



Татарский институт содействия бизнесу

Кафедра информационных технологий

***Структуры и алгоритмы
обработки данных***

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ



КАЗАНЬ – 2003

Рекомендовано к печати научно-
методическим советом ТИСБИ

Автор: А.Н. Козин

Рецензент: профессор В.С. Моисеев

© Козин А.Н., 2003

© Татарский институт содействия бизнесу, 2003

Оглавление

Введение

Раздел 1. Основные структуры данных

1. Введение в структуры данных. Динамическое распределение памяти
 - 1.1. Классификация структур данных
 - 1.2. Переменные-указатели и динамическое распределение памяти
 - 1.3. Дополнительные вопросы использования переменных-указателей
 - 1.4. Контрольные вопросы по теме
2. Структуры данных “стек” и “очередь”
 - 2.1. Что такое стек и очередь?
 - 2.2. Статическая реализация стека
 - 2.3. Динамическая реализация стека
 - 2.4. Статическая реализация очереди
 - 2.5. Динамическая реализация очереди
 - 2.6. Практические задания
 - 2.7. Контрольные вопросы по теме
3. Основы реализации списковых структур
 - 3.1. Структуры данных типа “линейный список”
 - 3.2. Первый способ статической реализации списка
 - 3.3. Второй способ статической реализации списка
 - 3.4. Управление памятью при статической реализации списка
 - 3.5. Динамическая реализация линейных списков
 - 3.6. Практические задания
 - 3.7. Контрольные вопросы
4. Усложненные списковые структуры
 - 4.1. Двухнаправленные линейные списки
 - 4.2. Комбинированные структуры данных: массивы и списки указателей
 - 4.3. Комбинированные структуры данных: массивы и списки списков
 - 4.4. Практические задания
 - 4.5. Контрольные вопросы
5. Основные понятия о древовидных структурах
 - 5.1. Основные определения
 - 5.2. Двоичные деревья
 - 5.3. Идеально сбалансированные деревья
 - 5.4. Практические задания
 - 5.5. Контрольные вопросы
6. Реализация поисковых деревьев
 - 6.1. Двоичные деревья поиска
 - 6.2. Добавление вершины в дерево поиска
 - 6.3. Удаление вершины из дерева поиска
 - 6.4. Практические задания
 - 6.5. Контрольные вопросы
7. Дополнительные вопросы обработки деревьев. Графы.
 - 7.1. Проблемы использования деревьев поиска

- 7.2. Двоичные деревья с дополнительными указателями
- 7.3. Деревья общего вида (не двоичные)
- 7.4. Представление графов
- 7.5. Практические задания
- 7.6. Контрольные вопросы

Раздел 2. Алгоритмы сортировки и поиска

- 1. Классификация методов. Простейшие методы сортировки
 - 1.1. Задача оценки и выбора алгоритмов
 - 1.2. Классификация задач сортировки и поиска
 - 1.3. Простейшие методы сортировки: метод обмена
 - 1.4. Простейшие методы сортировки: метод вставок
 - 1.5. Простейшие методы сортировки: метод выбора
 - 1.6. Практическое задание
 - 1.7. Контрольные вопросы
- 2. Улучшенные методы сортировки массивов
 - 2.1. Метод Шелла
 - 2.2. Метод быстрой сортировки
 - 2.3. Пирамидальная сортировка
 - 2.4. Практическое задание
 - 2.5. Контрольные вопросы
- 3. Специальные методы сортировки
 - 3.1. Простейшая карманная сортировка
 - 3.2. Карманная сортировка для случая повторяющихся ключей
 - 3.3. Поразрядная сортировка
 - 3.4. Практическое задание
 - 3.5. Контрольные вопросы
- 4. Поиск с использованием хеш-функций
 - 4.1. Основные понятия
 - 4.2. Разрешение конфликтов: открытое хеширование
 - 4.3. Разрешение конфликтов: внутреннее хеширование
 - 4.4. Практические задания
 - 4.5. Контрольные вопросы
- 5. Внешний поиск и внешняя сортировка
 - 5.1. Особенности обработки больших наборов данных
 - 5.2. Организация внешнего поиска с помощью Б-деревьев
 - 5.3. Б-дерево как структура данных
 - 5.4. Поиск элемента в Б-дереве
 - 5.5. Добавление вершины в Б-дерево
 - 5.6. Удаление вершины из Б-дерева
 - 5.7. Внешняя сортировка
 - 5.8. Практические задания
 - 5.9. Контрольные вопросы

Основные термины и понятия

Литература

Введение

В учебном пособии рассматриваются вопросы программной реализации основных важнейших структур данных, таких как стеки, очереди, списки, деревья и их различных комбинаций. Для большинства структур приводятся различные способы реализации – статические и динамические, даются рекомендации по их практическому применению. Во второй части приводятся алгоритмы сортировки данных, причем основное внимание уделяется методам сортировки массивов, включая все основные простейшие и улучшенные методы и несколько наиболее известных специальных.

Пособие содержит большое число примеров фрагментов программ, а также задания для самостоятельного практического решения. Выполнение практических заданий является абсолютно необходимым, поскольку именно оно развивает навыки самостоятельного написания, выполнения и отладки программ. Наиболее сложные задания имеют рекомендации по программной реализации, облегчающие их выполнение.

Материал учебного пособия соответствует содержанию Государственного образовательного стандарта по одноименному учебному курсу для студентов специальности «Программное обеспечение вычислительной техники и автоматизированных систем» и предполагает знание студентами основ классического структурного программирования. Отдельные темы пособия могут быть полезны и для студентов других специальностей, связанных с информационными и компьютерными технологиями.

Раздел 1. Основные структуры данных

Тема 1. Введение в структуры данных. Динамическое распределение памяти

1.1. Классификация структур данных

Типизация данных является одним из фундаментальных понятий современного программирования. Отнесение переменных к тому или иному типу позволяет установить внутренний формат хранения значений этой переменной и набор допустимых операций. Все распространенные языки программирования имеют набор базовых простейших типов данных (целочисленные, вещественные, символьные, логические) и возможность объединения их в составные наборы – массивы, записи, файлы. Понятие структуры данных определяется двумя моментами:

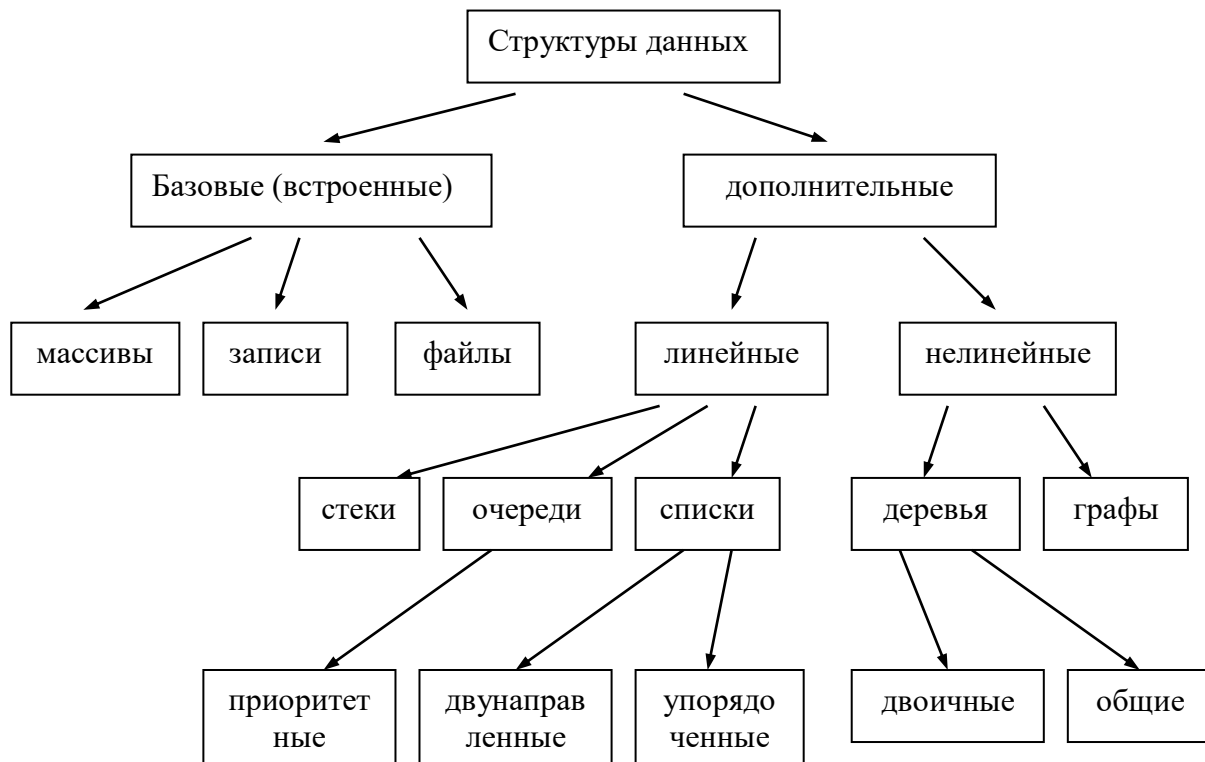
- способом объединения отдельных компонент в единую структуру
- способами обработки как отдельных компонент структуры так и всей структуры в целом.

Например, классическая структура МАССИВ есть объединение однотипных компонент, причем каждая компонента имеет фиксированный порядковый номер-индекс размещения в массиве. Доступ к элементам массива выполняется именно по этому индексу. Аналогично, структура ЗАПИСЬ является объединением разнотипных компонент-полей, доступ к которым выполняется по имени поля.

Обе эти стандартные структуры служат основой построения других важных структур, широко используемых в различных приложениях: стеки, очереди, списки, деревья, графы. Каждая из них имеет свои особенности объединения компонент и особенности обработки. Основные структуры данных представлены на следующей схеме.

Большинство дополнительных структур данных можно реализовать двумя способами:

- статически на основе массива
- динамически с помощью специальных переменных-указателей



Каждый из этих способов имеет свои преимущества и недостатки, которые будут рассмотрены в последующих разделах при детальном описании каждой из структур данных.

Поскольку весьма часто возможны несколько способов реализации одной и той же структуры данных, возникает вопрос о едином унифицированном способе описания подобных структур. Такое описание принято называть **абстрактным типом данных (АТД)**.

АТД — это формализованное описание (модель), определяющее организацию и набор возможных операций с описываемыми данными. Важность АТД определяется тем, что они позволяют отделить описание данных от их реализации и скрыть от пользователя детали реализации, оставив ему только открытый слой взаимодействия с данными — так называемый **интерфейс**. Даже простейшие встроенные типы данных можно описать в терминах АТД, хоть это и не имеет большого практического значения, т.к. простейшие типы встроены в сам язык программирования. Понятие АТД получило свое дальнейшее развитие в объектно-ориентированном программировании в виде

фундаментального понятия “класс”, рассмотрение которого выходит за рамки данного пособия.

Прежде чем перейти к детальному описанию основных структур данных, необходимо рассмотреть вопрос использования в программах переменных указательного типа и механизма динамического распределения памяти.

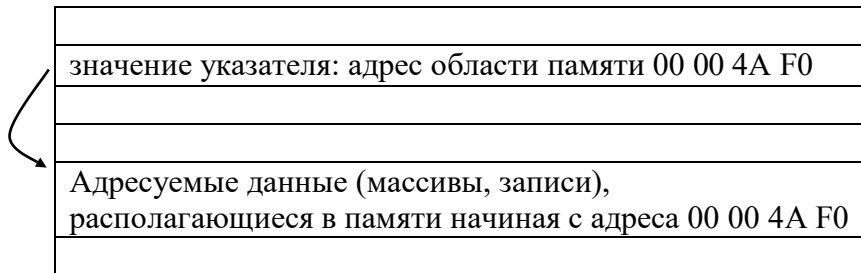
1.2. Переменные-указатели и динамическое распределение памяти.

Как известно, основными рабочими объектами в программах являются переменные. Имя переменной заменяет адрес области памяти, где при выполнении программы хранится значение этой переменной. Одной из важнейших функций компилятора является назначение адресов памяти всем переменным программы. Такое распределение памяти называют **статическим** – оно выполняется при компиляции программы на язык машины. Недостатком такого способа распределения памяти является его жесткость – для изменения объема памяти, выделенной под какую-либо переменную необходимо перекомпилировать программу.

Во многих задачах подобная статичность является очень неудобной, поэтому многие языки программирования, в частности С/С++ и Паскаль, реализуют другой способ создания переменных – **динамический**. Он позволяет создавать переменные в процессе выполнения программы в ответ на вызов специальных стандартных функций. При этом динамически выделяются необходимые области памяти, в которые заносятся значения соответствующих переменных. Когда необходимость в использовании таких переменных исчезает, соответствующие области памяти можно освободить для других целей.

Механизм динамического распределения памяти основан на использовании специальных переменных, которые называют **указателями**. **Переменные-указатели** или **переменные ссылочного типа** - это специальные переменные, значениями которых являются **адреса областей памяти**. Каждый указатель может адресовать некоторую область памяти, в которой могут находиться

любые данные. Чаще всего под адрес отводится 4 байта. Необходимо четко различать **переменную-указатель** и **адресуемую ею область памяти**.



Для описания переменных-указателей необходимо:

- ввести имя переменной
- связать эту переменную с адресуемыми данными, указав их тип или имя с помощью специального символа **^** :

var указатель : **^** базовый тип;

Например, указатели на простейшие базовые типы вводятся следующим образом:

pInt : ^integer; {указатель на отдельное целое число}

pChar : ^char; {указатель на отдельный символ}

pString1, pStr2 : ^string; {два указателя на символьные строки}

Ссылочные типы можно предварительно ввести в разделе описания типов, а потом объявить соответствующую переменную. **Например**:

type TpString = **^string;** {ссылочный тип для адресации текстовых строк}

var pStr1, pStr2 : TpString; {переменные-указатели на строки}

Можно ввести указатели на структурные типы (массивы, записи), используя для этого предварительно объявленные типы:

type TMyArray = **array** [1..100] **of integer;**

var pMyArray1 : **^TMyArray;** {указатель на начало базового массива}

Необходимо понимать, что объявление переменной-указателя **НЕ** приводит к выделению памяти для самих адресуемых данных: память выделяется **ТОЛЬКО** для указателя (например – 4 байта). Поэтому для правильного использования механизма динамического распределения памяти **НЕ** следует

объявлять переменные базового типа (точнее, их объявлять можно, но к динамическому распределению памяти это никакого отношения не имеет).

Например:

```

type TMyRecord = record
    Field1 : SomeType1;
    Field2 : SomeType2;
end;

var MyRec : TMyRecord;    {обычная переменная-запись со статическим
                           выделением памяти}

    pMyRec : ^TMyRecord;  {переменная-указатель на будущую структуру-
                           запись с выделением памяти только для указателя,
                           но не для самой записи!}
  
```

Динамическое создание переменных базового типа выполняется вызовом стандартной подпрограммы **New** с параметром-указателем, связанным с данным базовым типом:

```

New ( pMyRec );
New (pStr1 );
  
```

Данный вызов выполняет два следующих действия:

- обращается к операционной системе для выделения области памяти, необходимой для размещения переменной базового типа
- устанавливает начальный адрес этой области памяти в качестве значения переменной-указателя

Вызов **New** можно выполнять в любом месте в теле программы любое число раз, в том числе — циклически. Это позволяет при выполнении программы динамически создавать любое необходимое число переменных базового типа.

После выделения области памяти в неё можно записать необходимое значение, используя ссылочную переменную и символ **^** после ее имени:

```

pStr1^ := 'Привет';
pInt^ := 1;
  
```

Комбинация имени переменной-указателя и символа \wedge превращает указатель в переменную базового типа. С такой переменной можно делать все те операции, которые разрешены для базового типа. **Например:**

```
pInt $\wedge$  := pInt $\wedge$  + 1;
```

```
pChar $\wedge$  := 'A';
```

```
ReadLn ( pStr1 $\wedge$ );
```

```
pMyArray $\wedge$ [ i ] := 10;
```

```
pMyRec $\wedge$ .Field1 := соответствующее значение;
```

Обращаем внимание на последние два присваивания, конструкцию которых можно прокомментировать следующим образом:

- pMyArray и pMyRec – это **имена ссылочных переменных**
- pMyArray \wedge и pMyRec \wedge - это условные обозначения **объектов базового типа**, т.е. массива и записи соответственно
- pMyArray \wedge [i] и pMyRec \wedge .Field1 – это обозначения **отдельных элементов** базового типа, т.е. i-го элемента массива и поля записи с именем Field1

С переменными ссылочного типа допустимы две операции:

1. Присваивание значения одной ссылочной переменной другой ссылочной переменной того же базового типа:

```
pInt1 := pInt2;
```

После этого оба указателя будут адресовать одну и ту же область памяти.

2. Сравнение двух ссылочных переменных на равенство или неравенство:

```
if pInt2 = pInt1 then ...
```

В практических задачах довольно часто приходится использовать “пустые указатели”, которые никуда не указывают. Для задания такого пустого указателя используется специальное служебное слово **nil**. **Например:**

```
pInt1 := nil;    {указатель ничего не адресует}
```

```
if pStr1 = nil then ... {проверка указателя на пустоту}
```

Для освобождения динамически распределенной памяти используется вызов **Dispose** с параметром-указателем:

Dispose (pStr1);

После этого вызова соответствующая переменная-указатель становится **неопределенной** (НЕ путать с ПУСТЫМ указателем!) и ее нельзя использовать до тех пор, пока она снова не будет инициализирована с помощью вызова **New** или инструкции присваивания. Использование вызова **Dispose** может приводить к весьма неприятной ситуации, связанной с появлением так называемых “висячих” указателей: если два указателя адресовали одну и ту же область памяти (а это встречается очень часто), то вызов **Dispose** с одной из них оставляет в другой переменной адрес уже освобожденной области памяти и повторное использование этой переменной приведет либо к неверному результату, либо к аварийному завершению программы.

1.3. Дополнительные вопросы использования переменных-указателей

Иногда бывает удобно присваивать ссылочной переменной адрес базового объекта, используя для этого непосредственно сам объект. Для этого можно использовать специальный **оператор взятия адреса @**:

```
var x : real;           {обычная переменная вещественного типа}
    pReal : ^real;      {указатель на вещественный тип}
begin
    x := 1.5;           {обычная переменная x получает значение 1.5}
    pReal := @x;        {указатель получает адрес размещения числа 1.5}
    Write(pReal^ : 6 : 2); {вывод числа 1.5 с помощью указателя pReal}
end;
```

Принципиальное отличие данного способа использования указателя от ранее рассмотренного состоит в том, что здесь НЕ используется механизм динамического распределения памяти, т.е. НЕ используются стандартные функции **New** и **Dispose**: переменная *x* является обычной статической переменной, размещаемой в статической области памяти, просто доступ к ней по каким-то причинам организуется не напрямую, а с помощью ссылочной переменной.

Довольно мощным использованием указателей является объединение их в массив. Это можно сделать в силу того, что все переменные-указатели с одним и тем же базовым типом являются однотипными. Массивы указателей можно обрабатывать с помощью циклов. В качестве примера рассмотрим массив указателей на записи некоторой структуры.

```

type TRec = record           {описание базового типа-записи}
    x, y : integer;
    name : string;
end;

TpRec = ^TRec;                {описание ссылочного типа}

var ArrOfPointer : array[1..100] of TpRec;
                               {объявление массива указателей на записи}

begin
  for i := 1 to 100 do ArrOfPointer [ i ]^.x := Random(100);
                               {цикл установки значений в поле x}
end.

```

При использовании указателей есть еще одна весьма мощная, но опасная возможность – так называемые **нетипизированные** указатели, которые не связаны ни с каким базовым типом и поэтому могут адресовать объекты любых типов. Подобные указатели объявляются с помощью служебного слова **pointer**:

```
var pAll : pointer;
```

Распределение и освобождение памяти для нетипизированных указателей производится с помощью специальных стандартных функций **GetMem** и **FreeMem**, которые имеют по два параметра – имя нетипизированного указателя и байтовый размер выделяемой области памяти. Для задания этого размера удобно использовать стандартную функцию **SizeOf**, которая принимает имя типа данных, а возвращает необходимый размер области памяти. **Например**:

```

type TRec = record    {описание базового типа-записи}
    x, y : integer;
    name : string;
end;

var p1, p2 : pointer;    {объявление двух нетипизированных указателей}

begin
    GetMem ( p1, SizeOf ( TRec ) );    {распределение памяти под объект-запись}
    p1^.name := 'Text';
    p2 := p1;                    {оба указателя адресуют одну и ту же область}
    FreeMem ( p1, SizeOf ( TRec ) );
                                {освобождение памяти и деактуализация указателя p1}
end.

```

В заключение необходимо отметить, что использование динамической памяти требует большой осторожности и может быть причиной трудноуловимых ошибок времени выполнения.

1.4. Контрольные вопросы по теме

1. Что включает в себя понятие структуры данных?
2. Назовите основные линейные структуры данных и их разновидности
3. Назовите основные нелинейные структуры данных и их разновидности
4. В чем состоят отличия статического и динамического распределения памяти?
5. Что такое переменные ссылочного типа (указатели)?
6. Что включает описание переменных-указателей?
7. Приведите примеры описания простых переменных-указателей
8. Как вводится переменная-указатель на текстовые строки (2 способа)?
9. Как вводится переменная-указатель на массив?
10. Как вводится переменная-указатель на структуру-запись?
11. Какая память выделяется транслятором при объявлении переменных-указателей?

12. Какие стандартные подпрограммы используются для динамического выделения и освобождения памяти?
13. Что происходит при выполнении подпрограммы **New**?
14. Как выполняется установка необходимого значения в динамически выделенную область памяти?
15. Если p – переменная-указатель, то что определяет выражение p^{\wedge} ?
16. Если p – переменная-указатель на массив, то как можно определить i -ый элемент массива?
17. Если p – переменная-указатель на запись, то как можно определить некоторое поле этой записи?
18. Какие операции разрешены с переменными-указателями?
19. Что такое “пустой указатель” и как он задается?
20. Что происходит при вызове стандартной подпрограммы **Dispose**?
21. Какие неприятности могут возникать при использовании подпрограммы **Dispose**?
22. Как можно переменной-указателю непосредственно присвоить адрес базового объекта?
23. Как задается оператор взятия адреса и для чего он используется?
24. Как задается массив указателей?
25. Если `Pointers` есть массив указателей, то что определяет выражение `Pointers [j]^` ?
26. Что такое нетипизированные указатели и как они описываются?
27. Какие стандартные функции используются для распределения и освобождения памяти с нетипизированными указателями?
28. Приведите пример записи функции для распределения памяти с нетипизированным указателем.
29. Приведите пример записи функции для освобождения памяти с нетипизированным указателем.

Тема 2. Структуры данных “стек” и “очередь”

2.1. Что такое стек и очередь?

Стек – это линейная структура данных, в которую элементы добавляются и удаляются **только с одного конца**, называемого **вершиной** стека. Стек работает по принципу “элемент, помещенный в стек последним, извлечен будет первым”. Иногда этот принцип обозначается сокращением **LIFO** (Last In – First Out, т.е. последним зашел – первым вышел).

Очередь – это линейная структура данных, в которую элементы добавляются с **одного** конца (конец очереди), а удаляются - с **другого** (начало очереди). Очередь работает по принципу “элемент, помещенный в очередь первым, извлечен будет тоже первым”. Иногда этот принцип обозначается сокращением **FIFO** (First In – First Out, т.е. первым зашел – первым вышел). Элементами стеков и очередей могут быть любые **однотипные** данные. В простейшем случае – целые числа, чаще всего – записи заранее определенной структуры. Стеки и очереди очень широко используются в системных программах, в частности – в операционных системах и компиляторах. Программная реализация стеков и очередей возможна двумя способами:

- статически с помощью массива
- динамически с помощью механизма указателей

2.2. Статическая реализация стека

Пусть в стеке требуется сохранять целые числа, причем заранее известно их максимальное количество. Тогда для реализации стека надо объявить массив и одну переменную – **указатель вершины стека** (SP – Stack Pointer). Будем считать, что стек-массив заполняется (растет) от первых элементов массива к последним. Тогда указатель SP может определять либо последнюю занятую ячейку массива, либо первую свободную ячейку. Выберем второй способ. Тогда для пустого стека переменная $SP = 1$ (если индексация элементов массива начинается с 1), и при каждом добавлении нового элемента переменная SP

увеличивается на 1, а при удалении – уменьшается на 1. Последовательность операций для массива из пяти элементов показана на следующей схеме

| | | | | | | |
|-------------------------------------|---|----|----|----|--|--|
| 1. Пустой стек: sp=1 | <table><tr><td></td><td></td><td></td><td></td><td></td></tr></table> | | | | | |
| | | | | | | |
| 2. Добавлено первое число 15, sp=2 | <table><tr><td>15</td><td></td><td></td><td></td><td></td></tr></table> | 15 | | | | |
| 15 | | | | | | |
| 3. Добавлено второе число 33, sp=3 | <table><tr><td>15</td><td>33</td><td></td><td></td><td></td></tr></table> | 15 | 33 | | | |
| 15 | 33 | | | | | |
| 4. Добавлено третье число 07, sp=4 | <table><tr><td>15</td><td>33</td><td>07</td><td></td><td></td></tr></table> | 15 | 33 | 07 | | |
| 15 | 33 | 07 | | | | |
| 5. Удалено с вершины число 07, sp=3 | <table><tr><td>15</td><td>33</td><td></td><td></td><td></td></tr></table> | 15 | 33 | | | |
| 15 | 33 | | | | | |
| 6. Удалено с вершины число 33, sp=2 | <table><tr><td>15</td><td></td><td></td><td></td><td></td></tr></table> | 15 | | | | |
| 15 | | | | | | |

Со стеком связываются две основные операции: добавление (вталкивание, PUSH) элемента в стек и удаление (выталкивание, POP) элемента из стека.

Добавление включает в себя:

- проверку возможности добавления (стек-массив заполнен полностью?)
- размещение нового элемента в ячейке, определяемой значением переменной SP
- увеличение SP на 1

Удаление включает в себя:

- проверку возможности удаления (стек пустой?)
- уменьшение SP на 1
- извлечение элемента из ячейки, определяемой значением переменной SP

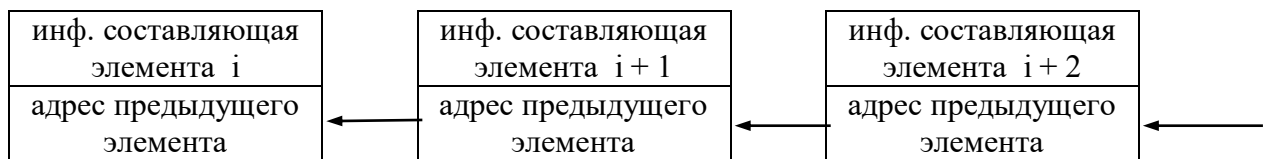
2.3. Динамическая реализация стека

В отличие от статической реализации на основе массива, при использовании механизма динамического выделения памяти в стек можно занести **любое** число элементов. Ограничением является только размер области памяти, выделяемой для размещения динамически создаваемых переменных. При динамической реализации элементы стека могут располагаться в **ЛЮБЫХ** свободных областях памяти, но при этом необходимо как-то связывать соседние элементы друг с другом. Это приводит к необходимости добавления в

каждый элемент стека нового связующего поля для хранения адреса соседнего элемента. Тем самым, каждый элемент стека должен представлять собой запись, состоящую из двух компонент:

- информационная составляющая с полезной смысловой информацией
- связующая составляющая для хранения адреса соседнего элемента

Учитывая специфику стека, указатели должны следовать от последнего элемента (вершина стека) к первому (дно стека).



В этом случае физическое размещение элементов в памяти НЕ обязано всегда соответствовать логическому порядку следования элементов. Логический порядок элементов определяется адресными частями при проходе от последнего элемента к первому. Структура оперативной памяти в этом случае может выглядеть следующим образом:



Для построения логического порядка следования элементов достаточно знать вершинный элемент, все остальное восстанавливается по адресным частям элементов независимо от их реального размещения в памяти.

