Тема 3

Структуры данных

(часть 2)

Запись

Запись – агрегат, составляющие которого (поля) имеют имя и могут быть различного типа

```
type
        TPerson = record
                Name: string;
                Address: string;
                Index:  longint;
        end;
var Person: TPerson;
```

Можно выполнять операции над отдельным полем

```
Person.Index := 117997;
Person.Name := 'РЭУ им. Плеханова';
Person.Adress := 'Москва, Стремянный переулок, д.36';
```

Операция присваивания для записей в целом – записи должны быть одного типа

```
Person_List[24] := Person;
```


Данные типа указатель

Тип указателя представляет собой адрес ячейки памяти

В языках программирования высокого уровня определена специальная константа nil, которая означает пустой указатель или указатель, не содержащий какой-либо конкретный адрес

Указатели применяются:

- при необходимости представить одну и ту же область памяти (одни и те же физические данные) как данные разной логической структуры
- при работе с динамическими структурами данных

Могут быть типизированными и нетипизированными

```
var                    pointer
ipt: ^integer;
cpt: ^char;
```


Основные операции:

- **Присваивание** – двухместная операция, оба операнда – указатели
  Если оба указателя, участвующие в операции присваивания типизированные, то оба должны указывать на данные одного и того же типа
- **Получение адреса** – одноместная, ее операнд может иметь любой тип
  Результат – типизированный указатель (в соответствии с типом операнда), содержащий адрес операнда
- **Выборка** – одноместная, ее операндом является типизированный указатель
  Результат – данные, выбранные из памяти по адресу, заданному операндом. Тип результата определяется типом указателя

Линейные списки

Список – структура данных, представляющая собой логически связанную последовательность элементов списка

Динамическая структура данных – структура данных, определяющие характеристики которой могут изменяться на протяжении ее существования

Отличие динамических структур от статических:

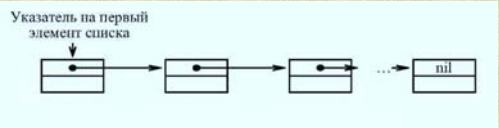
- в них нельзя обеспечить хранение в заголовке всей информации о структуре, поскольку каждый элемент должен содержать информацию, логически связывающую его с другими элементами структуры
- для них зачастую неудобно использовать единый массив смежных элементов памяти, поэтому необходимо предусматривать какую-либо схему динамического управления памятью

Процедура создания
`New(Current);`

Процедура освобождения
`Dispose(Current);`

Линейный однонаправленный список

В этом списке любой элемент имеет один указатель, который указывает на следующий элемент в списке или является пустым указателем у последнего элемента



При выполнении любых операций с линейным однонаправленным списком необходимо обеспечивать позиционирование какого-либо указателя на первый элемент

В противном случае часть или весь список будет недоступен

Операции:

- вставка элемента
- просмотр
- поиск
- удаление элемента

```
type
        PElement = ^TypeElement; {указатель на тип элемента}
        TypeElement = record {тип элемента списка}
                Data: TypeData; {поле данных элемента}
                Next: PElement; {поле указателя на следующий элемент}
        end;
var
ptrHead: PElement; {указатель на первый элемент списка}
ptrCurrent: PElement; {указатель на текущий элемент}
```

Процедуры вставки InsFirst_LineSingleList и Ins_LineSingleList

Входные параметры	Выходные параметры
• данные для заполнения создаваемого элемента • указатель на начало списка • указатель на текущий элемент в списке (при необходимости)	• указатель на начало списка • указатель текущего элемента (показывает на вновь созданный элемент, при вставке первого элемента указателем на него будет указатель на заголовок списка)


```
procedure Ins_LineSingleList(DataElem: TypeData;
                                var ptrHead, ptrCurrent: PElement);
{Вставка непервого элемента в линейный однонаправленный список}
{справа от элемента, на который указывает ptrCurrent}
var ptrAddition: PElement; {вспомогательный указатель}
begin
        New(ptrAddition);
        ptrAddition^.Data := DataElem;
        if ptrHead = nil then
                begin {список пуст, создаем первый элемент списка}
                        ptrAddition^.Next := nil;
                        ptrHead := ptrAddition;
                end
                        else
                begin
                {список не пуст, вставляем элемент списка справа от
                элемента, на который указывает ptrCurrent}
                        ptrAddition^.Next := ptrCurrent^.Next;
                        ptrCurrent^.Next := ptrAddition;
                end;
        ptrCurrent := ptrAddition;
end;
```



```
procedure InsFirst_LineSingleList(DataElem: TypeData;
                                        var ptrHead: PElement);
{Вставка первого элемента в линейный однонаправленный список}
var ptrAddition: PElement; {вспомогательный указатель}
begin
        New(ptrAddition);
        ptrAddition^.Data := DataElem;
        if ptrHead = nil then
                begin {список пуст, создаем первый элемент списка}
                        ptrAddition^.Next := nil;
                end
                        else
                begin {список не пуст - вставляем новый элемент слева
                        (перед) первым элементом}
                        ptrAddition^.Next := ptrHead;
                end;
        ptrHead := ptrAddition;
end;
```

При неправильном переопределении указателей возможен разрыв списка или потери указателя на первый элемент, что приводит к потере доступа к части или всему списку

Операция просмотра списка

```
procedure Scan_LineSingleList(ptrHead: PElement);
{Просмотр линейного однонаправленного списка}
var ptrAddition: PElement; {вспомогательный указатель}
begin
        ptrAddition := ptrHead;
        while ptrAddition <> nil do
        begin {пока не конец списка}
                writeln(ptrAddition^.Data); {Вывод значения элемента}
                ptrAddition := ptrAddition^.Next;
        end;
end;
```

Операция поиска элемента в списке

Входные параметры	Выходной параметр
• значение, которое должен содержать искомый элемент • указатель на список	• указатель, который устанавливается на найденный элемент или остается без изменений, если элемента в списке нет

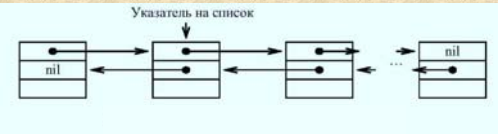

```
function Find_LineSingleList(DataElem: TypeData;
var ptrHead, ptrCurrent: PElement): boolean;
{Поиск элемента в линейном однонаправленном списке}
var ptrAddition:PElement; {Дополнительный указатель}
begin
  if (ptrHead <> nil) then
    begin {Если существует список}
        ptrAddition := ptrHead;
        while (ptrAddition <> nil) and (ptrAddition^.Data <> DataElem) do
        {пока не конец списка и не найден элемент}
        ptrAddition := ptrAddition^.Next;
        {Если элемент найден, то результатом работы функции
        является true}
        if ptrAddition <> nil then
                begin
                        Find_LineSingleList := true;
                        ptrCurrent := ptrAddition;
                end
                        else
                begin Find_LineSingleList := false; end;
    end
                        else
                begin Find_LineSingleList:= false; end;
  end;
```


