Министерство образования Республики Беларусь

Учреждение образования

«Белорусский государственный университет информатики и радиоэлектроники»

Филиал

«Минский радиотехнический колледж»

Учебная дисциплина «Структуры и алгоритмы обработки данных»

**Инструкция**

по выполнению лабораторной работы

«Реализация линейного списка и основных алгоритмов его обработки»

Минск

2018

**Лабораторная работа № 4**

**Тема работы: «Реализация линейного списка и основных алгоритмов его обработки»**

**1. Цель работы**

Изучение принципов работы с линейными списками.

**2. Задание**

1. Составить программу, которая на основе заданного списка формирует два других, помещая в первый из них положительные, а во второй — отрицательные элементы исходного списка. Выполнить задание по варианту:

Вариант 1. Подсчитать сумму элементов исходного списка.

Вариант 2. Подсчитать количество элементов в исходном списке.

2. Определить в двунаправленном списке количество элементов, у которых соседи справа и слева отрицательны.

**3. Оснащение работы**

ПК, Pascal.

**4. Основные теоретические сведения**

Линейные связные списки являются простейшими динамическими структурами данных.

Из всего многообразия связанных списков можно выделить следующие основные:

- однонаправленные (односвязные) списки;

- двунаправленные (двусвязные) списки;

- циклические (кольцевые) списки.

В основном они отличаются видом взаимосвязи элементов и/или допустимыми операциями.

С реализациями линейных списков в императивных языках программирования могут выполняться следующие операции:

- получение доступа к некоторому элементу списка для проверки и/или изменения содержимого его полей;

- вставка нового элемента сразу перед или после произвольного элемента;

- удаление произвольного элемента;

- объединение в одном списке двух (или более) линейных списков;

- разбиение линейного списка на два (или более) списка;

- создание копии линейного списка;

- определение количества элементов в списке;

- сортировка элементов списка;

- поиск элементов с заданным значением.

*В одной программе крайне редко возникает необходимость использовать все девять типов операций. При этом достаточно трудно создать единую реализацию линейных списков, при которой эффективно выполнялись бы все эти операции. Поэтому линейные списки могут быть реализованы по-разному в зависимости от класса операций, которые наиболее часто должны с ними выполняться в данной программе, или наиболее критичных к времени выполнения.*

Однонаправленный (односвязный) список – это структура данных, представляющая собой последовательность элементов, в каждом из которых хранится значение и указатель на следующий элемент списка.

*Каждый элемент связанного списка, во-первых, хранит какую-либо информацию, во-вторых, указывает на следующий за ним элемент.* Так как элемент списка хранит разнотипные части (хранимая информация и указатель), то его естественно представить записью, в которой в одном поле располагается объект, а в другом – указатель на следующую запись такого же типа. Такая запись называется звеном, а структура из таких записей называется списком или цепочкой.

Лишь на самый первый элемент списка (голову) имеется отдельный указатель. Последний элемент списка никуда не указывает.

*Пример описания списка:*

Type**U= ^ S;**

**S=**record

**Inf: BT {базовый тип};**

**Next: U;**

End**;**

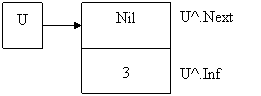
Для того, чтобы создать список, нужно создать сначала первый элемент списка, а затем при помощи функции добавить к нему остальные элементы. При относительно небольших размерах списка наиболее изящно и красиво использование рекурсивной функции. Добавление может выполняться как в начало, так и в конец списка.

**1. Создание первого элемента списка:**

New (u); {выделяем место в памяти}

u^. Next:= nil; {указатель пуст}

u^. Inf:=3;



**2. Добавление звена в начало списка**

|  |  |
| --- | --- |
| Добавление звена в начало списка | {Процедура добавления звена в начало списка; в x содержится добавляемая информация}  Procedure V\_Nachalo(Var First : U; X : BT);  Var Vsp : U;  Begin  New(Vsp);  Vsp^.Inf := X;  Vsp^.Next := First; {То звено, что было заглавным, становится вторым по счёту}  First := Vsp; {Новое звено становится заглавным}  End; |