Для программной реализации элемент стека надо объявить как запись, содержащую по крайней мере два поля – информационное и связующее. Для простоты будем считать, что информационное поле каждого элемента содержит только одно целое число.

Как реализовать связующее поле? Поскольку в связующих полях должны храниться адреса, то следует использовать переменные ссылочного типа.

Что должны адресовать эти переменные? Элементы стека, т.е. записи определенного типа. Следовательно, прежде всего надо ввести ссылочный тип, связанный с базовым типом-записью, а затем – описать базовый тип как запись с необходимыми полями, одно из которых должно быть ссылочного типа:

```
type pStackItem = ^ TStackItem;
```

```
    {ссылочный тип для адресации элементов стека}
```

```
    TStackItem = record
```

```
        {базовый тип, определяющий структуру элементов стека}
```

```
        inf : integer;    {информационная часть}
```

```
        next : pStackItem;
```

```
        {ссылочная часть: поле для адреса следующего элемента}
```

```
    end;
```

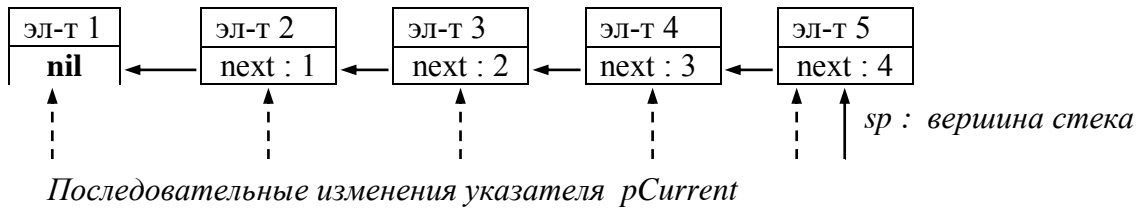
Какие ссылочные переменные необходимы для поддержки работы стека? Во-первых, всегда необходимо знать адрес элемента, находящегося на вершине стека, т.е. помещенного в стек самым последним:

```
var sp : pStackItem;
```

Тогда конструкция `sp^.inf` будет представлять саму информационную часть, а конструкция `sp^.next` – адрес предыдущего элемента, который был помещен в стек непосредственно перед текущим.

Кроме того, для прохода по стеку от вершинного элемента к самому первому элементу необходима вспомогательная ссылочная переменная (например – с именем `pCurrent`). Она на каждом шаге прохода по стеку должна определять адрес текущего элемента. В самом начале прохода надо установить значение `pCurrent = sp`, а затем циклически менять его на значение `pCurrent^.next` до тех пор, пока не будет достигнут первый элемент стека. Очевидно, что для прохода надо использовать цикл с неизвестным числом повторений, а признаком его завершения должно быть получение в поле `pCurrent^.next` пустой ссылки `nil`. Отсюда следует, что ссылочное поле самого первого элемента стека должно содержать значение `nil`.

Тогда схематично проход по стеку можно представить следующим образом:



$pCurrent := sp;$ {начинаем проход с вершины стека}

While $pCurrent \neq nil$ **do**

begin

Writeln ($pCurrent^.Inf$); {вывод инф. части текущего элемента}

$pCurrent := pCurrent^.next;$ {переход к следующему элементу}

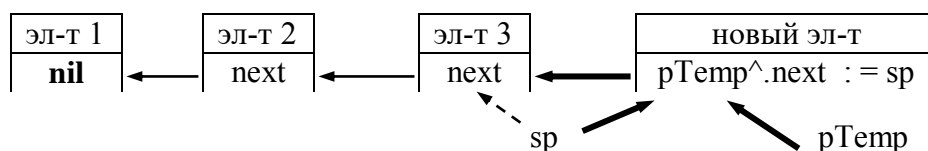
end;

Как выполняется **добавление нового элемента** в вершину стека?

Необходимые шаги:

- выделить память для размещения нового элемента с помощью вспомогательной ссылочной переменной $pTemp$ и стандартной программы **new**($pTemp$); адрес этой области памяти сохраняется как значение переменной $pTemp$
- заполнить информационную часть нового элемента (например: **ReadLn**($pTemp^.inf$))
- установить адресную часть нового элемента таким образом, чтобы она определяла адрес бывшего вершинного элемента: $pTemp^.next := sp;$
- изменить адрес вершины стека так, чтобы он определял в качестве вершины новый элемент: $sp := pTemp;$

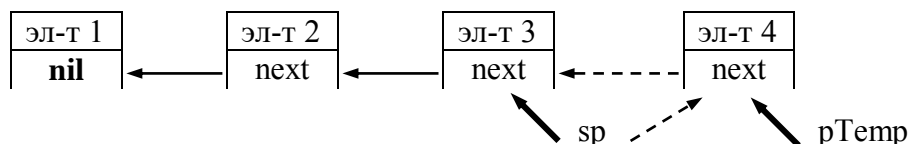
В этой последовательности шагов важен их порядок. Перестановка шагов 3 и 4 приведет к неправильной работе алгоритма вставки, т.к. на шаге 4 происходит изменение указателя sp , который перед этим на шаге 3 используется для установки правильного адреса в ссылочной части нового элемента.



Как выполняется **удаление элемента** с вершины стека?

Необходимые шаги:

- с помощью вспомогательной переменной `pTemp` адресуем удаляемый элемент:
`pTemp := sp;`
- изменяем значение переменной `sp` на адрес новой вершины стека:
`sp := sp^.next;`
- каким-то образом обрабатываем удаленный с вершины элемент, например – просто освобождаем занимаемую им память вызовом **Dispose(pTemp)**, или включаем его во вспомогательную структуру (например – стек удаляемых элементов).



Сравнение статической и динамической реализации стека: при статической реализации расходуется меньше памяти, но требуется знание максимального числа элементов в стеке-массиве; динамическая реализация более гибкая, но каждый элемент стека дополнительно расходует память на ссылочную часть (чаще всего – 4 байта), что при большом числе элементов может стать весьма ощутимым.

2.4. Статическая реализация очереди

Пусть в очереди требуется сохранять целые числа, причем заранее известно их максимальное количество. Тогда для реализации очереди надо объявить массив и две переменные – указатель начала очереди `First` и указатель конца очереди `Last`. Будем считать, что очередь-массив заполняется (растет) от

первых элементов массива к последним. Тогда указатель First будет определять первую **занятую** ячейку массива, а указатель Last - первую **свободную** ячейку. Тогда пустую очередь определим как $First = Last = 1$ (если индексация элементов массива начинается с 1), и при каждом добавлении нового элемента переменная Last увеличивается на 1, а при удалении на 1 увеличивается указатель First. Последовательность операций для массива из пяти элементов показана на следующей схеме:

| | | | | | | |
|---|--|-----------|-----------|-----------|-----------|--|
| 1. Пустая очередь: First = Last =1 | <table><tr><td></td><td></td><td></td><td></td><td></td></tr></table> | | | | | |
| | | | | | | |
| 2. Добавлено первое число 15, First = 1, Last =2 | <table><tr><td>15</td><td></td><td></td><td></td><td></td></tr></table> | 15 | | | | |
| 15 | | | | | | |
| 3. Добавлено второе число 33, First = 1, Last = 3 | <table><tr><td>15</td><td>33</td><td></td><td></td><td></td></tr></table> | 15 | 33 | | | |
| 15 | 33 | | | | | |
| 4. Добавлено третье число 07, First = 1, Last = 4 | <table><tr><td>15</td><td>33</td><td>07</td><td></td><td></td></tr></table> | 15 | 33 | 07 | | |
| 15 | 33 | 07 | | | | |
| 5. Удалено число 15, First = 2, Last = 4 | <table><tr><td></td><td>33</td><td>07</td><td></td><td></td></tr></table> | | 33 | 07 | | |
| | 33 | 07 | | | | |
| 6. Удалено число 33, First = 3 , Last = 4 | <table><tr><td></td><td></td><td>07</td><td></td><td></td></tr></table> | | | 07 | | |
| | | 07 | | | | |
| 7. Добавлено число 44, First = 3 , Last = 5 | <table><tr><td></td><td></td><td>07</td><td>44</td><td></td></tr></table> | | | 07 | 44 | |
| | | 07 | 44 | | | |

Рассмотренная выше простейшая реализация очереди-массива имеет один существенный недостаток: освобождающиеся при удалении ячейки в начале массива НЕ используются при последующих добавлениях, и поэтому при интенсивном использовании очереди быстро может возникнуть ситуация, когда указатель Last выходит за пределы массива, тогда как в начале массива есть свободные ячейки.

Для устранения этого недостатка можно использовать два подхода:

- при очередном удалении элемента из начала очереди сдвигать все элементы влево на одну ячейку, что при большом числе элементов в очереди может привести к большим вычислительным затратам
- более эффективно использовать так называемую **кольцевую** очередь, в которой при достижении указателем Last конца массива добавление производится в начало массива:

8. Добавлено число 11, First = 3 , Last = 6 (= 1)

| | | | | |
|--|--|----|----|-----------|
| | | 07 | 44 | 11 |
|--|--|----|----|-----------|

9. Новое число 22 добавляется в первую ячейку:
First = 3 , Last = 2

| | | | | |
|-----------|--|----|----|----|
| 22 | | 07 | 44 | 11 |
|-----------|--|----|----|----|

10. Добавлено число 99, First = 3 , Last = 3

| | | | | |
|----|-----------|----|----|----|
| 22 | 99 | 07 | 44 | 11 |
|----|-----------|----|----|----|

В этом случае добавление становится невозможным только если в массиве нет ни одной свободной ячейки, как в рассмотренном примере.

Для программной реализации удобно ввести переменную-счетчик числа элементов в очереди, с помощью которой легко отслеживаются состояния пустой и заполненной очереди.

Само **добавление элемента** в очередь выполняется следующим образом:

- проверить возможность добавления (в массиве есть свободные ячейки?)
- добавить элемент в массив по индексу Last
- изменить указатель Last на 1
- если Last выходит за пределы массива, то установить Last в 1
- увеличить счетчик числа элементов в очереди

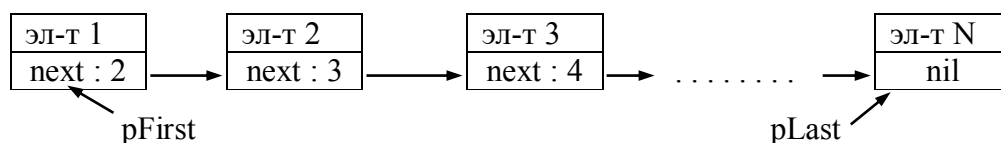
Удаление элемента из очереди:

- проверить возможность удаления (в очереди есть элементы?)
- извлечь элемент из массива по индексу First и выполнить с ним необходимые действия
- увеличить указатель First на 1
- если First выходит за пределы массива, то установить First в 1
- уменьшить счетчик числа элементов в очереди

2.5. Динамическая реализация очереди

Аналогично стеку, каждый элемент очереди должен иметь ссылку на следующий за ним элемент, поэтому элемент очереди объявляется как запись с двумя полями – информационное поле и связующее поле. Но для реализации операций с очередью необходимы уже две переменные: указатель pFirst на начало очереди и указатель pLast на конец очереди.

Приведенная ниже схема элементов очереди отражает логический порядок следования элементов, физически же элементы могут находиться в любых свободных областях памяти.



Основные операции с динамической очередью:

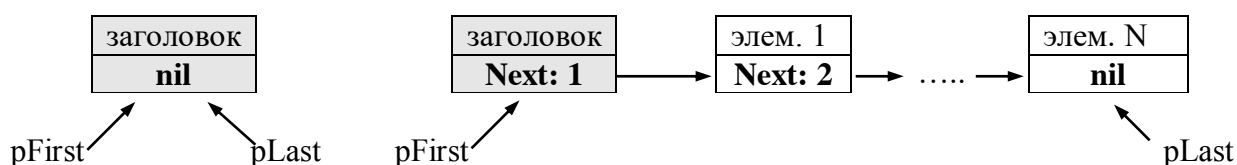
- проверка отсутствия элементов в очереди
- добавление нового элемента в конец очереди
- удаление элемента из начала очереди
- проход по очереди от начала к концу

Добавление нового элемента немного по-разному реализуется для пустой и непустой очереди. Аналогично, по-разному выполняется удаление из очереди, содержащей один или более одного элемента. Для того, чтобы эти операции во всех случаях выполнялись единообразно, часто используется следующий прием: в начало очереди вводится фиктивный **элемент-заголовок**, который НИКОГДА не удаляется и всегда содержит адрес первого элемента очереди.

В этом случае **создание пустой очереди** включает в себя:

- выделение памяти для заголовка с помощью указателя **pFirst**
- занесение в ссылочную часть заголовка пустого указателя **nil**
- установка указателя конца очереди **pLast = pFirst**

Схемы пустой и непустой очереди приводятся на следующих рисунках:



Необходимые описания типов и переменных:


```
type pQueueItem = ^ TQueueItem;
```

{ссылочный тип для адресации элементов очереди}

```
TQueueItem = record           {базовый тип: структура элемента очереди}
```

```
    inf : integer;           {информационная часть}
```

```
    next : pQueueItem;
```

{ссылочная часть: адрес следующего элемента}

```
    end;
```

```
var pFirst, pLast : pQueueItem;
```

Тогда условие пустой очереди можно записать следующим образом:

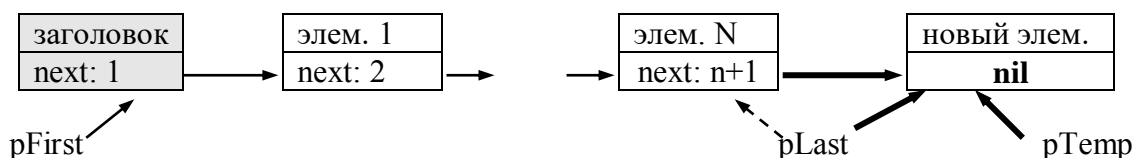
```
pFirst^.next = nil
```

Для **прохода по очереди** от первого реального элемента к последнему необходимо:

- ввести вспомогательную ссылочную переменную pTemp
- установить pTemp в адрес первого реального элемента: pTemp := pFirst^.next
- организовать цикл по условию достижения конца очереди
- в цикле обработать очередной элемент с помощью указателя pTemp и изменить этот указатель: pTemp := pTemp^.next

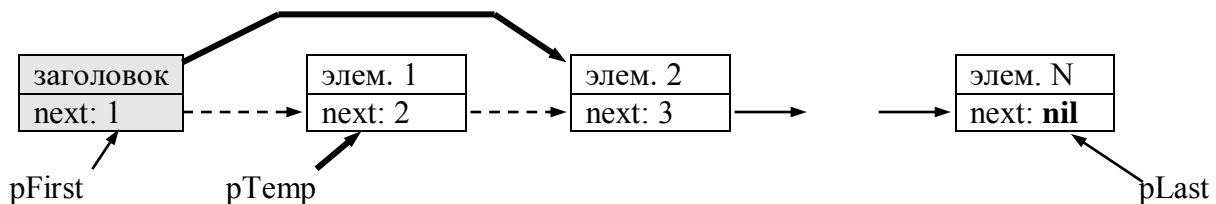
Добавление элемента в конец очереди выполняется следующим образом:

- выделить память для нового элемента с помощью стандартной функции **New** и вспомогательной ссылочной переменной pTemp:
- заполнить поля нового элемента, в частности в связующую часть установить значение **nil**: pTemp^.next := **nil**
- изменить связующую часть бывшего последнего элемента таким образом, чтобы она адресовала новый добавленный элемент: pLast^.next := pTemp;
- изменить значение указателя pLast так, чтобы он указывал новый последний элемент: pLast := pTemp;



Удаление элемента из начала очереди (но после заголовка!) выполняется следующим образом:

- адресуем удаляемый элемент с помощью вспомогательной переменной `pTemp`: `pTemp := pFirst^.next`;
- изменить связующую часть заголовка так, чтобы она указывала на второй элемент очереди, который теперь должен стать первым: `pFirst^.next := pTemp^.next`
- если после удаления в списке не остаётся реальных элементов, то необходимо изменить указатель `pLast`: `pLast := pFirst`
- обработать удаленный элемент, например - освободить занимаемую им память с помощью стандартной подпрограммы **Dispose** (`pTemp`) или включить его во вспомогательную очередь удаленных элементов



Сравнение двух способов реализации очереди полностью аналогично стеку. Интересной разновидностью очереди являются приоритетные очереди, в которых элементы выстраиваются не только в порядке поступления, но и в соответствии с их приоритетами: сначала идут более приоритетные элементы, потом – все менее приоритетные. Одноприоритетные элементы располагаются в порядке поступления. Это требует изменения процедуры добавления элемента в очередь: надо прежде всего найти место в очереди, куда должен вставиться новый элемент в соответствии с его приоритетом, а потом уже выполнить саму вставку. Фактически, приоритетную очередь можно рассматривать как разновидность упорядоченного линейного списка. Реализация линейных списков подробно рассматривается в следующей теме.

2.6. Практические задания

Задание 1. Реализовать программу, выполняющую стандартный набор операций со стеком на основе массива:

- проверку пустоты стека
- проверку заполненности стекового массива
- добавление элемента в вершину стека
- удаление элемента из вершины стека
- вывод текущего состояния стека на экран

Требования:

- все действия должны быть оформлены как процедуры или функции
- добавлению/удалению должна предшествовать проверка возможности выполнения этих операций
- главная программа реализует следующий набор действий:
 - инициализация пустого стека
 - организация диалогового цикла с пользователем

Задание 2. Реализовать тот же набор действий на основе динамического распределения памяти.

Требования аналогичны заданию 1, за исключением того, что проверку заполненности стека проводить не надо. Пустой стек задается установкой `sp := nil`.

Задание 3. Добавить в предыдущую программу возможность занесения в стек сразу нескольких значений. Количество вводимых значений должно запрашиваться у пользователя, а сами значения можно формировать случайным образом с помощью функции **Random** (не забыть предварительно вызвать функцию **Randomize**). Проверить работоспособность программы при различных количествах вводимых элементов, в том числе – для больших значений (десятки тысяч элементов).

Задание 4 (дополнительно). Добавить в предыдущую программу следующие возможности:

- при удалении элемента из основного стека запросить у пользователя, что делать далее с этим элементом: действительно удалить с освобождением памяти или включить его в вершину вспомогательного стека удаленных элементов
- при добавлении нового элемента запросить у пользователя происхождение этого элемента: действительно создание нового элемента или выбор его с вершины вспомогательного стека
- вывод содержимого вспомогательного стека удаленных элементов

Задание 5. Реализовать программу, выполняющую стандартный набор операций с кольцевой очередью на основе массива:

- проверку пустоты очереди
- проверку заполненности очереди
- добавление элемента в конец очереди
- удаление элемента из начала очереди
- вывод текущего состояния очереди на экран

Требования к программе:

- все действия должны быть оформлены как процедуры или функции
- добавлению/удалению должна предшествовать проверка возможности выполнения этих операций
- главная программа реализует следующий набор действий:
 - инициализация пустой очереди
 - организация диалогового цикла с пользователем

Задание 6. Реализовать тот же набор действий на основе динамического распределения памяти.

Требования аналогичны заданию 1, за исключением того, что проверку заполненности очереди проводить не надо. Пустая очередь содержит только заголовочный элемент

Задание 7. Написать программу для моделирования работы очереди со случайным числом добавляемых и удаляемых элементов.

Пусть за единицу времени (например – 1 минуту) в очередь либо добавляется, либо удаляется случайное число элементов в количестве от 1 до 3-х. Одновременно добавление и удаление НЕ происходит. Для наглядности элементами пусть являются большие латинские буквы (коды ASCII от 65 до 90), выбираемые случайным образом. Тип операции с очередью (добавление или удаление) также задается случайно. Это приводит к тому, что очередь случайно растет и убывает. Программа должна наглядно показывать состояние очереди в каждый момент времени.

Рекомендации по разработке программы.

За основу взять предыдущую программу, внося в ее процедуры следующие изменения:

- процедура добавления вместо запроса у пользователя информационной части нового элемента должна получать ее как входной параметр
- процедура удаления должна выполнять удаление одного элемента только если очередь НЕ пустая

Главная программа должна:

- после создания пустой очереди, содержащей только заголовок, инициировать датчик псевдослучайных чисел (процедура **Randomize**) и вывести соответствующее сообщение
- организовать цикл с выходом при вводе пользователем какого-либо специального символа (например – буквы q)
- сгенерировать случайное число в диапазоне от 1 до 100 и проверить его четность с помощью операции взятия остатка от деления этого числа на 2
- если число четное – реализовать операцию добавления, если нечетное – операцию удаления
- в том и другом случае выполнить:
 - генерацию случайного числа добавляемых или удаляемых символов в диапазоне от 1 до 3-х
 - вывод сообщения о выполняемом действии и количестве добавляемых/удаляемых элементов

- выполнение цикла для добавления или удаления заданного числа элементов с вызовом соответствующих процедур работы с очередью, причем при добавлении новый символ должен генерироваться случайно с помощью его кода (диапазон 65 – 90) с последующим преобразованием кода в сам символ с помощью стандартной функции **CHR**
- вывести текущее состояние очереди
- запросить у пользователя символ для окончания цикла (при выполнении программы для продолжения работы цикла можно просто дважды нажать клавишу ввода)

После отладки программы надо **выполнить ее несколько раз** для следующих ситуаций:

- случайное число добавляемых и удаляемых элементов одинаково (например – от 1 до 3-х)
- число добавляемых элементов чуть больше (в среднем!) числа удаляемых (например, добавляется случайное количество элементов в диапазоне от 1 до 4-х, а удаляется – от 1 до 3-х)
- число удаляемых элементов чуть больше числа добавляемых

Для каждого случая выполнить программу при достаточно большом числе добавлений/удалений (30-40) и сделать вывод о поведении очереди.

2.7. Контрольные вопросы по теме

1. Что такое стековый принцип сохранения элементов?
2. Какие основные операции реализуются для стеков?
3. Какие шаги выполняются при добавлении элемента в стек-массив?
4. Какие шаги выполняются при удалении элемента из стека-массива?
5. Какие особые ситуации возможны при реализации стека с помощью массива?
6. Какую структуру должен иметь элемент стека при динамической реализации?

7. Что такое физический и логический порядок следования элементов в стеке?
8. Как между собой связываются соседние элементы стека?
9. Какие типы данных необходимы для динамической реализации стека?
10. Какие переменные необходимы для реализации операций с динамическим стеком?
11. Как реализуется проход по динамическому стеку?
12. Как выполняется добавление элемента в динамический стек?
13. Как выполняется удаление элемента из динамического стека?
14. Сравните динамическую и статическую реализации стека
15. Что такое структура данных типа очередь?
16. Какие основные операции необходимы для реализации структур типа очередь?
17. Как реализуется очередь с помощью массива?
18. Как работает кольцевая очередь, реализованная с помощью массива?
19. Какие особые ситуации возможны при реализации очереди с помощью массива?
20. Как выполняется добавление элемента в очередь-массив?
21. Как выполняется удаление элемента из очереди-массива?
22. На чем основана динамическая реализация очереди?
23. Зачем в очередь вводится фиктивный заголовочный элемент?
24. Что включает в себя создание пустой динамической очереди с заголовком?
25. Какие типы данных необходимы для динамической реализации очереди?
26. Какие переменные используются для реализации основных операций с динамической очередью?
27. Как реализуется проход по динамической очереди?
28. Как реализуется добавление элемента в динамическую очередь?
29. Как реализуется удаление элемента из динамической очереди?
30. Какие особенности имеет приоритетная очередь?
31. Сравните статическую и динамическую реализации очереди.

Тема 3. Основы реализации списковых структур

3.1. Структуры данных типа “линейный список”

Линейный список – это набор связанных **однотипных** элементов, в котором каждый элемент каким-то образом определяет следующий за ним элемент. В отличие от стека и очереди, добавление нового элемента возможно в **любом месте** списка, также можно удалить **любой** элемент списка. Ясно, что списковые структуры являются более гибкими, но и немного более сложными в реализации. Фактически, стеки и очереди можно считать частными случаями списков, в которых добавление и удаление элементов может выполняться только на концах.

Стандартный набор операций со списком включает:

- добавление нового элемента после заданного или перед заданным элементом с проверкой возможности добавления элемента
- удаление заданного элемента
- проход по списку от первого элемента к последнему с выполнением заданных действий
- поиск в списке заданного элемента

Как обычно, возможна статическая и динамическая реализация списков.

При этом статическая реализация на базе массива может быть выполнена двумя способами.

3.2. Первый способ статической реализации списка.

Он основан на том, что логический номер элемента списка **полностью соответствует** номеру ячейки массива, где этот элемент хранится: первый элемент списка – в первой ячейке массива, второй – во второй, третий – в третьей и т.д.

| | | | | | |
|-----------|-----------|-----------|-------|-----------|--|
| Элемент 1 | Элемент 2 | Элемент 3 | | Элемент N | |
|-----------|-----------|-----------|-------|-----------|--|

Проверка наличия в массиве хотя бы одного элемента и хотя бы одной свободной ячейки выполняется элементарно с помощью счетчика числа

элементов списка. Проход по списку – также элементарен. Поиск заданного элемента сводится к обычному поиску в массиве.

Как выполнить вставку нового элемента внутри списка, например – в ячейку с номером $i < N$? Для освобождения ячейки i все элементы списка начиная с номера i и до N надо **сдвинуть вправо** на одну ячейку (конечно, если в массиве есть еще хотя бы одна свободная ячейка). Копирование надо начинать с последней ячейки, т.е. N – на место $N+1$, $N-1$ – на место N , и.т.д. i – на место $i+1$. В освободившуюся ячейку i записывается новый элемент.

Аналогично (с точностью до наоборот) выполняется удаление любого внутреннего элемента: освобождается ячейка i и все последующие элементы **сдвигаются влево** на одну ячейку, т.е. $i+1$ – на место i , $i+2$ – на место $i+1$, и т.д., элемент N – в ячейку $N-1$.

Возникает вопрос о трудоемкости выполнения подобных операций перемещения ячеек массива. Если каждый элемент списка, размещенный в одной ячейке массива представляет собой запись с большим числом объемистых полей, то затраты на подобное перемещение могут стать слишком большими. Здесь выходом может быть изменение структуры элемента списка: в массиве хранятся ТОЛЬКО УКАЗАТЕЛИ (АДРЕСА) информационных частей и перемещение производится только этих указателей, сами информационные части остаются на своих местах. Учитывая все возрастающие скорости работы современных процессоров и наличие в их архитектуре быстрых команд группового копирования байтов, можно считать данный метод реализации списков вполне работоспособным.

3.3. Второй способ статической реализации списка.

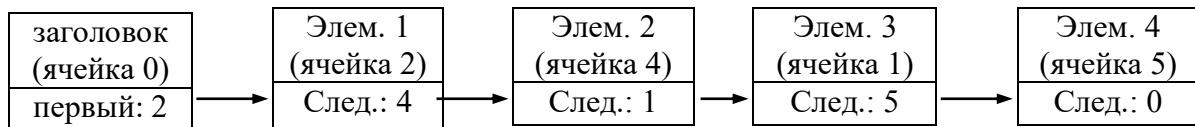
Второй способ реализации списка на основе массива использует принцип **указателей** (но БЕЗ динамического распределения памяти). В этом случае каждый элемент списка (кроме последнего) должен содержать **номер ячейки** массива, в которой находится **следующий** за ним элемент. Это позволяет **РАЗЛИЧАТЬ** физический и логический порядок следования элементов в

списке. Удобно (но НЕ обязательно) в начале массива ввести фиктивный элемент-заголовок, который всегда занимает нулевую ячейку массива, никогда не удаляется и указывает индекс первого реального элемента списка. В этом случае последний элемент списка (где бы он в массиве не располагался) должен в связующей части иметь некоторое специальное значение-признак, например – индекс 0.