Операция удаления элемента

```
procedure Del_LineSingleList(var ptrHead, ptrCurrent: PElement);
{Удаление элемента из линейного однонаправленного списка}
var ptrAddition: PElement; {вспомогательный указатель}
begin
        if ptrCurrent <> nil then
                begin {вх.параметр корректен}
                        if ptrCurrent = ptrHead then
                                begin {удаляем первый}
                                ptrHead := ptrHead^.Next;
                                dispose(ptrCurrent);
                                ptrCurrent := ptrHead;
                                end
                                        else
                                begin {удаляем непервый}
{устанавливаем вспомогательный указатель на элемент, предшествующий
удаляемому}                     ptrAddition := ptrHead;
                                while ptrAddition^.Next <> ptrCurrent do
                                ptrAddition := ptrAddition^.Next;
                                {непосредственное удаление элемента}
                                ptrAddition^.Next := ptrCurrent^.Next;
                                dispose(ptrCurrent);
                                ptrCurrent := ptrAddition;
                                end;
                end;
end;
```

Линейный двунаправленный список

В этом списке любой элемент имеет два указателя, один из которых указывает на следующий элемент в списке или является пустым указателем у последнего элемента, а второй – на предыдущий элемент в списке или является пустым указателем у первого элемента



В отличие от однонаправленного списка здесь нет необходимости обеспечивать позиционирование какого-либо указателя именно на первый элемент списка, так как благодаря двум указателям в элементах можно получить доступ к любому элементу списка из любого другого элемента, осуществляя переходы в прямом или обратном направлении

Операции:

- вставка элемента
- просмотр
- поиск
- удаление элемента

```
type
PElement = ^TypeElement; {указатель на тип элемента}
TypeElement = record {тип элемента списка}
        Data: TypeData; {поле данных элемента}
        Next, {поле указателя на следующий элемент}
        Last: PElement; {поле указателя на предыдущий элемент}
end;
var
ptrHead: PElement; {указатель на первый элемент списка}
ptrCurrent: PElement; {указатель на текущий элемент}
```

Операция вставки реализовывается с помощью двух процедур, аналогичных процедурам вставки для линейного однонаправленного списка:
InsFirst_LineDubleList и Ins_LineDubleList
Однако при вставке последующего элемента придется учитывать особенности добавления элемента в конец списка


```
procedure Ins_LineDubleList(DataElem: TypeData; var ptrHead, ptrCurrent: PElement);
{Вставка непервого элемента в линейный двунаправленный список}
{справа от элемента, на который указывает ptrCurrent}
var ptrAddition: PElement; {вспомогательный указатель}
begin
        New(ptrAddition);
        ptrAddition^.Data := DataElem;
        if ptrHead = nil then
                begin {список пуст - создаем первый элемент списка}
                        ptrAddition^.Next := nil;
                        ptrAddition^.Last := nil;
                        ptrHead := ptrAddition;
                end
                        else
                begin
{список не пуст - вставляем элемент списка справа от элемента,}
{на который указывает ptrCurrent}
                        if ptrCurrent^.Next <> nil then {вставляем не последний}
                        ptrCurrent^.Next^.Last := ptrAddition;
                        ptrAddition^.Next := ptrCurrent^.Next;
                        ptrCurrent^.Next := ptrAddition;
                        ptrAddition^.Last := ptrCurrent;
                end;
        ptrCurrent := ptrAddition;
end;
```



```
procedure InsFirst_LineDubleList(DataElem: TypeData; var ptrHead:
PElement);
{Вставка первого элемента в линейный двунаправленный список}
var ptrAddition: PElement; {вспомогательный указатель}
begin
        New(ptrAddition);
        ptrAddition^.Data := DataElem;
        ptrAddition^.Last := nil;
        if ptrHead = nil then
                begin {список пуст - создаем первый элемент списка}
                        ptrAddition^.Next := nil;
                end
                        else
                begin
{список не пуст - вставляем новый элемент слева (перед) первым
элементом}
                        ptrAddition^.Next := ptrHead;
                        ptrHead^.Last := ptrAddition;
                end;
        ptrHead := ptrAddition;
end;
```


Операция удаления элемента

```
procedure Del_LineDubleList(var ptrHead, ptrCurrent: PElement);
{Удаление элемента из линейного двунаправленного списка}
var ptrAddition: PElement; {вспомогательный указатель}
begin
  if ptrCurrent <> nil then
        begin {входной параметр корректен}
                if ptrCurrent = ptrHead then
                  begin {удаляем первый}
                        ptrHead := ptrHead^.Next;
                        dispose(ptrCurrent);
                        ptrHead^.Last := nil;
                        ptrCurrent := ptrHead;
                end
                        else
                begin {удаляем непервый}
                  if ptrCurrent^.Next = nil then
                        begin {удаляем последний}
                                ptrCurrent^.Last^.Next := nil;
                                dispose(ptrCurrent);
                                ptrCurrent := ptrHead;
                        end
                                        else
                        begin {удаляем непоследний и непервый}
                                ptrAddition := ptrCurrent^.Next;
                                ptrCurrent^.Last^.Next := ptrCurrent^.Next;
                                ptrCurrent^.Next^.Last := ptrCurrent^.Last;
                                dispose(ptrCurrent);
                                ptrCurrent := ptrAddition;
                        end;
                end;
        end;
end;
```


Циклические списки

Линейные списки	Циклические списки
можно выделить первый и последний элементы имеют пустые указатели обязательно – указатель на начало списка алгоритмы вставки и удаления крайних и средних элементов списка отличаются друг от друга	нет элементов, содержащих пустые указатели нельзя выделить крайние элементы все элементы являются «средними»

Циклический однонаправленный список

Похож на линейный однонаправленный список, но его последний элемент содержит указатель, связывающий его с первым элементом

Для полного обхода такого списка достаточно иметь указатель на произвольный элемент

Полезно выделить некоторый элемент как «первый» путем установки на него специального указателя

Операции:

- вставка элемента
- просмотр
- поиск
- удаление элемента

Объявления данных – те же, что и для линейного однонаправленного списка

```
type
PElement = ^TypeElement; {указатель на тип элемента}
TypeElement = record {тип элемента списка}
        Data: TypeData; {поле данных элемента}
        Next, {поле указателя на следующий элемент}
        Last: PElement; {поле указателя на предыдущий элемент}
end;
var
ptrHead: PElement; {указатель на первый элемент списка}
ptrCurrent: PElement; {указатель на текущий элемент}
```

