Требования к программам

1. В программе должны быть реализованы следующие структуры данных:

   - Контейнер данных объектов типа `student`:

```
enum class io_status
{
  success,
  eof,
  format,
  memory,
  open,
  create,
};
class student
  {
  private:
    char * name  = nullptr;
    int    value = 0;
  public:
    student () = default;
    ...
    io_status read (FILE * fp = stdin);
    void print (FILE * fp = stdout) const;
    int operator< (const student& b) const;
  };
```

   - B-дерево

```
template <class T> class b_tree_node;
template <class T> class b_tree;
template <class T>
class b_tree_node
  {
    T* values = nullptr;               // 2*m elements array
    int size = 0;                      // number of children
    b_tree_node** children = nullptr; // 2*m+1 elements array
  public:
    b_tree_node () = default;
    b_tree_node (const b_tree_node& x) = delete;
    b_tree_node (b_tree_node&& x)
      {
        values = x.values;
        children = x.children;
        size = x.size;

        x.erase_links ();
      }
    b_tree_node& operator= (const b_tree_node& x) = delete;
    b_tree_node& operator= (b_tree_node&& x)
      {
```

```
      if (this == &x)
        return *this;
      delete_node ();

      values = x.values;
      children = x.children;
      size = x.size;

      x.erase_links ();
      return *this;
    }
  ~b_tree_node ()
    {
      delete_node ();
    }
  void print (int p = 0)
    {
      for (int i = 0; i < size; ++i)
        {
          for (int j = 0; j < p; j++)
            printf("  ");
          printf ("values[%2d] ", i + 1);
          values[i].print ();
        }
    }
  // Look for position for insert
  int bin_search (const T& x) const
    {
      int l = 0, r = size, m;
      while (l != r)
        {
          m = (l + r) / 2;
          if (values[m] < x)
            l = m + 1;
          else
            r = m;
        }
      return l;
    }

  friend class b_tree<T>;

private:
  void erase_links ()
    {
      values   = nullptr;
      children = nullptr;
      size     = 0;
    }
  void delete_node ()
```

```
      {
        if (values != nullptr)
          delete[] values; // destructor for each value[i]
        if (children != nullptr)
          // each children[i] is T* - standard type
          delete[] children;
        erase_links ();
      }
    io_status init (int m)
      {
        // default constructor for T is called
        values = new T[2 * m];
        if (values == nullptr)
          return io_status::memory;
        children = new b_tree_node*[2 * m + 1];
        if (children == nullptr)
          {
            delete[] values;
            values = nullptr;
            return io_status::memory;
          }
        for (int i = 0; i < 2 * m + 1; ++i)
          // there is no default constructor for T*
          children[i] = nullptr;
        size = 0;
        return io_status::success;
      }
    // Insert value 'x' with children 'down' at position 'index'
    // into the node (assuming enough space)
    void add_value (T& x, b_tree_node<T>* down, int index)
      {
        for (int i = size; i > index; i--)
          {
            values[i] = static_cast<T&&>(values[i - 1]);
            children[i + 1] = children[i];
          }
        values[index] = static_cast<T&&>(x);
        children[index + 1] = down;
        size++;
      }
};

template <class T>
class b_tree
  {
  private:
    int m = 0; // B-tree base
    b_tree_node<T> * root = nullptr;
  public:
    b_tree (int i = 0) { m = i; }
```

```
  ...
  io_status read (FILE * fp = stdin)
    { ... }
  void print (int r, FILE * fp = stdout) const
    {
      print_subtree (root, 0, r, fp);
    }
  ~b_tree()
    {
      delete_subtree (root);
      erase_links ();
    }
private:
 void erase_links ()
    {
      m    = 0;
      root = nullptr;
    }
  static void delete_subtree (b_tree_node<T> * curr)
    {
      if (curr == nullptr)
        return;
      for (int i = 0; i <= curr->size; i++)
        delete_subtree (curr->children[i]);
      delete curr;
    }
  static void print_subtree (b_tree_node<T> * curr,
    int level, int r, FILE * fp = stdout) const
    {
      if (curr == nullptr || level > r)
        return;
      curr->print (level, fp);
      for (int i = 0; i <= curr->size; i++)
        {
          if (curr->children[i] && level + 1 <= r)
            {
              for (int j = 0; j < level; j++)
                fprintf (fp, "  ");
              fprintf (fp, "children[%2d]\n", i);
              print_subtree (curr->children[i], level + 1, r, fp);
            }
        }
    }
  // Add element x to tree
  io_status add_value (T& x)
    {
      if (root == nullptr)