Схема физического размещения элементов списка в массиве:

| Номер ячейки | 0 | 1 | 2 | 3 | 4 | 5 |
|-----------------|-----------|---------|---------|---|---------|---------|
| Информац. часть | заголовок | Элем. 3 | Элем. 1 | | Элем. 2 | Элем. 4 |
| Следующий эл-нт | 2 | 5 | 4 | | 1 | 0 |

Соответствующая схема логического следования элементов списка:



Тогда необходимые объявления могут быть следующими:

Const N = 100;

Type TListItem = record

Inf : <описание>;

Next : integer;

end;

Var StatList : array [0 .. N] of TListItem;

Рассмотрим реализацию основных списковых операций.

Условие пустоты списка: StatList [0].Next = 0;

Проход по списку:

- ввести вспомогательную переменную Current для отслеживания текущего элемента списка и установить Current := StatList [0].Next;
- организовать цикл по условию Current = 0, внутри которого обработать текущий элемент StatList [Current].Inf и изменить указатель Current на следующий элемент: Current := StatList [Current].Next

Поиск элемента аналогичен проходу, но может заканчиваться до достижения конца списка:

Current := StatList [0].Next;

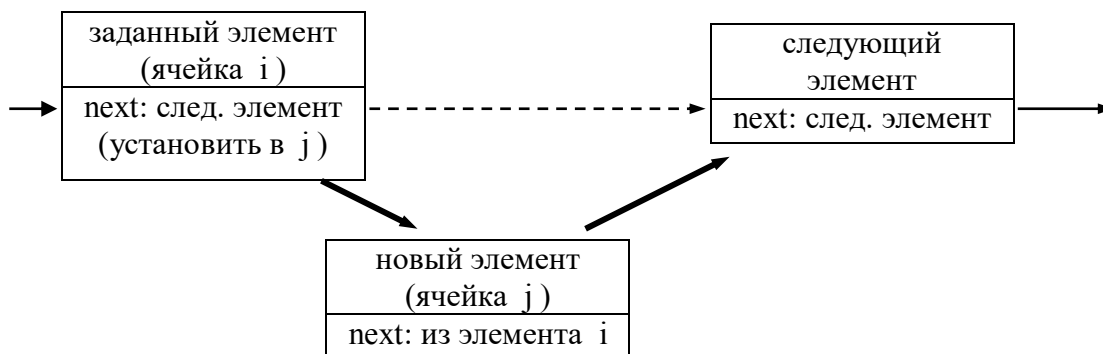
While (Current \neq 0) **and** (StatList [Current].Inf \neq 'значение') **do**

Current := StatList [Current].Next;

If Current = 0 **then** 'поиск неудачен' **else** 'элемент найден';

Добавление элемента **после** заданного:

- проверка возможности добавления с помощью счетчика текущего числа элементов в списке
- определение каким-то образом элемента, после которого надо добавить новый элемент (например – запрос у пользователя)
- поиск этого элемента в списке; пусть его индекс есть i
- определение номера свободной ячейки массива для размещения нового элемента (методы определения будут рассмотрены ниже); пусть этот номер равен j
- формирование связующей части нового элемента, т.е. занесение туда номера ячейки из связующей части элемента i : StatList [j].next := StatList [i].next;
- изменение связующей части элемента i на номер j : StatList [i].next := j ;
- занесение данных в информационную часть нового элемента StatList[j].inf;



Аналогично производится **добавление** нового элемента **перед** заданным, правда здесь приходится дополнительно узнавать номер ячейки, в которой находится элемент, **предшествующий** заданному. Это требует небольшого изменения процедуры поиска заданного элемента: вместе с индексом искомого элемента должен определяться индекс его предшественника. Для этого вводится вспомогательная переменная целого типа, которая в процессе поиска заданного элемента “отстает” на один элемент и тем самым всегда указывает предшественника искомого элемента.

Алгоритм добавления элемента перед заданным включает следующие шаги:

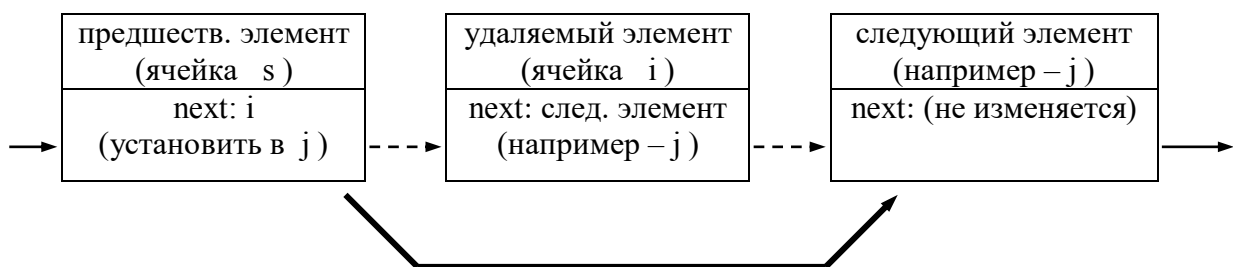
- проверка возможности добавления с помощью счетчика текущего числа элементов в списке
- определение каким-то образом элемента, перед которым надо добавить новый элемент (например – запрос у пользователя)
- поиск этого элемента в списке с одновременным отслеживанием элемента-предшественника; пусть индекс заданного элемента есть i , а индекс его предшественника - s
- определение номера свободной ячейки массива для размещения нового элемента (методы определения будут рассмотрены ниже); пусть этот номер равен j
- формирование связующей части нового элемента, т.е. занесение туда индекса i : `StatList [j].next := i`;
- изменение связующей части элемента-предшественника с индекса i на индекс j : `StatList [s].next := j`;
- занесение данных в информационную часть нового элемента `StatList[j].inf;`



Удаление заданного элемента (естественно, в случае его наличия в списке) также требует изменения связующей части у элемента-предшественника. Это изменение позволяет “обойти” удаляемый элемент и тем самым исключить его из списка.

Необходимые шаги:

- определение каким-то образом удаляемого элемента (например – запрос у пользователя)
- поиск удаляемого элемента в списке с одновременным отслеживанием элемента-предшественника; пусть индекс удаляемого элемента есть i , а индекс его предшественника - s
- изменение связующей части элемента-предшественника с индекса i на индекс-значение связующей части удаляемого элемента i : `StatList [s].next := StatList [i].next;`
- обработка удаляемого элемента (например – вывод информационной части)
- включение удаленного элемента во вспомогательный список без его уничтожения или освобождение ячейки i с включением ее в список свободных ячеек (методы поддержки свободной памяти рассматриваются ниже)



3.4. Управление памятью при статической реализации списков

Реализация списков на основе массивов с указателями-индексами требует постоянного отслеживания свободных и занятых ячеек массива. Можно предложить два подхода.

Первый - более простой, но менее эффективный: все свободные ячейки в связующей части содержат какой-либо специальный номер, например – число (-1). Тогда при начальном создании пустого списка во все ячейки (кроме нулевой с заголовком) в связующие части помещается значение (-1). Если при удалении элемента из списка соответствующая ячейка должна стать свободной, то в ее связующую часть записывается значение (-1), что возвращает эту ячейку в набор свободных. При добавлении нового элемента поиск свободной ячейки организуется просмотром всего массива до обнаружения первой ячейки со значением (-1). Именно эта операция в некоторых случаях может приводить к росту вычислительных затрат, если случайно все свободные ячейки оказались в конце массива, а сам массив является достаточно большим (например, содержит сотни тысяч ячеек).

Второй - более эффективный, но и более сложный способ состоит в том, что все свободные ячейки связываются во **вспомогательный список**, из которого они берутся при добавлении нового элемента в основной список, и куда они возвращаются при удалении элементов из основного списка. При этом вспомогательный список может иметь самое простейшее поведение – например стековое: последняя освободившаяся ячейка массива будет использована для размещения нового элемента в первую очередь.

Какие дополнительные действия необходимы для реализации данного способа? Прежде всего, при создании пустого списка все ячейки массива (кроме нулевой) связываются во вспомогательный список свободных ячеек:

```
for i := 1 to N-1 do StatList [ i ]. Next := i + 1;
StatList [ N ]. Next := 0;
```

Начало вспомогательного списка задается переменной StartFree с начальным значением 1. Удаление элемента из основного списка приводит к изменению связующей части удаленного элемента и изменению значения переменной StartFree на индекс удаленного элемента. При добавлении нового элемента свободная ячейка определяется по значению переменной StartFree с последующим ее изменением.

На следующей схеме показаны три состояния базового массива для реализации списка на 10 элементов.

Состояние пустого списка: $\text{StartFree} = 1$

↓

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|----|----|
| | | | | | | | | | | |
| 0 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 0 |

Состояние списка с шестью элементами:

- Занятые ячейки массива: $3 - 1 - 2 - 8 - 10 - 5$
- Свободные ячейки: $6 - 9 - 4 - 7$, $\text{StartFree} = 6$

↓

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|-------|-------|-------|---|-------|---|---|-------|---|-------|
| | Инф.2 | Инф.3 | Инф.1 | | Инф.6 | | | Инф.4 | | Инф.5 |
| 3 | 2 | 8 | 1 | 7 | 0 | 9 | 0 | 10 | 4 | 5 |

Состояние списка со всеми десятью элементами:

- Занятые ячейки: $8 - 2 - 5 - 6 - 7 - 10 - 1 - 4 - 3 - 9$
- Свободных ячеек нет: $\text{StartFree} = -1$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|-------|-------|-------|-------|-------|-------|-------|-------|--------|-------|
| | Инф.7 | Инф.2 | Инф.9 | Инф.8 | Инф.3 | Инф.4 | Инф.5 | Инф.1 | Инф.10 | Инф.6 |
| 8 | 4 | 5 | 9 | 3 | 6 | 7 | 10 | 2 | 0 | 1 |

3.5. Динамическая реализация линейных списков

Динамическая реализация линейного списка, также как стека и очереди, основана на динамическом выделении и освобождении памяти для элементов списка. Логическая последовательность элементов списка создается ссылочными переменными с адресами последующих элементов (последний элемент имеет пустую ссылку **nil**).

Опять же для удобства реализации будем считать, что список ВСЕГДА содержит хотя бы один элемент-заголовок с адресом первого реального элемента списка. Это позволяет унифицировать процедуры добавления и удаления крайних элементов и устранить некоторые проверки. Адрес элемента-заголовка задается переменной-указателем **rHead**. Эта переменная

устанавливается при первоначальном создании списка и в дальнейшем НЕ изменяется. Для реализации основных действий используются вспомогательные ссылочные переменные.

Необходимые объявления:

```
type pDinItem = ^ TDinItem; {ссылочный тип для адресации элементов списка}
      TDinItem = record
          {базовый тип, определяющий структуру элемента списка}
          inf : <тип информационной части>;
          next : pDinItem;      {адрес следующего элемента}
      end;

var pHead : pDinItem;
```

Создание пустого списка включает:

- выделение памяти под заголовок: **new**(pHead);
- установку пустой ссылочной части заголовка: pHead^.next := **nil**;

Проход по списку от первого элемента к последнему практически не отличается от прохода по очереди.

Поиск заданного элемента включает:

- установку вспомогательного указателя в адрес первого элемента списка
- организацию цикла прохода по списку с завершением либо по совпадению информационной части элемента с заданным значением, либо по достижению конца списка
- после завершения цикла проверить значение вспомогательного указателя и сделать вывод об успешности поиска

```
pCurrent := pHead^.next;
```

```
while (pCurrent <> nil) and (pCurrent^.inf <> 'заданное значение')
```

```
  do pCurrent := pCurrent^.next;
```

```
if pCurrent = nil then 'Элемента нет' else 'элемент найден';
```

Удаление заданного элемента включает:

- поиск удаляемого элемента с определением адреса элемента-предшественника (для этого вводится еще одна вспомогательная

ссылочная переменная $pPrev$, инициализированная адресом заголовка и изменяющая свое значение внутри цикла)

- если удаляемый элемент найден, то изменяется ссылочная часть его предшественника:
 $pPrev^{next} := pCurrent^{next}$
- удаляемый элемент обрабатывается необходимым образом, т.е. либо освобождается занимаемая им память, либо он включается во вспомогательный список

Добавление нового элемента ПОСЛЕ заданного включает:

- поиск заданного элемента с помощью вспомогательного указателя $pCurrent$
- выделение памяти для нового элемента с помощью еще одного указателя $pTemp$
- формирование полей нового элемента, в частности – настройка ссылочной части
 $pTemp^{next} := pCurrent^{next};$
- изменение ссылочной части текущего элемента на адрес нового элемента
 $pCurrent^{next} := pTemp;$

Добавление нового элемента ПЕРЕД заданным включает:

- поиск заданного элемента с одновременным определением элемента-предшественника (используются указатели $pCurrent$ и $pPrev$)
- выделение памяти для нового элемента с помощью еще одного указателя $pTemp$
- формирование полей нового элемента, в частности – настройка ссылочной части: $pTemp^{next} := pCurrent;$
- изменение ссылочной части элемента-предшественника на адрес нового элемента: $pPrev^{next} := pTemp;$

Если при использовании списка часто приходится добавлять или удалять элементы в конце списка, то для уменьшения расходов на просмотр всего списка можно ввести второй основной указатель - на последний элемент

списка. Это потребует изменения процедур удаления и добавления для отслеживания этого указателя.

Довольно часто используется упорядоченная разновидность линейного списка, в котором элементы выстраиваются в соответствии с заданным порядком, например – целые числа по возрастанию, текстовые строки по алфавиту. Для таких списков изменяется процедура добавления – новый элемент должен вставляться в соответствующее место для сохранения порядка элементов. Например, если порядок элементов определяется целыми числами по возрастанию, то при поиске подходящего места надо найти первый элемент, больший заданного и выполнить вставку ПЕРЕД этим элементом.

3.6. Практические задания

Задание 1. Реализовать программу для простейшего моделирования линейного списка с помощью массива. Должны быть реализованы все основные действия:

- проход по списку с выводом на экран информационных частей элементов
- поиск элемента с заданной информационной частью
- добавление нового элемента после заданного и перед заданным со сдвигом (при необходимости) хвостовой части вправо для освобождения ячейки массива
- удаление заданного элемента со сдвигом (при необходимости) хвостовой части влево для заполнения образовавшейся пустой ячейки.

Выполнение всех операций предусматривает необходимые проверки (наличие в списке хотя бы одного элемента, наличие свободных ячеек, наличие искомого элемента). Все основные операции оформляются как подпрограммы с параметрами. Главная программа создает пустой список, устанавливая счетчик числа элементов в списке в ноль, и организует диалог для реализации всех операций.

Проверить работу программы для небольшого массива (до 10 элементов).

Задание 2. Изменить предыдущую программу для создания упорядоченного списка. Для этого надо изменить логику работы подпрограммы добавления элемента. Подпрограмма должна находить соответствующее место для нового элемента, т.е. поиск должен останавливаться при обнаружении первого элемента, большего заданного. Предусмотреть обработку особых случаев:

- если в списке нет ни одного элемента, вставка производится в первую ячейку массива
- если все элементы меньше заданного, вставка производится в конец списка

Проверить работу программы для двух случаев: список целых чисел по возрастанию и список коротких текстовых строк по алфавиту.

Задание 3. Реализовать линейный список на базе массива с указателями-индексами. Список должен иметь заголовок (нулевая ячейка массива) с номером первого элемента списка. Набор операций - стандартный. Для отслеживания свободных ячеек использовать простейшую схему – отмечать свободные ячейки специальным значением индекса (-1). Главная программа при создании пустого списка должна отметить все ячейки массива (кроме нулевой) как свободные.

Задание 4. Реализовать линейный динамический список со стандартным набором операций. Пустой список содержит только элемент-заголовок, который создается главной программой в начале работы и содержит значение **nil** в ссылочной части. У непустого списка в ссылочной части хранится адрес первого реального элемента. Адрес заголовка сохраняется в глобальной ссылочной переменной. Все действия оформляются как подпрограммы.

Подпрограмма для добавления элемента после заданного должна работать и для пустого списка – в этом случае новый (он же первый реальный!) элемент должен добавляться сразу после заголовка. Для этого проверить пустоту списка, после чего для пустого списка установить глобальный указатель

текущего элемента в адрес заголовка, для непустого списка вызвать процедуру поиска. Само добавление выполняется обычным образом.

Задание 5. Изменить предыдущую программу так, чтобы удаляемый из основного списка элемент добавлялся во вспомогательный список с возможностью просмотра вспомогательного списка. Работа со вспомогательным списком может выполняться по стековому принципу.

В предыдущую программу надо внести следующие изменения:

- добавить глобальную ссылочную переменную для хранения адреса вершины стека
- в начале главной программы создать пустой вспомогательный список (стек)
- в основное меню добавить возможность просмотра вспомогательного списка (стека)
- изменить процедуру удаления элемента из основного списка, заменив операцию освобождения памяти операциями добавления удаленного элемента во вспомогательный список (стек)
- добавить обычную процедуру вывода вспомогательного списка (стека)

Задание 6. Изменить предыдущую программу для поддержки упорядоченных списков (см. задание 2).

3.7. Контрольные вопросы по теме

1. В чем состоит отличие списковых структур от стека и очереди?
2. Что включает в себя стандартный набор операций со списком?
3. В чем состоит простейший способ реализации списка с помощью массива?
4. Как выполняется вставка элемента при простейшей реализации списка на базе массива?
5. Как выполняется удаление элемента при простейшей реализации списка на базе массива?
6. В чем состоят преимущества и недостатки простейшего способа реализации списков с помощью массивов?

7. За счет чего можно повысить эффективность простейшей реализации списка с помощью массива?
8. В чем смысл реализации статического списка с указателями-индексами?
9. Какую структуру имеют элементы массива при статической реализации списка?
10. Какие описания необходимы для статической реализации списка?
11. Как выполняется проход по статическому списку?
12. Как выполняется поиск элемента в статическом списке?
13. Какие особые ситуации возможны при статической реализации списка?
14. Что такое пустой статический список?
15. Как выполняется добавление элемента после заданного в статическом списке?
16. Как выполняется добавление элемента перед заданным в статическом списке?
17. Как выполняется удаление элемента в статическом списке?
18. Как в простейшем случае можно отслеживать свободные ячейки массива при реализации статического списка?
19. В чем недостатки простейшего способа отслеживания свободных ячеек при реализации статического списка?
20. Как используется вспомогательный список свободных ячеек при статической реализации списка?
21. Как инициализируется вспомогательный список свободных ячеек при создании пустого статического списка?
22. Что является основой реализации динамических списков?
23. Какую структуру имеют элементы динамического списка?
24. Какие описания необходимы для реализации динамического списка?
25. Какие переменные используются для реализации операций с динамическими списками?
26. Что включает в себя создание пустого динамического списка?
27. Как выполняется проход по динамическому списку?

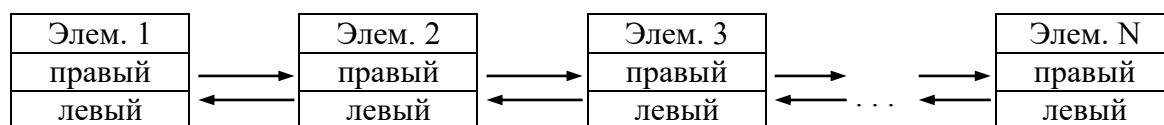
28. Как выполняется поиск элемента в динамическом списке?
29. Как выполняется удаление элемента в динамическом списке?
30. Как выполняется добавление элемента после заданного в динамическом списке?
31. Как выполняется добавление элемента перед заданным в динамическом списке?
32. Какие особенности возникают при обработке упорядоченных списков?

Тема 4. Усложненные списковые структуры

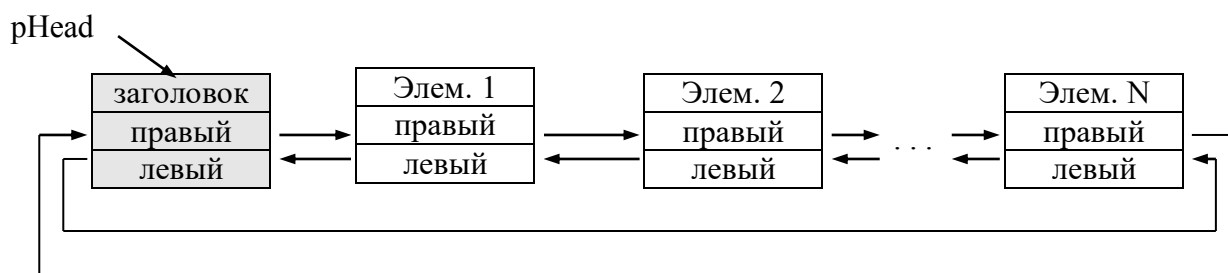
4.1. Двухнаправленные линейные списки

Недостатком рассмотренных выше списковых структур является их однонаправленность от первого элемента к последнему. Если при обработке списков часто бывает необходимо переходить от текущего элемента к его предшественнику, то такая односторонняя организация становится неудобной. Выходом является построение **двухнаправленных** списков, в которых каждый элемент “знает” **обоих** своих соседей, как левого, так и правого.

Для этого каждый элемент должен иметь не одно, а два связующих поля: указатель на элемент слева и указатель на элемент справа.



Аналогично обычным спискам, удобно ввести фиктивный заголовочный элемент, а поскольку двухнаправленные списки являются симметричными, удобно сделать их замкнутыми, кольцевыми: правая ссылка последнего элемента указывает на заголовок, а левая ссылка заголовка – на последний элемент. Адрес заголовка определяется указателем pHead.



Отметим достоинства и недостатки двунаправленных списков. Достоинство – простота перехода от текущего элемента к любому его соседу. Недостатки – увеличиваются затраты памяти и число операций на поддержку дополнительного указателя.

Двунаправленный список можно реализовать как на основе массива (причем – обоими способами), так и динамически.

В последнем случае описание структуры данных выглядит следующим образом:

```
type pDin2Item = ^ TDin2Item;    {ссылочный тип}
      TDin2Item = record          {базовый тип}
                  inf : <тип информационной части>;
                  left, right : pDin2Item;      {адреса соседних элементов}
      end;
```

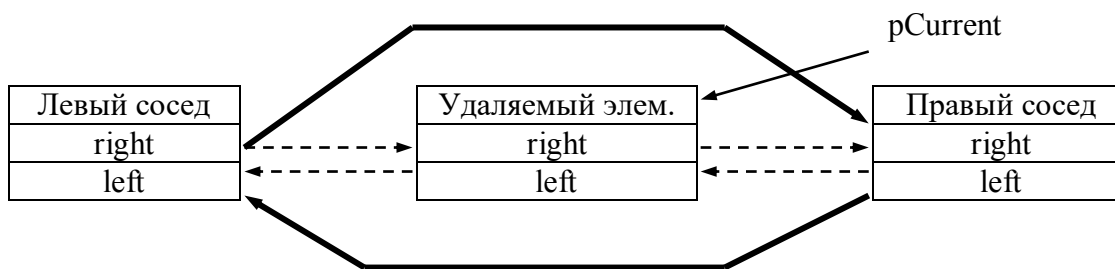
Если pHead есть указатель на заголовок, то пустой список создается так:

- выделяется память под заголовок, адресуемая указателем pHead
 - оба ссылочных поля заголовка устанавливаются в адрес самого заголовка: pHead^.left := pHead; pHead^.right := pHead;
- (при этом пустая ссылка **nil** нигде НЕ используется!)

Набор операций расширяется просмотром и поиском не только в прямом, но и в **обратном** направлениях. Немного изменяется условие достижения конца списка – вместо условия появления пустой ссылки надо использовать условие получения на текущем шаге адреса заголовка. Например, проход в **обратном** направлении реализуется так:

- устанавливаем начальное значение указателя текущего элемента на последний элемент списка: pCurrent := pHead^.left;
 - организуем цикл прохода до достижения заголовка
- ```
while pCurrent <> pHead do pCurrent := pCurrent^.left;
```

**Удаление** заданного элемента требует изменения большего числа ссылочных полей. Надо изменить правый указатель у левого соседа удаляемого элемента и левый указатель у правого соседа.

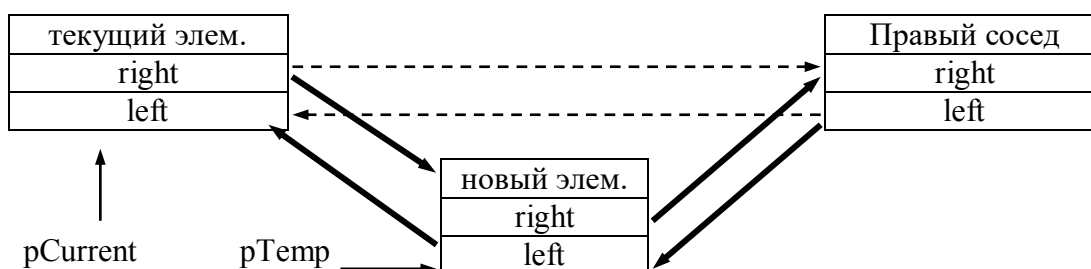


Если удаляемый элемент адресуется указателем  $pCurrent$ , то  $pCurrent^{left}$  определяет адрес левого соседа, а  $pCurrent^{right}$  – адрес правого соседа. Тогда необходимые изменения реализуются так:

$$pCurrent^{left}.right := pCurrent^{right};$$

$$pCurrent^{right}.left := pCurrent^{left};$$

Аналогично выполняется **добавление** нового элемента, например – после заданного указателем  $pCurrent$ . Пусть как обычно новый элемент определяется указателем  $pTemp$ . Тогда для вставки его в список надо настроить оба его ссылочных поля, изменить правое ссылочное поле у текущего элемента и левое ссылочное поле у его правого соседа.



Необходимые присваивания (порядок следования важен!):

$$pTemp^{right} := pCurrent^{right};$$

$$pTemp^{left} := pCurrent;$$

$$pCurrent^{right}.left := pTemp;$$

$$pCurrent^{right} := pTemp;$$



Аналогично реализуется **добавление** нового элемента **перед** заданным элементом, только вместо правого соседа обрабатывается левый сосед текущего элемента.

#### 4.2. Комбинированные структуры данных: массивы и списки указателей

Рассмотренные выше способы объединения элементов могут комбинироваться друг с другом, образуя достаточно сложные структуры. Самый простой случай такого взаимодействия уже был упомянут выше – имеется в виду массив указателей на объекты базового типа. Для удобства повторим соответствующие описания:

```

type TRec = record {описание базового типа-записи}
 x, y : integer;
 name : string;
 end;

 TpRec = ^TRec; {описание ссылочного типа}
var ArrOfPointer : array[1..100] of TpRec;
 {объявление массива указателей на записи}

```

**Поиск** заданного элемента в данном случае реализуется очень просто:

```

i := 1;
While (i <= 100) and (ArrOfPointer [i]^.name <> ‘заданное значение’)
 do i := i + 1;

```

**Добавление** нового элемента предполагает выделение памяти для этого элемента и включение в массив ссылок адреса элемента:

```

ArrOfPointer [i] := pTemp;

```

Массив указателей позволяет быстро и эффективно производить **перестановку** элементов. Например, для перестановки элементов *i* и *j* достаточно выполнить три простых присваивания:

```

pTemp := ArrOfPointer [i];
ArrOfPointer [i] := ArrOfPointer [j];
ArrOfPointer [j] := pTemp;

```

Эти присваивания переставляют лишь значения адресов в соответствующих элементах, НЕ перемещая сами базовые объекты, которые могут занимать значительные объемы памяти.

Вместо массива можно организовать линейный список указателей на базовые объекты. Каждый элемент такого списка содержит два поля: указатель на соседний элемент и указатель на базовый объект. Поскольку структура базового объекта отличается от структуры элемента списка, их надо описывать отдельно и вводить два ссылочных типа данных.

