Вопросы по второму модулю.

1. Опишите хеш-функцию целочисленных ключей, реализованную методом деления. Какие значения параметров приемлемы?
   h(k) = k mod M, M - capacity
   M = 2^k — значение кэша не зависит от старших байтов.
   M = 2^8 - 1 — значение кэша не зависит от перестановки байтов.
   В иделе M — простое число.
2. Опишите хеш-функцию целочисленных ключей, реализованную методом умножения. Как вычислять ее значения для целочисленных ключей без использования операций над числами с правающей точкой?
   h(k) = [M * {k * A}]
   Пусть M - степень двойки, тогда h(k) = (k * s mod 2^32) >> (32 - p)
3. Опишите классичискую хеш-функцию для строк и метод Горнера для ее реализации. Какие значения параметров приемлемы?
   h(s) = [s0 + s1 * a + s2 * a^2 + … + s(n-1) * a^(n-1)] mod M, где M - степень двойки
   h(s) = [s0 * a^(n-1) + s1 * a^(n-2) + … + s(n-2) * a + s(n-1)] mod M, где M - степень двойки
   Необходимо, чтобы a и M были взаимнопростыми.
   Метод Горнера:
   ```
   int firstHash(std::string &str, int M) {
           int hash = 0;
           unsigned long size = str.size();
           for (int i = 0; i < size; i++)
       hash = (hash * FIRST_COEF + str[i]) % M;
           return hash;
   };
   ```
4. Опишите операции поиска, добавления и удаления ключей в хеш-таблицу, реализованную методом цепочек.

```
class LinkedHashEntry {
private:
    int key;
    int value;
    LinkedHashEntry *next;
public:
    LinkedHashEntry(int key, int value) {
        this->key = key;
        this->value = value;
        this->next = NULL;
    }

    int getKey() {
        return key;
    }

    int getValue() {
        return value;
    }

    void setValue(int value) {
        this->value = value;
    }
```

```
    LinkedHashEntry *getNext() {
        return next;
    }

    void setNext(LinkedHashEntry *next) {
        this->next = next;
    }


};
```

5. Оцените среднее время работы операции поиска в хеш-таблице, реализованной методом цепочек.

   Среднее время работы операций поиска, вставки (с проверкой на дубликаты) и удаления в хэш-таблице, реализованной методом цепочек - O(1 + a), где a — коэффициент заполнения таблицы.

6. Опишите операции поиска, добавления и удаления ключа в хеш-таблицу, реализованную методом открытой адресации.

```
bool addElem(std::string str) {

   if ((double) elemCount / capacity >= RESIZE_COEF)
      reHash();

   int hashVal = firstHash(str, capacity);
   int hashStep = secondHash(str, capacity);
   int check = 0;

   if (hasElem(str)) { return false; }
   else {
      while (!table[hashVal].empty && check < capacity) {
         hashVal = (hashVal + hashStep) % capacity;
         check++;
      }

      if (table[hashVal].deleted || table[hashVal].empty) {
         table[hashVal].value = str;
         table[hashVal].empty = false;
         table[hashVal].deleted = false;
      }
      elemCount++;
      return true;
   }
};
```

```
bool delElem(std::string str) {
    int hashVal = firstHash(str, capacity);
    int hashStep = secondHash(str, capacity);
    int check = 0;

    while (table[hashVal].value != "" && check < capacity) {
        if (table[hashVal].value == str && !table[hashVal].deleted){
            table[hashVal].deleted = true;
            return true;
        }
        hashVal = (hashVal + hashStep) % capacity;
        check++;
    }
    return false;
};

bool hasElem(std::string str) {
    int hashVal = firstHash(str, capacity);
    int hashStep = secondHash(str, capacity);
    int check = 0;

    while (table[hashVal].value != "" && check < capacity) {
        if (table[hashVal].value == str && !table[hashVal].deleted)
            return true;
        hashVal = (hashVal + hashStep) % capacity;
        check++;
    }
    return false;
};
```

