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.

```
program main;
uses listUnit, biListUnit;

function block(val : ListNodeType) : Boolean;
    begin
        block := (val.value = 'D') or (val.value = 'E');
    end;
function conditionBlock(val1, val2 : ListNodeType) : Boolean;
    begin
        conditionBlock := (val1.value >= val2.value);
    end;
var aText : Text;
    list : ListType;
    aChar : Char;
begin
    assign(aText, 'text.txt');
    reset(aText);
    readList(aText, list);
    close(aText);
    unshift(list, #10);
    unshift(list, #64);
    add(list, #81);
    add(list, #80);
    writeList(list);
    writeln(findByBlock(list, @block).value);
    reset(aText);
    while not EOF(aText) do begin
        read(aText, aChar);
        insertBy(@conditionBlock, list, aChar);
    end;
    close(aText);
    writeList(list);
    findByBlock(list, @block);
    deleteBy(@block, list);
    writeList(list);
    delete(list);
end.

unit listUnit;
interface
    type ListNodeType = record
                            next  : ^ListNodeType;
                            value : Char;
                        end;
    type ListType     = record first  : ^ListNodeType; end;
    type BlockType = function(val : ListNodeType) : Boolean;
    type ConditionBlockType = function(val1, val2 : ListNodeType) : Boolean;
        procedure init(var list : ListType); overload;
```

```
        function add(var list  : ListType;
                     var value : ListNodeType) : ListType; overload;
        function add(var list  : ListType;
                     value     : Char) : ListType; overload;
        function unshift(var list  : ListType;
                                var value : ListNodeType) : ListType; overload;

        function unshift(var list  : ListType;
                               value : Char) : ListType; overload;
        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;
        function findNodeWithIndex(const node : ListNodeType;
                                   var index : Integer;
                                      number : Integer) : ListNodeType; overload;
        // inserts
            function insert(var list : ListType;
                                    what : ListNodeType;
                                   where : Integer) : ListType; overload;
    function delete(var list : ListType; where : Integer) : ListType; overload;
    procedure recursiveDelete(var node : ListNodeType); overload;
    procedure delete(var list  : ListType); overload;
    function findByBlock(const node : ListNodeType; block : BlockType) :
ListNodeType; overload;
    function findByBlock(const list : ListType; block : BlockType) : ListNodeType;
overload;
    function insertBy(block : ConditionBlockType; node : ListNodeType; given :
ListNodeType) : ListNodeType; overload;
    function insertBy(block : ConditionBlockType; const list : ListType; node :
ListNodeType) : ListType; overload;
    function insertBy(block : ConditionBlockType; const list : ListType; val :
Char) : ListType; overload;
    procedure deleteBy(block : BlockType; var node : ListNodeType); overload;
    function deleteBy(block : BlockType; var list : ListType) : ListType; overload;
    function readList(var aFile : Text; var list : ListType) : ListType; overload;
implementation
    procedure init(var list : ListType);
        begin
            list.first := nil;
        end;

    function add(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;
        // writeln('List#add -> List#to_s');
        // writeln(toString(list));
        // writeln('List#to_s -> List#add');
        // writeln;
        add := list;
    end;

function add(var list  : ListType;
             value     : Char) : ListType; overload;
    var valueNode : ListNodeType;
    begin
        valueNode.value := value;
        valueNode.next := nil;
        add := add(list, valueNode);
    end;

function unshift(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;
        unshift := list;
    end;

function unshift(var list  : ListType;
                     value : Char) : ListType; overload;
    var valueNode : ListNodeType;
    begin
        valueNode.value := value;
        valueNode.next := nil;
        unshift := unshift(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;
    begin
        // write('Node#to_s');
        toString := String(node.value);
    end;

function inspect(const node : ListNodeType) : String; overload;
    begin
        inspect := node.value;
    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 insert(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;
            insert := 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;

    function insertBy(block : ConditionBlockType; node : ListNodeType; given :
ListNodeType) : ListNodeType; overload;
        begin
            if Pointer(node.next) = nil
            then begin
                node.next := @given;
                given.next := @node;
            end
            else
            if block(node, given) and block(given, node.next^)
            then begin
                given.next := node.next;
                node.next := @given;