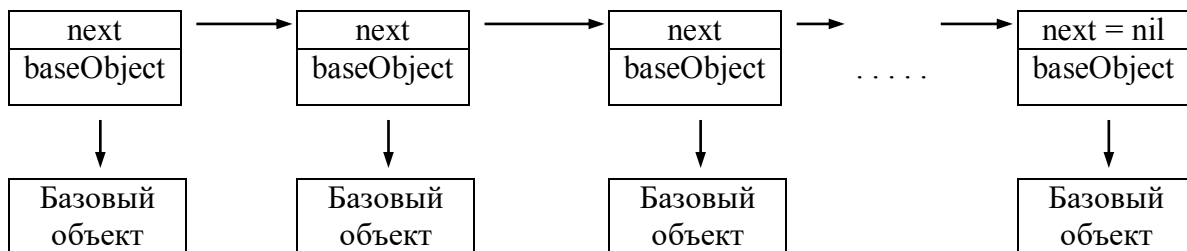
```

Type pInf = ^TInf; {ссылочный тип для адресации базовых объектов}
 TInf = record {тип-базовая структура}
 <описание полей базовой структуры данных>
 end;

pItem = ^TItem;
 {ссылочный тип для адресации элементов списка указателей}

TItem = record {описание структуры элемента списка указателей}
 next : pItem; {поле-указатель на соседний элемент списка}
 baseObject : pInf; {поле-указатель на базовый объект}
end;

```

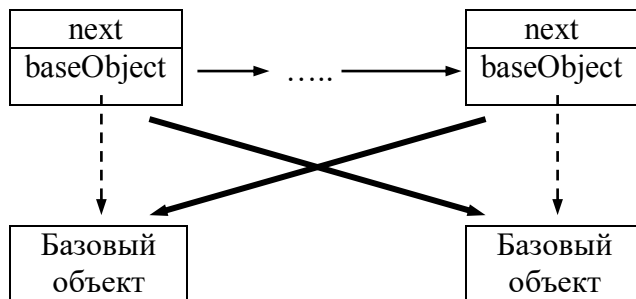


Естественно, что список указателей может иметь заголовок и быть двунаправленным. В любом случае обработка полей базовых объектов выполняется с помощью соответствующих указателей (поле `baseObject`) в элементах списка. Например, если `pCurrent` определяет адрес текущего элемента списка, то выражение `pCurrent^.baseObject` определяет адрес соответствующего базового объекта, а выражение `pCurrent^.baseObject^` - сам базовый объект.

**Добавление** нового элемента требует двукратного выделения памяти: сначала для самого базового объекта (переменная-указатель `pTempObject` типа `pInf`), потом – для элемента списка (переменная-указатель `pTemp` типа `pItem`). Основные операции:

- `new (pTempObject);`
- “заполнение полей базовой структуры `pTempObject^`”;
- `new (pTemp);`
- `pTemp^.baseObject := pTempObject;`
- “добавление элемента в список”;

Аналогично, **удаление** элемента требует двукратного освобождения памяти: сначала – для базового объекта, потом – для элемента списка. Как и в случае использования массива, очень легко производится **перестановка** двух элементов за счет взаимной замены адресных полей `baseObject` без перемещения самих базовых объектов.



Пусть `pCurrent1` и `pCurrent2` - указатели на элементы списка, у которых надо обменять базовые объекты. Тогда сам обмен реализуется с помощью вспомогательной переменной `pTempObject` типа `pInf` следующим образом:

```

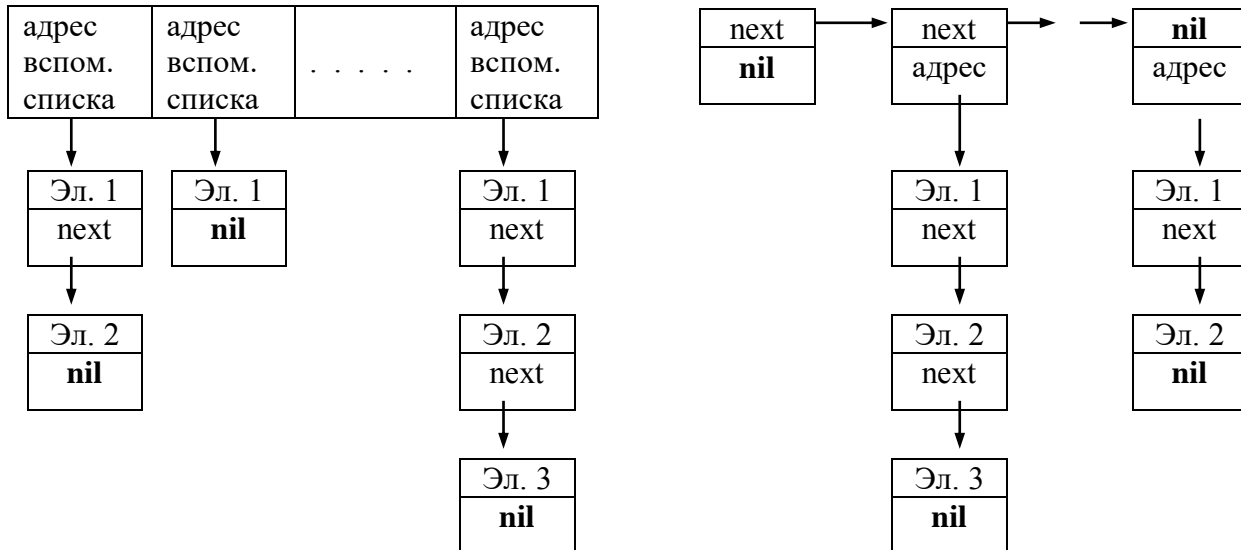
pTempObject := pCurrent1^.baseObject;
pCurrent1^.baseObject := pCurrent2^.baseObject;
pCurrent2^.baseObject := pTempObject;

```

Отметим, что операции перестановки очень часто используются в алгоритмах сортировки массивов и списков.

### 4.3. Комбинированные структуры данных: массивы и списки списков

Более сложным случаем является использование массива списков или списка списков. Здесь каждый элемент массива или основного списка является началом вспомогательного списка, причем эти вспомогательные списки могут содержать разное число элементов (но, конечно, одного типа).



В обоих случаях в первую очередь вводится тип данных, описывающий структуру элементов вспомогательного списка:

**Type** pSubList = ^ TSubList;

TSubList = **record**

<описание информационных полей>;

next : pSubList;

**end;**

После этого описывается либо основной массив указателей, либо структура элементов основного списка:

**Type** TMainArray = **array** [ 1 .. N ] **of** pSubList;

pMainList = ^ TMainList;

TMainList = **record**

NextMain : pMainList;

FirstSub : pSubList;

**end;**

Обработка таких структур включает больше операций, поскольку практически любая типовая операция (поиск, просмотр, добавление, удаление) может выполняться как с основным массивом или списком, так и с любым вспомогательным списком. Например, полный поиск или полный проход реализуется двойным циклом: внешний цикл проходит по элементам основного списка, а внутренний обрабатывает отдельно каждый вспомогательный список. Для этого необходимы две ссылочные переменные: `pCurrentMain` для прохода по основному списку и `pCurrentSub` – для прохода по вспомогательному списку.

```
pCurrentMain := "адрес первого элемента основного списка";
while pCurrentMain <> nil do
 begin
 pCurrentSub := pCurrentMain^.FirstSub;
 while pCurrentSub <> nil do pCurrentSub := pCurrentSub^.next;
 end;
 pCurrentMain := pCurrentMain^.NextMain;
```

Добавление и удаление элементов во вспомогательных списках выполняется обычным образом. Небольшие особенности имеет удаление элемента из основного списка, поскольку в этом случае как правило надо удалить и весь вспомогательный список. Поэтому прежде всего организуется проход по вспомогательному списку с удалением каждого элемента, а потом уже производится удаление элемента основного списка.

Иногда удобно в элементах основного списка хранить не только адрес первого элемента вспомогательного списка, но и адрес последнего элемента. Естественно, что при необходимости как основной, так и вспомогательные списки могут быть двунаправленными.

Кроме того, поскольку стеки и очереди можно рассматривать как частные случаи списков, легко можно реализовать такие структуры как массив или список стеков, массив или список очередей и.т.д.

#### 4.4. Практические задания.

**Задание 1.** Реализовать линейный динамический двунаправленный список со следующим набором операций:

- просмотр списка в прямом и обратном направлениях
- поиск заданного элемента в прямом и обратном направлениях
- добавление элемента перед или после заданного
- удаление заданного элемента

Список должен иметь заголовок и быть кольцевым. Пустой список содержит только заголовок, оба ссылочных поля которого указывают на сам заголовок. Адрес заголовка задается глобальной ссылочной переменной. Все операции оформляются как подпрограммы. Добавление нового элемента после заданного должно работать и для пустого списка (см. задание 4 из предыдущей темы).

**Задание 2.** Реализовать набор подпрограмм для выполнения основных операций с массивом списков. Каждый элемент массива хранит только указатель на начало связанного списка. Сам базовый массив работает на основе сдвиговых операций. Основные операции:

- полный проход по всей структуре
- поиск заданного элемента
- добавление нового элемента в массив с пустым связанным списком
- добавление нового элемента в связанный список
- удаление элемента из связанного списка
- удаление элемента из базового массива

**Задание 3.** Реализовать набор подпрограмм для выполнения основных операций со списком списков. Требования аналогичны предыдущему заданию.

#### 4.5. Контрольные вопросы по теме

1. Какие преимущества и недостатки имеют двунаправленные списки?
2. Какую структуру имеют элементы двунаправленных списков?
3. Почему двунаправленные списки чаще всего делают кольцевыми?

4. Как используются ссылочные поля заголовка двунаправленного списка?
5. Какие описания необходимы для динамической реализации двунаправленных списков?
6. Какие переменные используются при реализации операций с динамическими двунаправленными списками?
7. Как создается пустой динамический двунаправленный список?
8. Как реализуется проход в прямом направлении по динамическому двунаправленному списку?
9. Как реализуется проход в обратном направлении по динамическому двунаправленному списку?
10. Как реализуется поиск в прямом направлении по динамическому двунаправленному списку?
11. Как реализуется поиск в обратном направлении по динамическому двунаправленному списку?
12. Как реализуется удаление элемента в динамическом двунаправленном списке?
13. Как реализуется добавление элемента после заданного в динамическом двунаправленном списке?
14. Как реализуется добавление элемента перед заданным в динамическом двунаправленном списке?
15. Как выполняется перестановка элементов в массиве указателей?
16. Какие описания необходимы для реализации списка указателей?
17. Как выполняется добавление элемента в список указателей?
18. Как выполняется удаление элемента из списка указателей?
19. Как выполняется перестановка элементов в списке указателей?
20. Что такое массив списков и как он описывается?
21. Что такое список списков и как он описывается?
22. Как выполняется полный проход по массиву списков?
23. Как выполняется полный проход по списку списков?
24. Как выполняется поиск элемента в массиве списков?

25. Как выполняется поиск элемента в списке списков?
26. Как выполняется удаление элемента из массива списков?
27. Как выполняется удаление элемента из списка списков?
28. Какие особенности имеет операция добавление элемента в массив или список списков?

## Тема 5. Основные понятия о древовидных структурах

### 5.1. Основные определения

Структуры данных типа “дерево” исключительно широко используются в программной индустрии. В отличие от списковых структур деревья относятся к **нелинейным** структурам. Любое дерево состоит из элементов – узлов или вершин, которые по определенным правилам связаны друг с другом рёбрами. В списковых структурах за текущей вершиной (если она не последняя) всегда следует только одна вершина, тогда как в древовидных структурах таких вершин может быть **несколько**. Математически дерево рассматривается как частный случай графа, в котором отсутствуют замкнутые пути (циклы).

Дерево является типичным примером **рекурсивно определённой структуры** данных, поскольку оно определяется в терминах самого себя.

Рекурсивное определение дерева с базовым типом  $T$  – это:

- либо **пустое** дерево (не содержащее ни одного узла)
- либо некоторая **вершина** типа  $T$  с конечным числом связанных с ней отдельных деревьев с базовым типом  $T$ , называемых **поддеревьями**

Отсюда видно, что в любом непустом дереве есть одна особая вершина – **корень** дерева, которая как бы определяет “начало” всего дерева. С другой стороны, существуют и вершины другого типа, не имеющие связанных с ними поддеревьев. Такие вершины называют **терминальными** или листьями.

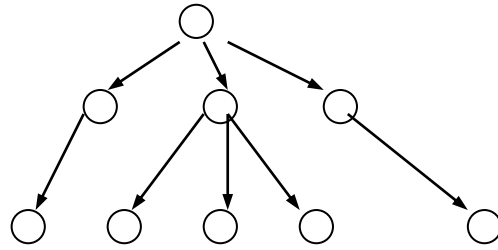
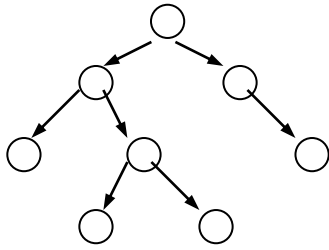
Классификацию деревьев можно провести по разным признакам.

1. По числу возможных потомков у вершин различают **двоичные (бинарные)** или **недвоичные (сильноветвящиеся)** деревья.

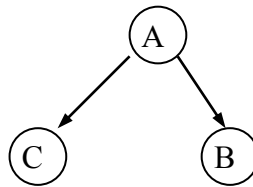
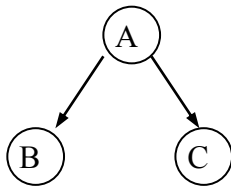
Двоичное дерево: каждая вершина может иметь не более двух потомков.



Недвоичное дерево: вершины могут иметь любое число потомков.



2. Если в дереве важен порядок следования потомков, то такие деревья называют **упорядоченными**. Для них вводится понятие левый и правый потомок (для двоичных деревьев) или более левый/правый (для недвоичных деревьев). В этом смысле два следующих простейших упорядоченных дерева с **одинаковыми** элементами считаются **разными**:



При использовании деревьев часто встречаются такие понятия как **путь** между начальной и конечной вершиной (последовательность проходимых ребер или вершин), **высота** дерева (наиболее длинный путь от корневой вершины к терминальным).

При рассмотрении дерева как структуры данных необходимо четко понимать следующие два момента:

1. Все вершины дерева, рассматриваемые как переменные языка программирования, должны быть одного и того же типа, более того – записями с некоторым информационным наполнением и необходимым количеством связующих полей
2. В силу естественной логической разветвленности деревьев (в этом весь их смысл!) и отсутствия единого правила выстраивания вершин в порядке друг за другом, их логическая организация не совпадает с физическим размещением вершин дерева в памяти.

Дерево как абстрактная структура данных должна включать следующий набор операций:

- добавление новой вершины
- удаление некоторой вершины
- обход всех вершин дерева
- поиск заданной вершины

## 5.2. Двоичные деревья

Двоичные деревья (ДД) используются наиболее часто и поэтому представляют наибольший практический интерес. Каждая вершина ДД должна иметь **два связующих поля** для адресации двух своих возможных потомков.

ДД можно реализовать двумя способами:

- на основе массива записей с использованием индексных указателей
- на базе механизма динамического распределения памяти с сохранением в каждой вершине адресов ее потомков (если они есть)

Второй способ является значительно более удобным и поэтому используется наиболее часто. В этом случае каждая вершина описывается как запись, содержащая как минимум три поля: информационную составляющую и два ссылочных поля для адресации потомков:

**Type** Tr = ^TNode;      {объявление ссылочного типа данных}

TNode = **record**

Inf : <описание информационной части>;

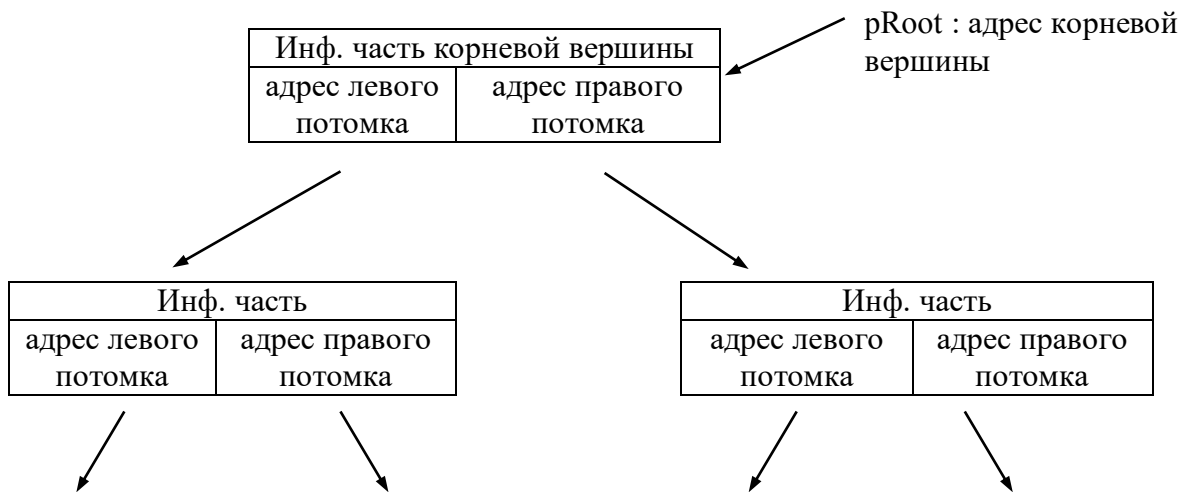
Left, Right : Tr;

**end;**

Для обработки дерева достаточно знать адрес корневой вершины. Для хранения этого адреса надо ввести ссылочную переменную:

**Var** pRoot : Tr;

Тогда пустое дерево просто определяется установкой переменной pRoot в нулевое значение (например – **nil**).



Реализацию основных операций с ДД удобно начать с **процедур обхода**. Поскольку дерево является нелинейной структурой, то НЕ существует **единственной** схемы обхода дерева. Классически выделяют три основных схемы:

- обход в прямом направлении
- симметричный обход
- обход в обратном направлении

Для объяснения каждого из этих правил удобно воспользоваться простейшим ДД из трех вершин. Обход всего дерева следует проводить за счет последовательного выделения в дереве подобных простейших поддеревьев и применением к каждому из них соответствующего правила обхода. Выделение начинается с корневой вершины.

Сами правила обхода носят рекурсивный характер и формулируются следующим образом:

### 1. Обход в прямом направлении:

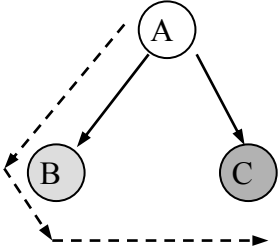
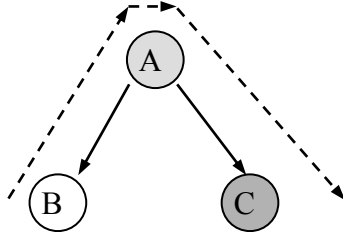
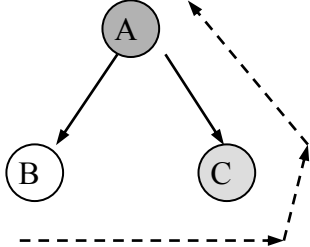
- обработать корневую вершину текущего поддерева
- перейти к обработке левого поддерева таким же образом
- обработать правое поддерево таким же образом

### 2. Симметричный обход:

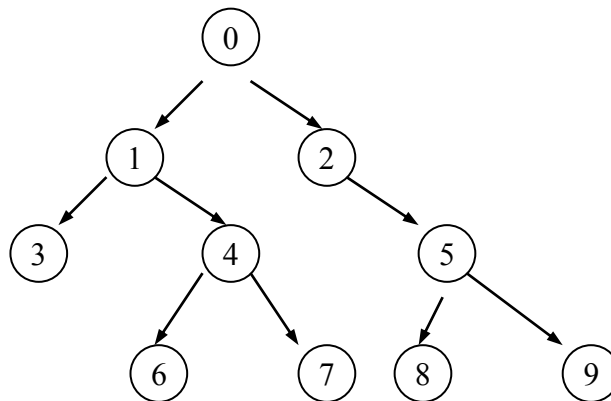
- рекурсивно обработать левое поддерево текущего поддерева
- обработать вершину текущего поддерева
- рекурсивно обработать правое поддерево

### 3. Обход в обратном направлении:

- рекурсивно обработать левое поддерево текущего поддерева
- рекурсивно обработать правое поддерево
- затем – вершину текущего поддерева

|                                                                                   |                                                                                   |                                                                                     |
|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
|  |  |  |
| Прямой обход: A - B - C                                                           | Симметричный обход: B - A - C                                                     | Обратный обход: B - C - A                                                           |

В качестве **примера** по шагам рассмотрим обход следующего ДД с числовыми компонентами (10 вершин):



Обход в прямом порядке:

1. Выделяем поддерево 0-1-2
2. обрабатываем его корень – вершину **0**
3. переходим к левому потомку и выделяем поддерево 1-3-4
4. обрабатываем его корень – вершину **1**
5. выделяем левое поддерево 3-\*-\* (здесь \* обозначает пустую ссылку)
6. обрабатываем его корень – вершину **3**
7. т.к. левого потомка нет, обрабатываем правое поддерево
8. т.к. правого поддерева нет, возвращаемся к поддереву 1-3-4
9. выделяем поддерево 4-6-7

10. обрабатываем его корень – вершину **4**
11. выделяем левое поддерево 6-\*-\*
12. обрабатываем его корень – вершину **6**
13. т.к. левого потомка нет, обрабатываем правое поддерево
14. т.к. правого потомка нет, то возвращаемся к поддереву 4-6-7
15. выделяем правое поддерево 7-\*-\*
16. обрабатываем его корень – вершину **7**
17. т.к. левого поддерева нет, обрабатываем правое поддерево
18. т.к. правого поддерева нет, то возвращаемся к поддереву 4-6-7
19. т.к. поддерево 4-6-7 обработано, то возвращаемся к поддереву 1-3-4
20. т.к. поддерево 1-3-4 обработано, возвращаемся к поддереву 0-1-2
21. выделяем правое поддерево 2-\*-\*5
22. обрабатываем его корень – вершину **2**
23. т.к. левого потомка нет, обрабатываем правого потомка
24. выделяем поддерево 5–8–9
25. обрабатываем его корень – вершину **5**
26. выделяем левое поддерево 8-\*-\*
27. обрабатываем его корень – вершину **8**
28. т.к. левого поддерева нет, обрабатываем правое поддерево
29. т.к. правого поддерева нет, то возвращаемся к поддереву 5-8-9
30. выделяем правое поддерево 9-\*-\*
31. обрабатываем его корень – вершину **9**
32. т.к. левого поддерева нет, обрабатываем правое поддерево
33. т.к. правого поддерева нет, то возвращаемся к поддереву 5-8-9
34. т.к. поддерево 5-8-9 обработано, то возвращаемся к поддереву 2-\*-\*5
35. т.к. поддерево 2-\*-\*5 обработано, то возвращаемся к поддереву 0-1-2
36. т.к. поддерево 0-1-2 полностью обработано, то обход закончен

В итоге получаем следующий порядок обхода вершин: **0-1-3-4-6-7-2-5-8-9**

В более краткой записи симметричный обход дает следующие результаты:

|                    |                     |                     |                     |
|--------------------|---------------------|---------------------|---------------------|
| 1. поддерево 0-1-2 | 6. поддерево 4-6-7  | 10. вершина 7       | 15. поддерево 8-*-* |
| 2. поддерево 1-3-4 | 7. поддерево 6-*-*  | 11. вершина 0       | 16. вершина 8       |
| 3. поддерево 3-*-* | 8. вершина 6        | 12. поддерево 2-*5  | 17. вершина 5       |
| 4. вершина 3       | 9. вершина 4        | 13. вершина 2       | 18. поддерево 9-*-* |
| 5. вершина 1       | 10. поддерево 7-*-* | 14. поддерево 5-8-9 | 19. вершина 9       |

Итого: **3-1-6-4-7-0-2-8-5-9**

Аналогично, обход в обратном порядке дает:

|                    |                     |               |
|--------------------|---------------------|---------------|
| 1. поддерево 0-1-2 | 6. вершина 7        | 11. вершина 8 |
| 2. поддерево 1-3-4 | 7. вершина 4        | 12. вершина 9 |
| 3. вершина 3       | 8. вершина 1        | 13. вершина 5 |
| 4. поддерево 4-6-7 | 9. поддерево 2-*5   | 14. вершина 2 |
| 5. вершина 6       | 10. поддерево 5-8-9 | 15. вершина 0 |

Итого: **3-6-7-4-1-8-9-5-2-0**

Видно, что результирующая последовательность вершин существенно зависит от правила обхода. Иногда используются разновидности трех основных правил, например – обход в обратно-симметричном порядке: правое поддерево – корень – левое поддерево.

Учитывая рекурсивный характер правил обхода, программная их реализация наиболее просто может быть выполнена с помощью рекурсивных подпрограмм. **Каждый рекурсивный вызов отвечает за обработку своего текущего поддерева.** Из приведенных выше примеров видно, что после полной обработки текущего поддерева происходит возврат к поддереву более высокого уровня, а для этого надо запоминать и в дальнейшем восстанавливать адрес корневой вершины этого поддерева. Рекурсивные вызовы позволяют выполнить это запоминание и восстановление автоматически, если описать адрес корневой вершины поддерева как формальный параметр рекурсивной подпрограммы.

Каждый рекурсивный вызов прежде всего должен проверить переданный ей адрес на **nil**. Если этот адрес равен **nil**, то очередное обрабатываемое поддерево является пустым и никакая его обработка не нужна, поэтому просто происходит возврат из рекурсивного вызова. В противном случае в соответствии с реализуемым правилом обхода производится либо обработка вершины, либо рекурсивный вызов для обработки левого или правого поддерева.

Рекурсивная реализация обхода в прямом направлении:

**Procedure** Forward ( pCurrent : Tp );

**Begin**

**If** pCurrent  $\neq$  nil **then**

**Begin**

“обработка корневой вершины pCurrent^”;

Forward (pCurrent^.Left);

Forward (pCurrent^.Right);

**End;**

**End;**

Первоначальный вызов рекурсивной подпрограммы производится в главной программе, в качестве стартовой вершины задаётся адрес корневой вершины дерева: Forward (pRoot).

Остальные две процедуры обхода с именами Symmetric и Back отличаются только порядком следования трех основных инструкций в теле условного оператора.

Для симметричного прохода:

Symmetric (pCurrent^.Left);

“обработка корневой вершины pCurrent^”;

Symmetric (pCurrent^.Right);

Для обратного прохода:

Back (pCurrent^.Left);

Back (pCurrent^.Right);

“обработка корневой вершины pCurrent^”;

В принципе, достаточно легко реализовать **нерекурсивный** вариант процедур обхода, если учесть, что рекурсивные вызовы и возвраты используют **стековый** принцип работы. Например, рассмотрим схему реализации нерекурсивного симметричного обхода. В соответствии с данным правилом сначала надо обработать всех левых потомков, т.е. спустится влево максимально глубоко. Каждое продвижение вниз к левому потомку приводит к

запоминанию в стеке адреса бывшей корневой вершины. Тем самым для каждой вершины в стеке запоминается путь к этой вершине от корня дерева.

Обращение к рекурсивной процедуре для обработки левого потомка надо заменить помещением в стек адреса текущей корневой вершины и переходом к левому потомку этой вершины. Обработка правого потомка заключается в извлечении из стека адреса некоторой вершины и переходе к её правому потомку.

Для нерекурсивного обхода дерева необходимо объявить вспомогательную структуру данных – стек. В информационной части элементов стека должны храниться адреса узлов этого дерева, поэтому ее надо описать с помощью соответствующего ссылочного типа Тр.

Схематично нерекурсивный симметричный обход выглядит следующим образом:

```
pCurrent := pRoot; {начинаем с корневой вершины дерева}
```

```
Stop := false; {вспомогательная переменная}
```

```
while (not stop) do {основной цикл обхода}
```

```
begin
```

```
 while pCurrent <> nil do {обработка левых потомков}
```

```
 begin
```

```
 “занести pCurrent в стек”;
```

```
 pCurrent := pCurrent^.left;
```

```
 end;
```

```
 if “стек пуст” then stop:= true {обход закончен}
```

```
 else
```

```
 begin
```

```
 “извлечь из стека адрес и присвоить его pCurrent ”;
```

```
 “обработка узла pCurrent ”;
```

```
 pCurrent := pCurrent^.right;
```

```
 end;
```

```
end;
```



На основе процедур обхода легко можно реализовать поиск в дереве вершины с заданным информационным значением. Для этого каждая текущая вершина проверяется на совпадение с заданным значением и в случае успеха происходит завершение обхода.