Процедура вставки Ins_CicleSingleList	
Входные параметры	Выходные параметры
• данные для заполнения создаваемого элемента • указатель на начало списка • указатель на текущий элемент в списке, после которого осуществляется вставка	• указатель на начало списка (может измениться) • указатель текущего элемента (показывает на вновь созданный элемент)

Операция вставки элемента

```
procedure Ins_CicleSingleList(DataElem: TypeData; var ptrHead, ptrCurrent: PElement);
{Вставка элемента в циклический однонаправленный список}
{справа от элемента, на который указывает ptrCurrent}
var ptrAddition: PElement; {вспомогательный указатель}
begin
        New(ptrAddition);
        ptrAddition^.Data := DataElem;
        if ptrHead = nil then
          begin {список пуст – создаем первый элемент списка}
                ptrAddition^.Next := ptrAddition; {цикл из 1 элемента}
                ptrHead := ptrAddition;
          end
                        else
          begin {список не пуст – вставляем элемент списка справа от элемента,}
          {на который указывает ptrCurrent}
                ptrAddition^.Next := ptrCurrent^.Next;
                ptrCurrent^.Next := ptrAddition;
          end;
  ptrCurrent := ptrAddition;
end;
```

Порядок следования операторов присваивания процедуры очень важен

При неправильном переопределении указателей возможен разрыв списка или потеря указателя на первый элемент, что приводит к потере доступа к части или всему списку

Операция просмотра списка

В отличие от линейного однонаправленного списка здесь признаком окончания просмотра списка будет возврат к элементу, выделенному как «первый»

```
procedure Scan_CicleSingleList(ptrHead: PElement);
{Просмотр циклического однонаправленного списка}
var ptrAddition: PElement; {вспомогательный указатель}
begin
  if ptrHead <> nil do
  begin {список не пуст}
        ptrAddition := ptrHead;
        repeat
                writeln(ptrAddition^.Data); {Вывод значения элемента}
                ptrAddition := ptrAddition^.Next;
        until ptrAddition = ptrHead;
  end;
end;
```


Операция поиска элемента

Последовательный просмотр всех элементов списка до тех пор, пока текущий элемент не будет содержать заданное значение или пока не достигнут «первый» элемент списка (фиксируется отсутствие искомого элемента в списке, функция принимает значение false)

Входные параметры	Выходной параметр
• значение, которое должен содержать искомый элемент • указатель на список	• указатель, который устанавливается на найденный элемент или остается без изменений, если элемента в списке нет


```
function Find_CicleSingleList(DataElem: TypeData;
                              var ptrHead, ptrCurrent: PElement): boolean;
{Поиск в циклическом однонаправленном списке}
var ptrAddition: PElement; {вспомогательный указатель}
begin
  if ptrHead <> nil do
  begin {список не пуст}
        ptrAddition := ptrHead;
        repeat
                ptrAddition := ptrAddition^.Next;
        until (ptrAddition = ptrHead) or {прошли весь список}
                                (ptrAddition^.Data = DataElem) {элемент найден}
        if ptrAddition^.Data = DataElem then
        begin
                Find_CicleSingleList := true;
                ptrCurrent := ptrAddition;
        end
                                else
        begin Find_CicleSingleList := false; end;
  end
                else
  begin
        Find_CicleSingleList := false;
  end;
end;
```

Операция удаления элемента

Удаляет элемент, на который установлен указатель текущего элемента

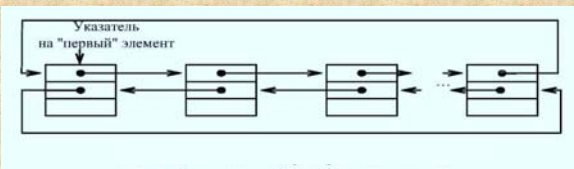
После удаления указатель текущего элемента устанавливается на элемент списка, следующий за удаляемым

В случае удаления «первого» элемента необходимо соответствующий указатель переместить на следующий элемент


```
procedure Del_CicleSingleList(var ptrHead, ptrCurrent: PElement);
{Удаление элемента из циклического однонаправленного списка}
var ptrAddition: PElement; {дополнительный указатель}
begin
  if ptrCurrent <> nil then
  begin {входной параметр корректен}
        if ptrHead^.Next <> ptrHead then
                begin {Если удаляемый элемент не единственный в списке}
                {устанавливаем вспомогательный указатель на элемент,
                предшествующий удаляемому}
                        ptrAddition := ptrHead;
                        while ptrAddition^.Next <> ptrCurrent do
                                ptrAddition := ptrAddition^.Next;
                                {непосредственное удаление элемента}
                                ptrAddition^.Next := ptrCurrent^.Next;
                                if ptrHead = ptrCurrent then {удаляем первый}
                                        ptrHead := ptrCurrent^.Next;
                                dispose(ptrCurrent);
                                ptrCurrent := ptrAddition^.Next;
                                end
                                        else
                                begin
                                        ptrHead:=nil;
                                        dispose(ptrCurrent);
                                        ptrCurrent:=nil;
                                end;
                end;
end;
```


Циклический двунаправленный список

Любой элемент имеет два указателя, один из которых указывает на следующий элемент в списке, а второй – на предыдущий элемент



Операции:
- вставка элемента
- просмотр
- поиск
- удаление элемента

Объявления данных – те же, что и для линейного двунаправленного списка

```
type
PElement = ^TypeElement; {указатель на тип элемента}
TypeElement = record {тип элемента списка}
  Data: TypeData; {поле данных элемента}
  Next, {поле указателя на следующий элемент}
  Last: PElement; {поле указателя на предыдущий элемент}
end;
var
ptrHead: PElement; {указатель на первый элемент списка}
ptrCurrent: PElement; {указатель на текущий элемент}
```


Операция вставки элемента Ins_CicleDubleList

Входные параметры	Выходные параметры
данные для заполнения создаваемого элемента указатель на начало списка указатель на текущий элемент в списке, после которого осуществляется вставка	указатель на начало списка (может измениться) указатель текущего элемента (показывает на вновь созданный элемент)

```
procedure Ins_CicleDubleList(DataElem: TypeData; var ptrHead, ptrCurrent: PElement);
{Вставка элемента в циклический двунаправленный список
справа от элемента, на который указывает ptrCurrent}
var ptrAddition: PElement; {вспомогательный указатель}
begin
  New(ptrAddition);
  ptrAddition^.Data := DataElem;
  if ptrHead = nil then
  begin {список пуст – создаем первый элемент списка}
        ptrAddition^.Next := ptrAddition; {петля из 1 элемента}
        ptrAddition^.Last := ptrAddition;
        ptrHead := ptrAddition;
  end
                else
  begin {список не пуст – вставляем элемент списка справа от элемента.}
        {на который указывает ptrCurrent}
        ptrAddition^.Next := ptrCurrent^.Next;
        ptrCurrent^.Next := ptrAddition;
        ptrAddition^.Last := ptrCurrent;
        ptrAddition^.Next^.Last := ptrAddition;
  end;
ptrCurrent := ptrAddition;
end;
```