7. Опишите способ квадратичного пробирования.

```
            while (mas_cap > i_prob) {
    if (!str_mas[key].deleted && !str_mas[key].empty && str_mas[key].val == val) {
        return false;
    }
    if (str_mas[key].empty) {
        str_mas.insert(
            Hash_val(val),
            key
        );

        return true;
    } else {
        key = (key + i_prob + 1) % str_mas.get_capacity();
        i_prob += 1;
    }
}
```

8. Опишите способ пробирования методом двойного хеширования.

```
        bool addElem(std::string str) {

    if ((double) elemCount / capacity >= RESIZE_COEF)
        reHash();

    int hashVal = firstHash(str, capacity);
    int hashStep = secondHash(str, capacity);
    int check = 0;

    if (hasElem(str)) { return false; }
    else {
        while (!table[hashVal].empty && check < capacity) {
            hashVal = (hashVal + hashStep) % capacity;
            check++;
        }

        if (table[hashVal].deleted || table[hashVal].empty) {
            table[hashVal].value = str;
            table[hashVal].empty = false;
            table[hashVal].deleted = false;
        }
        elemCount++;
        return true;
    }
};
```

9. Преимущества и недостатки метода цепочек по сравнению с методом открытой адресации.

   Плюсы:

   — Открытая адресация не тратит время на хранение указателей списка.

   — Нет элементов, хранящихся вне таблицы.

   Минусы:

   — Хэш-таблица может оказаться заполненной. Коэффициент заполнения “альфа” не может быть больше 1.

   — При приближении коэффициента заполнения “альфа” к 1 среднее время работы поиска, добавления и удаления стремится к N.

   — Сложное удаление.

10. Опишите процесс динамического изменения размера хеш-таблицы.

```
        void reHash() {
    int newCapacity = capacity * 2;
    Elem *newTable = new Elem[newCapacity];
    for (int i = 0; i < capacity; i++) {
        if (!table[i].empty) {
            if (!table[i].deleted) {
                int hashVal = firstHash(table[i].value, newCapacity);
                int hashStep = secondHash(table[i].value, newCapacity);
                int check = 0;

                while (!newTable[hashVal].empty && check < newCapacity) {
```

```
                hashVal = (hashVal + hashStep) % newCapacity;
                ++check;
            }
            newTable[hashVal] = table[i];
        }
      }
    }
    table = newTable;
    capacity = newCapacity;
};
```

11. Опишите обход двоичного дерева в глубину (pre-order, post-order, in-order).

```
void preOrder(Node* node)
if (node != nullptr) {
        std::cout << node->value << "  ";
        preOrder(node->left);
        preOrder(node->right);
}

void postOrder(Node* node)
if (node != nullptr) {
        postOrder(node->left);
        postOrder(node->right);
        cout << node->value << " ";
}

void inOrder(Node* node)
if (node != nullptr) {
        inOrder(node->left);
        cout << node->value << " ";
        inOrder(node->right);
}
```

12. Опишите обход двоичного дерева в ширину.

```
void Traverse(Node* root)
if (root == nullptr)
        return;
queue<Node*> q;
q.put(root);
while (!q.empty())
        Node* node = q.pop()
        visit(node);
        if (node->left != nullptr)
                q.push(node->left)
        if (node->right != nullptr)
                q.push(node->right)
```

13. Опишите операцию поиска вершины с минимальным (максимальным) ключом в двоичном дереве поиска.

```
Node* minimum(Node* node)
while (node != nullptr)
        node = node->left;
```

return node;

14. Опишите наивный способ добавления элемента в двоичное дерево поиска.

```
void insert(Node* node, int value)
if (node == nullptr)
        node = new Node(value)
        return;
if (node->data > value)
        insert(node->left, value)
else
        insert(node->right, value)
```

15. Опишите наивный способ удаления элемента из двоичного дерева поиска.

```
bool delete(Node* &node, int value)
        if (node == 0)
                return false
        if (node->data == value)
                deleteNode(node)
                return true
        return delete(node->data > value ? node->left : node->right, value)

void deleteNode(Node* &node)
        if (node->left == 0)
                Node* right = node->right;
                delete node;
                node = right;
        else if (node->right == 0)
                Node* left = node->left;
                delete node;
                node = left;
        else
                Mode* parent = node;
                Node* min = node->right;
                while(min->left != 0)
                        parent = min
                        min = min->left
                node->data = min->data
                (parent->left == min ? parent->left : parent->right) = min->right
                delete min;
```