Еще одним интересным применением процедур обхода является уничтожение всего дерева с освобождением занимаемой вершинами памяти. Ясно, что в простейшем поддереве надо сначала удалить левого и правого потомка, а уже затем – самую корневую вершину. Здесь наилучшим образом подходит правило обхода в обратном направлении.

Разные правила обхода часто используются для вывода структуры дерева в наглядном графическом виде. Например, для рассмотренного выше дерева с десятью вершинами применение разных правил обхода позволяет получить следующие представления дерева:

|              |                      |                     |
|--------------|----------------------|---------------------|
|              |                      |                     |
| Прямой обход | Обратно-симметричный | Симметричный проход |

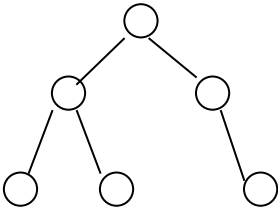
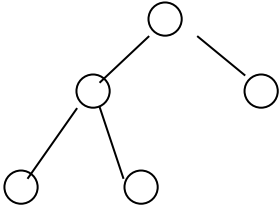
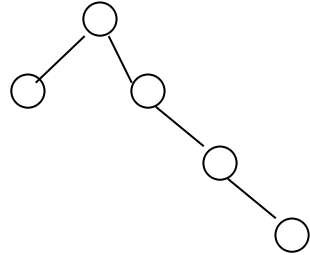
Из этих примеров видно, что наличие нескольких правил обхода дерева вполне обоснованно, и в каждой ситуации надо выбирать подходящее правило.

### 5.3. Идеально сбалансированные деревья

В заключение данной темы рассмотрим один частный случай ДД – так называемое **идеально сбалансированное дерево (ИСД)**. Как будет отмечено в дальнейшем, эффективное использование деревьев на практике часто требует управления ростом дерева для устранения крайних случаев, когда дерево

вырождается в линейный список и тем самым теряет всю свою привлекательность (с вычислительной точки зрения, разумеется).

В этом смысле ИСД полностью оправдывает свое название, поскольку вершины в нем распределяются наиболее равномерно и тем самым ИСД имеет **минимально возможную высоту**. Более точно, ДД называется идеально сбалансированным, если для **каждой** вершины число вершин в левом и правом ее поддеревьях отличаются не более чем на единицу. Обратим внимание, что данное условие должно выполняться для всех вершин дерева!

|                                                                                   |                                                                                   |                                                                                     |
|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
|  |  |  |
| Это дерево - ИСД                                                                  | Это – не ИСД (нарушение условия для корня!)                                       | Это тоже не ИСД (совсем плохо, почти список!)                                       |

ИСД легко строится, если заранее известно количество вершин  $N$  в этом дереве. В этом случае ИСД можно построить с помощью следующего рекурсивного алгоритма:

- взять первую по порядку вершину в качестве корневой
- найти количество вершин в левых и правых поддеревьях:  
 $N_l = N \text{ div } 2;$   
 $N_r = N - N_l - 1;$
- построить левое поддерево с  $N_l$  вершинами точно таким же образом (пока не получим  $N_l = 0$ )
- построить правое поддерево с  $N_r$  вершинами точно таким же образом (пока не получим  $N_r = 0$ )

Естественно, что реализация рекурсивного алгоритма наиболее просто выполняется в виде рекурсивной подпрограммы. При этом между этой процедурой и процедурами обхода есть одно принципиальное различие: процедуры обхода лишь **используют** существующую структуру дерева, не

изменяя ее, и поэтому их формальные параметры являются лишь входными, тогда как процедура построения ИСД должна **СОЗДАВАТЬ** вершины и каждый раз возвращать в вызвавшую ее подпрограмму адрес очередной созданной вершины. Поэтому формальный параметр ссылочного типа должен быть объявлен как **параметр-переменная**. Кроме того, второй формальный параметр-значение принимает число вершин в текущем строящемся поддереве.

```
procedure AddNodes (var pCurrent : Tp; aN : integer);
var pTemp : Tp;
 Nl, Nr : integer;
begin
 if aN = 0 then { вершин для размещения нет }
 pCurrent := nil { формируем пустую ссылку }
 else
 begin
 Nl := aN div 2; { сколько вершин будет слева? }
 Nr := aN - Nl - 1; { сколько вершин будет справа? }
 New (pTemp); { создаем корень поддерева }
 AddNodes (pTemp^.left, Nl); { уходим на создание левого поддерева }
 AddNodes (pTemp^.right, Nr); { уходим на создание правого поддерева }
 pCurrent := pTemp; { возвращаем адрес созданного корня }
 end
 end;
```

Запуск процесса построения как обычно выполняется из главной программы с помощью вызова `AddNodes(pRoot, N)`. В этом вызове фактический параметр `N` обязательно должен иметь конкретное значение, например – заданное пользователем количество вершин в строящемся ИСД. Однако, первый фактический параметр `pRoot`, являясь выходным, получит свое значение лишь **после отработки** всех рекурсивных вызовов, при возврате в главную программу.

Для понимания работы приведенной процедуры целесообразно вручную расписать шаги ее работы для простейшего дерева из трех вершин. Пусть элементами дерева являются символы A, B, C. В результате работы мы должны получить: pRoot -> A, A.left -> B, A.right -> C, B.left -> **nil**, B.right -> **nil**, C.left -> **nil**, C.right -> **nil**.

Тогда схема построения ИСД будет:

- Главная программа: вызов AddNodes (pRoot, 3)
  - П/п 1: Nl = 1, Nr = 1, создание вершины A, вызов AddNodes (A.left, 1)
    - П/п 1-2: сохранение в стеке значений Nl = 1, Nr = 1, адреса A
    - П/п 1-2: Nl = 0, Nr = 0, создание вершины B, вызов AddNodes (B.left, 0)
      - П/п 1-2-3: сохранение в стеке значений Nl = 0, Nr = 0, адреса B
      - П/п 1-2-3: aN = 0, pCurrent = **nil**, возврат к 1-2
    - П/п 1-2: восстановление Nl = 0, Nr = 0, вых. параметр B.left = **nil**
    - П/п 1-2: вызов AddNodes (B.right, 0)
      - П/п 1-2-3: сохранение в стеке значений Nl = 0, Nr = 0, адреса B
      - П/п 1-2-3: aN = 0, pCurrent = **nil**, возврат к 1-2
    - П/п 1-2: восстановление Nl = 0, Nr = 0, вых. параметр B.right = **nil**
    - П/п 1-2: pCurrent = адрес B, возврат к 1
  - П/п 1: восстановление Nl = 1, Nr = 1, вых. параметр A.left=адрес B
  - П/п 1: вызов AddNodes (A.right, 1)
    - П/п 1-2: сохранение в стеке значений Nl = 1, Nr = 1, адреса A
    - П/п 1-2: Nl = 0, Nr = 0, создание вершины C, вызов AddNodes (C.left, 0)
      - П/п 1-2-3: сохранение в стеке значений Nl = 0, Nr = 0, адреса C
      - П/п 1-2-3: aN = 0, pCurrent = **nil**, возврат к 1-2
    - П/п 1-2: восстановление Nl = 0, Nr = 0, вых. параметр C.left = **nil**
    - П/п 1-2: вызов AddNodes (C.right, 0)
      - П/п 1-2-3: сохранение в стеке значений Nl = 0, Nr = 0, адреса C

- П/п 1-2-3:  $aN = 0$ ,  $pCurrent = \mathbf{nil}$ , возврат к 1-2
- П/п 1-2: восстановление  $Nl = 0$ ,  $Nr = 0$ , вых. параметр  $C.right = \mathbf{nil}$
- П/п 1-2:  $pCurrent = \text{адрес } C$ , возврат к 1
- П/п 1: восстановление  $Nl = 1$ ,  $Nr = 1$ , вых. параметр  $A.right = \text{адрес } C$
- П/п 1:  $pCurrent = \text{адрес } A$ , возврат в главную программу
- Главная программа: установка выходного параметра  $pRoot = \text{адрес } A$

#### 5.4. Практические задания

**Задание 1.** Построение и обход идеально сбалансированных двоичных деревьев. Реализовать программу, выполняющую следующий набор операций:

- построение идеально сбалансированного двоичного дерева с заданным числом вершин
- построчный вывод дерева на основе процедуры обхода в прямом порядке
- построчный вывод дерева на основе процедуры обхода в симметричном порядке
- построчный вывод дерева на основе процедуры обхода в обратнo-симметричном порядке

Рекомендации:

- для простоты построения дерева можно информационную часть формировать как случайное целое число в интервале от 0 до 99
- глобальные переменные: указатель на корень дерева и число вершин
- алгоритмы построения ИСД и его обхода оформляются как подпрограммы, вызываемые из главной программы
- все процедуры обхода должны выводить вершины с числом отступов, пропорциональным уровню вершины: корень дерева не имеет отступов, вершины первого уровня выводятся на 5 отступов правее, вершины 2-го уровня – еще на 5 отступов правее и т.д. Для этого в рекурсивные подпрограммы обхода надо ввести второй формальный параметр - уровень этой вершины

- Все процедуры обхода имеют похожую структуру. Например, процедура обхода в прямом направлении должна:
  - проверить пустоту очередного поддерева
  - вывести в цикле необходимое число пробелов в соответствии с уровнем вершины
  - вывести информационную часть текущей вершины
  - вызвать рекурсивно саму себя для обработки своего левого поддерева с увеличением уровня на 1
  - вызвать рекурсивно саму себя для обработки своего правого поддерева с увеличением уровня на 1

Сравнение рассмотренных правил вывода двоичного дерева приводится в следующей таблице

| Исходное дерево                                                                | Вывод в прямом порядке                                       | Вывод в симметричном порядке                                                                                 | Вывод в обратнo-симметричном порядке                                                                        |
|--------------------------------------------------------------------------------|--------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|
| <pre>       10      /  \     15   9    /  \ /  \   13 22 5 17           </pre> | <pre> 10  15     13     22   9     5     17           </pre> | <pre>       13      /  \     15   22    /      \   10       5      \      \      9       17           </pre> | <pre>       17      /  \     9    5    /      \   10       22      \      \      15     13           </pre> |

Главная программа реализует следующий набор действий:

- запрос числа вершин в дереве
- запуск рекурсивной подпрограммы построения идеально сбалансированного дерева со следующими фактическими параметрами: указатель на корень дерева (при построении дерева этот параметр является выходным!) и заданное число вершин
- последовательный вызов подпрограмм обхода дерева со следующими фактическими входными параметрами: указатель на корень дерева, ноль в качестве уровня корневой вершины дерева.

**Задание 2.** Добавить в программу **нерекурсивный** вариант процедуры обхода дерева в симметричном порядке.

Замена рекурсии циклом основана на использовании вспомогательного стека для хранения последовательности пройденных вершин от корня до текущей вершины и уровня этих вершин в дереве (напомним, что уровень используется только для задания правильного числа отступов при построчном выводе дерева). Исходя из этого, создаваемая подпрограмма должна объявить необходимые локальные типы данных для динамической реализации вспомогательного стека. Каждый элемент стека должен хранить: указатель на пройденную вершину дерева, уровень вершины, указатель на следующий элемент стека. Для реализации стека, как обычно, требуются две ссылочные переменные: указатель на вершину стека и вспомогательный указатель, используемый при добавлении или удалении элементов в стек.

Кодовая часть подпрограммы должна начинаться с инициализации необходимых переменных: текущая вершина дерева – его корень, вспомогательный стек – пустой, начальный уровень равен (-1). После этого запускается основной цикл обработки дерева, включающий выполнение следующих действий:

- циклическое сохранение очередной вершины в стеке (пока не будет достигнута пустая вершина) следующим образом:
  - увеличение уровня вершины на 1
  - создание нового элемента стека, заполнение всех его полей и добавление его в стек
  - переход к левому потомку текущей вершины
- проверка пустоты стека: если стек пуст, то основной цикл следует завершить, а иначе – перейти к обработке вершины
  - извлечь из стека адрес текущей обрабатываемой вершины и ее уровень
  - вывести вершину с необходимым числом пробелов
  - удалить элемент из стека с освобождением памяти
  - перейти к правому потомку текущей вершины

Для проверки правильности работы подпрограммы сравнить ее результаты с рекурсивным аналогом.

### Задание 3. Обработка произвольных двоичных деревьев

Реализовать программу, выполняющую следующий набор операций с двоичными деревьями:

- поиск вершины с заданным значением информационной части
- добавление левого или правого потомка для заданной вершины
- построчный вывод дерева с помощью основных правил обхода
- уничтожение всего дерева

Рекомендации:

- объявить необходимые глобальные переменные: указатель на корень дерева, указатель на родительскую вершину, признак успешности поиска
- объявить и реализовать рекурсивную подпрограмму поиска. В качестве основы можно взять любой из известных вариантов обхода дерева. Основное отличие рекурсивного поиска от рекурсивного обхода состоит в необходимости прекращения обхода дерева в случае совпадения информационной части текущей вершины с заданным значением. Один из способов прекращения обхода основан на использовании булевского признака, который перед запуском обхода устанавливается в **false** и переключается в **true** при обнаружении искомой вершины. Для каждой текущей вершины подпрограмма поиска должна выполнять следующие основные действия:

- проверить необходимость продолжения поиска
- проверить текущую ссылочную переменную на **nil**
- сравнить информационную часть текущей вершины с заданным значением
- если эти величины совпадают, то установить признак окончания поиска и установить адрес родительской переменной в адрес текущей вершины



- в противном случае продолжить поиск сначала в левом поддереве текущей вершины, вызвав рекурсивно саму себя с адресом левого потомка, а потом – в правом поддереве, вызвав саму себя с адресом правого потомка

Результат поиска можно проверить с помощью глобального признака. В случае успешного поиска становится известен адрес найденной вершины, который можно использовать в дальнейшем для добавления к этой вершине одного из потомков.

- Объявить и реализовать подпрограмму добавления новой вершины в дерево как потомка заданной вершины.

Подпрограмма должна:

- проверить пустоту дерева: если указатель корня имеет значение **nil**, то надо создать корень дерева
  - выделить память
  - запросить значение информационной части корня
  - сформировать пустые ссылочные поля на потомков
- если дерево не пустое, то организовать поиск родительской вершины:
  - запросить искомое значение информационной части родительской вершины
  - установить признак поиска и вызвать подпрограмму поиска
  - если поиск удачен, то проверить число потомков у найденной родительской вершины
  - если вершина-родитель имеет двух потомков, то добавление невозможно
  - если родительская вершина имеет только одного потомка, то сообщить о возможности добавления одного из потомков, выделить память для новой вершины и заполнить все три ее поля, настроить соответствующее ссылочное поле у родительской вершины

- если родительская вершина не имеет ни одного потомка, то запросить тип добавляемой вершины (левый или правый потомок) и выполнить само добавление
- Объявить и реализовать рекурсивную подпрограмму для построчного вывода дерева в обратно-симметричном порядке. Эту подпрограмму без каких-либо изменений можно взять из предыдущей работы.
- Объявить и реализовать подпрограмму для уничтожения всего дерева с освобождением памяти. Основой подпрограммы является рекурсивный обход дерева, причем – по правилу **обратного** обхода: сначала посещается и удаляется левый потомок текущей вершины, потом посещается и удаляется правый потомок, и лишь затем удаляется сама текущая вершина. Такой обход позволяет не разрывать связи между родительской вершиной и потомками до тех пор, пока не будут удалены оба потомка. Подпрограмма удаления имеет один формальный параметр – адрес текущей вершины. Подпрограмма должна проверить указатель на текущую вершину, и если он не равен **nil**, то:
  - вызвать рекурсивно саму себя с адресом левого поддерева
  - вызвать рекурсивно саму себя с адресом правого поддерева
  - вывести для контроля сообщение об удаляемой вершине
  - освободить память с помощью процедуры **Dispose** и текущего указателя

Главная программа должна:

- создать пустое дерево
- организовать цикл для добавления вершины с вызовом соответствующей подпрограммы и последующим построчным выводом дерева
- предоставить возможность в любой момент вызвать подпрограмму удаления дерева с фактическим параметром, равным адресу корня дерева, что запускает механизм рекурсивного удаления всех вершин, включая и корневую; поскольку после удаления корневой вершины

соответствующий указатель становится неопределенным, можно инициировать его пустым значением, что позволит повторно создать дерево с новой корневой вершиной

### **5.5. Контрольные вопросы по теме**

1. В чем состоит основное отличие древовидных структур от списковых?
2. Как рекурсивно определяется дерево?
3. Какие типы вершин существуют в деревьях?
4. Какие можно выделить типы деревьев?
5. Какие деревья называются двоичными?
6. Какие деревья называются упорядоченными?
7. Какие основные понятия связываются с деревьями?
8. Какие основные операции характерны при использовании деревьев?
9. Какую структуру имеют вершины двоичного дерева?
10. Почему для деревьев существует несколько правил обхода вершин?
11. Какие правила обхода вершин дерева являются основными?
12. Как выполняется обход дерева в прямом направлении?
13. Как выполняется обход дерева в симметричном направлении?
14. Как выполняется обход дерева в обратном направлении?
15. Как выполняется обход дерева в обратно-симметричном направлении?
16. Почему рекурсивная реализация правил обхода является наиболее удобной?
17. Что происходит при рекурсивном выполнении обхода дерева?
18. Как программно реализуется обход дерева в прямом направлении?
19. Как программно реализуется обход дерева в симметричном направлении?
20. Как программно реализуется обход дерева в обратном направлении?
21. Какой формальный параметр необходим для рекурсивной реализации правил обхода и как он используется?
22. В чем состоит суть нерекурсивной реализации процедур обхода?

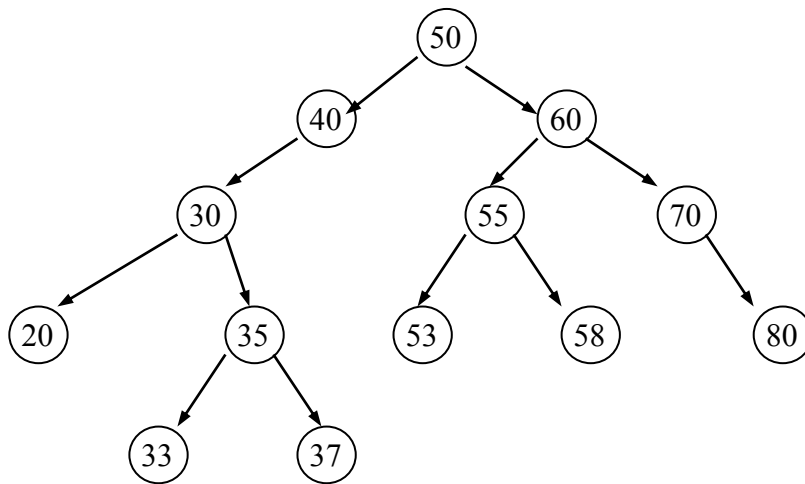
23. Какая вспомогательная структура данных необходима для нерекурсивной реализации обхода дерева и как она используется?
24. Опишите схему процедуры для нерекурсивного обхода дерева.
25. Как выполняется поиск в дереве вершины с заданным ключом?
26. Как правильно выполнить уничтожение всей древовидной структуры?
27. Какое дерево называется идеально сбалансированным?
28. В чем заключается значимость идеально сбалансированных деревьев с точки зрения организации поиска?
29. Опишите алгоритм построения идеально сбалансированного дерева.
30. В чем состоит принципиальное отличие алгоритмов обхода деревьев от алгоритма построения идеально сбалансированного дерева?
31. Почему ссылочный параметр в рекурсивной процедуре построения идеально сбалансированного дерева должен быть параметром-переменной?
32. Какие формальные параметры должна иметь рекурсивная подпрограмма построения идеально сбалансированного дерева и для чего они используются?
33. Приведите программную реализацию процедуры построения идеально сбалансированного дерева.

## Тема 6. Реализация поисковых деревьев

### 6.1. Двоичные деревья поиска.

Деревья поиска – частный, но практически, пожалуй, наиболее важный вид двоичных деревьев. Будем считать, что каждая вершина имеет некое **ключевое** поле, позволяющее **упорядочить** множество вершин. Двоичное дерево называется деревом поиска или поисковым деревом, если для **каждой** вершины дерева все ключи в ее **левом** поддереве **меньше** ключа этой вершины, а все ключи в ее **правом** поддереве **больше** ключа вершины.

Пример дерева поиска с целыми ключами представлен на следующем рисунке.



Деревья поиска являются одной из наиболее эффективных структур построения **упорядоченных** данных. Как известно, в упорядоченном массиве очень эффективно реализуется поиск (дихотомия!), но очень тяжело выполнить добавление и удаление элементов. Наоборот, в упорядоченном списке легко реализуется добавление и удаление элементов, но не эффективна реализация поиска из-за необходимости последовательного просмотра всех элементов, начиная с первого. Деревья поиска позволяют **объединить преимущества** массивов и линейных списков: легко реализуется добавление и удаление элементов, а также эффективно выполняется поиск.

Эффективность процесса поиска в ДП определяется дихотомической структурой самого дерева. На каждом шаге поиска, начиная с корня, **отбрасывается одно из поддеревьев** - левое или правое. Двоичный поиск наиболее эффективен для идеально сбалансированных деревьев или близких к ним. В самом плохом случае число сравнений равно высоте дерева. При этом, как и в методе двоичного поиска в упорядоченном массиве, сравнительная эффективность поиска в ДП быстро увеличивается при увеличении числа вершин в этом дереве.

Например, если идеально сбалансированное ДП имеет 1000 вершин, то даже в наихудшем случае потребуется не более 10 сравнений ( $2$  в степени  $10$  есть  $1024$ ), а если число вершин 1 миллион, то потребуется не более 20 сравнений.

Можно сделать следующий **вывод**: ДП следует использовать для представления упорядоченных данных, когда число их достаточно велико и часто приходится выполнять операции добавления, удаления и поиска.

Интересно заметить, что обход ДП в симметричном порядке позволяет получить исходный набор данных в соответствии с заданным упорядочиванием, а обход в обратно-симметричном порядке – в обратном порядке.

**Алгоритм поиска** в ДП очень прост:

Начиная с корневой вершины для каждого текущего поддерева надо выполнить следующие шаги:

- сравнить ключ вершины с заданным значением  $x$
- если заданное значение меньше ключа вершины, перейти к левому потомку, иначе перейти к правому поддереву.

Поиск прекращается при выполнении одного из двух условий:

- либо если найден искомый элемент
- либо если надо продолжать поиск в пустом поддереве, что является признаком отсутствия искомого элемента

Интересно, что поиск очень легко можно реализовать простым циклом, без использования рекурсии:

```
pCurrent := pRoot; {начинаем поиск с корня дерева}
```

```
Stop := false;
```

```
While (pCurrent <> nil) and (not Stop) do
```

```
 if $x < \text{pCurrent}^{\wedge}.\text{inf}$ then pCurrent := pCurrent^.left else
```

```
 if $x > \text{pCurrent}^{\wedge}.\text{inf}$ then pCurrent := pCurrent^.right else Stop := true;
```

## 6.2. Добавление вершины в дерево поиска

Немного сложнее реализуется операция добавления нового элемента в ДП. Прежде всего, надо найти подходящее место для нового элемента, поэтому добавление неразрывно связано с процедурой поиска. Будем считать, что в дерево могут добавляться элементы с одинаковыми ключами, и для этого с

каждой вершиной свяжем счетчик числа появления этого ключа. В процесс поиска может возникнуть одна из двух ситуаций:

- найдена вершина с заданным значением ключа, и в этом случае просто увеличивается счетчик
- поиск надо продолжать по пустой ссылке, что говорит об отсутствии в дереве искомой вершины, более того, тем самым определяется место в дереве для размещения новой вершины.

Само добавление включает следующие шаги:

- выделение памяти для новой вершины
- формирование информационной составляющей
- формирование двух пустых ссылочных полей на будущих потомков
- формирование в родительской вершине левого или правого ссылочного поля – адреса новой вершины

Здесь только последняя операция вызывает некоторую сложность, поскольку для доступа к ссылочному полю родительской вершины надо знать ее адрес. Ситуация аналогична добавлению элемента в линейный список перед заданным, когда для отслеживания элемента-предшественника при проходе по списку использовался дополнительный указатель. Этот же прием можно использовать и для деревьев, но есть более элегантное решение – **рекурсивный** поиск с добавлением новой вершины при необходимости. Каждый рекурсивный вызов отвечает за обработку очередной вершины дерева, начиная с корня, а вся последовательность вложенных вызовов позволяет автоматически запоминать путь от корня до любой текущей вершины. Процедура поиска должна иметь формальный параметр-переменную ссылочного типа, который отслеживает адрес текущей вершины дерева и как только этот адрес становится пустым, создается новая вершина и ее адрес возвращается в вызвавшую процедуру, тем самым автоматически формируя необходимую ссылку в родительской вершине.

**Procedure** AddNode ( **var** pCurrent : Tp);

**begin**

```

if pCurrent = nil then {место найдено, создать новую вершину}
begin
 New (pCurrent); {параметр pCurrent определяет адрес новой вершины}
 pCurrent^.inf := “необходимое значение”;
 pCurrent^.left := nil; pCurrent^.right := nil;
 “установка значения поля счетчика в 1 “;
end
else {просто продолжаем поиск в левом или правом поддереве}
 if x < pCurrent^.inf then AddNode (pCurrent^.left)
 else if x > pCurrent^.inf then AddNode (pCurrent^.right)
 else “увеличить счетчик”

```

**end;**

Запуск процедуры выполняется в главной программе вызовом AddNode(pRoot). Если дерево пустое, т.е. pRoot = **nil**, то первый вызов процедуры создаст корневую вершину дерева, к которой потом можно аналогичными вызовами добавить любое число вершин.

Рассмотрим **нерекурсивный** вариант процедуры добавления вершины в ДП. Необходимо объявить две ссылочные переменные для отслеживания адреса текущей вершины и адреса ее родителя:

```
pCurrent, pParent : Tp;
```

Удобно отдельно рассмотреть случай пустого дерева и дерева хотя бы с одной корневой вершиной:

```

if pRoot = nil then
 begin
 New (pRoot); pRoot^.left := nil; pRoot^.right := nil;
 “заполнение остальных полей”;
 end
else
 begin
 pCurrent := pRoot; {начинаем поиск с корня дерева}

```



```

while (pCurrent <> nil) do
begin
 pParent := pCurrent; {запоминаем адрес родительской вершины}
 if (x < pCurrent^.inf) then pCurrent := pCurrent^.left
 else if (x > pCurrent^.inf) then pCurrent := pCurrent^.right
 else begin {вершина найдена, добавлять не надо, закончить цикл}
 pCurrent := nil;
 “увеличить счетчик”;
 end;
end;
if (x < pParent^.inf) then
begin {добавляем новую вершину слева от родителя}
 New (pCurrent);
 “заполнить поля новой вершины”;
 pParent^.left := pCurrent;
end
else
 if (x > pParent^.inf) then
 begin {добавляем новую вершину справа от родителя}
 New (pCurrent);
 “заполнить поля новой вершины”;
 pParent^.right := pCurrent;
 end
end;

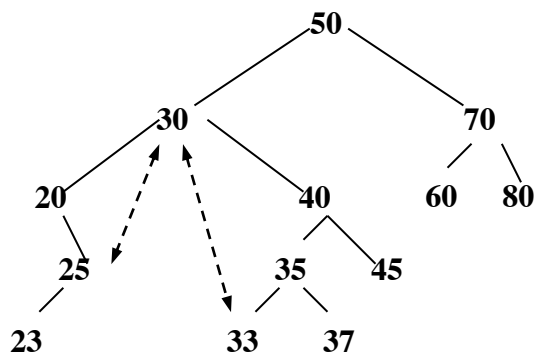
```

### 6.3. Удаление вершины из дерева поиска

Теперь рассмотрим удаление вершины из ДП. По сравнению с добавлением удаление реализуется более сложным алгоритмом, поскольку добавляемая вершина **всегда** является **терминальной**, а удаляться может

**ЛЮБАЯ**, в том числе и нетерминальная. При этом может возникать несколько различных ситуаций.

Рассмотрим фрагмент ДП с целыми ключами.



**Ситуация 1.** Удаляемая вершина **не имеет ни одного потомка**, т.е. является терминальной. Удаление реализуется очень легко обнулением соответствующего указателя у родителя. Например, для удаления вершины с ключом 23 достаточно установить  $25.\text{left} = \text{nil}$ .