При неправильном переопределении указателей возможен разрыв списка или потеря указателя на первый элемент, что приводит к потере доступа к части или всему списку

Операция просмотра и операция поиска для циклического двунаправленного списка реализуются абсолютно аналогично соответствующим процедурам для циклического однонаправленного списка
Просматривать и искать здесь можно в обоих направлениях

Операция удаления элемента

```
procedure Del_CicleDubleList(var ptrHead, ptrCurrent: PElement);
{Удаление элемента из циклического двунаправленного списка}
var ptrAddition: PElement; {дополнительный указатель}
begin
  if ptrCurrent <> nil then
  begin {входной параметр корректен}
        if ptrHead^.Next <> ptrHead then
                begin {Если удаляемый элемент, не единственный в списке}
                        ptrAddition := ptrCurrent^.Next;
                        ptrCurrent^.Last^.Next := ptrCurrent^.Next;
                        ptrCurrent^.Next^.Last := ptrCurrent^.Last;
                        if ptrHead = ptrCurrent then {удаляем первый}
                                        ptrHead := ptrCurrent^.Next;
                        dispose(ptrCurrent);
                        ptrCurrent := ptrAddition;
                end
                        else
                begin
                        ptrHead:=nil;
                        dispose(ptrCurrent);
                        ptrCurrent:=nil;
                end;
  end;
end;
```

Использование двух указателей позволяет ускорить операции, связанные с передвижением по списку за счет двунаправленности этого движения

Разреженные матрицы

Разреженная матрица – двумерный массив, большинство элементов которого равны между собой, так что хранить в памяти достаточно лишь небольшое число значений, отличных от основного (фонового) значения остальных элементов

Разреженные матрицы

- **Матрицы, в которых местоположения элементов со значениями, отличными от фонового, могут быть математически описаны**
- **Матрицы со случайным расположением элементов**

Вопросы размещения в памяти реализуются с учетом типа матриц

Матрицы с математическим описанием местоположения элементов

В расположении элементов есть какая-либо закономерность

Элементы, значения которых являются фоновыми, называют нулевыми
Элементы, значения которых отличны от фонового, называют ненулевыми
(фоновое значение не всегда равно нулю)

Ненулевые значения хранятся, как правило, в одномерном массиве (векторе), а связь между местоположением в разреженной матрице и в новом, одномерном, описывается математически с помощью формулы, преобразующей индексы матрицы в индексы вектора

Для работы разрабатываются функции

- Для преобразования индексов матрицы в индекс вектора
- Для получения значения элемента матрицы из ее упакованного представления по двум индексам (строка, столбец)
- Для записи значения элемента матрицы в ее упакованное представление по двум индексам

Пример
Имеется двумерная разреженная матрица, в которой все ненулевые элементы расположены в шахматном порядке, начиная со второго элемента

Формула вычисления индекса элемента в линейном представлении:

$$I = ((y - 1) \cdot X_m + x) / 2$$

где
I – индекс в линейном представлении;
x, y – индексы соответственно строки и столбца в двумерном представлении;
X_m – количество элементов в строке исходной матрицы

Матрицы со случайным расположением элементов

Матрицы, у которых местоположение элементов со значениями, отличными от фонового, не могут быть математически описано – в их расположении нет какой-либо закономерности

Пусть в матрице A размерности 5x7 из 35 элементов только 10 отличны от нуля

0	0	6	0	9	0	0
2	0	0	7	8	0	4
10	0	0	0	0	0	0
0	0	12	0	0	0	0
0	0	0	3	0	0	5

Способ хранения – запоминание ненулевых элементов в одномерном массиве записей – последовательное представление разреженных матриц

Row (*i*)	Column (j)	Value
1	3	6
1	5	9
2	1	2
2	4	7
2	5	8
2	7	4
3	1	10
4	3	12
5	4	3
5	7	5

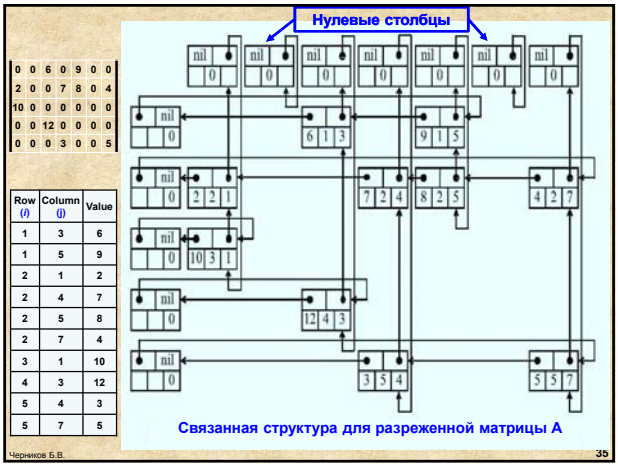
Row (*i*)	Column (j)	Value
1	3	6
1	5	9
2	1	2
2	4	7
2	5	8
2	7	4
3	1	10
4	3	12
5	4	3
5	7	5

Для ускорения поиска элементы матрицы обязательно запоминаются в порядке возрастания номеров строк

Включение и исключение новых элементов матрицы вызывает необходимость перемещения большого числа существующих элементов

Если включение новых элементов и их исключение осуществляется часто, то можно использовать метод связанных структур – переводит статическую структуру матрицы в динамическую, реализуемую в виде циклических списков

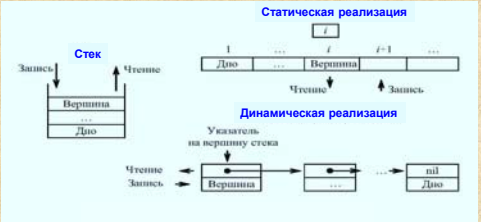
```
type
    PElement = ^TypeElement; {указатель на тип элемента}
    TypeElement = record {тип элемента списка}
        Left: PElement; {указатель на предыдущий элемент в строке}
        Up: PElement; {указатель на предыдущий элемент в столбце}
        Value: TypeData; {значение элемента матрицы}
        Row: integer; {индекс строки матрицы}
        Colum: integer; {индекс столбца матрицы}
    end;
```

Стек

Стек – структура данных, в которой новый элемент всегда записывается в ее начало (вершину) и очередной читаемый элемент также всегда выбирается из ее начала
Используется принцип «последним пришел – первым вышел»
(**LIFO:** Last Input – First Output)