16. Опишите операцию Split в декартовом дереве.

```
void split(Node* current, int key, Node* &left, Node* &right)
        if (current == 0)
                left = 0
                right = 0
        else if (current->key <= key)
                split(current->right, key, current->right, right)
                left = current
        else
                split(current->left, key, left, current->left)
                right = current
```

17. Опишите операцию Merge в декартовом дереве.

```
void merge(Node* left, Node* right)
        if (left == 0 || right == 0)
                return left == 0 ? Right : left
        if (left->priority > right->priority)
                left->RightChild = merge(left->RightChild, right)
                return left
        else
                right->LeftChild = merge(left, right->Left)
                return right
```

18. Опишите операцию вставки в декартово дерево без помощи слияния.

```
void insert(tree_node &*root, tree_node *vertex) {
   if (root == nullptr) {
      root = vertex;
      return;
   }
   if (root->priority > vertex->priority) {
      if (vertex->key < root->key)
         insert(root->left, vertex);
      else
         insert(root->right, vertex);
      return;

   }
   split(root, vertex->left, vertex->right);
   root = vertex;
}
```

19. Опишите операцию удаления из декартова дерева без помощи разрезания.

```
void erase(tree_node &*root, int key) {
        assert(root != nullptr);
        if (key < root->key)
                erase(root->left, key);
        else if (key > root->key)
                erase(root->right, key);
        else
                root = merge(root->left, root->right);
}
```

20. Опишите операцию добавления вершины в АВЛ-дерево.

```
        node* insert(node* p, int k) // вставка ключа k в дерево с корнем p
                {
                if( !p ) return new node(k);
                if( k<p->key )
                        p->left = insert(p->left,k);
                else
                        p->right = insert(p->right,k);
                return balance(p);
                }
////////////////////////////////////////////////////////////
        node* balance(node* p) // балансировка узла p
        {
                fixheight(p);
                if( bfactor(p)==2 )
                {
```