**3. Добавление звена в произвольное место списка, отличное от начала (после звена, указатель на которое задан)**

|  |  |
| --- | --- |
| Добавление звена в произвольное место списка, отличное от начала (после звена, на которое ссылается указатель Pred) | {Процедура добавления звена в список после звена,  на которое ссылается указатель Pred;  в x содержится информация для добавления}  Procedure V\_Spisok(Pred : U; X : BT);  Var Vsp : U;  Begin  New(Vsp); {Создаем пустое звено}  Vsp^.Inf := X; {Заносим информацию}  Vsp^.Next := Pred^.Next; {Теперь это звено ссылается на то,  что было следом за звеном Pred}  Pred^.Next := Vsp; {Теперь новое звено встало вслед за звеном Pred}  End; |

**4. Удаление звена из начала списка**

|  |  |
| --- | --- |
| Удаление звена из начала списка | {Процедура удаления звена из начала списка;  в x содержится информация из удалённого звена}  Procedure Iz\_Nachala(Var First : U; Var X : BT);  Var Vsp : U;  Begin  Vsp := First; {Забираем ссылку на текущее заглавное звено}  First := First^.Next; {То звено, что было вторым по счёту, становится заглавным}  X := Vsp^.Inf; {Забираем информацию из удаляемого звена}  Dispose(Vsp); {Уничтожаем звено}  End; |

**5. Удаление звена из произвольного места списка, отличного от начала (после звена, указатель на которое задан)**

|  |  |
| --- | --- |
| Удаление звена из произвольного места списка, отличного от начала (после звена, на которое ссылается указатель Pred) | {Процедура удаления звена из списка после звена,  на которое ссылается указатель Pred;  в x содержится информация из удалённого звена}  Procedure Iz\_Spiska(Pred : U; Var X : BT);  Var Vsp : U;  Begin  Vsp := Pred^.Next; {Забираем ссылку на удаляемое звено}  {Удаляем звено из списка, перенаправив ссылку на следующее  за ним звено}  Pred^.Next := Pred^.Next^.Next;  X := Vsp^.Inf; {Забираем информацию из удаляемого звена}  Dispose(Vsp); {Уничтожаем звено}  End; |

Операция **поиска** элемента в списке заключается в последовательном просмотре всех элементов списка до тех пор, пока текущий элемент не будет содержать заданное значение или пока не будет достигнут конец списка.

Операция **печати** списка заключается в последовательном просмотре всех элементов списка и выводе их значений на экран. Для обработки списка организуется функция, в которой нужно переставлять указатель на следующий элемент списка до тех пор, пока указатель не станет равен NIL, то есть будет достигнут конец списка.

*Пусть необходимо найти сумму элементов списка.*

summa:= 0;

x:= u;

while x<> nil do

begin

summa:= summa+ x^.inf;

x:= x^.next;

end;

*Приведём полный текст модуля.*

{Язык Pascal}

Unit Spisok;

Interface

Type BT = LongInt;

U = ^Zveno;

Zveno = Record Inf : BT; Next: U End;

Procedure V\_Nachalo(Var First : U; X : BT);

Procedure Iz\_Nachala(Var First : U; Var X : BT);

Procedure V\_Spisok(Pred : U; X : BT);

Procedure Iz\_Spiska(Pred : U; Var X : BT);

Procedure Ochistka(Var First: U);

Function Pust(First : U) : Boolean;

Procedure Print(First : U);

Implementation

Procedure V\_Nachalo;

Var Vsp : U;

Begin

New(Vsp);

Vsp^.Inf := X;

Vsp^.Next := First;

First := Vsp;

End;

Procedure Iz\_Nachala;

Var Vsp : U;

Begin

Vsp := First;

First := First^.Next;

X := Vsp^.Inf;

Dispose(Vsp);

End;

Procedure V\_Spisok;

Var Vsp : U;

Begin

New(Vsp);

Vsp^.Inf := X;

Vsp^.Next := Pred^.Next;

Pred^.Next := Vsp;

End;

Procedure Iz\_Spiska;

Var Vsp : U;

Begin

Vsp := Pred^.Next;

Pred^.Next := Pred^.Next^.Next;

X := Vsp^.Inf;

Dispose(Vsp);

End;

Procedure Ochistka;

Var Vsp : BT;

Begin

While Not Pust(First) Do Iz\_Nachala(First, Vsp)

End;

Function Pust;

Begin

Pust := First = Nil

End;

Procedure Print;

Var Vsp : U;

Begin

Vsp := First;

While Vsp <> Nil Do

Begin

Write(Vsp^.Inf : 6);

Vsp := Vsp^.Next

End; WriteLn

End;

Begin

End.

