Задача 1

Полностью объяснить задачу из второго задания № 10. Под «полностью объяснить» подразумевается все — от свойств используемых структур данных до строчек кода.

Формат сдачи — предоставленный исходный код плюс устная беседа по нему.

Задача 2

Разобрать алгоритм Морриса-Пратта поиска подстроки в строке (см. ниже). Оценить асимптотическую сложность алгоритма.

Формат сдачи — названная асимптотическая сложность (как «О большое от чего-то») плюс устная беседа по алгоритму.

Задача 3

На базе любой удобной задачи второго задания реализовать структуру данных и интерфейс к ней (указаны ниже). Программа при запуске должна читать команды из файла commands.txt, выполнять их, печатать результат на экран и завершаться. Никаких интерактивных действий от пользователя не предполагается.

Структура данных: бинарное дерево

Тип данных: строка не более 15 символов

Состав инструкций в commands.txt:

INSERT A — добавить элемент со значением A, ничего не печатать. Если такой элемент уже есть — добавить еще одну копию.

FIND A — найти элемент со значением A. Если элемент найден, напечатать FOUND. Если не найден — напечатать NOT FOUND.

DELETE A — найти и удалить элемент со значением A. Если элемента нет — ничего не делать. Если элементов несколько — удалить одну из копий.

DEDUP A — из всех вхождений элемента со значением A оставить только одно (первое), остальные удалить.

Пример:

Инструкции в файле:

INSERT alice

INSERT alice

INSERT alice

DEDUP alice

DELETE alice

FIND alice

Вывод программы:

NOT FOUND (так как после DEDUP осталась только одна alice, которую потом удалили)

Формат сдачи — предоставленный исходный код плюс устная беседа по нему.

Алгоритм Морриса-Пратта поиска подстроки в строке

Простейшие алгоритм поиска подстроки — алгоритм грубой силы. Этот алгоритм заключается в проверке всех позиций текста с 0 по n - m на предмет совпадения с началом образца. Если совпадает - смотрим следующую букву и т.д.

Пример реализации:

```
#define EOS '\0'

void BF(char *x, char *y, int m) {

 int i;

 /* Ищем до конца, вообще говоря можно до позиции n-m */

 for(i=0; *y!=EOS; i++, y++) if(memcmp(y,x,m) == 0) OUTPUT(i);

}
```

Из анализа этого алгоритма возник алгоритм Морриса-Пратта. Исследователи хотели найти способы более полно использовать информацию, полученную во время сканирования (алгоритм грубой силы ее просто выбрасывает ;-( ).

Итак, давайте и мы взглянем на него поближе. Оказывается, размер сдвига образца можно увеличить, одновремеменно запомнив части текста, совпадающие с образцом. Это позволит нам избежать ненужных сравнений и, тем самым, резко увеличить скорость поиска.

Рассмотрим сравнение на позиции i, где образец x[0, m - 1] сопоставляется с частью текста y[i, i + m - 1]. Предположим, что первое несовпадение произошло между y[i + j] и x[j] , где 1 < j < m. Тогда y[i, i + j - 1] = x[0, j - 1] = u и a = y[i + j] =/= x[j] = b.

При сдвиге вполне можно ожидать, что префикс образца u сойдется с каким-нибудь суффиксом подслова текста u. Наиболее длинный такой префикс - граница u (Он встречается на обоих концах u). Это приводит нас к следующему алгоритму:
пусть mp_next[j] - длина границы x[0, j - 1]. Тогда после сдвига мы можем возобновить сравнения с места y[i + j] и x[j - mp_next[j]] без потери возможного местонахождения образца. Таблицаmp_next может быть вычислена за O(m) перед самим поиском. Максимальное число сравнений на 1 символ - m.

Пример реализации:

```
/* Preprocessing */

void PRE_MP( char *x, int m, int mp_next[] ) {

  int i, j;

  i=0;

  j=mp_next[0]=-1;

  while ( i < m ) {

     while ( j > - 1 && x[i] != x[j] ) j=mp_next[j];

    mp_next[++i]=++j;

   }

  }


  void MP( char *x, char *y, int n, int m ) {

    int i, j, mp_next[XSIZE];
```

```
  /* Preprocessing */
  PRE_MP(x, m, mp_next );
  /* Searching */
  i=j=0;
  while ( i < n ) {
    while ( j > -1 && x[j] != y[i] ) j=mp_next[j];
    i++;
    j++;
    if ( j >= m ) {
       OUTPUT( i - j );
      j = mp_next[ j ];
    }
   }
}
```