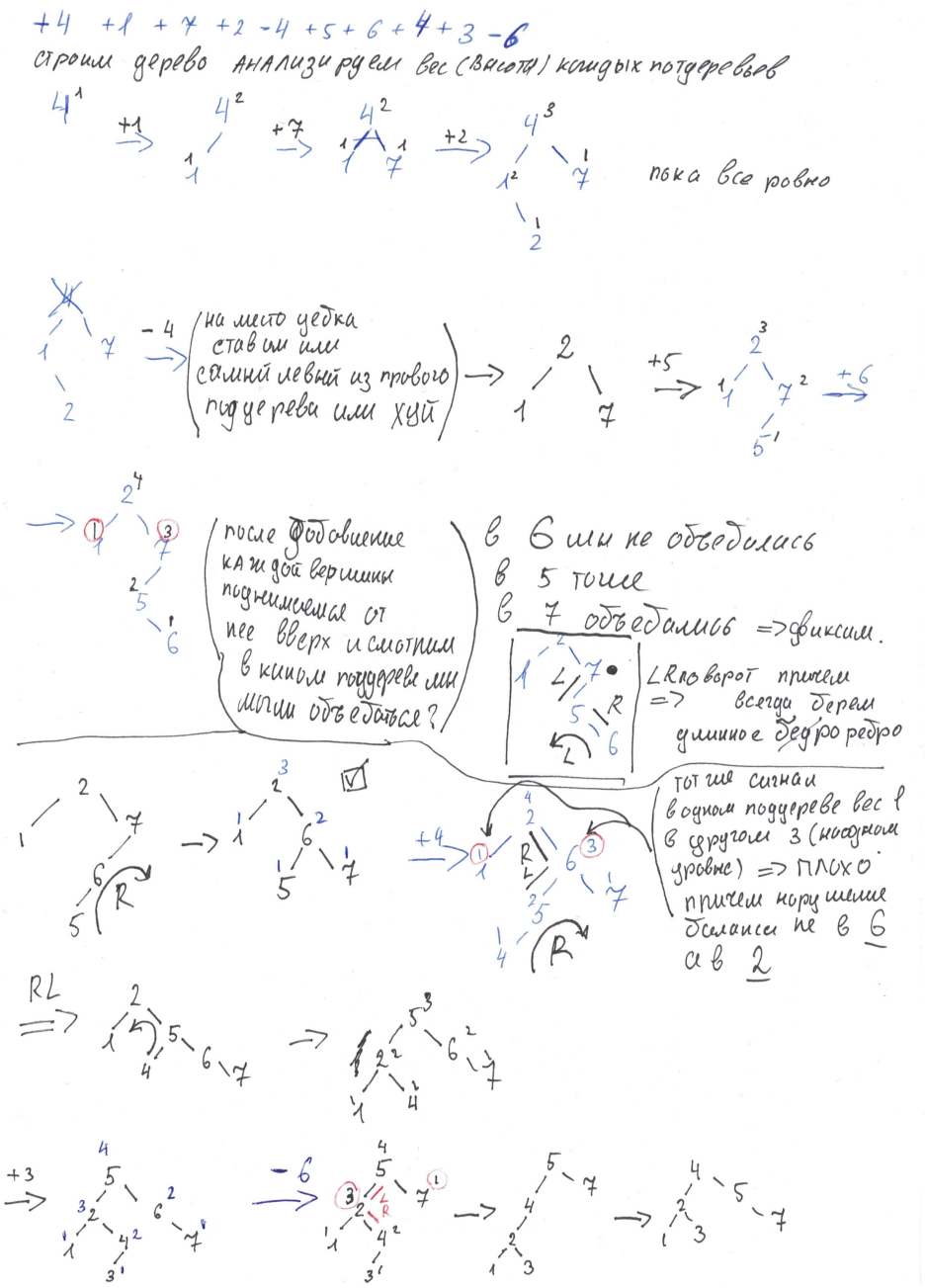
```
                if( bfactor(p->right) < 0 )
                        p->right = rotateright(p->right);
                return rotateleft(p);
        }
        if( bfactor(p)==-2 )
        {
                if( bfactor(p->left) > 0  )
                        p->left = rotateleft(p->left);
                return rotateright(p);
        }
        return p; // балансировка не нужна
}
//////////////////////////////////////////////////////////
int bfactor(node* p) //вычисляет balance factor заданного узла
{
        return height(p->right)-height(p->left);
}
//////////////////////////////////////////////////////////
unsigned char height(node* p) //вычисляет высоту
{
        return p?p->height:0;
}
//////////////////////////////////////////////////////////
node* rotateright(node* p) // правый поворот вокруг p
{
        node* q = p->left;
        p->left = q->right;
        q->right = p;
        fixheight(p);
        fixheight(q);
        return q;
}
//////////////////////////////////////////////////////////
node* rotateleft(node* q) // левый поворот вокруг q
{
        node* p = q->right;
        q->right = p->left;
        p->left = q;
        fixheight(q);
        fixheight(p);
        return p;
}
```

21. Опишите операцию удаления вершины из АВЛ-дерева.

```
node* remove(node* p, int k) // удаление ключа k из дерева p
{
        if( !p ) return 0;
        if( k < p->key )
                p->left = remove(p->left,k);
        else if( k > p->key )
                p->right = remove(p->right,k);
        else //  k == p->key
        {
                node* q = p->left;
                node* r = p->right;
                delete p;
                if( !r ) return q;
                node* min = findmin(r);
```

```
                min->right = removemin(r);
                min->left = q;
                return balance(min);
            }
            return balance(p);
        }
```

22. Постройте АВЛ-дерево по следующей последовательности команд: +4, +1, +7, + 2, -4, +5, +6, +4, +3, -6. Команда +k означает добавление узла с ключом k, команда -k означает удаление узла с ключом k.



23. Опишите операцию добавления вершины в красно-черное дерево.
24. Опишите операцию удаления вершины из красно-черного дерева.
25. Постройте красно-черное дерево по следующей последовательности команд: +4, +1, +7,

+ 2, +4, +5, +6, +4, +3, +6. Команда +k означает добавление узла с ключом k.

26. Опишите АТД “Ассоциативный массив”. Сравните время работы его операций в реализациях с помощью хеш-таблицы и деревом поиска.