Для ускорения многих операций целесообразно применять переходы между элементами списка в обоих направлениях. Это реализуется с помощью двунаправленных списков, которые являются сложной динамической структурой. Для этого добавим в каждое звено списка еще одно поле, значением которого будет ссылка на предыдущее звено.

**Type U= ^S;**

**S= record**

**Inf: BT;**

**Next: U;**

**Pred: U;**

**End;**

**Двунаправленный (двусвязный) список** – это структура данных, состоящая из последовательности элементов, каждый из которых содержит информационную часть и два указателя на соседние элементы. При этом два соседних элемента должны содержать взаимные ссылки друг на друга.

Свойства двунаправленных списков:

1) по списку можно двигаться в любом направлении;

2) если **List** есть указатель на любой элемент двунаправленного списка, то выполняются следующие свойства

**List = List^.pNext^.pPrev**

**и**

**List = List^.pPrev ^.pNext**

3) возможность легкого исключения узла из списка, в котором он находится, по известному указателю на этот узел. Алгоритмы, в которых требуется исключать узлы из середины списка, встречаются достаточно часто, и как раз этим обстоятельством и объясняется распространенное использование списков с двумя связями.

Описание компоненты списка дадим следующим образом:

**Type**

**PtrRec = ^Rec;**

**Rec = record**

**Element : TypeElement; {поле данных}**

**pNext : PtrRec; {прямой указатель}**

**pPrev : PtrRec; {обратный указатель}**

**End;**

Для формирования списка и работы с ним необходимо описать четыре переменных типа указатель, пусть: **pBegin** определяет начало списка, **pEnd** определяет конец списка, **pCKey**, **pAux** – вспомогательные:

**Var**

**pBegin, pEnd, pCKey, pAux : PtrRec;**

*Рассмотрим процедуру на языке Рascal:*

**Рrocedure Create\_List2 (var pBegin:PtrRec; var pEnd:PtrRec);**

**Var pAux : PtrRec;**

**Elem : TyрeElement;**

**Begin**

**{Создание заглавного звена}**

**New (pBegin);**

**pBegin^.pNext := Nil;**

**pBegin^.pPrev := Nil;**

**{Определяем ссылку на последнее звено}**

**pEnd := pBegin;**

**Elem := Random (6); Write (Elem:4);**

**While Elem <> 0 do**

**Begin**

**New (pEnd^.pNext);**

**pEnd^.pNext^.pPrev := pEnd;**

**pEnd := pEnd^.pNext;**

**pEnd^.pNext := Nil;**

**pEnd^.Element := Elem;**

**Elem := Random (6); Write (Elem:4)**

**End;**

**Writeln**

**End;**

*Приведем соответствующую процедуру вывода списка от его начала:*

**Рrocedure Print\_Forward (pBegin: PtrRec);**

**var pAux : PtrRec;**

**Begin**

**pAux := pBegin^.pNext; {Исключаем из просмотра заглавное звено}**

**If pAux = Nil**

**Then WriteLn ('Двунаправленный список пуст.')**

**Else While pAux <> Nil do**

**Begin**

**Write (pAux^.Element:5);**

**pAux := pAux^.pNext;**

**End;**

**WriteLn;**

**End;**

*вместо оператора* ***Write (pAux^.Element : 5);*** *можно использовать другие операторы, обрабатывающие элементы списка*

*Приведем процедуру прохода по двунаправленному списку от его конца:*

**Рrocedure Print\_Back (pEnd : PtrRec);**

**{pEnd - указатель на "хвост" двунаправленного списка}**

**var pAux: PtrRec;**

**Begin**

**pAux := pEnd;**

**If pAux^.pPrev = Nil**

**Then Writeln ('Двунаправленный список пуст.')**

**else While pAux^.pPrev <> Nil do**

**Begin**

**Write (pAux^.Element : 5);**

**pAux := pAux^.pPrev;**

**end;**

**Writeln**

**End;**

*Приведем не рекурсивную процедуру "очистки" двунаправленного списка:*

Рrocedure **Free (pBegin : PtrRec;** var **pEnd: PtrRec);**

Begin

While **pEnd <> pBegin** do

Begin

**pEnd := pEnd^.pPrev;**

**Disрose (pEnd^.pNext);**

**pEnd^.pNext :=** Nil;

End;

