где `<длина записи n>` есть четырехбайтовое поле, содержащее значение числа типа `unsigned long` — длины соответствующей записи. Последняя запись в файле (т.е. `<запись k>`) соответствует вершине стека.

Реализуйте функции со следующими прототипами

```
int     OpenFileStack (char *filename);
int     PushRecord (void *record, size_t length);
          // функция помещает в стек length байтов,
          // размещенных начиная с адреса record
int     PopRecord  (void *record, size_t *length);
          // функция берет вершину стека и помещает ее
          // в области памяти по адресу record,
          // значение длины записи помещается по адресу length
size_t  TopLength  (void);
          // функция возвращает длину записи
          // на вершине стека
int     CloseFileStack (void);
```

Функции возвращают 0 в случае успешного выполнения и -1 при отказе.

3.19. Постройте реализацию файлового стека строк на основе объединения идей задач 3.17, 3.18 Строки частично хранятся в памяти, частично — на диске. В памяти выделяется два блока — первый и второй. Строки накапливаются сначала в первом блоке, а когда он заполнится — во втором. При заполнении второго блока первый блок записывается в файл (по стековому принципу) и блоки обмениваются своими номерами: второй (занятый) блок становится первым, а первый (теперь свободный) становится вторым. При удалении процедура обращается: пока существуют строки в блоках, работа идет в памяти, а когда оба блока становятся пустыми, в первый считываются данные из файлового стека блоков.

Дек. Эту структуру можно рассматривать как обобщение стека за счет возможности добавлять и удалять элементы "с обоих концов" набора данных, т.е. имеется два текущих элемента, к которым разрешен доступ и которые называются "начало" и "конец" дека.

Дисциплина доступа к началу и концу дека аналогична дисциплине работы с вершиной стека: разрешено чтение, добавление и извлечение элементов.

3.20. Постройте реализацию дека элементов абстрактного типа Type.

Идеи реализации. Будем размещать элементы в выделенном участке памяти вплотную друг к другу и хранить два указателя — на начало и конец дека. При добавлении или удалении элементов эти указатели будем перемещать на позицию следующего или предыдущего элементов. В этом случае для корректной работы дека нужно формально "замкнуть" левую и правую границы выделенной области: при необходимости новая позиция "перебрасывается" с позиции за концом области в начало, или с позиции перед началом — в конец. В итоге массив оказывается логически замкнутым в кольцо. Если принять, что начало, конец выделенной области памяти, голова, хвост и размер дека задаются переменными

```
      Type  *begmem, *endmem;
      Type  *head, *tail;
      int    size;
```

то процедуры определения следующего и предыдущего элементов могут выглядеть, например, так

```
Type * Next (Type *pos)
{ return (pos==endmem) ? begmem : pos+1; }
Type * Prev (Type *pos)
{ return (pos==begmem) ? endmem : pos-1; }
```

Реализуйте дек на языке C со следующим интерфейсом:

```
/*---   Файл  dequeue.h  ---------------------------------------*/
typedef ....  Type;
int     Dq_Init (int maxsize);   /* создать дек                 */
void    Dq_Term (void);          /* закончить работу            */
int     Dq_PushHead (Type val);  /* добавить элемент  val       */
int     Dq_PushTail (Type val);  /*    в начало или конец дека  */
int     Dq_PopHead  (Type *dst); /* взять начало или конец дека */
int     Dq_PopTail  (Type *dst); /* и поместить по указателю dst */
Type  * Dq_Head (void);          /* указатель на начало         */
Type  * Dq_Tail (void);          /*     или конец дека          */
int     Dq_Size (void);          /* текущее кол-во элементов    */
int     Dq_Room (void);          /* кол-во свободных элементов  */
/*---   конец файла dequeue.h   ---------------------------------*/
```

Реализуйте дек на языке C++ в виде следующего класса:

```
//---  Файл  dequeue.h  ---------------------------------
template <class Type>
class   Dequeue
{  private:
      Type  *begmem, *endmem;
      Type  *head, *tail;
      int    size;
      Type * Next (Type *pos);
      Type * Prev (Type *pos);
   public:
      Dequeue (int maxsize);
     ~Dequeue ();
      int    Success  ();
      int    PushHead (Type val);
      int    PushTail (Type val);
      int    PopHead  (Type &dst);
      int    PopTail  (Type &dst);
      Type & Head ();
      Type & Tail ();
      int    Size ();
      int    Room ();
};
//---  конец файла dequeue.h  -----------------------------
```

По поводу функции `Success()` см. замечание к задаче 3.1 о реализации стека.

