Требования к программам

1. Программа работает с деревом с переменным числом ветвей объектов типа `student`:

```
class student
{
  private:
    char * name  = nullptr;
    int    value = 0;
  public:
    student () = default;
    student (const student& x) = delete;
    student (student&& x)
      {
        name = x.name; x.name = nullptr;
        value = x.value; x.value = 0;
      }
    ~student ()
      {
        erase ();
      }
    student& operator= (const student& x) = delete;
    student& operator= (student&& x)
      {
        if (this == &x)
          return *this;
        erase ();
        name = x.name; x.name = nullptr;
        value = x.value; x.value = 0;
        return *this;
      }
    void print () const;
    int read (FILE * fp);
    int cmp (const student& x) const
      {
        if (name == nullptr)
          {
            if (x.name != nullptr)
              return 1;
            return value - x.value;
          }
        if (x.name == nullptr)
          return -1;
        int res = strcmp (name, x.name);
        if (res)
          return res;
        return value - x.value;
      }
    int operator<  (const student& x) const { return cmp (x) <  0; }
    int operator<= (const student& x) const { return cmp (x) <= 0; }
    int operator>  (const student& x) const { return cmp (x) >  0; }
    int operator>= (const student& x) const { return cmp (x) >= 0; }
    int operator== (const student& x) const { return cmp (x) == 0; }
    int operator!= (const student& x) const { return cmp (x) != 0; }
```

```
  private:
    int init (const char * n, int v);
    void erase ();
};

class tree;
class tree_node : public student
  {
  private:
    tree_node * down  = nullptr;
    tree_node * level = nullptr;
  public:
    tree_node () = default;
    tree_node (const tree_node& x) = delete;
    tree_node (tree_node&& x) : student ((student&&)x)
      {
        erase_links ();
        x.erase_links ();
      }
    ~tree_node ()
      {
        erase_links ();
      }
    tree_node& operator= (const tree_node& x) = delete;
    tree_node& operator= (tree_node&& x)
      {
        if (this == &x)
          return *this;
        (student&&) *this = (student&&) x;
        erase_links ();
        x.erase_links ();
        return *this;
      }

    friend class tree;
  private:
    void erase_links ()
      {
        down  = nullptr;
        level = nullptr;
      }
  };

class tree
  {
  private:
    tree_node * root = nullptr;
  public:
    tree () = default;
    tree (const tree& x) = delete;
    tree (tree&& x)
      {
```

```
      root = x.root; x.root = nullptr;
    }
  ~tree ()
    {
      delete_subtree (root);
      root = nullptr;
    }
  tree& operator= (const tree& x) = delete;
  tree& operator= (tree&& x)
    {
      if (this == &x)
        return *this;
      delete_subtree (root);
      root = x.root; x.root = nullptr;
      return *this;
    }
  void print (int r) const
    {
      print_subtree (root, 0, r);
    }
  int read (FILE * fp);
private:
  static void delete_subtree (tree_node * curr)
    {
      if (curr == nullptr)
        return;
      tree_node * p, * next;
      for (p = curr->down; p; p = next)
        {
          next = p->level;
          delete_subtree (p);
        }
      delete curr;
    }
  static void print_subtree (tree_node * curr, int level, int r)
    {
      if (curr == nullptr || level > r)
        return;
      int spaces = level * 2;
      for (int i = 0; i < spaces; i++)
        printf (" ");
      curr->print ();
      for (tree_node * p = curr->down; p; p = p->level)
        print_subtree (p, level + 1, r);
    }
  static void add_node_subtree (tree_node* curr, tree_node* x)
    {
      if (curr->down == nullptr)
        { // No any child node
          curr->down = x;
          return;
        }
      if (*x < *curr)
```

```
        {
          if (*curr->down < *curr)
            // head of the list of child nodes < *curr
            add_node_subtree (curr->down, x);
          else
            {
              x->level = curr->down;
              curr->down = x;
            }
        }
      else if (*x == *curr)
        {
          if (curr->down->level != nullptr)
            {
              x->level = curr->down->level;
              curr->down->level = x;
            }
          else if (*curr->down < *curr)
            {
              curr->down->level = x;
            }
          else
            {
              x->level = curr->down;
              curr->down = x;
            }
        }
      else // *x > *curr
        {
          tree_node * p;
          for (p = curr->down; p->level; p = p->level);
          if (*p > *curr)
            // tail of the list of child nodes > *curr
            add_node_subtree (p, x);
          else
            p->level = x; // append at end of the list
        }
    }
};
```

Все функции в задании являются членами класса "дерево".