        {
          root = new b_tree_node<T>();
          if (root == nullptr)
```

```
            return io_status::memory;
          if (root->init(m) != io_status::success)
            {
              delete root;
              return io_status::memory;
            }
          root->values[0] = static_cast<T&&>(x);
          root->size = 1;
          return io_status::success;
        }

      b_tree_node<T> *curr = root, *down = nullptr;
      io_status r = add_value_subtree (curr, down, x, m);

      if (r == io_status::memory)
        return io_status::memory;
      if (r == io_status::success)
        return io_status::success;
      // иначе r == io_status::create и был создан новый узел
      // который возвращается в 'down'
      // Создаем новый корень с одним значением 'x'
      // и двумя потомками 'curr' and 'down'

      b_tree_node<T>* p = new b_tree_node<T>();
      if (p == nullptr)
        return io_status::memory;
      if (p->init (m) != io_status::success)
        {
          delete p;
          return io_status::memory;
        }

      p->values[0] = static_cast<T&&>(x);
      p->children[0] = curr;
      p->children[1] = down;
      p->size = 1;

      root = p;

      return io_status::success;
    }
    // Insert value 'x' with children 'down' at subtree 'curr'
    static io_status add_value_subtree
      (b_tree_node<T>*& curr, b_tree_node<T>*& down, T& x, int m)
      {
        int index = curr->bin_search (x);
        b_tree_node<T>* p = curr->children[index];

        if (p != nullptr)
          { // Есть потомок
```

```
    io_status r = add_value_subtree (p, down, x, m);
    if (r == io_status::memory)
        return io_status::memory;
    if (r == io_status::success)
        return io_status::success;
    // иначе r == io_status::create
    // и был создан новый узел,
    // который возвращается в 'down'
  }
else
    down = nullptr;

if (curr->size < 2 * m)
  { // Есть свободное место в текущем узле
    curr->add_value (x, down, index);
    return io_status::success;
  }
else
  { // Создаем новый узел
    b_tree_node<T>* p = new b_tree_node<T>();
    if (p == nullptr)
      return io_status::memory;
    if (p->init (m) != io_status::success)
      {
        delete p;
        return io_status::memory;
      }

    if (index == m)
      { // 'x' ровно посередине.
        // Перемещаем вторую половину в новый узел
        for (int i = 1; i <= m; i++)
          {
            p->values[i - 1]
              = static_cast<T&&>(curr->values[i + m - 1]);
            p->children[i] = curr->children[i + m];
            curr->children[i + m] = nullptr;
          }
        p->children[0] = down; // меньше всех
    }

    if (index < m)
      { // 'x' в первой половине.
        // Перемещаем вторую половину в новый узел
        for (int i = 0; i < m; i++)
          {
            p->values[i]
              = static_cast<T&&>(curr->values[i + m]);
            p->children[i] = curr->children[i + m];
            curr->children[i + m] = nullptr;
```

```
      }
    p->children[m] = curr->children[2 * m];
    curr->children[2 * m] = nullptr;
    // сдвигаем элементы вправо
    // и вставляем 'x' в позицию 'index'
    for (int i = m; i > index; i--)
      {
        curr->values[i]
          = static_cast<T&&> (curr->values[i - 1]);
        curr->children[i + 1] = curr->children[i];
      }
    curr->children[index + 1] = down;
    curr->values[index] = static_cast<T&&>(x);
    x = static_cast<T&&> (curr->values[m]);
    // новый 'x' - максимальный
  }

if (index > m)
  { // 'x' во второй половине.
    // Перемещаем вторую половину до 'index'
    // в новый узел
    p->children[0] = curr->children[m + 1];
    curr->children[m + 1] = nullptr;
    for (int i = 1 ; i < index - m; i++)
      {
        p->values[i - 1]
          = static_cast<T&&> (curr->values[i + m]);
        p->children[i] = curr->children[i + m + 1];
        curr->children[i + m + 1] = nullptr;
      }
    // Вставляем 'x' в нужную позицию
    p->values[index - m - 1] = static_cast<T&&> (x);
    p->children[index - m] = down;
    // Перемещаем остаток второй половины в новый узел
    for (int i = index - m + 1; i <= m; i++)
      {
        p->values[i - 1]
          = static_cast<T&&> (curr->values[i + m - 1]);
        p->children[i] = curr->children[i + m];
        curr->children[i + m] = nullptr;
      }
    x = static_cast<T&&> (curr->values[m]);
    // новый 'x' - максимальный
  }

down = p;
p->size = m;
curr->size = m;
return io_status::create;
// создан новый узел, он возвращается в 'down'
```

```
        }

        return io_status::success;
    }
};
```

2. Все функции в задании являются членами класса "дерево".