            end
            else insertBy := insertBy(block, node.next^, given);
        end;

    function insertBy(block : ConditionBlockType; const list : ListType; node :
ListNodeType) : ListType; overload;
        begin
            insertBy(block, list.first^, node);
            insertBy := list;
        end;
    function insertBy(block : ConditionBlockType; const list : ListType; val :
Char) : ListType; overload;
        var node : ListNodeType;
        begin
            node.value := val;
            node.next := nil;
            insertBy := insertBy(block, list, node);
        end;
    procedure deleteBy(block : BlockType; var node : ListNodeType); overload;
        begin
            if Pointer(node.next) <> nil
            then begin
                if block(node.next^)
                then node.next := node.next^.next;
                deleteBy(block, node.next^);
            end;
        end;
    function deleteBy(block : BlockType; var list : ListType) : ListType; overload;
        begin
            deleteBy(block, list.first^);
            deleteBy := list;
        end;
```

```
    function readList(var aFile : Text; var list : ListType) : ListType; overload;
        var aChar : Char;
        begin
            aChar := #0;

            while not EOF(aFile) do begin
                read(aFile, aChar);
                add(list, aChar);
            end;
            readList := list;
        end;
end.

{$MODE OBJFPC}
unit biListUnit;
interface
    uses sysUtils;
    type BiListNodeType = record next, last : ^BiListNodeType;
                                 value      : Char;
                         end;
    type BiListType = record first, last : ^BiListNodeType; end;
    type BlockType = function(val : BiListNodeType) : Boolean;
    type ConditionBlockType = function(val1, val2 : BiListNodeType) : Boolean;
    function last(const node : BiListNodeType) : BiListNodeType; overload;
    function init(var list : BiListType) : BiListType; overload;
    function last(const list : BiListType) : BiListNodeType; overload;
    function add(var list : BiListType; value : char) : BiListType;
    function add(var list : BiListType;
                     node : BiListNodeType
                )         : BiListType; overload;
    function unshift(var list : BiListType;
                         node : BiListNodeType
                    )         : BiListType; overload;
    function get(var list : BiListType; which : Integer) : BiListNodeType;
overload;
    function get(var node           : BiListNodeType;
                     counter, which : Integer
                )                   : BiListNodeType; overload;
    procedure deleteBy(block : BlockType; var node : BiListNodeType); overload;
    function deleteBy(block : BlockType; var list : BiListType) : BiListType;
overload;
    function readList(var aFile : Text; var list : BiListType) : BiListType;
overload;
    function insertBy(block : ConditionBlockType; const list : BiListType; node :
BiListNodeType) : BiListType; overload;
implementation
    function init(var list : BiListType) : BiListType; overload;
        begin
```

```
        list.first := nil;
        list.last  := nil;
        init := list;
    end;

function last(const node : BiListNodeType) : BiListNodeType; overload;
    begin
        if Pointer(node.next) = nil
        then last := node
        else last := last(node.next^);
    end;
function last(const list : BiListType) : BiListNodeType; overload;
    begin
        last := list.last^;
    end;
function add(var list : BiListType; value : char) : BiListType;
    var node : BiListNodeType;
    begin
        node.last := nil;
        node.next := nil;
        node.value := value;
        add := add(list, node)
    end;
function add(
        var list : BiListType;
        node : BiListNodeType
    ) : BiListType; overload;
    var last : BiListNodeType;
    begin
        if Pointer(list.first) = nil
        then begin
            list.first := @node;
            list.last  := @node;
        end
        else begin
            last := list.last^;
            last.next := @node;
            node.last := @last;
            list.last := @node;
        end;
        add := list;
    end;

function unshift(var list : BiListType;
                     node : BiListNodeType
                )         : BiListType; overload;
    var first : BiListNodeType;
    begin
```

```
            if Pointer(list.first) = nil
            then begin
                list.first := @node;
                list.last  := @node;
            end
            else begin
                first := list.first^;
                first.last := @node;
                node.next  := @first;
                list.first := @node;
            end;
            unshift := list;
        end;
    function get(var node           : BiListNodeType;
                     counter, which : Integer
                )                  : BiListNodeType; overload;
        begin
            if counter = which
            then get := node
            else
            if Pointer(node.next) = nil
            then writeln('No such node!')