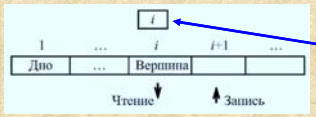
Стек можно реализовывать
- как статическую структуру данных в виде одномерного массива
- как динамическую структуру – в виде линейного списка



Статическая реализация

Необходимо резервировать массив, длина которого равна максимально возможной глубине стека:

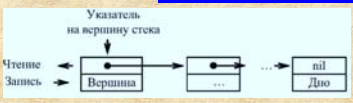
- неэффективное использование памяти
- работать с такой реализацией проще и быстрее



Необходимо отдельно хранить значение индекса элемента массива, являющегося вершиной стека

Можно обойтись без отдельного хранения индекса, если в качестве вершины стека всегда использовать первый элемент массива, но в этом случае, при записи или чтении из стека, необходимо будет осуществлять сдвиг всех остальных элементов, что приводит к дополнительным затратам вычислительных ресурсов

Динамическая реализация



Можно организовать на основе линейного списка

Достаточно хранить указатель вершины стека, который указывает на первый элемент списка

Если стек пуст, то списка не существует и указатель принимает значение nil

При описании динамической реализации используем определения и операции, приведенные в линейных однонаправленных списках:

```
type
        PElement = ^TypeElement; {указатель на тип элемента}
        TypeElement = record {тип элемента списка}
                Data: TypeData; {поле данных элемента}
                Next: PElement; {поле указателя на следующий элемент}
        end;
var
ptrHead: PElement; {указатель на первый элемент списка}
ptrCurrent: PElement; {указатель на текущий элемент}
```


Операции:

- записать (положить в стек)
- прочитать (снять со стека)
- очистить стек
- проверка пустоты стека

```
procedure PushStack(NewElem: TypeData; var ptrStack: PElement);
{Запись элемента в стек (положить в стек)}
begin
        InsFirst_LineSingleList(NewElem, ptrStack);
end;
procedure PopStack(var NewElem: TypeData, ptrStack: PElement);
{Чтение элемента из стека (снять со стека)}
begin
  if ptrStack <> nil then
        begin
                NewElem := ptrStack^.Data;
                Del_LineSingleList(ptrStack, ptrStack); {удаляем вершину}
        end;
end;
procedure ClearStack(var ptrStack: PElement); {Очистка стека}
begin
        while ptrStack <> nil do
                Del_LineSingleList(ptrStack, ptrStack); {удаляем вершину}
end;
function EmptyStack(var ptrStack: PElement): boolean; {Проверка пустоты стека}
begin
        if ptrStack = nil then EmptyStack := true
                          else EmptyStack := false;
end;
```

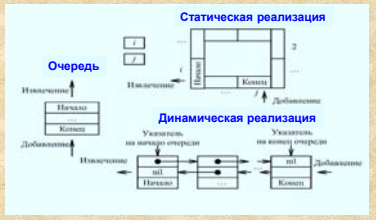

Очередь

Очередь – структура данных, представляющая собой последовательность элементов, образованная в порядке их поступления

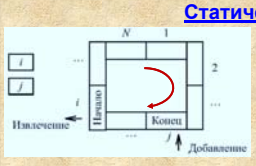
Используется принцип «первым пришел – первым вышел»
(**FIFO:** First Input – First Output)

Очередь можно реализовывать

- как статическую структуру данных в виде одномерного массива
- как динамическую структуру – в виде линейного списка



Статическая реализация



Необходимо резервировать массив, длина которого равна максимально возможной длине очереди, что приводит к неэффективному использованию памяти

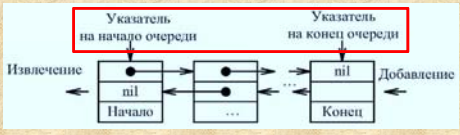
Начало очереди располагается в первом элементе массива, а рост очереди осуществляется в сторону увеличения индексов

С течением времени будет происходить миграция элементов очереди из начала массива в сторону его конца. Это может привести к быстрому исчерпанию массива и невозможности добавлении новых элементов в очередь при наличии свободных мест в начале массива

Способы борьбы с этим:

- после извлечения очередного элемента из начала очереди осуществлять сдвиг всей очереди на один элемент к началу массива
- представить массив в виде циклической структуры, где первый элемент массива следует за последним

Динамическая реализация



Предпочтительно использовать линейный двунаправленный список

```
type
PElement = ^TypeElement; {указатель на тип элемента}
TypeElement = record {тип элемента списка}
        Data: TypeData; {поле данных элемента}
        Next, {поле указателя на следующий элемент}
        Last: PElement; {поле указателя на предыдущий элемент}
end;
var
ptrHead: PElement; {указатель на первый элемент списка}
ptrCurrent: PElement; {указатель на текущий элемент}
ptrBeginQueue, ptrEndQueue: PElement;
                {указатели на начало и конец очереди}
```

Операции:

- добавить элемент
- извлечь элемент
- очистить очередь
- проверка пустоты очереди

```
procedure InQueue(NewElem: TypeData; var ptrBeginQueue, ptrEndQueue: PElement);
{Добавление элемента в очередь}
begin Ins_LineDoubleList(NewElem, ptrBeginQueue, ptrEndQueue); end;

procedure FromQueue(var NewElem: TypeData;var ptrBeginQueue: PElement);
{Извлечение элемента из очереди}
begin
        if ptrBeginQueue <> nil then
                begin
                        NewElem := ptrEndQueue^.Data;
                        Del_LineDoubleList(ptrBeginQueue, ptrBeginQueue);
                end;
end;

procedure ClearQueue(var ptrBeginQueue, ptrEndQueue: PElement);
{Очистка очереди}
begin
        while ptrBeginQueue <> nil do
                Del_LineDoubleList(ptrBeginQueue, ptrBeginQueue);
        ptrEndQueue := nil;
end;

function EmptyQueue(var ptrBeginQueue: PElement): boolean;
{Проверка пустоты очереди}
begin
        if ptrBeginQueue = nil then EmptyQueue := true
                        else EmptyQueue := false;
end;
```

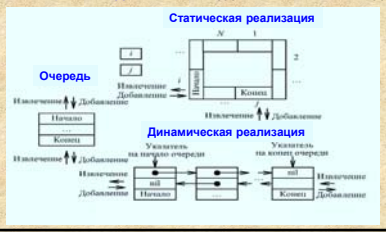

Дек

Дек – структура данных, представляющая собой последовательность элементов, в которой можно добавлять и удалять в произвольном порядке элементы с двух сторон

Первый и последний элементы дека соответствуют входу и выходу дека

Ограниченные деки:

- дек с ограниченным входом – из конца дека можно только извлекать элементы
- дек с ограниченным выходом – в конец дека можно только добавлять элементы



В деке, как и в очереди, осуществляется работа с обоими концами структуры, поэтому используются те же подходы к организации дека, что применялись и для очереди

Описание элементов дека аналогично описанию элементов линейного двунаправленного списка, но также с дополнением

```
type
PElement = ^TypeElement; {указатель на тип элемента}
TypeElement = record {тип элемента списка}
  Data: TypeData; {поле данных элемента}
  Next, {поле указателя на следующий элемент}
  Last: PElement; {поле указателя на предыдущий элемент}
end;
var
ptrHead: PElement; {указатель на первый элемент списка}
ptrCurrent: PElement; {указатель на текущий элемент}
ptrBeginDeck, ptrEndDeck: PElement; {указатели на начало и конец дека}
```

Операции:

- добавить элемент в начало
- добавить элемент в конец
- извлечь элемент из начала
- извлечь элемент из конца
- очистить дек
- проверка пустоты дека

Операции добавления элементов в начало и конец, удаления элемента из начала

```
procedure InBeginDeck(NewElem: TypeData; var ptrBeginDeck: PElement);
{Добавление элемента в начало дека}
begin
        InsFirst_LineDoubleList(NewElem, ptrBeginDeck);
end;

procedure InEndDeck(NewElem: TypeData; var ptrBeginDeck, ptrEndDeck: PElement);
{Добавление элемента в конец дека}
begin
        Ins_LineDoubleList(NewElem, ptrBeginDeck, ptrEndDeck);
end;

procedure FromBeginDeck(NewElem: TypeData; var ptrBeginDeck: PElement);
{Извлечение элемента из начала дека}
  begin
    if ptrBeginDeck <> nil then
        begin
                NewElem := ptrBeginDeck^.Data;
                Del_LineDoubleList(ptrBeginDeck, ptrBeginDeck); {удаляем первый}
        end;
end;
```

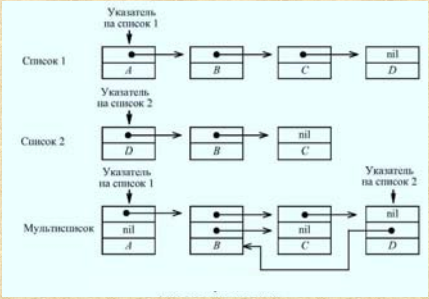

Операции удаления элементов из конца, очистки дека и проверки его на пустоту

```
procedure FromEndDeck(NewElem: TypeData, var ptrBeginDeck, ptrEndDeck:
PElement);
{Извлечение элемента из конца дека}
begin
  if ptrBeginDeck <> nil then
        begin
                NewElem := ptrEndDeck^.Data;
                Del_LineDoubleList(ptrBeginDeck, ptrEndDeck); {удаляем конец}
        end;
end;

procedure ClearDeck(var ptrBeginDeck: PElement);
{Очистка дека}
begin
        while ptrBeginDeck <> nil do
                Del_LineDoubleList(ptrBeginDeck, ptrBeginDeck);
        ptrEndDeck := nil;
end;

function EmptyDeck(var ptrBeginDeck: PElement): boolean;
{Проверка пустоты дека}
begin
        if ptrBeginDeck = nil then EmptyDeck := true
                        else EmptyDeck := false;
end;
```


НЕЛИНЕЙНЫЕ СТРУКТУРЫ ДАННЫХ

Мультисписки

Мультисписок – структура данных, состоящая из элементов, содержащих такое число указателей, которое позволяет организовать их одновременно в виде нескольких различных списков



Использование мультисписков позволяет устранить нерациональное использование памяти из-за дублирования динамических элементов, хранящих повторяющиеся данные

Поиск в мультисписке аналогичен поиску в линейном списке, но при этом используется только один указатель, соответствующий списку, в котором осуществляется поиск

Добавление элемента, принадлежащего только одному из списков, аналогично добавлению в линейный список, за исключением того, что поля указателей, относящиеся к другим спискам, устанавливаются в *nil*

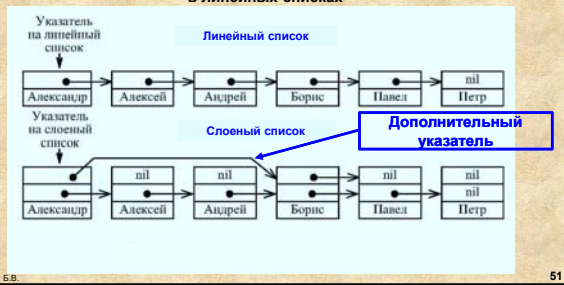
При **добавлении элемента, принадлежащего сразу нескольким спискам**, необходимо аккуратно осуществлять определение значений соответствующих указателей

Алгоритм выполнения такой операции сильно зависит от количества списков и места вставки нового элемента

Слоеные списки

Слоеные (skip), или разделенные, списки – связные списки, которые позволяют перескакивать через некоторое количество элементов

Это позволяет преодолеть ограничения последовательного поиска, являющейся основной причиной низкой эффективности поиска в линейных списках



Графы

Граф G – упорядоченная пара (V, E), где *V* – непустое множество вершин, *E* – множество пар элементов множества V, называемое множеством ребер

Упорядоченная пара элементов из V называется дугой
Если все пары в E упорядочены, то граф называется ориентированным (орграфом)

Путь – любая последовательность вершин орграфа такая, что в этой последовательности вершина b может следовать за вершиной a, только если существует дуга, следующая из a в b

Путь, начинающийся и заканчивающийся в одной и той же вершине, называется циклом

Граф, в котором отсутствуют циклы, называется ациклическим

Петля – дуга, соединяющая некоторую вершину сама с собой

Деревья

Деревом называется орграф, для которого:
- существует узел, в который не входит ни одной дуги (корень)
- в каждую вершину, кроме корня, входит одна дуга

Корень – вершина, в которую не входит ни одной дуги;
Узлы – вершины, в которые входит одна дуга и выходит одна или более дуг
Листья – вершины, в которые входит одна дуга и не выходит ни одной дуги

Все вершины, в которые входят дуги, исходящей из одной вершины, называются потомками этой вершины, а сама вершина – предком

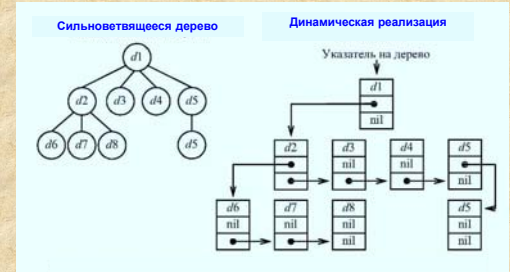
Поддеревом называется вершина со всеми ее потомками