3. Программа должна получать все параметры в качестве аргументов командной строки. Аргументы командной строки:

   1) m – порядок B-дерева,
   2) r – максимальное количество выводимых уровней в дереве,
   3) k – параметр задачи,
   4) `filename` – имя файла, откуда надо прочитать дерево.

   Например, запуск

   ```
   ./a.out 2 4 3 a.txt
   ```

   означает, что B-дерево порядка 2 надо прочитать из файла `a.txt`, выводить не более 4-х уровней дерева, и вычислить результат задач для $k = 3$.

4. Класс "дерево" должен содержать функцию ввода дерева из указанного файла.

5. Ввод дерева из файла. В указанном файле находится дерево в формате:

Слово-1	Целое-число-1
Слово-2	Целое-число-2
…	…
Слово-n	Целое-число-n

   где слово – последовательность алфавитно-цифровых символов без пробелов. Длина слова неизвестна, память под него выделяется динамически. Все записи в файле различны (т.е. нет двух, у которых совпадают все поля). B-дерево заполняется как упорядоченное дерево. **Никакие другие функции, кроме функции ввода дерева, не используют упорядоченность дерева**. Концом ввода считается конец файла. Программа должна выводить сообщение об ошибке, если указанный файл не может быть прочитан или содержит данные неверного формата.

6. Класс "дерево" должен содержать подпрограмму вывода на экран не более чем r уровней дерева. Эта подпрограмма используется для вывода исходного дерева после его инициализации. Подпрограмма выводит на экран не более, чем r уровней дерева, где r – параметр этой подпрограммы (аргумент командной строки). Каждый элемент дерева должен печататься на новой строке и так, чтобы структура дерева была понятна.

7. Поскольку дерево не изменяется функциями из задач, то для всех задач надо сделать **одну функцию** `main`, в которой создается объект `b_tree<student>`, вводится из указанного файла, выводится указанное число уровней на экран, вызываются все функции задач, выводятся результаты и время их работы; затем удаляется этот объект. После компиляции должен получиться один исполняемый файл `a.out`, а не несколько.

8. Схема функции `main`:

```
int main (int argc, char *argv[])
{
  // Ввод аргументов командной строки
  ...
  b_tree<student> *a = new b_tree<student>;
  // Работа с деревом b_tree<student>
  ...
  delete a;
  return 0;
}
```

9. Вывод результата работы функции в функции `main` должен производиться по формату:

```
printf ("%s : Task = %d M = %d K = %d Result = %d Elapsed = %.2f\n",
        argv[0], task, m, k, res, t);
```

где

- `argv[0]` – первый аргумент командной строки (имя образа программы),
- `m` – параметр m командной строки,
- `k` – параметр k командной строки,
- `task` – номер задачи (1–6),
- `res` – результат работы функции, реализующей решение этой задачи,
- `t` – время работы функции, реализующей решение этой задачи.

Вывод должен производиться в точности в таком формате, чтобы можно было автоматизировать обработку запуска многих тестов.

Задачи

1. Написать функцию – член класса "B-дерево", получающую в качестве аргумента целое число k, и возвращающую целое значение, равное количеству элементов типа `student` в узлах, имеющих ровно k потомков.

2. Написать функцию – член класса "B-дерево", получающую в качестве аргумента целое число k, и возвращающую целое значение, равное количеству элементов типа `student` в поддеревьях, имеющих не более k вершин.

3. Написать функцию – член класса "B-дерево", получающую в качестве аргумента целое число k, и возвращающую целое значение, равное количеству элементов типа `student` в поддеревьях, имеющих не более k уровней.

4. Написать функцию – член класса "B-дерево", получающую в качестве аргумента целое число k, и возвращающую целое значение, равное количеству элементов типа `student` в поддеревьях, имеющих не более k узлов в любом уровне.

5. Написать функцию – член класса "B-дерево", получающую в качестве аргумента целое число k, и возвращающую целое значение, равное количеству элементов типа `student` в его k-м уровне.

6. Написать функцию – член класса "B-дерево", получающую в качестве аргумента целое число k, и возвращающую целое значение, равное количеству элементов типа `student` во всех ветвях длины k, начиная с корня.