            else get := get(node.next^, counter + 1, which);
        end;
    function get(var list : BiListType; which : Integer) : BiListNodeType;
overload;
        begin
            get := get(list.first^, 0, which);
        end;
    function insert(var list      : BiListType;
                        node      : BiListNodeType;
                        afterWhat : Integer
                   )              : BiListType; overload;
        var nodeBefore, nodeAfter : BiListNodeType;
        begin
            nodeBefore := get(list, afterWhat);
            nodeAfter  := nodeBefore.next^;
            nodeBefore.next := @node;
            nodeAfter.last  := @node;
            node.last := @nodeBefore;
            node.next := @nodeAfter;
            insert := list;
        end;
    function findBy(block : BlockType; node : BiListNodeType) : BiListNodeType;
overload;
        begin
            if block(node)
            then findBy := node
```

```
            else
            if Pointer(node.next) = nil
            then writeln('Not found')
            else findBy := findBy(block, node.next^);
        end;
    function rFindBy(block : BlockType; node : BiListNodeType) : BiListNodeType;
overload;
        begin
            if block(node)
            then rFindBy := node
            else
            if Pointer(node.last) = nil
            then writeln('Not found')
            else rFindBy := rFindBy(block, node.last^);
        end;
    function rFindBy(block : BlockType; const list : BiListType) : BiListNodeType;
overload;
        begin rFindBy := rFindBy(block, list.last^); end;
    function findBy(block : BlockType; var list : BiListType) : BiListNodeType;
overload;
        begin
            findBy := findBy(block, list.first^);
        end;

    function insertBy(block : ConditionBlockType; node : BiListNodeType; given :
BiListNodeType) : BiListNodeType; overload;
        begin
            if Pointer(node.next) = nil
            then begin
                node.next := @given;
                given.last := @node;
            end
            else
            if block(node, given) and block(given, node.next^)
            then begin
                given.next := node.next;
                node.next := @given;
                given.last := @node;
            end
            else insertBy := insertBy(block, node.next^, given);
        end;

    function insertBy(block : ConditionBlockType; const list : BiListType; node :
BiListNodeType) : BiListType; overload;
        begin
            insertBy(block, list.first^, node);
            insertBy := list;
        end;
```

```
    procedure deleteBy(block : BlockType; var node : BiListNodeType); overload;
        begin
            if block(node)
            then begin
                node.last^.next := node.next;
                node.next^.last := node.last;
            end;
            if Pointer(node.next) <> nil
            then deleteBy(block, node.next^);
        end;
    function deleteBy(block : BlockType; var list : BiListType) : BiListType;
overload;
        begin
            deleteBy(block, list.first^);
            deleteBy := list;
        end;
    function readList(var aFile : Text; var list : BiListType) : BiListType;
overload;
        var aChar : Char;
        begin
            aChar := #0;

            while not EOF(aFile) do begin
                read(aFile, aChar);
                add(list, aChar);
            end;
            readList := list;
        end;
    procedure writeList(const node : BiListNodeType); overload;
        begin
            writeln(node.value);
            if Pointer(node.next) <> nil
            then writeList(node.next^);
        end;
    procedure writeList(const list : BiListType); overload;
        begin
            if Pointer(list.first) <> nil
            then writeList(list.first^)
            else writeln('Nil list@first');
        end;

    procedure rWriteList(const node : BiListNodeType); overload;
        begin
            writeln(node.value);
            if Pointer(node.last) <> nil
            then rWriteList(node.last^);
        end;
    procedure rWriteList(const list : BiListType); overload;
```

```
        begin
            if Pointer(list.last) <> nil
            then rWriteList(list.last^)
            else writeln('Nil list@last');
        end;
end.
```