**Ситуация 2.** Удаляемая вершина имеет **только одного потомка**. В этом случае удаляемая вершина вместе со своим потомком и родителем образуют фрагмент линейного списка. Удаление реализуется простым изменением указателя у родительского элемента. Например, для удаления вершины с ключом 20 достаточно установить  $30.\text{left} = 20.\text{right} = 25$ .

**Ситуация 3.** Пусть удаляемая вершина имеет **двух потомков**. Этот случай наиболее сложен, поскольку нельзя просто в родительской вершине изменить соответствующее ссылочное поле на адрес одного из потомков удаляемой вершины. Это может нарушить структуру дерева поиска. Например, замена вершины 30 на одного из ее непосредственных потомков 20 или 40 сразу нарушает структуру дерева поиска.

Существует специальное правило для определения вершины, которая должна **заменить** удаляемую. Это правило состоит из двух взаимоисключающих действий:

- либо войти в **левое** поддереву удаляемой вершины и в этом поддереве спустится как можно глубже, придерживаясь только **правых** потомков;

это позволяет найти в дереве ближайшую **меньшую** вершину (например, для вершины 30 это будет вершина 25)

- либо войти в **правое** поддерево удаляемой вершины и спустится в нем как можно глубже придерживаясь только **левых** потомков; это позволяет найти ближайшую **большую** вершину (например, для той же вершины 30 это будет вершина 33).

Перейдем к программной реализации процедуры удаления. Поскольку при удалении могут изменяться связи между внутренними вершинами дерева, удобно (но, конечно же, не обязательно) использовать рекурсивную реализацию. Основная процедура DeleteNode рекурсивно вызывает сама себя для поиска удаляемой вершины. После нахождения удаляемой вершины процедура DeleteNode определяет число ее потомков. Если потомков не более одного, выполняется само удаление. Если потомков два, то вызывается вспомогательная рекурсивная процедура Changer для поиска вершины-заменителя.

```
Procedure DeleteNode (var pCurrent : Tp);
```

```
Var pTemp : Tp;
```

```
 Procedure Changer (var p : Tp);
```

```
 begin
```

```
 {реализация рассматривается ниже}
```

```
 end;
```

```
begin
```

```
 if pCurrent = nil then “удаляемой вершины нет, ничего не делаем”
```

```
 else
```

```
 if x < pCcurrent^.inf then DeleteNode (pCcurrent^.left)
```

```
 else
```

```
 if x > pCurrent^.inf then DeleteNode (pCurrent^.right)
```

```
 else
```

```
 {удаляемая вершина найдена}
```

```
 begin
```

```
 pTemp := pCurrent;
```

```

if pTemp^.right = nil then pCurrent := pTemp^.left
else
 if pTemp^.left = nil then pCurrent := pTemp^.right
 else {два потомка, ищем заменитель}
 Changer (pCurrent^.left); { а можно и pCurrent^.right }
 Dispose (pTemp);
end;
end;

```

Схема процедуры Changer:

```

begin
 if p^.right <> nil then Changer (p^.right)
 else {правое поддереву пустое, заменитель найден, делаем обмен}
 begin
 pTemp^.inf := p^.inf; {заменяем информац. часть удаляемой вершины}
 pTemp := p; {pTemp теперь определяет вершину-заменитель}
 p := p^.left; {выходной параметр адресует левого потомка заменителя}
 end;
 end;
end;

```

Дополнительный комментарий к процедурам. Подпрограмма Changer в качестве входного значения получает адрес левого (или правого) потомка удаляемой вершины и рекурсивно находит вершину-заменитель. После этого информационная часть удаляемой вершины заменяется содержимым вершины-заменителя, т.е. **фактического удаления вершины не происходит**. Это позволяет сохранить неизменными обе связи этой вершины с ее потомками. Удаление реализуется относительно **вершины-заменителя**, для чего ссылка pTemp после обмена устанавливается в адрес этой вершины. Кроме того, в самом конце процедуры Changer устанавливается новое значение выходного параметра p: оно равно адресу левого потомка вершины-заменителя. Это необходимо для правильного изменения адресной части вершины-родителя для

вершины-заменителя. Само изменение происходит при отработке механизма возвратов из рекурсивных вызовов процедуры `Changer`. Когда все эти возвраты отработают, происходит возврат в основную процедуру `DeleteNode`, которая освобождает память, занимаемую вершиной-заменителем. Отметим, что приведенная выше реализация процедуры `Changer` ориентирована на поиск в левом поддереве удаляемой вершины и требует симметричного изменения для поиска заменителя в правом поддереве.

Для окончательного уяснения данного алгоритма настоятельно рекомендуем провести ручную пошаговую отработку его для небольшого примера, как это было сделано для значительно более простого алгоритма построения идеально сбалансированного дерева.

#### 6.4. Практические задания

**Задание 1.** Построение и обработка двоичных деревьев поиска.

Реализовать программу, выполняющую следующий набор операций с деревьями поиска:

- поиск вершины с заданным значением ключа с выводом счетчика числа появлений данного ключа
- добавление новой вершины в соответствии со значением ее ключа или увеличение счетчика числа появлений
- построчный вывод дерева в наглядном виде с помощью обратнo-симметричного обхода
- вывод всех вершин в одну строку по порядку следования ключей с указанием для каждой вершины значения ее счетчика появлений
- удаление вершины с заданным значением ключа

Рекомендации:

- Объявить и реализовать подпрограмму поиска. Поиск начинается с корня дерева и в цикле для каждой вершины сравнивается ее ключ с заданным значением. При совпадении ключей поиска заканчивается с выводом

значения счетчика числа появлений данного ключа. При несовпадении поиск продолжается в левом или правом поддереве текущей вершины

- Объявить и реализовать рекурсивную подпрограмму добавления новой вершины в дерево. Подпрограмма использует один параметр-переменную, определяющую адрес текущей вершины. Если при очередном вызове подпрограммы этот адрес равен **nil**, то производится добавление нового элемента с установкой всех необходимых полей. В противном случае продолжается поиск подходящего места для новой вершины за счет рекурсивного вызова подпрограммы с адресом левого или правого поддерева. При совпадении ключей надо просто увеличить значение счетчика появлений
- Объявить и реализовать не-рекурсивный вариант подпрограммы добавления новой вершины в дерево. Необходимы две ссылочные переменные – адрес текущей вершины и адрес ее родителя. Сначала в цикле ищется подходящее место за счет сравнения ключей и перехода влево или вправо. Поиск заканчивается либо по пустому значению текущего адреса, либо в случае совпадения ключей (здесь просто увеличивается счетчик числа появлений). После окончания поиска либо создается корневая вершина, либо добавляется новая вершина как левый или правый потомок родительской вершины.
- Объявить и реализовать рекурсивную подпрограмму для построчного вывода дерева в обратном-симметричном порядке. Эту подпрограмму без каких-либо изменений можно взять из предыдущей работы.
- Объявить и реализовать рекурсивную подпрограмму для вывода всех вершин в одну строку в соответствии с возрастанием их ключей. Основа – обход в симметричном порядке. Дополнительно рядом со значением ключа в скобках должно выводиться значение счетчика повторений данного ключа.

Например:

5(1) 11(2) 19(5) 33(1) 34(4) . . . . .

- Объявить и реализовать подпрограмму удаления вершины: запрашивается ключ вершины, организуется ее поиск, при отсутствии вершины выводится сообщение, при нахождении вершины проверяется число ее потомков и при необходимости выполняется поиск вершины-заменителя

Главная программа должна предоставлять следующие возможности:

- создание дерева с заданным числом вершин со случайными ключами
- добавление в дерево одной вершины с заданным пользователем значением ключа
- поиск в дереве вершины с заданным ключом
- построчный вывод дерева в наглядном виде
- вывод всех вершин в порядке возрастания их ключей
- удаление вершины с заданным ключом

**Задание 2.** Построение таблицы символических имен с помощью дерева поиска. Постановка задачи формулируется следующим образом.

Деревья поиска широко используются компиляторами при обработке текстов программ на языках высокого уровня. Одной из важнейших задач любого компилятора является выделение во входном тексте программы всех **символических имен** (ключевых слов, пользовательских идентификаторов для обозначения типов, переменных, подпрограмм) и построение их в виде **таблицы** с запоминанием номеров строк, где эти имена встречаются. Поскольку каждое выделенное из текста имя должно отыскиваться в этой таблице, а число повторений одного и того же имени в тексте программы может быть весьма большим, важным для скорости всего процесса компиляции становится скорость поиска в таблице имен. Поэтому очень часто подобные таблицы реализуются в виде дерева поиска. **Ключом** поиска в таких деревьях являются **текстовые имена**, которые добавляются в дерево поиска в соответствии с обычными правилами упорядочивания по алфавиту.

Необходимо реализовать программу, выполняющую следующие действия:

- запрос имени исходного файла с текстом анализируемой программы

- чтение очередной строки и выделение из нее всех символических имен (будем считать, что имя – любая непрерывная последовательность букв и цифр, начинающаяся с буквы и не превышающая по длине 255 символов)
- запоминание очередного имени в дереве поиска вместе с номером строки исходного текста, где это имя найдено; поскольку одно и то же имя в тексте может встречаться многократно в разных строках, приходится с каждым именем-вершиной связывать вспомогательный линейный список номеров строк
- вывод построенной таблицы имен по алфавиту с помощью процедуры обхода дерева в симметричном порядке
- строчный вывод построенного дерева в наглядном представлении с помощью процедуры обхода дерева в обратно-симметричном порядке

Для упрощения программной реализации будем считать, что обрабатываемая программа удовлетворяет следующим непринципиальным условиям:

- в именах используются только строчные (малые) буквы
- отсутствуют комментарии
- отсутствуют текстовые константы, задаваемые с помощью кавычек ‘ ’

Например, пусть исходная программа имеет следующий вид:

```
program test;
var x, y : integer;
 str : string;
begin
 x := 1;
 y := x + 2;
 write(x, y);
end.
```

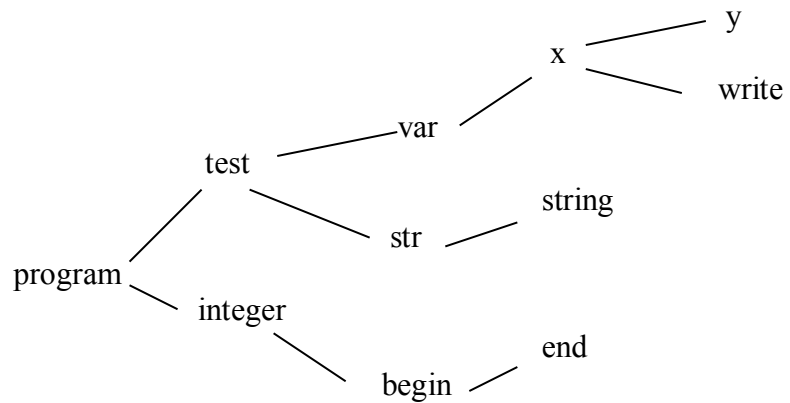
Тогда для нее должна быть построена следующая таблица имен и дерево поиска:



```

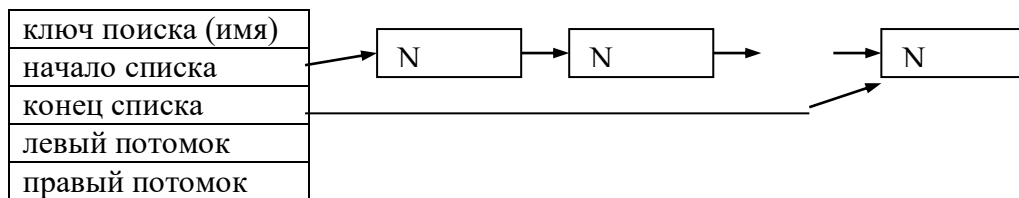
begin 4
end 8
integer 2
program 1
str 3
string 3
test 1
var 2
write 7
x 2 5 6 7
y 2 6 7

```



### Рекомендации:

1. В разделе описания типов ввести два ссылочных типа данных для динамической реализации дерева поиска и вспомогательных линейных списков. Описать связанные с ними базовые типы, определяющие структуру вершин дерева и узлов списка. Вершина дерева должна содержать строковое ключевое поле, указатели на двух потомков и два указателя на начало и конец вспомогательного линейного списка.



2. Объявить необходимые глобальные переменные, такие как номер обрабатываемой строки исходного текста и ее длина, номер обрабатываемого символа в строке (счетчик символов), сама текущая строка, текущее формируемое имя, имя исходного файла, файловая переменная, указатель на корень дерева
3. Объявить и реализовать рекурсивную подпрограмму добавления вершины в дерево поиска. За основу можно взять рассмотренную в предыдущей работе процедуру добавления, внеся в нее следующие небольшие дополнения:

- при вставке новой вершины создать первый (пока единственный!) узел вспомогательного линейного списка, заполнить его поля и поля созданной вершины дерева необходимыми значениями
  - в случае совпадения текущего имени с ключом одной из вершин дерева добавить в конец вспомогательного списка новый узел с номером строки, где найдено данное имя
4. Объявить и реализовать рекурсивную подпрограмму для вывода построенной таблицы в порядке возрастания имен. Основа – обход дерева в симметричном порядке. Дополнительно для каждой вершины выводится список номеров строк, где используется данное имя. Для этого организуется проход по вспомогательному линейному списку от его начала до конца.
  5. Объявить и реализовать рекурсивную подпрограмму для построчного вывода дерева в обратно-симметричном порядке. Эту подпрограмму без каких-либо изменений можно взять из предыдущей работы.
  6. Реализовать главную программу, выполняющую основную работу по выделению имен из строк обрабатываемого текста. Формируемое имя объявляется как **String**, но обрабатывается как массив символов. Основной механизм формирования имени – посимвольная обработка текущей строки. Как только в строке после не-буквенных символов распознается буква, начинается выделение всех последующих буквенно-цифровых символов и формирование очередного имени. Обрабатываемая строка (тип **String**) рассматривается как массив символов. Удобно перед обработкой строки добавить в ее конце символ пробела. Номер текущего обрабатываемого символа задается переменной-счетчиком. Для отслеживания символов в формируемом имени также необходима переменная-счетчик.

Алгоритм работы главной программы:

- инициализация обработки исходного файла (запрос имени файла, открытие для чтения)

- цикл обработки строк исходного файла:
  - ✓ чтение очередной строки и определение ее длины
  - ✓ добавление в конец строки дополнительного символа пробела
  - ✓ инициализация счетчика символов
  - ✓ организация циклической обработки символов строки по следующему алгоритму:
    - a. если очередной символ есть буква, то:
      - запомнить символ как первую букву имени
      - организовать цикл просмотра следующих символов в строке, пока они являются буквами или цифрами, с добавлением этих символов к формируемому имени
      - после окончания цикла установить длину сформированного имени (можно использовать нулевой элемент массива символов)
      - вызвать подпрограмму для поиска сформированного имени в дереве
    - b. увеличить значение счетчика символов для перехода к анализу следующего символа
- вывод построенной таблицы в алфавитном порядке
- построчный вывод дерева поиска

### **Контрольные вопросы по теме**

1. Какое дерево называется деревом поиска?
2. В чем состоит практическая важность использования деревьев поиска?
3. Какие преимущества имеет использование деревьев поиска для хранения упорядоченных данных по сравнению с массивами и списками?
4. Почему наивысшая эффективность поиска достигается у идеально сбалансированных деревьев?
5. Как находится максимально возможное число шагов при поиске в идеально сбалансированном дереве?

6. Приведите алгоритм поиска в дереве поиска.
7. Как программно реализуется поиск в дереве поиска?
8. Как выполняется добавление новой вершины в дерево поиска?
9. В чем смысл рекурсивной реализации алгоритма добавления вершины в дерево поиска?
10. Какой формальный параметр имеет рекурсивная процедура добавления вершины в дерево поиска и как он используется?
11. Приведите программную реализацию рекурсивной процедуры добавления вершины в дерево поиска.
12. Какие ссылочные переменные необходимы для нерекурсивной реализации процедуры добавления вершины в дерево поиска?
13. Как программно реализуется добавление нерекурсивная процедура добавления вершины в дерево поиска?
14. Какие ситуации могут возникать при удалении вершины из дерева поиска?
15. Что необходимо выполнить при удалении из дерева поиска вершины, имеющей двух потомков?
16. Какие правила существуют для определения вершины-заменителя при удалении из дерева поиска?
17. Опишите алгоритм удаления вершины из дерева поиска.
18. Приведите программную реализацию алгоритма удаления вершины из дерева поиска.

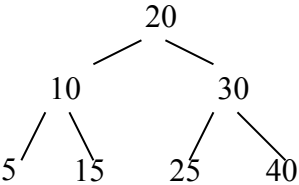
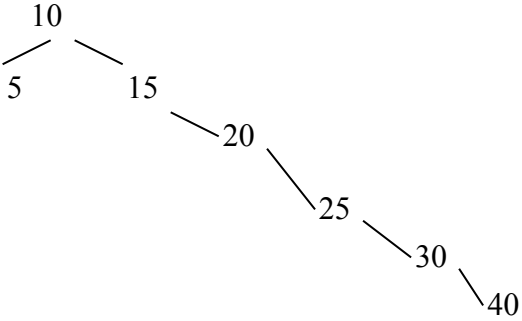
## **Тема 7. Дополнительные вопросы обработки деревьев. Графы.**

### **7.1. Проблемы использования деревьев поиска**

Как было отмечено выше, эффективность поиска с помощью ДП сильно зависит от структуры дерева: чем ближе дерево к идеально сбалансированному, тем меньше высота дерева и тем выше эффективность поиска. К сожалению, входные данные при построении дерева могут быть таковыми, что дерево начинает деформироваться влево или вправо - становится разбалансированным. В крайнем случае, дерево превращается в обычный линейный список, т.е.

становится вырожденным. Отсюда следует, что процесс построения дерева должен **контролироваться** соответствующими алгоритмами, которые при выполнении процедур добавления или удаления могли бы выполнять **перестройку** дерева с целью сохранения его структуры максимально близкой к идеально сбалансированной.

Например, один и тот же входной набор числовых ключей, но поступающий в разном порядке, может приводить к существенно разным ДП.

|                                                                                                                                                                                                              |                                                                                                                                                                                                               |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  <pre> graph TD     20 --&gt; 10     20 --&gt; 30     10 --&gt; 5     10 --&gt; 15     30 --&gt; 25     30 --&gt; 40 </pre> |  <pre> graph TD     10 --&gt; 5     10 --&gt; 15     15 --&gt; 20     20 --&gt; 25     25 --&gt; 30     30 --&gt; 40 </pre> |
| <p>“Хорошая” входная последовательность:<br/>20 – 10 – 30 – 15 – 25 – 5 – 40</p>                                                                                                                             | <p>“Плохая” входная последовательность:<br/>10 – 15 – 20 – 5 – 25 – 30 – 40</p>                                                                                                                               |

Данная проблема исследуется уже около 40 лет, существует несколько методов сохранения “хорошей” структуры дерева. К сожалению, требование идеальной сбалансированности дерева оказалось слишком сильным и до сих пор нет хороших алгоритмов поддержания структуры дерева всегда в идеально сбалансированном виде. Вместо этого сильного требования было предложено несколько более простых, но удобных с вычислительной точки зрения критериев “хорошего” дерева.

Одним из наиболее известных методов балансировки является метод, предложенный в 60-е годы советскими математиками Адельсон–Вельским и Ландисом. Они предложили вместо понятия идеально сбалансированных деревьев использовать понятие **АВЛ-сбалансированных** деревьев, которые хоть и уступают немного идеально сбалансированным по эффективности поиска, но зато имеют не очень сложную программную реализацию.

Дерево поиска называется АВЛ-сбалансированным, если для **каждой** вершины справедливо требование: **высоты** левого и правого поддеревьев отличаются **не больше чем на единицу**.

Очевидно, что любое идеально сбалансированное дерево является также и АВЛ-сбалансированным, но не наоборот.

|                                                                                                            |                                                                                           |                                                                                                                             |
|------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| <pre> graph TD     30 --&gt; 20     30 --&gt; 40     20 --&gt; 10     20 --&gt; 25     40 --&gt; 50 </pre> | <pre> graph TD     30 --&gt; 20     30 --&gt; 40     20 --&gt; 10     20 --&gt; 25 </pre> | <pre> graph TD     30 --&gt; 20     30 --&gt; 40     20 --&gt; 10     20 --&gt; 25     25 --&gt; 22     25 --&gt; 27 </pre> |
| Идеально сбалансированное<br>и оно же АВЛ-<br>сбалансированное                                             | АВЛ-сбалансированное,<br>но не идеально<br>сбалансированное                               | Не АВЛ-сбалансированное<br>(нарушение – для корня)                                                                          |

Для реализации алгоритмов АВЛ-балансировки в каждой вершине дерева удобно хранить так называемый **коэффициент балансировки (КБ)** как разность высот правого и левого поддеревьев. Для АВЛ-деревьев у всех вершин значение КБ равно  $-1$ ,  $0$  или  $+1$ .

|                                                                                                                                              |                                                                                                                                                                                                     |
|----------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> graph TD     30["30 (-1)"] --&gt; 20["20 (0)"]     30 --&gt; 40["40 (0)"]     20 --&gt; 10["10 (0)"]     20 --&gt; 25["25 (0)"] </pre> | <pre> graph TD     30["30 (-2)"] --&gt; 20["20 (+1)"]     30 --&gt; 40["40 (0)"]     20 --&gt; 10["10 (0)"]     20 --&gt; 25["25 (0)"]     25 --&gt; 22["22 (0)"]     25 --&gt; 27["27 (0)"] </pre> |
| АВЛ-сбалансированное дерево                                                                                                                  | несбалансированное дерево                                                                                                                                                                           |

Поскольку коэффициент балансировки используется для каждой вершин, удобно ввести его в структуру данных, описывающих эту вершину:

**Type** Tp = ^TNode;

TNode = **record**

Inf : <описание>;

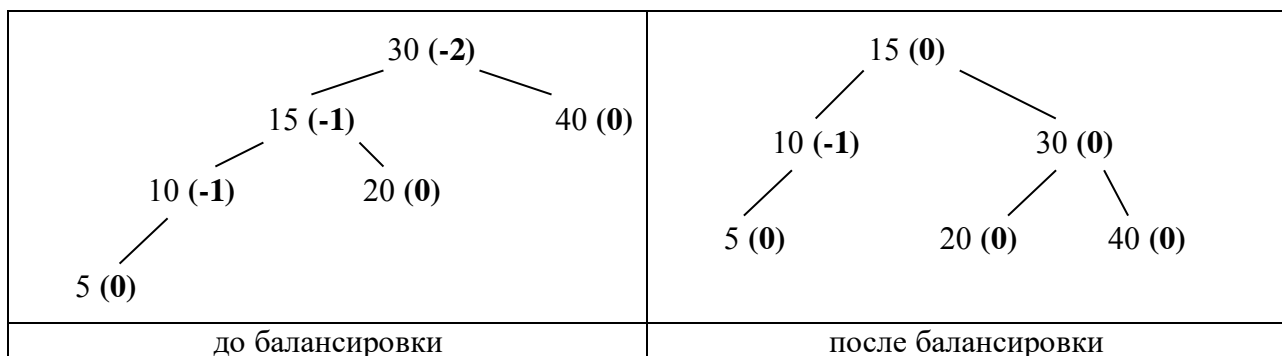
Left, right : Tp;

Balance : **shortInt**;  
**end**;

Алгоритм AVL-балансировки при добавлении вершины.

- выполняется поиск в дереве места для новой добавочной вершины, при этом для каждой вершины высчитывается коэффициент балансировки
- если необходимо, то выполняется добавление самой вершины с заполнением всех полей, в том числе и КБ = 0 (т.к. потомков у вновь созданной вершины пока нет)
- изменяется на 1 коэффициент балансировки у родителя добавленной вершины:  $KB = KB + 1$  если добавлен правый потомок, иначе  $KB = KB - 1$
- проверяем новое значение КБ у родительской вершины: если он имеет допустимое значение, то переходим еще на уровень выше для изменения и анализа КБ у деда новой вершины и т.д – до корневой вершины (иногда условие балансировки может нарушиться только для корневой вершины, поэтому приходится проверять все вершины на пути от вновь добавленной до корневой)
- если у какой либо вершины нарушается условие балансировки, запускается специальный **алгоритм перестановки вершин**, который восстанавливает AVL-балансировку дерева и в то же время сохраняет упорядоченную структуру дерева поиска

Алгоритм перестановки вершин основан на так называемых **поворотах** вершин. При этом возможны две ситуации: более простая требует однократного поворота, более сложная – двукратного. Например, пусть обнаружено следующее поддерево с нарушенной балансировкой для его корневой вершины:



Данная ситуация является более простой и определяется следующими условиями:

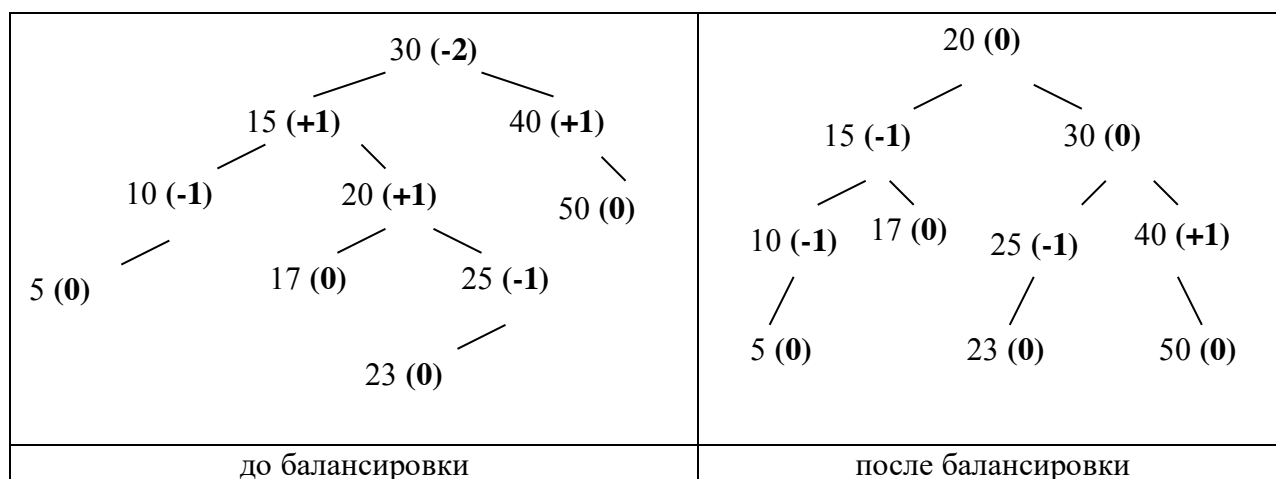
- вершина с нарушенным условием балансировки деформирована влево,  $KB(30) = -2$
- ее левый потомок также перевешивает в левую сторону,  $KB(15) = -1$

Для исправления этой ситуации выполняется **однократный поворот вправо**:

- вершина 15 заменяет вершину 30
- левое поддерево вершины 15 не изменяется ( $15 - 10 - 5$ )
- правое поддерево вершины 15 образуется корневой вершиной 30
- у вершины 30 правое поддерево не изменяется ( $30 - 40$ )
- левое поддерево вершины 30 образуется бывшим правым потомком вершины 15, т.е. 20

Аналогично выполняется однократный поворот **влево**, если вершина с нарушенным условием балансировки перевешивает вправо ( $KB = 2$ ), а ее правый потомок тоже перевешивает вправо ( $KB = 1$ ).

Более сложные случаи возникают при **несогласованном** перевешивании корневой вершины и ее потомков (коэффициенты балансировки имеют **разные** знаки). Например, рассмотрим случай, когда корневая вершина поддерева с нарушенным условием перевешивает **влево** ( $KB = -2$ ), но ее левый потомок перевешивает **вправо** ( $KB = 1$ ).