End**;**

*Приведем рекурсивную процедуру "очистки" двунаправленного списка:*

Рrocedure **Free\_Rec (**var **pRing: PtrRec);**

Begin

If **pRing <>** Nil

Then Begin

**Free\_Rec (pRing^.pNext);**

If **pRing <>** Nil

Then Begin

**Disрose (pRing);**

**pRing :=** Nil;

End;

End;

End**;**

Поиск в прямом направлении: от начала двунаправленного списка к его хвосту:

*Приведем функцию, в которой следующие параметры:*

***pBegin*** *- указатель на список (ссылка на заглавное звено);*

***Key*** *- искомый элемент;*

***pCKey*** *- ссылочная переменная, которой присваивается ссылка на первое по порядку звено, содержащее искомый элемент.*

**Function Find\_Forward (pBegin : PtrRec; Key : TypeElement;**

**var pCKey: PtrRec): Boolean;**

**var pAux: PtrRec;**

**Flag : Boolean;**

**Begin**

**Flag := False; pCKey := Nil; {предположим, что не нашли...}**

**pAux := pBegin^.pNext; {исключим заглавное звено из рассмотрения}**

**While (pAux<>Nil) and (not Flag) do**

**begin**

**If pAux^.Element = Key**

**then**

**begin**

**Flag := True;**

**pCKey := pAux;**

**end;**

**pAux := pAux^.pNext**

**end;**

**Find\_Forward := Flag**

**End;**

Поиск в обратном направлении: от хвоста списка к заглавному звену:

*Приведем функцию на языке Рascal, параметры которой:*

***pEnd*** *– указатель на "хвост" списка (ссылка на последнее звено);*

***Key*** *- искомый элемент;*

***pCKey*** *- ссылочная переменная, которой присваивается ссылка на первое от хвоста звено, содержащее искомый элемент.*

**Function Find\_Back (pEnd : PtrRec; Key : TypeElement; var pCKey: PtrRec): Boolean;**

**var pAux : PtrRec;**

**Flag : Boolean;**

**Begin**

**Flag := False; pAux := pEnd; pCKey := Nil;**

**While (pAux<>Nil) and (not Flag) do**

**Begin**

**If pAux^.Element = Key**

**then begin**

**Flag := True;**

**pCKey := pAux**

**end;**

**pAux := pAux^.pPrev;**

**end;**

**Find\_Back := Flag**

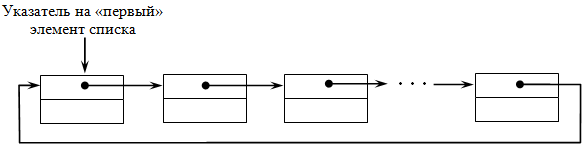
**End;**

**Циклический (кольцевой) список** – это структура данных, представляющая собой последовательность элементов, последний элемент которой содержит указатель на первый элемент списка, а первый (в случае двунаправленного списка) – на последний.

Основная особенность такой организации состоит в том, что в этом списке нет элементов, содержащих пустые указатели, и, следовательно, нельзя выделить крайние элементы.

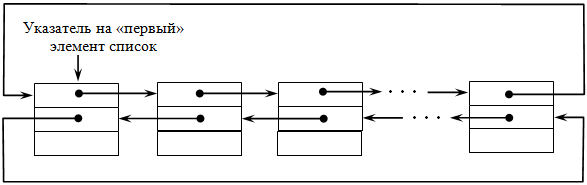
Циклические списки, так же как и линейные, бывают однонаправленными и двунаправленными.

***Циклический однонаправленный список*** *похож на линейный однонаправленный список, но его последний элемент содержит указатель, связывающий его с первым элементом.*



Построение кольцевого списка выполняется так же, как и в случае линейного однонаправленного списка с заглавным звеном, только после окончания ввода элементов кольцевого списка, в поле указателя последнего звена списка помещается адрес звена, следующего за заглавным.

