Лабораторная работа 11

Липкин Григорий, А-18-21

Разработать модули однонаправленного и двунаправленного списка из элементов в диапазоне и, возможно, упорядоченных заданным образом.
Модули должны обеспечивать следующие возможности работы со списками:
- инициализацию
- добавление:
  - в начало
  - в конец
  - после заданного элемента
  - до заданного (двунаправленный)
  - так, чтобы сохранялся определённый порядок
- удаление элемента
- удаление списка
- ввод из текстового файла (до конца файла);
- вывод на экран (
  - однонаправленный в прямом порядке
  - двунаправленный в прямом и обратном порядке
-)
- поиск в списке элемента, что `условие`:
  - в однонаправленном первый, удовлетворяющий `условию`
  - в двунаправленном первый и последний, удовлетворяющие `условию`
- удаление элементов списка, удовлетворяющих условию.

Разработать программу, к которой будут поочерёдно подключаться разработанные модули.
Программа должна сделать следующее:
- ввести данные из файла с сохранением порядка элементов таким, какой есть в файле
- добавить несколько элементов в начало списка и несколько элементов в конец списка
- найти первый (и последний) элемент по заданному условию
- удалить элементы из списка по заданному условию
- удалить список
- ввести данные из файла, добавляя элементы в список так, чтобы соблюдался порядок, указанный в задании;
- найти первый (и последний) элемент по заданному условию;
- удалить элементы из списка по заданному условию;
- удалить список.
- После каждого изменения выводить содержимое списка.