**Двукратный поворот** включает в себя:

- вершина 30 заменяется вершиной 20, т.е. правым потомком левого потомка
- левый потомок вершины 20 (вершина 17) становится правым потомком вершины 15
- левое поддерево с корнем 15 без изменений остается левым поддеревом вершины 20
- правое поддерево вершины 20 начинается с вершины 30
- правое поддерево вершины 30 не меняется (30 – 40 – 50)
- левое поддерево вершины 30 образуется правым поддеревом вершины 20 (25 – 23)

**Удаление** вершин из AVL-дерева выполняется аналогично:

- ищется удаляемая вершина
- если требуется – определяется вершина-заменитель и выполняется обмен
- после удаления вершины пересчитываются КБ и при необходимости производится балансировка точно по таким же правилам

В отличие от добавления, при удалении возможны ситуации, когда балансировка потребуется для всех вершин на пути от удаленной вершины до корня дерева.

Практика использования AVL-деревьев показала, что балансировка требуется примерно в половине случаев вставки и удаления вершин.

Общий вывод: учитывая достаточно высокую трудоемкость выполнения балансировки, AVL-деревья следует использовать тогда, когда вставка и удаление выполняются значительно реже, чем поиск.

## **7.2. Двоичные деревья с дополнительными указателями**

Классические двоичные деревья и, в частности, деревья поиска, в каждой вершине имеют по две ссылки на левого и правого потомка. В некоторых практических задачах удобно использовать деревья, вершины которых

содержат **большее** число ссылок. Одной из разновидностей подобных деревьев являются деревья с дополнительными указателями на **родительские** вершины.

В таком дереве каждая вершина имеет дополнительное поле – указатель на своего родителя. Описание соответствующей структуры данных:

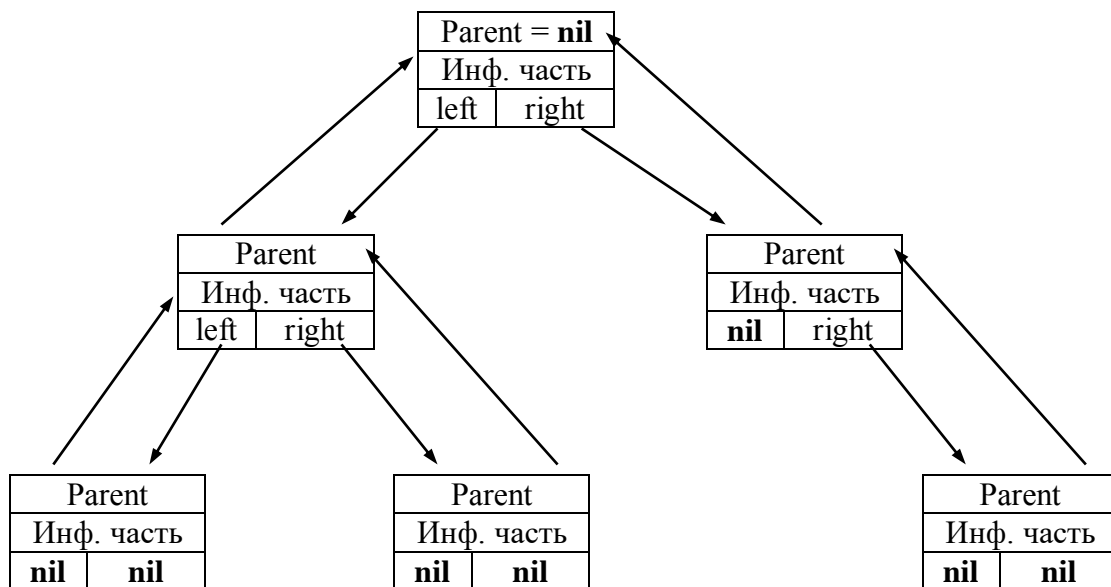
**Type** Tp = ^TNode;

TNode = **record**

Inf : <описание>;

Left, right, parent : Tp;

**end;**



Преимуществом такого дерева является простота реализации прохода по дереву как вниз, так и вверх, что напоминает двунаправленный список.

Недостатками являются:

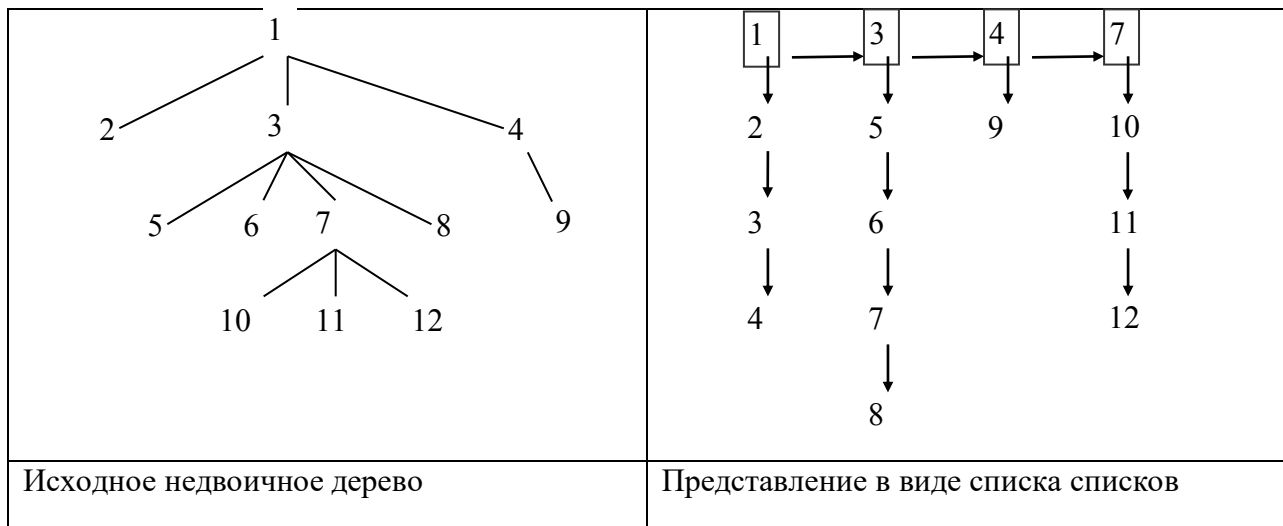
- Увеличение затрат памяти на поддержку в каждой вершине дополнительного указателя (4 байта на вершину), т.е. в каждой вершине 12 байт занимает служебная информация, что при большом числе вершин (сотни тысяч) может стать весьма ощутимым
- Увеличивается число операций при добавлении и удалении вершин за счет поддержки дополнительных ссылочных полей

Использование таких деревьев позволяет упростить процедуры обработки деревьев, прежде всего – обхода и удаления вершин. Отпадает необходимость явного использования рекурсии или ее замены стековыми операциями, поскольку возврат от текущей вершины к ее родителю можно выполнить непосредственно по дополнительному ссылочному полю.

### 7.3. Деревья общего вида (не двоичные).

Особенность таких деревьев – заранее **неизвестное** число потомков вершин, что не позволяет объявить в каждой вершине какое-то разумное количество ссылочных полей.

Подобные деревья используются реже двоичных, но тем не менее есть круг задач, где они необходимы. Существует несколько способов описания подобных деревьев, например – представление их через набор двоичных деревьев. Другой способ использует понятие сложной списковой структуры. Сначала создаётся **основной** линейный **список**, содержащий все **вершины-родители** (не терминальные). С каждой родительской вершиной дополнительно связывается **список ее потомков**.



Поскольку любая вершина исходного дерева может быть как в основном списке родителей, так и в списке потомков, структура элементов всех списков должна быть одинаковой. Каждая вершина в этой структуре должна иметь информационное поле и два ссылочных поля: указатель на следующий элемент в родительском списке и указатель на следующий элемент списка потомков. В

зависимости от присутствия вершины в том или ином списке некоторые поля могут быть пустыми.

**Type** Tp = ^ TNode;

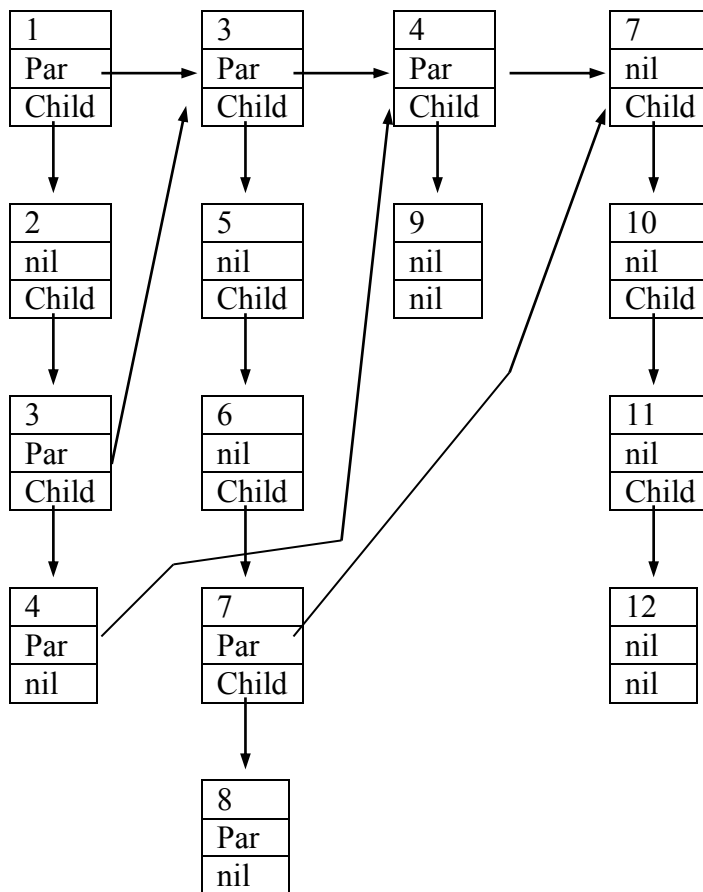
TNode = **record**

Inf : <описание>;

NextParent, NextChild : Tp;

**end;**

В списках потомков ссылочное поле NextParent в простейшем случае просто не используется (устанавливается в **nil**). Как вариант, можно в этом поле хранить указатель на одноименную вершину в родительском списке.



Схематично рассмотрим реализацию основных операций с подобным списковым представлением недвоичных деревьев.

#### 1. **Вывод** списка родителей с соответствующими списками потомков

реализуется двойным циклом: внешний цикл идет по списку родителей, а

внутренний позволяет для каждого родителя вывести список его потомков

$P_{CurrentParent} := pRoot;$

**While**  $p_{CurrentParent} \neq nil$  **do**

**begin**

“обработка вершины  $p_{CurrentParent}$ ”;

$p_{CurrentChild} := p_{CurrentParent}.NextChild;$

**while**  $p_{CurrentChild} \neq nil$  **do**

**begin**

“обработка вершины  $p_{CurrentChild}$ ”;

$p_{CurrentChild} := p_{CurrentChild}.NextChild;$

**end;**

$p_{CurrentParent} := p_{CurrentParent}.NextParent;$

**end;**

2. **Добавление** вершины  $X$  как потомка вершины  $Y$ :

- “поиск вершины  $Y$  обходом всех вершин”;
- **if** “вершина  $Y$  найдена в списке родителей” **then** “добавляем  $X$  в список потомков вершины  $Y$ ”;
- **if** “вершина  $Y$  найдена в списке потомков” **then**
  - ✓ “добавляем  $Y$  в список родителей”;
  - ✓ “создаем у вершины  $Y$  список потомков с вершиной  $X$ ”;

3. **Удаление** вершины  $X$ , если она не корневая для всего дерева:

- “поиск вершины  $X$  обходом всех вершин”;
- **if** “вершина  $X$  найдена в списке родителей” **then**
  - ✓ “просмотром всех списков найти родителя вершины  $X$ ”;
  - ✓ “удалить  $X$  из списка потомков”;
  - ✓ “к родителю  $X$  добавить список всех потомков  $X$ ”;
- **if** “вершина  $X$  есть только в списке потомков” **then**
  - ✓ “удалить  $X$  из списка потомков”;

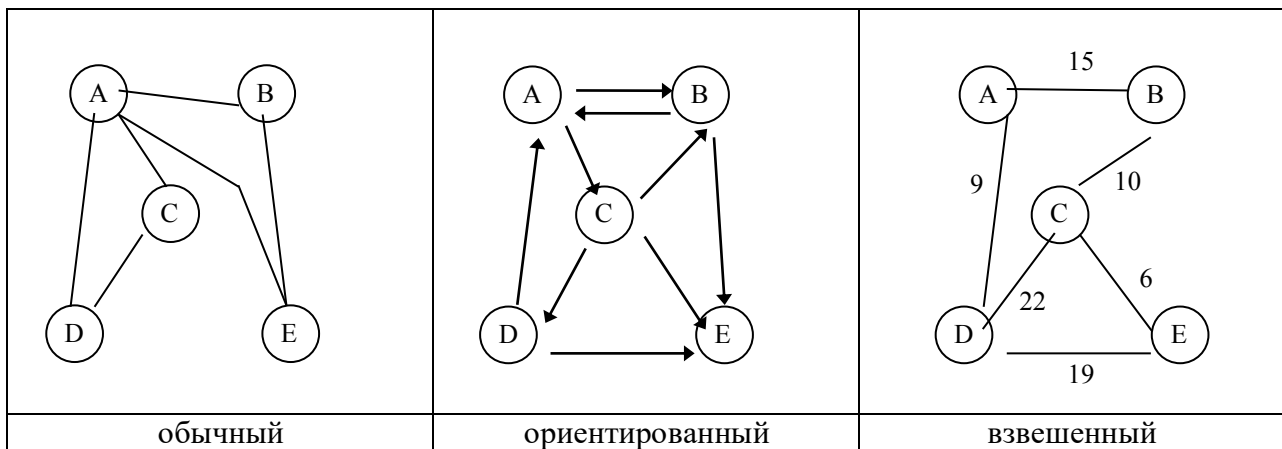
- ✓ **if** “список потомков пуст, то удалить родителя из списка родителей”;

#### 7.4. Представление графов

Граф – это множество однотипных **объектов** (вершин), некоторые из которых **связаны** друг с другом какими-либо связями (ребрами). Одна связь всегда соединяет только две вершины (иногда – вершину саму с собой). Основные разновидности графов:

- **неориентированные** (обычные), в которых важен только **сам факт** связи двух вершин
- **ориентированные** (орграфы), для которых важным является еще и **направление** связи вершин
- **взвешенные**, в которых важной информацией является еще и **степень** (величина, вес) связи вершин

Примеры графов разных типов:



Для описания графа как структуры данных используются два способа: **матрицы смежности** и **списки смежности**. Первый способ предполагает использование двумерного массива чисел, который для простых графов заполняется только значениями 0 (нет связи) и 1 (есть связь), а для взвешенного – значениями весов. Для обычного графа матрица смежности всегда является симметричной относительно главной диагонали, а для орграфа чаще всего эта матрица не симметрична, что отражает одностороннюю направленность связей. Для рассмотренных выше примеров матрицы смежности будут следующими:

|   | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 1 | 1 | 1 | 1 |
| B | 1 | 0 | 0 | 0 | 1 |
| C | 1 | 0 | 0 | 1 | 0 |
| D | 1 | 0 | 1 | 0 | 0 |
| E | 1 | 1 | 0 | 0 | 0 |

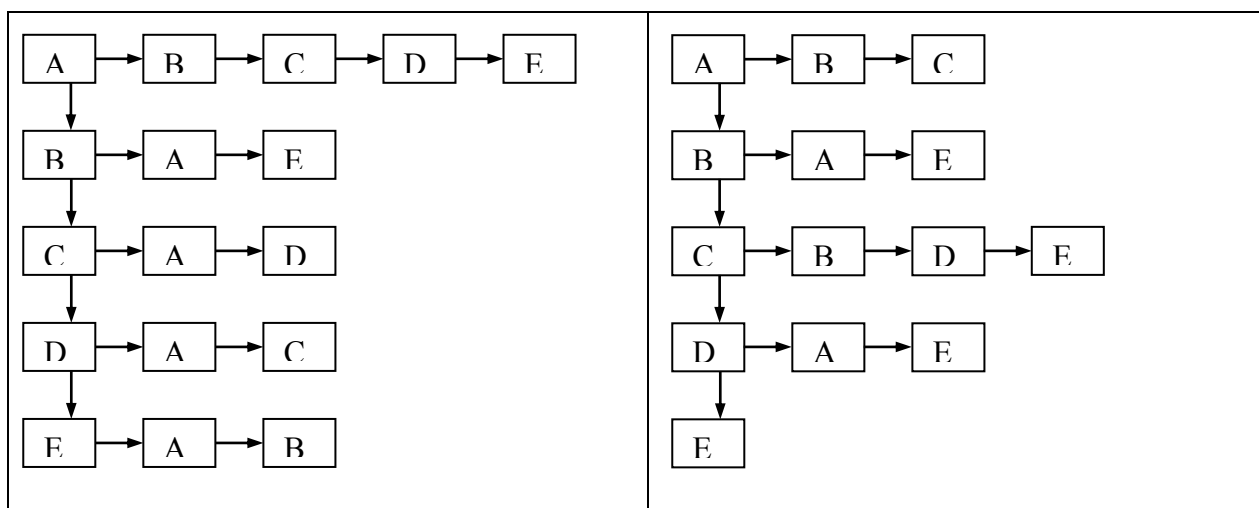
|   | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 1 | 1 | 0 | 0 |
| B | 1 | 0 | 0 | 0 | 1 |
| C | 0 | 1 | 0 | 1 | 1 |
| D | 1 | 0 | 0 | 0 | 1 |
| E | 0 | 0 | 0 | 0 | 0 |

|   | A  | B  | C  | D  | E  |
|---|----|----|----|----|----|
| A | 0  | 15 | 0  | 9  | 0  |
| B | 15 | 0  | 10 | 0  | 0  |
| C | 0  | 10 | 0  | 22 | 6  |
| D | 9  | 0  | 22 | 0  | 19 |
| E | 0  | 0  | 6  | 19 | 0  |

Недостатки данного способа:

- заранее надо знать хотя бы ориентировочное число вершин в графе
- для графов с большим числом вершин матрица становится слишком большой (например  $1000 \times 1000 = 1$  миллион чисел)
- при малом числе связующих ребер матрица заполнена в основном нулями

Этих недостатков во многом лишен второй способ, основанный на использовании списков смежных вершин. Здесь списки содержат ровно столько элементов, сколько ребер в графе, и кроме того вершины и ребра могут добавляться динамически. Список смежных вершин представляет собой **главный список всех вершин** и множество **вспомогательных списков**, содержащих перечень вершин, связанных с данной. Для рассмотренных выше обычного графа и ориентированного графа списки смежности будут следующими:



Описание подобной сложной списковой структуры выполняется обычным образом.

Операции добавления и удаления по сравнению с деревьями имеют следующие варианты:

- добавление новой связи (ребра) между заданной парой существующих вершин
- добавление новой вершины вместе со всеми необходимыми связями
- удаление связи (ребра) между двумя вершинами
- удаление вершины вместе со всеми ее связями

**Добавление нового ребра** включает в себя (на примере обычного графа):

- получение имен связываемых вершин
- поиск в основном списке первой связываемой вершины
- поиск в списке смежных ей вершин второй связываемой вершины и либо вывод сообщения об ошибке, либо добавление в этот список нового элемента с именем второй вершины
- поиск в основном списке второй связываемой вершины
- поиск в списке смежных ей вершин первой связываемой вершины и либо вывод сообщения об ошибке, либо добавление в этот список нового элемента с именем первой вершины

**Добавление новой вершины** включает в себя:

- запрос имени новой вершины вместе с именами всех связываемых с ней вершин
- поиск в основном списке имени новой вершины и в случае отсутствия ее -добавление в основной список
- формирование списка вершин, смежных вновь добавленной
- поиск в основном списке всех смежных вершин и добавление в их вспомогательные списки нового элемента с именем новой вершины

**Удаление ребра** производится следующим образом:

- запрос имен двух вершин, между которыми разрывается связь
- поиск в основном списке каждой из этих вершин
- поиск в каждом из двух вспомогательных списков имени соседней вершины и удаление соответствующего элемента

**Удаление вершины** производится следующим образом:



- запрос имени удаляемой вершины
- поиск ее в основном списке
- просмотр вспомогательного списка удаляемой вершины, для каждого элемента которого:
  - поиск смежной вершины в основном списке и удаление из ее вспомогательного списка элемента, соответствующего удаляемой вершине
  - удаление самого элемента из вспомогательного списка
- удаление вершины из основного списка

При обработке графов часто приходится выполнять **обход** всех его вершин.

**Правила обхода** графов похожи на обход деревьев. Существуют два основных правила обхода, известные как **поиск в глубину** и **поиск в ширину**.

Поиск в глубину использует две структуры – **стек** для запоминания еще не обработанных вершин и **список** для запоминания уже обработанных. Поиск выполняется следующим образом:

- задать стартовую вершину (аналог корневой вершины при обходе дерева)
- обработать стартовую вершину и включить ее во вспомогательный список обработанных вершин
- включить в стек все вершины, смежные со стартовой
- организовать цикл по условию опустошения стека и внутри цикла выполнить:
  - извлечь из стека очередную вершину
  - проверить по вспомогательному списку обработанность этой вершины
  - если вершина уже обработана, то извлечь из стека следующую вершину
  - если вершина еще не обработана, то обработать ее и поместить в список обработанных вершин
  - просмотреть весь список смежных с нею вершин и поместить в стек все еще не обработанные вершины

**Например**, для рассмотренного выше обычного графа получим:

- пусть стартовая вершина – В
- включаем В в список обработанных вершин: список = (В)
- помещаем в стек смежные с В вершины, т.е. А и Е: стек = (А, Е)
- извлекаем из стека вершину Е: стек = (А)
- т.к. Е нет в списке обработанных вершин, то обрабатываем ее и помещаем в список: список = (В, Е)
- смежные с Е вершины – это А и В, но В уже обработана, поэтому помещаем в стек только вершину А: стек = (А, А)
- извлекаем из стека вершину А: стек = (А)
- т.к. А нет в списке обработанных вершин, то помещаем ее туда: список = (В, Е, А)
- смежные с А вершины – это В, С, D, Е, из которых В и Е уже обработаны, поэтому помещаем в стек С и D: стек = (А, С, D)
- извлекаем из стека вершину D: стек = (А, С)
- т.к. D не обработана, то помещаем ее в список: список = (В, Е, А, D)
- смежные с D вершины – это А и С, из которых А уже обработана, поэтому помещаем в стек вершину С: стек = (А, С, С)
- извлекаем из стека вершину С: стек = (А, С)
- т.к. С не обработана, помещаем ее в список: список = (В, Е, А, D, С)
- смежные с С вершины – это А и D, но они обе уже обработаны, поэтому в стек ничего не заносим
- извлекаем из стека С, но она уже обработана
- извлекаем из стека А, но она тоже уже обработана
- т.к. стек стал пустым, то завершаем обход с результатом (В, Е, А, D, С)

**Поиск в ширину** работает немного по другому: сначала обрабатываются все вершины, **смежные** с текущей, а лишь потом – их **потомки**. Вместо стека для запоминания еще не обработанных вершин используется **очередь**. Последовательность действий:

- задать стартовую вершину (аналог корневой вершины при обходе дерева)
- обработать стартовую вершину и включить ее во вспомогательный список обработанных вершин
- включить в очередь все вершины, смежные со стартовой
- организовать цикл по условию опустошения очереди и внутри цикла выполнить:
  - извлечь из очереди очередную вершину
  - проверить по вспомогательному списку обработанность этой вершины
  - если вершина уже обработана, то извлечь из очереди следующую вершину
  - если вершина еще не обработана, то обработать ее и поместить в список обработанных вершин
  - просмотреть весь список смежных с нею вершин и поместить в очередь все еще не обработанные вершины

Тот же что и раньше **пример** даст следующий результат:

- пусть стартовая вершина – В
- включаем В в список обработанных вершин: список = (В)
- помещаем в очередь смежные с В вершины, т.е. А и Е: очередь = (А, Е)
- извлекаем из очереди вершину А: очередь = (Е)
- т.к. она не обработана, добавляем ее в список: список = (В, А)
- смежные с А вершины – это В, С, D и Е, помещаем в очередь вершины С, D и Е: очередь = (Е, С, D, Е)
- извлекаем из очереди вершину Е: очередь = (С, D, Е)
- т.к. Е не обработана, помещаем ее в список: список = (В, А, Е), т.е. в первую очередь обработаны обе смежные с В вершины
- смежные с Е вершины – это А и В, но обе они уже обработаны, поэтому очередь новыми вершинами не пополняется
- извлекаем из очереди вершину С: очередь = (D, Е)

- т.к. С не обработана, то помещаем ее в список: список = (В, А, Е, С)
- смежные с С вершины – это А и D, помещаем в очередь только D:  
очередь = (D, Е, D)
- извлекаем из очереди вершину D: очередь = (Е, D)
- т.к. D не обработана, помещаем ее в список: список = (В, А, Е, С, D)
- смежные с D вершины – это А и С, но обе они обработаны, поэтому очередь не пополняется
- извлекаем из очереди вершину Е, но она уже обработана: очередь = (D)
- извлекаем из очереди вершину D, но она уже обработана и т.к. очередь становится пустой, то поиск заканчивается с результатом (В, А, Е, С, D), что отличается от поиска в глубину.

В заключение отметим несколько наиболее часто встречающихся задач на графах:

- найти путь наименьшей (наибольшей) длины между двумя заданными вершинами
- выделить из графа дерево, имеющее наименьший суммарный вес ребер (кратчайшее покрывающее дерево)
- присвоить каждой вершине графа цвет таким образом, чтобы не было ни одной пары смежных вершин, имеющих одинаковый цвет, и при этом число используемых цветов было бы минимальным (задача раскраски графа)
- найти в графе такой путь, который проходит по всем вершинам ровно по 1 разу и имеет при этом наименьшую суммарную длину (задача бродячего торговца или коммивояжера).

## 7.5. Практические задания

**Задание 1.** Реализовать основные операции с недвоичными деревьями, представленными с помощью списка родителей и потомков. Будем считать, что начальное дерево содержит единственную корневую вершину. Необходимо реализовать следующие операции:

- добавление новой вершины как потомка заданной вершины
- удаление заданной вершины
- вывод всех вершин с указанием родительских связей
- поиск заданной вершины

Требования к подпрограммам и главной программе – стандартные.

**Задание 2.** Реализовать основные операции с простыми графами с использованием матричного и спискового представлений:

- формирование матрицы смежности
- преобразование матрицы смежности в список смежности
- формирование списка смежности
- преобразование списка смежности в матрицу смежности
- добавление нового ребра
- удаление заданного ребра
- добавление новой вершины
- удаление заданной вершины
- обход графа в глубину
- обход графа в ширину

Требования к подпрограммам и главной программе – стандартные.

### **7.6. Контрольные вопросы по теме**

1. Какие проблемы возникают при использовании деревьев поиска?
2. Как влияет на структуру дерева поиска разный порядок поступления одних и тех же входных ключей?
3. Почему при построении дерева поиска важно управлять его структурой?
4. Какие деревья называются AVL-сбалансированными?
5. Как связаны понятия “идеально сбалансированное дерево” и “AVL-сбалансированное дерево”?
6. Приведите примеры идеально сбалансированного, AVL-сбалансированного и не-AVL-сбалансированного деревьев.

7. Какой параметр используется для реализации AVL-сбалансированных деревьев?
8. По какому алгоритму выполняется AVL-балансировка дерева при добавлении вершины?
9. Какие ситуации возможны при необходимости балансировки некоторого поддерева?
10. Как выполняется однократный поворот?
11. Как выполняется двукратный поворот?
12. Как выполняется балансировка дерева при удалении вершины?
13. Как описывается структура вершины дерева с дополнительными указателями на родителей?
14. Какие преимущества и недостатки имеют деревья с дополнительными указателями на родителей?
15. Для чего можно использовать деревья с дополнительными указателями на родителей?
16. В чем состоит основная сложность реализации не-двоичных деревьев?
17. Какие списковые структуры можно использовать для реализации не-двоичных деревьев?
18. Какую структуру должны иметь вершины не-двоичных деревьев при реализации их с помощью списков?
19. Как реализуется вывод вершин не-двоичного дерева, представленного с помощью списков?
20. Как реализуется добавление вершины в не-двоичное дерево, представленное с помощью списков?
21. Как реализуется удаление вершины из не-двоичного дерева, представленного с помощью списков?
22. Какие существуют разновидности графов?
23. Какие способы можно использовать для представления графов как структур данных?
24. Что такое матрица смежности и как она описывается?