**Циклический двунаправленный список** похож на линейный двунаправленный список, но его любой элемент имеет два указателя, один из которых указывает на следующий элемент в списке, а второй указывает на предыдущий элемент. Это вносит свои коррективы в алгоритм добавления элемента к кольцевому списку.



*Для полного обхода такого списка достаточно иметь* указатель *на произвольный элемент, а не на первый, как в линейном однонаправленном списке. Понятие "первого" элемента здесь достаточно условно и не всегда требуется. Хотя иногда бывает полезно выделить некоторый элемент как "первый" путем установки на него специального указателя. Это требуется, например, для предотвращения "зацикливания" при просмотре списка.*

*Для описания элемента списка будем пользоваться следующей структурой:*

[type](http://pascal.net.ru/type)

**myType =** [**integer**](http://pascal.net.ru/Integer)**;**

**Ring = ^TRing;**

**TRing =** [record](http://pascal.net.ru/record)

**info: myType;**

**next: Ring;**

**prev: Ring;**

[end](http://pascal.net.ru/end)**;**

Добавляться новые элементы будут перед элементом, помеченным как first (то есть, перед "головой" списка) по следующему алгоритму:

1) поскольку выделение памяти под новый элемент должно производиться независимо от того, пуст список или в нем уже есть элементы, то сделаем это в самом начале процедуры добавления, еще перед проверкой на пустоту списка;

2) в зависимости от того, пуст наш кольцевой список или уже частично заполнен, дальнейшее выполнение процедуры пойдет по одному из двух путей:

2а) список пуст – устанавливаем указатели **prev** и **next** на новый элемент и "помечаем" его как первый (First);

2б) список не пуст – действовать нужно по-другому:

- поскольку добавляем в "конец" списка (перед элементом first), то поле **next** нового элемента указывает на "голову" списка, а поле **prev** - туда, куда раньше указывал **prev** "головы";

- не забываем про "бывший последним" элемент (элемент на который раньше указывал **first^.prev**). Поле **next** этого элемента должно указывать на добавляемый элемент, поскольку новый добавляется после "последнего" (перед "первым");

- ну, и наконец, "голова" списка должна теперь указывать **prev**-ом на новый элемент, он ведь находится в списке ПЕРЕД "головой"...

[**procedure**](http://pascal.net.ru/procedure)[**Append**](http://pascal.net.ru/Append)**([var](http://pascal.net.ru/var) First: Ring; value: myType);**

[**var**](http://pascal.net.ru/var) **p: Ring;**

[begin](http://pascal.net.ru/begin)

**{ 1 }**

[**new**](http://pascal.net.ru/New)**(p);**

**p^.info := value;**

[if](http://pascal.net.ru/if) **first =** [nil](http://pascal.net.ru/nil)[then](http://pascal.net.ru/if)[begin](http://pascal.net.ru/begin)

**{ 2а }**

**p^.next := p;**

**p^.prev := p;**

**first := p;**

[end](http://pascal.net.ru/end)

[else](http://pascal.net.ru/else)[begin](http://pascal.net.ru/begin)

**{ 2б }**

**p^.next := first;**

**p^.prev := first^.prev;**

**first^.prev^.next := p;**

**first^.prev := p;**

[end](http://pascal.net.ru/end)**;**

[end](http://pascal.net.ru/end)**;**

При удалении элемента из кольцевого списка также рассматриваем два случая:

1) список содержит один элемент: проверяем, нужно ли удалять этот элемент, и если нужно - просто удаляем его, и обнуляем First, нет элементов - нет "головы" списка;

2) список содержит несколько элементов: переназначаем указатели **prev^.next** и **next^.prev** удаляемого элемента так, чтобы ни предыдущий ни последующий элементы больше на **P** не указывали. После этого проверяем, не удаляется ли "головной" элемент (First), и в случае положительного ответа "продвигаем" First на один элемент вперед (если этого не сделать, то после удаления элемента **P** указатель на "голову" станет невалидным и его использование приведет к получению неправильных результатов). И только после того, как все вышесказанное выполнено, можно удалять элемент списка, на который указывает **P.**