Имена файлов передаются через параметры программы.

```
unit listUnit;
interface
    // uses
        uses sysUtils;
    // type definitions
        type ListNodeType = record
                                next  : ^ListNodeType;
                                value : Pointer;
                                valueType : String;
                            end;
        type ListType     = record
                                first  : ^ListNodeType;
                                length : Integer;
                            end;
        type BlockType = function(val : ListNodeType) : Boolean;
    // List methods
        procedure initList(var list : ListType); overload;
        // adds
            function addToList(var list  : ListType;
                               var value : ListNodeType) : ListType; overload;

            function addToList(var list  : ListType;
                               value     : Pointer;
                               valueType : String) : ListType; overload;

        // unshifts
            function unshiftList(var list  : ListType;
                                 var value : ListNodeType) : ListType; overload;

            function unshiftList(var list  : ListType;
                                     value : Pointer;
                                 valueType : String) : ListType; overload;
        // inspect/to_s/print_self
            function inspect( const node : ListNodeType) : String; overload;
            function toString(const node : ListNodeType) : String; overload;
            function inspect( const list : ListType)     : String; overload;
            function toString(const list : ListType)     : String; overload;
            procedure writeList(const list : ListType); overload;
        // finds
            function findNodeWithIndex(const node : ListNodeType;
                                       var index : Integer;
                                          number : Integer) : ListNodeType;
overload;
        // inserts
            function insertToList(var list : ListType;
                                      what : ListNodeType;
                                     where : Integer) : ListType; overload;
implementation
```

```
procedure initList(var list : ListType);
    begin
        list.first := nil;
        list.length := 0;
    end;

function addToList(var list  : ListType;
                   var value : ListNodeType) : ListType; overload;
    var elem : ListNodeType;
    begin
        // writeln('List#add');
        // writeln(toString(value));
        if (Pointer(list.first) = nil)
        then begin
            // writeln('List#add::then');
            list.first := @value;
            value.next := nil;
        end else begin
            // writeln('List#add::else');
            elem := ListNodeType(list.first^);
            while not (Pointer(elem.next) = nil) do begin
                elem := ListNodeType(elem.next^);
            end;
            elem.next := @value;
        end;
        list.length := list.length + 1;
        // writeln('List#add -> List#to_s');
        // writeln(toString(list));
        // writeln('List#to_s -> List#add');
        // writeln;
        addToList := list;
    end;

function addToList(var list  : ListType;
                   value     : Pointer;
                   valueType : String) : ListType; overload;
    var valueNode : ListNodeType;
    begin
        valueNode.value := value;
        valueNode.next := nil;
        valueNode.valueType := valueType;
        list.length := list.length + 1;
        addToList := addToList(list, valueNode);
    end;

function unshiftList(var list  : ListType;
                     var value : ListNodeType) : ListType; overload;
    begin
```

```
        if (Pointer(list.first) = nil)
        then value.next := nil
        else value.next := list.first;
        list.first := @value;
        unshiftList := list;
    end;

function unshiftList(var list  : ListType;
                         value : Pointer;
                     valueType : String) : ListType; overload;
    var valueNode : ListNodeType;
    begin
        valueNode.value := value;
        valueNode.next := nil;
        valueNode.valueType := valueType;
        unshiftList := unshiftList(list, valueNode);
    end;

function toString(const list : ListType) : String; overload;
    var toReturn : String;
        elem     : ListNodeType;
    begin
        // writeln('List#to_s');
        toReturn := '';
        if (Pointer(list.first) = nil)
        then begin
            // writeln('List#to_s::then');
        end else begin
            // writeln('List#to_s::else');
            elem := list.first^;
            if Pointer(elem.next) = nil
            then begin
                // writeln('List#to_s::else::then');
                // writeln('List#to_s -> Node#to_s');
                toReturn := toReturn + toString(elem);
                // writeln('Node#to_s -> List#to_s');
            end else repeat
                // writeln('List#to_s::else::else::repeat');
                // writeln('List#to_s -> Node#to_s');
                toReturn := toReturn + toString(elem);
                // writeln('Node#to_s -> List#to_s');
                if not (Pointer(elem.next) = nil)
                then toReturn := toReturn + #10;
                elem := elem.next^;
            until Pointer(elem.next) = nil;
        end;
        toString := toReturn;
    end;
```

```
    function toString(const node : ListNodeType) : String; overload;
        var toReturn : String;
        begin
            // write('Node#to_s');
            toReturn := '';
            case node.valueType of
                'String', 'string', 'Char', 'char'  :
                            toReturn := String(node.value^);
                'Integer' : toReturn := intToStr(Integer(node.value^));
                'Real'    : toReturn := FloatToStr(Real(node.value^));
                else begin writeln('::UNKNOWN TYPE!'); toReturn := 'ERR::NOTYPE';
end;
            end;
            toString := toReturn;
        end;

    function inspect(const node : ListNodeType) : String; overload;
        var toReturn : String;
        begin
            toReturn := '';
            case node.valueType of
                'String', 'string', 'Char', 'char'  :
                            toReturn := '"' + String(node.value^) + '"';
                'Integer' : toReturn := intToStr(Integer(node.value^));
                'Real'    : toReturn := FloatToStr(Real(node.value^));
                else toReturn := 'ERR::NOTYPE';
            end;
            inspect := toReturn;
        end;

    function inspect(const list : ListType) : String; overload;
        var toReturn : String;
            elem     : ListNodeType;
        begin
            // writeln('List#to_s');
            toReturn := 'List[';
            if (Pointer(list.first) = nil)
            then begin
                // writeln('List#to_s::then');
            end else begin
                // writeln('List#to_s::else');
                elem := list.first^;
                if Pointer(elem.next) = nil
                then begin
                    // writeln('List#to_s::else::then');
                    // writeln('List#to_s -> Node#to_s');
                    toReturn := toReturn + inspect(elem);
```

```
                // writeln('Node#to_s -> List#to_s');
            end else repeat
                // writeln('List#to_s::else::else::repeat');
                // writeln('List#to_s -> Node#to_s');
                toReturn := toReturn + inspect(elem);
                // writeln('Node#to_s -> List#to_s');
                if not (Pointer(elem.next) = nil)
                then toReturn := toReturn + ', ';
                elem := elem.next^;
            until Pointer(elem.next) = nil;
        end;
        inspect := toReturn + ']';
    end;

procedure writeList(const list : ListType); overload;
    begin
        write(toString(list));
    end;
function findNodeWithIndex(const node : ListNodeType;
                           var index : Integer;
                              number : Integer) : ListNodeType; overload;
    begin
        if index = number
        then findNodeWithIndex := node
        else begin
            index := index + 1;
            findNodeWithIndex := findNodeWithIndex(
                node.next^,
                index,
                number
            )
        end;
    end;
function insertToList(var list : ListType;
                         what : ListNodeType;
                        where : Integer) : ListType; overload;
    var index : Integer;
        node  : ListNodeType;
    begin
        index := 0;
        node := findNodeWithIndex(list.first^, index, where);
        what.next := node.next;
        node.next := @what;
        insertToList := list;
    end;
function delete(var list : ListType; where : Integer) : ListType; overload;
    var node  : ListNodeType;
        index : Integer;
```

```
        begin
            index := 0;
            node := findNodeWithIndex(list.first^, index, where - 1);
            if not (Pointer(node.next^.next) = nil)
            then node.next := node.next^.next
            else node.next := nil;
            delete := list;
        end;
    procedure recursiveDelete(var node : ListNodeType); overload;
        begin
            if not (Pointer(node.next) = nil)
            then recursiveDelete(node.next^);
            node.next := nil;
        end;
    procedure delete(var list  : ListType); overload;
        var elem : ListNodeType;
        begin
            if not (Pointer(list.first) = nil)
            then begin
                elem := list.first^;
                recursiveDelete(elem);
            end;
        end;

    function findByBlock(const node : ListNodeType; block : BlockType) :
ListNodeType; overload;
        begin
            if not(Pointer(node.next) = nil)
            then findByBlock := findByBlock(node.next^, block)
            else findByBlock := node;
        end;

    function findByBlock(const list : ListType; block : BlockType) : ListNodeType;
overload;
        begin
            if not(Pointer(list.first) = nil)
            then findByBlock := findByBlock(list.first^, block);
        end;
end.
```