Степенью вершины в дереве называется количество дуг, которое из нее выходит. Степень дерева равна максимальной степени вершины, входящей в дерево. Листьями в дереве являются вершины, имеющие степень нуль

По степени дерева

Двоичные – степень дерева не более двух

Сильноветвящиеся – степень дерева произвольная



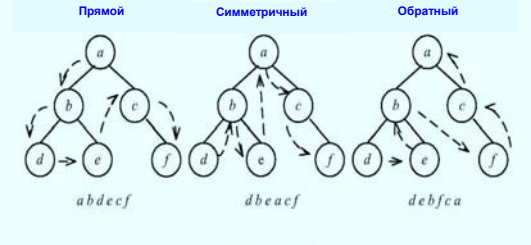
Списочное представление деревьев произвольной степени основано на элементах, соответствующих вершинам дерева

Каждый элемент имеет поле данных и два поля указателей:
- указатель на начало списка потомков вершины
- указатель на следующий элемент в списке потомков текущего уровня

При таком способе представления дерева обязательно следует сохранять указатель на вершину, являющуюся корнем дерева

```
type
  PTree = ^TTree;
  TTree = record
        Data: TypeElement; {поле данных}
        Childs, Next: PTree; {указатели на потомков и на следующий}
        end;
```

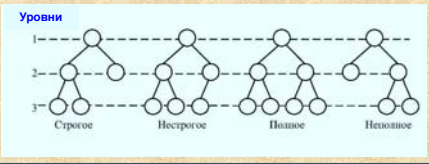

Обходы деревьев



Рекурсивные определения способов обхода:
- если дерево Tree является пустым деревом, то в список обхода заносится пустая запись
- если дерево Tree состоит из одной вершины, то в список обхода записывается эта вершина
- если Tree – дерево с корнем n и поддеревьями $Tree_1$, $Tree_2$, ..., $Tree_k$, то:
  - при прохождении в прямом порядке сначала посещается корень n, затем в прямом порядке – вершины поддерева $Tree_1$, далее в прямом порядке вершины поддерева $Tree_2$ и т. д. Последними в прямом порядке посещаются вершины поддерева $Tree_k$
  - при прохождении в обратном порядке сначала посещаются в обратном порядке вершины поддерева $Tree_1$, далее последовательно в обратном порядке посещаются вершины поддеревьев $Tree_2$, ., $Tree_k$. Последним посещается корень n
  - при прохождении в симметричном порядке сначала посещаются в симметричном порядке вершины поддерева $Tree_1$, далее корень *n*, затем последовательно в симметричном порядке вершины поддеревьев $Tree_2$, ..., $Tree_k$

Спецификация двоичных деревьев

Двоичные (бинарные) – деревья со степенью не более двух



По степени вершин двоичные деревья

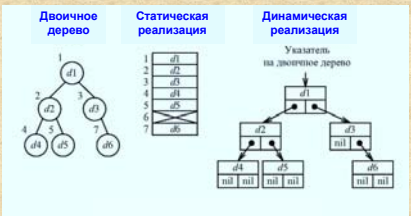
Строгие – вершины дерева имеют степень нуль (у листьев) или два (у узлов)

Нестрогие – вершины дерева имеют степень нуль (у листьев), один или два (у узлов)

Двоичное дерево, содержащее только полностью заполненные уровни (т. е. 2^{k-1} вершин на каждом k-м уровне), называется полным

Реализация двоичных деревьев

Двоичное дерево можно реализовывать
- как статическую структуру данных в виде одномерного массива
- как динамическую структуру – в виде списка



Каждый элемент представления имеет поле данных и два поля указателей. Один указатель используется для связывания элемента с правым потомком, а другой – с левым

Листья имеют пустые указатели потомков

При таком способе представления дерева обязательно следует сохранять указатель на узел, являющийся корнем дерева

```
type
  PTree = ^TTree;
  TTree = record
        Data: TypeElement; {поле данных}
        Left, Right: PTree; {указатели на левого и правого потомков}
  end;
```

Вершины можно пронумеровать слева направо последовательно по уровням и использовать эти номера в качестве индексов в одномерном массиве

```
type
  Tree: array[1..N] of TypeElement;
```

Адрес любой вершины в массиве вычисляется как адрес = $2^{k-1}+i-1$, где *k* – номер уровня вершины; *i* – номер на уровне *k* в полном двоичном дереве. Адрес корня будет равен единице. Для любой вершины, имеющей индекс *j* в массиве, можно вычислить адреса левого и правого потомков:

*адрес_левого = 2*j*

*адрес_правого = 2*j+1*

Основные операции

Рассмотрим операцию прямого обхода двоичного дерева в рекурсивной и нерекурсивной форме

```
procedure PreOrder_BinTree(Node: PTree);
{Рекурсивный обход двоичного дерева в прямом порядке}
begin
        writeln(Node^.Data);
        if Node^.Left <> nil then PreOrder_BinTree(Node^.Left);
        if Node^.Right <> nil then PreOrder_BinTree(Node^.Right);
end;
```

В процедуре, реализующей нерекурсивный обход двоичного дерева, используется стек, хранящий путь от корня дерева до предка текущей вершины. Два режима:

- обход по направлению к левым потомкам до тех пор, пока не встретится лист, при этом выполняется печать значений вершин, и занесение указателей на них в стек
- возврат по пройденному пути с поочередным извлечением указателей из стека до тех пор, пока не встретится вершина, имеющая еще не напечатанного правого потомка. Тогда процедура переходит в первый режим и исследует новый путь, начиная с этого потомка


```
procedure NR_PreOrder_BinTree(Tree: PTree);
{Нерекурсивный обход двоичного дерева в прямом порядке}
var
Node: Ptree; {Указатель на текущую вершину}
S: ^TypeElement; {Стек указателей вершин}
begin
{Инициализация}
        ClearStack(S);
        Node := Tree;
        while true do
          if Node <> nil then
                begin
                        writeln(Node^.Data);
                        PushStack(Node, S);
                {Исследование левого потомка вершины Node}
                        Node := Node^.Left;
                end
                        else
                begin
                {Завершено исследование пути, содержащегося в стеке}
                        if EmptyStack(S) then return;
                {Исследование правого потомка вершины Node}
                        PopStack(Node, S);
                        Node := Node^.Right;
                end;
end;
```