3.21. Постройте реализацию дека элементов абстрактного типа Type на основе структуры

```
struct Dequeue{
  Type * mem;          /* начало области элементов дека */
  int maxsize;         /* размер дека */
  int size;            /* текущее количество элементов */
  int left;            /* позиция для очередного элемента слева  */
  int right;           /* позиция для очередного элемента справа  */
};
```

Идеи реализации. Считая, что `maxsize>0`, будем размещать элементы дека в выделенном участке памяти `mem[0], ... , mem[maxsize-1]` и хранить два целых числа `left` и `right` из диапазона от 0 до `maxsize-1`, характеризующие текущие позиции головы и хвоста. В начальный момент `left=0;right=(left+1)%maxzise`. Таким образом, указатели `left,right` содержат номера ячеек, куда будут добавляться очередные элементы (при наличии свободного места). При добалении слева имеем:

```
if(size==maxsize)return -1;
mem[left]=val; size++; left--;
if(left<0)left=maxsize-1;
return 0;
```

Добавление справа осуществляется аналогично:

```
if(size==maxsize)return -1;
mem[right]=val; size++; rigt=(right+1)%maxsize;
return 0;
```

3.22. Выпуклой оболочкой множества точек называется наименьший выпуклый многоугольник с вершинами в некоторых точках этого множества, содержащий внутри себя все остальные точки множества. Требуется построить выпуклую оболочку заданного множества точек плоскости (т.е. указать точки, являющиеся ее вершинами).

Идеи реализации. Для хранения подмножества точек, составляющих вершины выпуклой оболочки, выделим дек. Точки, лежащие в начале и конце дека будут составлять текущее ребро выпуклой оболочки. Сначала в множестве точек находятся три точки, являющиеся вершинами некоторого невырожденного треугольника. Далее последовательно проверяется в какой полуплоскости относительно линии, содержащей ребро уже построенной выпуклой оболочки, лежит каждая следующая точка множества. Для этого последовательно рассматривается текущее ребро, которое меняется перекладыванием элементов дека, например, из начала в конец. Таким образом можно выяснить находится точка внутри или вне уже построенной части выпуклой оболочки и при необходимости включить новую точку в выпуклую оболочку, т.е. удалить некоторые стороны и добавить новые две стороны с общей вершиной в этой точке.

Очередь. Дисциплина работы с очередью часто обозначается сокращением FIFO (First In, First Out), т.е. элемент, первым добавленный в очередь (в ее конец), будет также первым взят из (начала) очереди. Как нетрудно видеть, очередь является частным случаем дека, если запретить удаление и доступ к элементу, находящемуся в конце, а также добавление элемента в начало. Доступ к значению хранимых данных разрешается только для элемента, расположенного в начале очереди.

3.23. Реализуйте очередь элементов абстрактного типа `Type` на языке C со следующим интерфейсом:

```
/*---   Файл  queue.h  ----------------------------------------*/
typedef ....  Type;
int     Qu_Init (int maxsize);   /* создать очередь             */
void    Qu_Term (void);          /* закончить работу            */
int     Qu_Add  (Type val);      /* добавить элемент в очередь  */
int     Qu_Take (Type *dst);     /* взять элемент из очереди    */
Type  * Qu_Head (void);          /* указатель на начало очереди */
int     Qu_Size (void);          /* текущее кол-во элементов    */
int     Qu_Room (void);          /* кол-во свободных элементов  */
/*---   конец файла queue.h  ---------------------------------*/
```

3.24. Реализуйте очередь на языке C++ в виде следующего класса:

```
//---   Файл  queue.h  ---------------------------------------
template <class Type>
class   Queue
{  private:
      Type  *begmem, *endmem;
      Type  *head, *tail;
      int    size;
      Type * Next (Type *pos);
   public:
      Queue (int maxsize);
     ~Queue ();
      int    Success  ();
      int    Add  (Type val);
      int    Take (Type &dst);
      Type & Head ();
      int    Size ();
      int    Room ();
};
//---  конец файла queue.h  -------------------------------
```

3.25. Считая, что время добавления и изъятия элементов из очереди есть случайные величины смоделируйте, как со временем изменяется длина очереди.