*Удалить из списка, начинающегося с First, элемент, на который указывает P:*

[procedure](http://pascal.net.ru/procedure) **RemoveItem(**[var](http://pascal.net.ru/var) **First: Ring; p: Ring);**

[begin](http://pascal.net.ru/begin)

[if](http://pascal.net.ru/if) **(first^.next = first)** [and](http://pascal.net.ru/DIV) **(p = first)** [then](http://pascal.net.ru/if)

[begin](http://pascal.net.ru/begin)

[**dispose**](http://pascal.net.ru/Dispose)**(first);**

**first :=** [nil](http://pascal.net.ru/nil)**;** [**exit**](http://pascal.net.ru/Exit)**;**

[end](http://pascal.net.ru/end)**;**

**p^.prev^.next := p^.next;**

**p^.next^.prev := p^.prev;**

[if](http://pascal.net.ru/if) **p = first** [then](http://pascal.net.ru/if) **first := p^.next;**

[**dispose**](http://pascal.net.ru/Dispose)**(p); p :=** [nil](http://pascal.net.ru/nil)**;**

[end](http://pascal.net.ru/end)**;**

Отсутствие "последнего" звена приводит к ещё большему упрощению операций добавления и удаления, по сравнению с 1- и 2-связным линейным списком. Например, для 1-связного кольцевого списка в процедуре удаления отсутствует оператор if - проверка на существование звена, следующего за заданным (в кольцевом списке такое звено всегда есть). Такие же операторы отсутствуют в процедурах добавления и удаления звеньев для 2-связного кольцевого списка.

**5. Порядок выполнения работы**

1. Создать линейный однонаправленный список, заполнить список (не менее 10 элементов) любым способом.

2. Составить программу, которая на основе созданного списка сформирует 2 дополнительных списка и заполнит их по следующему алгоритму: в первый дополнительный список поместить все положительные элементы исходного списка, а во второй дополнительный – все отрицательные.

3. Выполнить задание в соответствии с вариантом.

4. Создать линейный двунаправленный список. Заполнить список (не менее 10 элементов).

5. Составить программу для поиска элементов в списке, у которых соседние элементы справа и слева отрицательны. Подсчитать количество таких элементов.

**6. Форма отчета о работе**

*Лабораторная работа № \_\_\_*

*Номер учебной группы \_\_\_\_\_\_\_\_\_\_\_\_*

*Фамилия, инициалы учащегося \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*Дата выполнения работы \_\_\_\_\_\_\_\_\_\_\_\_\_*

*Тема работы: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*Цель работы: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*Оснащение работы: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*Результат выполнения работы: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

**7. Контрольные вопросы и задания**

1. Дайте определение однонаправленного линейного списка.

2. В чем отличия между линейными и кольцевыми списками?

3. Сформулируйте свойства двунаправленных линейных списков.

4. Перечислите основные операции, выполняемые с кольцевыми списками.

**8. Рекомендуемая литература**

1.Ахо, А.В.Структуры данных и алгоритмы / А.В. Ахо, Дж.Хопкрофт, Дж. Д.Ульман. – пер. с англ. – М.: Издательский дом "Вильямс", 2007.-400 с.

2. Вирт,Н. Алгоритмы и структуры данных / Н. Вирт. –СПб.:Невский диалект, 2008. – 352с.

3. Вирт, Н. Алгоритмы и структуры данных с примерами на Паскале / Н. Вирт[перевод с английского Д. Б. Подшивалова]; – 2-е изд., испр., – СПб.: Невский Диалект, 2005. – 352с.

4. Гагарина, Л.Г. Алгоритмы и структуры данных / Л.Г. Гагарина, В.Д. Колдаев. – учеб.пособие – М: Финансы и статистика, 2009. – 304с.

5. Котов, В.М. Алгоритмы и структуры данных: учеб.пособие / В.М. Котов, Е.П. Соболевская, А.А. Толстиков – Минск: БГУ, 2011. – 267с.

6. Макконнелл, Дж. Основы современных алгоритмов / Дж. Макконнелл – 2-е дополненное издание – М.:Техносфера, 2006. – 368с.

7. Окулов, С.М. Программирование в алгоритмах / С.М. Окулов. – 3-е изд. – М.: БИНОМ. Лаборатория знаний, 2007. – 383с.