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

   1) r – количество выводимых уровней в дереве,
   2) `filename` – имя файла, откуда надо прочитать дерево,
   3) k – параметр задачи.

   Например, запуск

   ```
   ./a.out 4 a.txt 10
   ```

означает, что дерево надо прочитать из файла `a.txt`, выводить не более 4-х уровней дерева, $k = 10$

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

4. 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. При заполнении первый объект типа `student` попадает в корень дерева, каждый новый объект типа `student` добавляется в поддерево, являющееся:

   - первым элементом списка потомков, если он меньше текущего узла относительно операции сравнения `tree_node::operator<`,
   - последним элементом списка потомков, если он больше текущего узла,
   - одним из "средних" элементов списка потомков, если он равен текущему узлу, при этом может оказаться, что у текущего узла нет потомков, которые его меньше или(и) больше, и этот "средний" элемент будет первым или последним; таких "средних" элементов списка потомков может быть несколько, если к текущему узлу многократно добавляется равный ему элемент.

   Никакие другие функции, кроме функции ввода дерева, не используют указанную выше 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.

5. Решение задачи должно быть оформлено в виде подпрограммы, находящейся в отдельном файле. Получать в этой подпрограмме дополнительную информацию извне через глобальные переменные, включаемые файлы и т.п. запрещается.

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

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

   где

   - `argv[0]` – первый аргумент командной строки (имя образа программы),
   - `task` – номер задачи (1–7), `k` – аргумент k,
   - `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.

7. Класс "дерево" должен содержать подпрограмму вывода на экран не более чем r уровней дерева. Эта подпрограмма используется для вывода исходного дерева после его инициализации, а также для вывода на экран результата. 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.

8. Программа должна выводить на экран время, затраченное на решение.

9. Поскольку дерево не изменяется всеми функциями из задач, кроме последней, то для всех задач надо сделать **одну функцию** `main`, в которой прочитать дерево, вывести указанное число уровней на экран, и вызвать все функции задач, выводя результаты и время их работы. Вызов функции, реализующей последнюю задачу, должен быть последним и сразу после него надо вывести указанное число уровней преобразованного дерева на экран. Другими словами, после компиляции должен получиться **один исполняемый файл** `a.out`, а не несколько.

Задачи

1. Написать функцию – член класса "дерево" с переменным числом ветвей, получающую в качестве аргумента целое число k, и возвращающую целое значение, равное количеству элементов в узлах, имеющих ровно k потомков.

2. Написать функцию – член класса "дерево" с переменным числом ветвей, получающую в качестве аргумента целое число k, и возвращающую целое значение, равное количеству элементов в поддеревьях, имеющих не более k вершин.

3. Написать функцию – член класса "дерево" с переменным числом ветвей, получающую в качестве аргумента целое число k, и возвращающую целое значение, равное количеству элементов в поддеревьях, имеющих не более k уровней.

4. Написать функцию – член класса "дерево" с переменным числом ветвей, получающую в качестве аргумента целое число k, и возвращающую целое значение, равное количеству элементов в поддеревьях, имеющих не более k узлов в любом уровне.

5. Написать функцию – член класса "дерево" с переменным числом ветвей, получающую в качестве аргумента целое число k, и возвращающую целое значение, равное количеству элементов в его k-м уровне.

6. Написать функцию – член класса "дерево" с переменным числом ветвей, получающую в качестве аргумента целое число k, и возвращающую целое значение, равное количеству элементов во всех ветвях длины не менее k, начиная с корня.

7. Написать функцию – член класса "дерево" с переменным числом ветвей, получающую в качестве аргумента целое число k, которая удаляет все узлы дерева, имеющие значение поля `value` не превышающие k. Узел удаляется вместе со всеми потомками (т.е. удаляется поддерево с вершиной в этом узле). Функция возвращает целое значение, равное количеству удаленных узлов в дереве (только самих узлов, имеющие значение поля `value` $\leq k$, без учета потомков, у которых значение поля `value` $> k$).