***Ответ:*** | |
|  |  |
| **3адание 9\*** | |
| ***Задача:*** | |
| Создать дерево отрезков (в узлах хранятся интервалы). Программа должна запрашивать l и r начало и конец интервала соответственно. Для готового дерева реализовать дополнительно операции:  а) подсчет суммы элементов на отрезке (a,b);  б) нахождение минимального элемента на отрезке (a,b);  в) поиск максимального элемента на отрезке (a,b);  г) подсчёт количества чётных и нечётных элементов на отрезке (a,b). | |
| ***Решение:*** | |
|  | |
| ***Ответ:*** | |
|  | |

1. **Ортогональные деревья**

|  |
| --- |
| **Теоретический материал** |
| Ортогональные деревья (range tree, дерево диапазона) - это структура данных, которая используется для представления иерархических отношений между элементами, где каждый узел дерева может иметь произвольное количество дочерних узлов. Основная идея ортогональных деревьев заключается в том, что они обеспечивают эффективное хранение и обработку данных, особенно в контексте задач, связанных с графами и сетями.  An example of a 1-dimensional range tree.  Пример одномерного дерева диапазона  Представим, что у нас есть набор точек (x,y). Чтобы быстро находить точки в каком-то диапазоне, мы строим два уровня структур:  1. Основное дерево по оси x  • Считаем, что каждая точка имеет ключ - свою координату x.  • Строим двоичное дерево поиска, где в узлах хранятся точки (или их ссылки), упорядоченные по возрастанию x.  2. В каждом узле - дополнительная структура по оси y  • Помимо «значения x», в узле хранится список или дерево тех же точек, но уже отсортированных по y. Сюда попадают все точки, находящиеся в поддереве данного узла (по x).  • Другими словами, у каждого узла есть «второе дерево» (или упорядоченный список), которое отвечает за координату y.  Поиск точек в диапазоне [x1..x2]x[y1..y2]  1. На уровне x находим все узлы, которые перекрывают диапазон [x1..x2]. Для этого в двоичном дереве (по x) делаем обычный «диапазонный» поиск.  2. На уровне y для каждого подходящего узла смотрим в его «второе дерево (или список)», чтобы отобрать точки, у которых y лежит в [y1..y2].  3. Объединяем результаты и получаем список всех искомых точек в прямоугольнике.  Таким образом, мы сужаем поиск сначала по x, а затем по y, что обычно гораздо быстрее, чем «перебирать» все точки подряд. |

|  |  |
| --- | --- |
| **Пример 3** | |
| ***Задача:*** | |
|  | Дано множество точек в двумерном пространстве P , состоящее из n точек, каждая из которых задана координатами (xᵢ, yᵢ). Напишите алгоритм, который принимает запрос в виде прямоугольника, заданного двумя углами (xₘᵢₙ, yₘᵢₙ) и (xₘₐₓ, yₘₐₓ), и возвращает все точки из P , которые лежат внутри этого прямоугольника |
| ***Решение:*** | |
|  |  |
| ***Ответ:*** | |
|  |  |

|  |  |
| --- | --- |
| **3адание 10\*\*\*** | |
| ***Задача:*** | |
|  | Дано множество точек в двумерном пространстве P и запрос в виде прямоугольника, заданного двумя углами (xₘᵢₙ, yₘᵢₙ) и (xₘₐₓ, yₘₐₓ). Напишите программу, который возвращает количество точек из P, лежащих внутри этого прямоугольника. Для реализации используйте range tree. |
| ***Решение:*** | |
|  |  |
| ***Ответ:*** | |
|  |  |