Файлы

Файл – поименованная область во внешней памяти

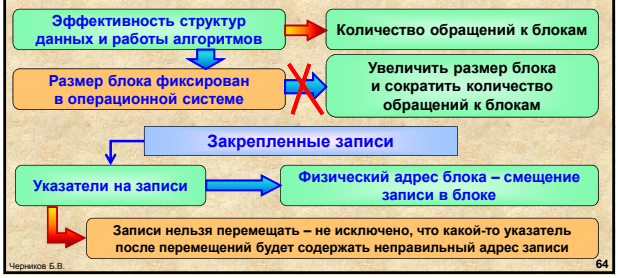
Файлы хранятся в виде определенной последовательности блоков; каждый такой блок содержит целое число записей файла

Базовые операции, выполняемые по отношению к файлам:

- перенос одного блока из внешней памяти в буфер
- перенос одного блока из буфера во внешнюю память

При чтении из файла указатель считывания указывает на одну из записей в блоке, который в данный момент находится в буфере. Когда этот указатель должен переместиться на запись, отсутствующую в буфере, происходит чтение очередного блока из внешней памяти в буфер

При записи в файл фактически происходит внесение записей в буфер файла непосредственно за записями, которые уже находятся там.
Если очередная запись не помещается в буфере, содержимое буфера переносится в свободный блок внешней памяти, который присоединяется к концу списка блоков данного файла.
После этого буфер становится свободным для помещения в него очередной порции записей



Организация

Последовательный файл

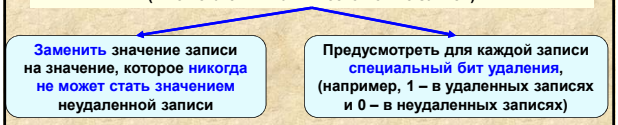
- Используются примитивы чтения и записи файлов – read() и write()
- Записи могут храниться в любом порядке

Поиск записи	Полный просмотр файла
Вставка в файл	Присоединение записи в конец файла
Изменение записи	Поиск требуемой записи, внесение изменений
Удаление записи	Поиск требуемой записи, затем – ее удаление

Сдвинуть все записи, следовавшие за удаленной записью, на одну позицию вперед

Не годится, если записи являются закрепленными, поскольку указатель на *i*-ю запись в файле после выполнения такой операции будет указывать на (i+1)-ю запись

Необходимо определенным образом помечать уделенные записи, но не смещать оставшиеся на место удаленных (и не вставлять на их место новые записи)



Недостаток последовательного файла – операции выполняются медленно

Существуют способы организации файлов, позволяющие обращаться к записи, считывая в основную память лишь небольшую часть файла

Такие способы предусматривают наличие у каждой записи файла ключа – поля (или совокупности полей), которое уникальным образом идентифицирует каждую запись

Хешированные файлы

Хеширование – широко распространенный метод обеспечения быстрого доступа к информации, хранящейся во внешней памяти

Организуется связный список блоков:
заголовок *i*-го блока содержит указатель на физический адрес (i+1)-го блока
Записи, хранящиеся в одном блоке, связывать друг с другом с помощью указателей не требуется – сама таблица представляет собой таблицу указателей на блоки

Структура оказывается эффективной, если в выполняемой операции указывается значение ключа

Среднее количество обращений к блокам равно $n / b \cdot k$,
где n – количество записей; b – количество записей в блоке;
k – длина таблицы

В среднем в *k раз меньше*, чем у последовательного файла

Вставка записи с ключом

1. Вычисляется хеш-функция по ключу – определяется строка таблицы указателей и просматривается соответствующая цепочка блоков
2. Для каждого блока осуществляется попытка вставки новой записи (при наличии места в блоке)
3. Если не удалось вставить ни в один блок цепочки, то у файловой системы запрашивается новый блок, который добавляется в конец цепочки и в него вставляется новая запись

Удаление записи

1. Вычисляется строка таблицы указателей
2. Находится запись в соответствующей цепочке блоков
3. Запись помечается как удаленная

Примечания

- Способы пометки записи – как в последовательных файлах
- Если записи не являются закрепленными, можно заменять удаляемую запись на последнюю запись в последнем блоке текущей цепочки
- Если в результате такой замены последний блок стал пустым, то его можно вернуть файловой системе для повторного использования

Индексированные файлы

Индексированный файл – файл, поддерживаемый в отсортированном (по значению ключа) порядке

Для облегчения процедуры поиска можно создать второй файл, называемый разреженным индексом, который состоит из пар (x, b),
где x – значение ключа, b – физический адрес блока,
в котором значение ключа первой записи равняется x
Этот индексный файл отсортирован по значению ключей

Поиск записи с заданным ключом *x*

1. Просмотреть индексный файл, отыскивая в нем пару (x, b)
2. Найти запись в блоке с физическим адресом b

Стратегии поиска

- Линейный поиск → Просмотр n блоков индексного файла
- Двоичный поиск → Просмотр $\log_2(n + 1)$ блоков индексного файла

Создание индексированных файлов

1. Записи сортируются по значениям их ключей
2. Записи распределяются по блокам в возрастающем порядке ключей

В каждый блок можно разместить столько записей, сколько в него помещается, но можно оставить место под записи, которые могут вставляться туда позже (это уменьшает вероятность переполнения и, следовательно, обращение к смежным блокам)

Вставка записи

1. С помощью индексного файла находят соответствующий блок основного файла
2. Если новая запись умещается в найденный блок, то она вставляется в него в правильной последовательности. Если новая запись становится первой записью в блоке, то необходима корректировка индексного файла
3. Если новая запись не умещается в найденный блок, то возможно применение нескольких стратегий

Стратегия размещения записи, НЕ ПОМЕЩАЮЩЕЙСЯ в блок

Перейти на следующий блок и узнать, можно ли последнюю запись найденного блока переместить в начало следующего (блок не заполнен полностью или не является последним)

Можно?

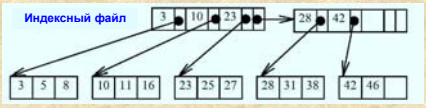
Да:
1. Осуществляем перенос (освобождая место в найденном блоке)
2. Вставляем новую запись на подходящее место в найденный блок
3. Корректируем индексный файл

Нет:
1. Запрашиваем у файловой системы новый блок
2. Помещаем этот блок за найденным блоком
3. В новый блок вставляем новую запись
4. Корректируем индексный файл

Плотный индекс (разновидность индексированного файла)

Плотный индекс – сохранение произвольного порядка записей в файле и создание другого файла, с помощью которого можно отыскивать требуемые записи

Состоит из пар (x, p), где p – указатель на запись с ключом x в основном файле. Эти пары отсортированы по значениям ключа



1. **Вставить новую запись** – отыскивают последний блок основного файла и туда вставляют новую запись
2. **Последний блок полностью заполнен** – запрашивают новый блок у файловой системы
3. Вставляют указатель на соответствующую запись в файле плотного индекса
4. **Удалить запись** – в ней устанавливают бит удаления и удаляют соответствующий указатель в плотном индексе