25. Какие структуры данных необходимы для реализации списков смежности?
26. Какие основные задачи возникают при использовании графов?
27. Как реализуются операции добавления и удаления ребер в графе?
28. Как реализуются операции добавления и удаления вершин в графе?
29. Какие шаги включает в себя алгоритм поиска в глубину?
30. Какие шаги включает в себя алгоритм поиска в ширину?

## **Раздел 2. Алгоритмы сортировки и поиска**

### **Тема 1. Классификация методов. Простейшие методы сортировки**

#### **1.1. Задача оценки и выбора алгоритмов**

Довольно часто при разработке программного обеспечения возникает ситуация, когда одну и ту же задачу можно решить с помощью нескольких разных алгоритмов. Например, существует несколько алгоритмов сортировки массивов, из которых надо выбрать для реализации только один. Для этого надо уметь анализировать алгоритмы и сравнивать их между собой по тем или иным критериям. Наиболее часто используются следующие критерии:

- точность решения задачи
- затраты машинного времени
- затраты оперативной памяти

Как правило, перечисленные критерии противоречат друг другу, поэтому в каждой практической ситуации приходится выбирать наиболее важный критерий. В настоящее время, в связи с резким ростом объемов доступной оперативной памяти в большинстве случаев на первое место выходит временной критерий. Поскольку задача выбора алгоритмов возникает на этапе разработки программы, надо уметь **оценивать** алгоритмы по их **трудоемкости** еще до написания соответствующих программ. Для этого можно выделить в алгоритме наиболее часто повторяющиеся элементарные базовые операции и оценить зависимость числа их повторений от размерности обрабатываемых данных. Интуитивно понятно, что чем больше объем обрабатываемых данных, тем больше времени занимает эта обработка. Это связано с тем, что

большинство алгоритмов в цикле повторяют одни и те же действия, и число повторов определяется объемом исходных данных. При этом часто разумно оценивать три величины:

- минимально возможное число повторов элементарных операций в наилучшем случае
- среднее число повторов
- максимальное число повторов для наихудшего случая

Например, в задаче поиска в неупорядоченном массиве или списке из  $n$  элементов базовой операцией является сравнение заданного значения с ключом элемента массива. Тогда минимальное число сравнений равно 1 (совпадение произошло на первом элементе массива), максимальное равно  $n$  (просмотрен весь массив), а среднее - около  $n/2$ .

В отличие от этого, двоичный поиск в упорядоченном массиве или в “хорошем” двоичном дереве поиска дает другие оценки: минимум – 1, максимум -  $\log_2 n$ , среднее –  $0,5 * (\log_2 n)$ .

Получение подобных оценок часто является сложной математической задачей. Наиболее полный анализ методов получения этих оценок для всех основных алгоритмов приводится в фундаментальной многотомной работе Дональда Кнута [1, 2].

В идеале каждый алгоритм должен оцениваться по всем 3 параметрам: наилучшему, наихудшему и среднему числу базовых операций. К сожалению, получение средних оценок часто бывает невозможно, и поэтому на практике больше всего используются оценки для наихудшего случая. При этом для сравнения алгоритмов важен не **точный** вид функциональной зависимости трудоемкости от объема входных данных, а скорее **характер** или поведение этой зависимости. Эта информация помогает ответить на следующий важнейший вопрос: **если рост трудоемкости с увеличением объема данных неизбежен, то НАСКОЛЬКО БЫСТРО происходит этот рост?**

Для оценивания трудоемкости алгоритмов была введена специальная система обозначений – так называемая **О-нотация**. Эта нотация позволяет



учитывать в функции  $f(n)$  лишь **наиболее значимые** элементы, отбрасывая второстепенные.

Например, в функции  $f(n) = 2n^2 + n - 5$  при достаточно больших  $n$  компонента  $n^2$  будет значительно превосходить остальные слагаемые, и поэтому характерное поведение этой функции определяется именно этой компонентой. Остальные компоненты можно отбросить и условно записать, что данная функция имеет оценку поведения (в смысле скорости роста ее значений) вида  $O(n^2)$ .

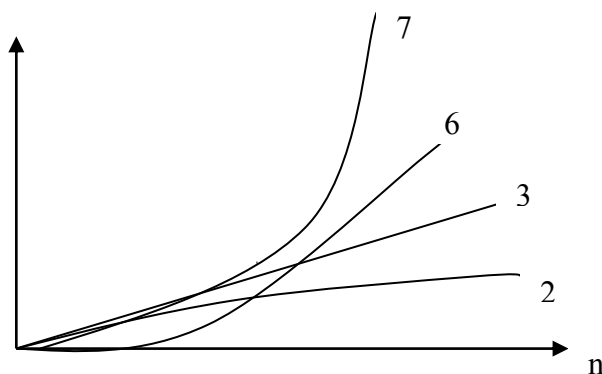
Аналогично, для функции  $f(n) = 2^n + 3n^3 - 10$  начиная с некоторого  $n$  первое слагаемое будет превосходить второе и поэтому данную функцию можно описать оценкой  $O(2^n)$ .

Важность  $O$ -оценивания состоит в том, что оно позволяет описывать характер поведения функции  $f(n)$  с ростом  $n$ : **насколько быстро** или медленно **растет** эта функция.

$O$ -оценка позволяет разбить все основные функции на ряд групп в зависимости от скорости их роста:

1. **постоянные** функции типа  $O(1)$ , которые с ростом  $n$  НЕ растут (в оценивании алгоритмов этот случай встречается крайне редко, но все-таки встречается!)
2. функции с **логарифмической** скоростью роста  $O(\log_2 n)$
3. функции с **линейной** скоростью роста  $O(n)$
4. функции с **линейно–логарифмической** скоростью роста  $O(n \cdot \log_2 n)$
5. функции с **квадратичной** скоростью роста  $O(n^2)$
6. функции со **степенной** скоростью роста  $O(n^a)$  при  $a > 2$
7. функции с **показательной** или **экспоненциальной** скоростью роста  $O(2^n)$
8. функции с **факториальной** степенью роста  $O(n!)$

В этом списке функции упорядочены именно по критерию скорости роста: сначала идут медленно растущие функции, потом – все более быстро растущие. Графики некоторых функций приведены на рисунке.



Отсюда можно сделать несколько **выводов**.

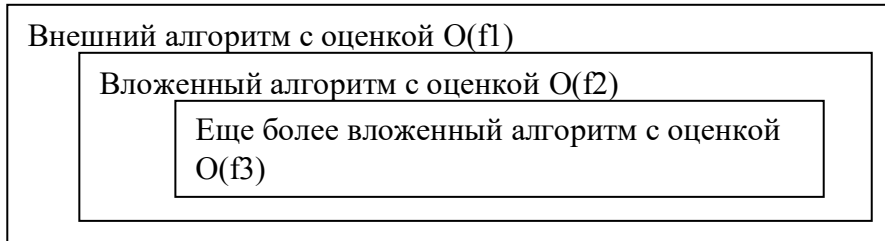
1. При выборе однотипных алгоритмов предпочтение (при прочих равных условиях) следует отдавать алгоритмам с **наименьшей** скоростью роста трудоемкости, поскольку они позволят за одно и то же время решить задачи с **большей размерностью**
2. Если заранее известно, что размерность решаемых задач невелика, но зато число их повторений очень большое, имеет смысл рассмотреть возможность использования алгоритмов не с самой лучшей оценкой, поскольку при малых  $n$  “лучшие” алгоритмы могут вести себя **хуже**, чем “плохие” (это можно заметить по графику в области начала координат)
3. Алгоритмы класса  $O(2^n)$  и  $O(n!)$  следует использовать с **большой осторожностью**, учитывая **катастрофический** рост их трудоемкости уже при  $n > 100$ . Например, если число базовых операций определяется соотношением  $2^n$ , то при  $n=100$  это число будет примерно равно  $10^{30}$ , и если одна базовая операция выполняется за 1 микросекунду, то это потребует около  $10^{24}$  секунд, т.е. порядка  $10^{16}$  лет. К сожалению, задачи с подобной трудоемкостью довольно часто встречаются на практике и их точное решение пока невозможно даже на сверхбыстрых суперкомпьютерах!

Как быть с оценкой трудоемкости программы **в целом**, если в программе используется **несколько** алгоритмов, решающих свои конкретные задачи? Есть два основных способа взаимодействия алгоритмов – **последовательное** и **вложенное**. При последовательном выполнении алгоритмов с оценками  $O(f_1)$ ,

$O(f_2), \dots, O(f_k)$  общая трудоемкость определяется трудоемкостью алгоритма с **максимальным** значением:

$$O(\text{программы}) = \text{Max}(O(f_1), O(f_2), \dots, O(f_k))$$

При вложенном выполнении общая трудоемкость есть **произведение** оценок вложенных друг в друга алгоритмов:  $O(\text{программы}) = O(f_1) * O(f_2) * O(f_3)$



## 1.2. Классификация задач сортировки и поиска

Пусть задано некоторое множество элементов  $a_1, a_2, a_3, \dots, a_n$  и требуется выстроить эти элементы по порядку в соответствии с заданной функцией предпочтения (например, по алфавиту). Очень часто значения функции предпочтения явно хранятся в исходных элементах в виде специального ключевого поля целого или строкового типа.

Все задачи сортировки делятся на две большие и принципиально различные группы: задачи **внутренней** и **внешней** сортировки. Внутренняя сортировка применима тогда, когда все входные данные **можно одновременно разместить в оперативной памяти**. Возможность такой загрузки определяется 3 факторами: располагаемым размером памяти, числом обрабатываемых элементов, объемом каждого элемента в байтах. Внутренняя сортировка, как правило, реализуется с помощью массивов и поэтому часто называется сортировкой массивов.

Если исходные данные **нельзя** одновременно разместить в основной памяти, то приходится использовать **дискковую память** и алгоритмы обработки файлов. Такая сортировка называется внешней или сортировкой файлов и будет рассмотрена позже. Пока остановимся на задаче сортировки массивов.

Все алгоритмы сортировки основаны на многократном повторении двух базовых операций: **сравнение ключей** у двух элементов и **перестановка** двух элементов. Подсчет именно этих операций лежит в основе методов оценивания трудоемкости алгоритмов сортировки.

Методы сортировки массивов можно разделить на две большие группы:

- **Универсальные** методы, не требующие никакой дополнительной информации об исходных данных и выполняющие сортировку “на месте”, т.е. без использования больших объемов дополнительной памяти (например – для размещения копии исходного массива); такие методы в лучшем случае дают оценку трудоемкости порядка  $O(n \cdot \log_2 n)$
- **Специальные** методы, которые либо за счет некоторой дополнительной информации об исходных данных, либо за счет использования большой дополнительной памяти позволяют получить более высокую производительность сортировки порядка  $O(n)$  (примеры – карманная и поразрядная сортировка)

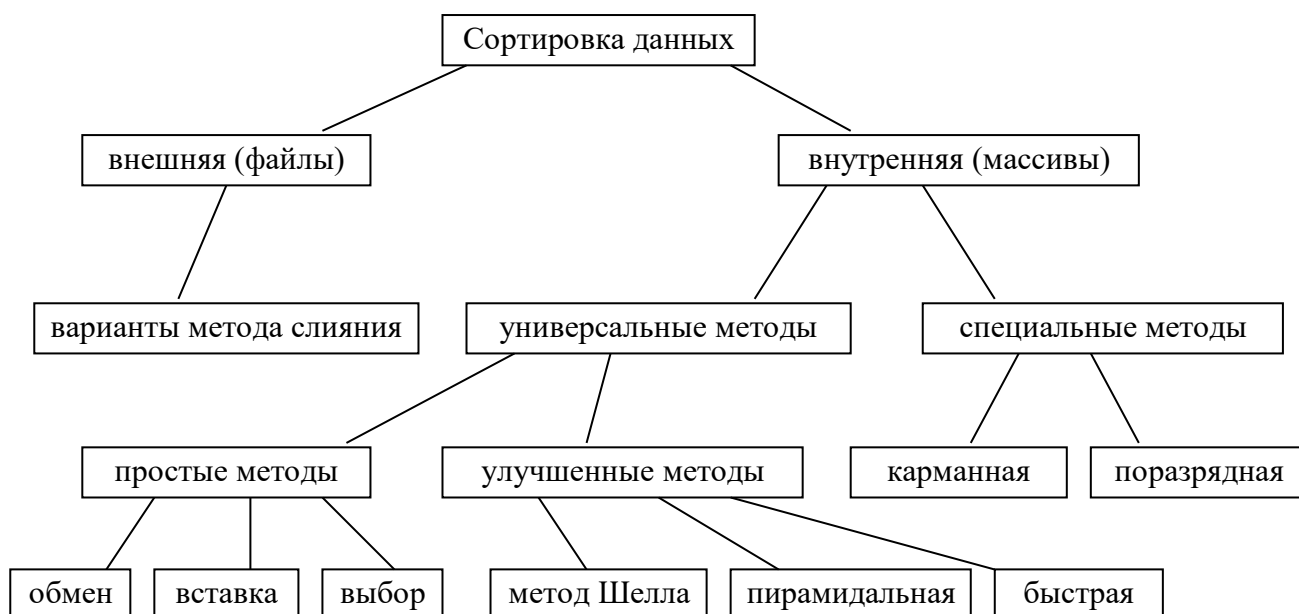
В свою очередь, универсальные методы сортировки делятся на две подгруппы:

- **Простейшие** методы с трудоемкостью порядка  $O(n^2)$ : сортировка обменом, сортировка выбором и сортировка вставками
- **Улучшенные** методы с трудоемкостью  $O(n \cdot \log_2 n)$ : метод Шелла, пирамидальная сортировка, быстрая сортировка

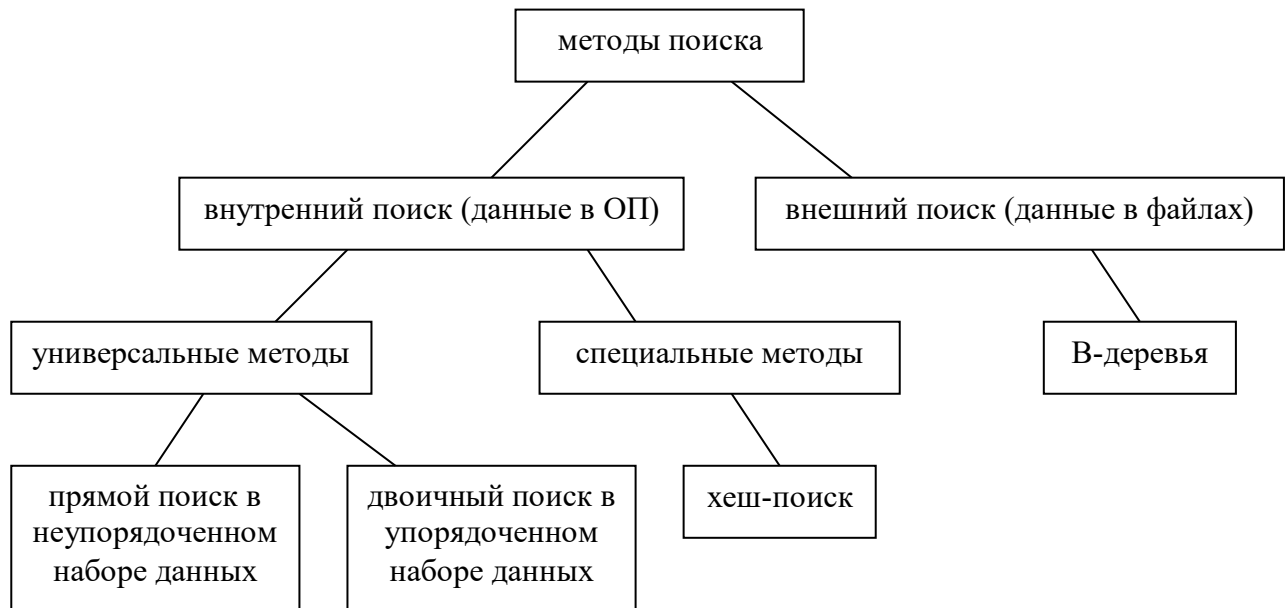
Возникает справедливый вопрос: зачем нужны простейшие методы, если есть более быстрые улучшенные методы? Как ни странно, возможны ситуации, когда простейшие методы оказываются **лучше** улучшенных. Подобные ситуации уже были упомянуты выше: если надо очень много (сотни тысяч или миллионы) раз повторить сортировку весьма небольших массивов (несколько десятков элементов), то использование простейших методов может дать некоторый выигрыш, поскольку компонента  $n^2$  оценочной функции при малых  $n$  не оказывает решающего влияния на общий результат. Кроме того, простейшие методы сортировки имеют исключительно простую и понятную

программную реализацию, что далеко не всегда можно сказать об улучшенных методах.

Общая классификация методов сортировки приводится на схеме. Описание всех методов приводится далее в пособии.



Аналогично можно провести классификацию методов поиска. Прежде всего – **внутренний** поиск и **внешний** поиск. Внутренний поиск – **универсальный** и **специальный**. Универсальный – поиск в **упорядоченном** или **неупорядоченном** наборе данных. Неупорядоченный набор – метод **простого перебора** (массив, список, иногда – дерево), упорядоченный – метод **двоичного поиска** в массиве или дереве. Специальный метод поиска в массиве (**хеш-поиск**) предполагает особую его организацию и используется при определенных ограничениях (правда, при выполнении этих ограничений данный метод дает потрясающую производительность – **одно сравнение** для поиска ЛЮБОГО элемента, независимо от объема входных данных!). Внешний поиск реализуется для **сверхбольших** объемов данных, которые нельзя одновременно разместить в основной памяти и требует использования внешних файлов (один из самых известных методов основан на использовании так называемых **В-деревьев**). Универсальные методы поиска были рассмотрены ранее, а остальные рассматриваются ниже, после описания методов сортировки.



### 1.3. Простейшие методы сортировки: метод обмена

Данный метод относится к классу простейших, занимая в нем последнее место по производительности. Тем не менее, он очень широко известен, видимо, благодаря своему одному легко запоминающемуся названию – метод **всплывающего пузырька** (или тонущего шарика, если кому-то так больше нравится). Работа алгоритма действительно похожа на всплывание наверх пузырьков воздуха: сначала на самый верх всплывает самый легкий элемент, потом за ним – чуть более тяжелый и т.д.

Пусть имеется  $n$  элементов  $a_1, a_2, a_3, \dots, a_n$ , расположенных в ячейках массива. Для простоты будем считать, что сам элемент совпадает с его ключом. Алгоритм состоит в повторении  $n-1$  шага, на каждом из которых в оставшемся необработанном наборе за счет попарного сравнения соседних элементов отыскивается минимальный элемент.

Шаг 1. Сравниваем  $a_n$  с  $a_{n-1}$  и если  $a_n < a_{n-1}$  то меняем их местами, потом сравниваем  $a_{n-1}$  с  $a_{n-2}$  и, возможно, переставляем их, сравниваем  $a_{n-2}$  и  $a_{n-3}$  и т.д. до сравнения и, возможно, перестановки  $a_2$  и  $a_1$ . В результате на первом месте в массиве оказывается самый минимальный элемент, который в дальнейшей сортировке не участвует

Шаг 2. Аналогично сравниваем  $a_n$  с  $a_{n-1}$ ,  $a_{n-1}$  с  $a_{n-2}$  и т.д.,  $a_3$  с  $a_2$ , в результате чего на месте  $a_2$  оказывается второй наименьший элемент, который вместе с  $a_1$  образует начальную часть упорядоченного массива

Шаг 3. Аналогичными сравнениями и перестановками среди элементов  $a_3, a_4, \dots, a_n$  находится наименьший, который занимает место  $a_3$

.....

Шаг  $n-1$ . К этому моменту первые  $n-2$  элемента в массиве уже упорядочены и остается “навести порядок” только между двумя последними элементами  $a_{n-1}$  и  $a_n$ . На этом сортировка заканчивается.

**Пример.** Дано 6 элементов – целые числа 15, 33, 42, 07, 12, 19.

|       | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | Выполняемые операции                                |
|-------|-------|-------|-------|-------|-------|-------|-----------------------------------------------------|
| шаг 1 | 15    | 33    | 42    | 07    | 12    | 19    | сравнение 19 и 12, обмена нет                       |
|       | 15    | 33    | 42    | 07    | 12    | 19    | сравнение 12 и 07, обмена нет                       |
|       | 15    | 33    | 07    | 42    | 12    | 19    | сравнение 07 и 42, меняем их                        |
|       | 15    | 07    | 33    | 42    | 12    | 19    | сравнение 07 и 33, меняем их                        |
|       | 07    | 15    | 33    | 42    | 12    | 19    | сравнение 07 и 15, меняем их; 07 - наименьший       |
| шаг 2 | 07    | 15    | 33    | 42    | 12    | 19    | сравнение 19 и 12, обмена нет                       |
|       | 07    | 15    | 33    | 12    | 42    | 19    | сравнение 12 и 42, меняем их                        |
|       | 07    | 15    | 12    | 33    | 42    | 19    | сравнение 12 и 33, меняем их                        |
|       | 07    | 12    | 15    | 33    | 42    | 19    | сравнение 12 и 15, меняем их, 12 – второй наим.     |
| шаг 3 | 07    | 12    | 15    | 33    | 19    | 42    | сравнение 19 и 42, меняем их                        |
|       | 07    | 12    | 15    | 19    | 33    | 42    | сравнение 19 и 33, меняем их                        |
|       | 07    | 12    | 15    | 19    | 33    | 42    | сравнение 19 и 15, обмена нет, 15 – третий наим.    |
| шаг 4 | 07    | 12    | 15    | 19    | 33    | 42    | сравнение 42 и 33, обмена нет                       |
|       | 07    | 12    | 15    | 19    | 33    | 42    | сравнение 33 и 19, обмена нет, 19 – четвертый элем. |
| шаг 5 | 07    | 12    | 15    | 19    | 33    | 42    | сравнение 42 и 33, обмена нет, сортировка закончена |
|       | 07    | 12    | 15    | 19    | 33    | 42    |                                                     |

Итого, для шести элементов сделано  $5+4+3+2+1 = 15$  сравнений и 8 перестановок.

В общем случае, на каждом из  $n-1$  шагов выполняется в среднем  $n/2$  сравнений, поэтому оценка для числа сравнений выражается соотношением  $n(n-1)/2$ , т.е. данный метод относится к классу  $O(n^2)$ . Аналогично, число перестановок тоже пропорционально  $n^2$ . Несмотря на то, что было предложено несколько улучшений данного метода (есть очень красивые названия – например, шейкер-сортировка), он остается самым неэффективным. Уже для

1000 элементов число сравнений выражается внушительной величиной порядка 500 тысяч.

Программная реализация включает двойной цикл: внешний реализует основные шаги алгоритма, внутренний сравнивает и переставляет элементы, начиная с конца массива.

```
for i := 2 to n do
 begin
 for j := n downto i do
 if a[j-1] > a[j] then
 begin temp := a[j-1]; a[j-1] := a[j]; a[j] := temp; end;
 end;
```

#### 1.4. Простейшие методы сортировки: метод вставок

Данный метод также относится к классу простейших, но по сравнению с методом пузырька имеет немного лучшие показатели.

Пусть имеется  $n$  элементов  $a_1, a_2, a_3, \dots, a_n$ , расположенных в ячейках массива. Сортировка выполняется за  $(n-1)$  шаг, причем шаги удобно нумеровать от 2 до  $n$ . На каждом  $i$ -ом шаге обрабатываемый набор разбивается на 2 части:

- левую часть образуют уже упорядоченные на предыдущих шагах элементы  $a_1, a_2, a_3, \dots, a_{i-1}$
- правую часть образуют еще не обработанные элементы  $a_i, a_{i+1}, a_{i+2}, \dots, a_n$

На шаге  $i$  для элемента  $a_i$  находится подходящее место в уже отсортированной последовательности. Поиск подходящего места выполняется поэлементными сравнениями и перестановками по необходимости: сравниваем  $a_i$  с  $a_{i-1}$ , если  $a_i < a_{i-1}$ , то переставляем их, потом сравниваем  $a_{i-1}$  с  $a_{i-2}$  и т. д. Сравнения и, возможно, перестановки продолжаются до тех пор, пока не будет выполнено одно из 2-х следующих условий:

- в отсортированном наборе найден элемент, меньший  $a_i$  (все остальные не просмотренные элементы будут еще меньше)



- достигнут первый элемент набора  $a_1$ , что произойдет в том случае, если  $a_i$  меньше всех элементов в отсортированном наборе и он должен занять первое место в массиве

**Пример.** Возьмем тот же исходный набор целых чисел: 15-33-42-07-12-19

|       | $a_1$     | $a_2$     | $a_3$     | $a_4$     | $a_5$     | $a_6$ | Выполняемые операции                               |
|-------|-----------|-----------|-----------|-----------|-----------|-------|----------------------------------------------------|
| шаг 2 | <b>15</b> | 33        | 42        | 07        | 12        | 19    | сравнение 15 и 33, обмена нет, 15 – пока первый    |
| шаг 3 | <b>15</b> | <b>33</b> | 42        | 07        | 12        | 19    | сравнение 33 и 42, обмена нет, 15 и 33 пока первые |
| шаг 4 | <b>15</b> | <b>33</b> | <b>07</b> | 42        | 12        | 19    | сравнение 07 и 42, меняем их                       |
|       | <b>15</b> | <b>07</b> | <b>33</b> | 42        | 12        | 19    | сравнение 07 и 33, меняем их                       |
|       | <b>07</b> | <b>15</b> | <b>33</b> | 42        | 12        | 19    | сравнение 07 и 15, меняем их; 07-15-33 пока первые |
| шаг 5 | <b>07</b> | <b>15</b> | <b>33</b> | <b>12</b> | 42        | 19    | сравнение 12 и 42, меняем их                       |
|       | <b>07</b> | <b>15</b> | <b>12</b> | <b>33</b> | 42        | 19    | сравнение 12 и 33, меняем их                       |
|       | <b>07</b> | <b>12</b> | <b>15</b> | <b>33</b> | 42        | 19    | сравнение 12 и 15, меняем их                       |
|       | <b>07</b> | <b>12</b> | <b>15</b> | <b>33</b> | 42        | 19    | сравнение 12 и 07, обмена нет, пока: 07-12-15-33   |
| шаг 6 | <b>07</b> | <b>12</b> | <b>15</b> | <b>33</b> | <b>19</b> | 42    | сравнение 19 и 42, меняем их                       |
|       | <b>07</b> | <b>12</b> | <b>15</b> | <b>19</b> | <b>33</b> | 42    | сравнение 19 и 33, меняем их                       |
|       | <b>07</b> | <b>12</b> | <b>15</b> | <b>19</b> | <b>33</b> | 42    | сравнение 19 и 15, обмена нет, все готово          |

Для данного примера было сделано 12 сравнений и 8 перестановок, что чуть лучше предыдущего метода. В среднем, число сравнений по данному методу примерно в 2 раза меньше, чем в методе пузырька, оставаясь тем не менее пропорциональным величине  $n^2$ . Наилучший результат этот метод показывает для уже упорядоченного исходного массива – всего  $(n-1)$  сравнение.

Программная реализация включает два вложенных цикла, но в отличие от предыдущего метода, внутренний цикл реализуется как **while-do** с возможностью остановки при обнаружении меньшего элемента.

**for**  $i := 2$  **to**  $n$  **do**

**begin**

temp :=  $a[i]$ ;  $j := i - 1$ ;

**While**  $(j > 0)$  **and**  $(temp < a[j])$  **do**

**begin**  $a[j+1] := a[j]$ ;  $j := j - 1$ ; **end**;

$a[j+1] := temp$ ;

**end**;

### 1.5. Простейшие методы сортировки: метод выбора

Данный метод из группы простейших имеет лучшие характеристики по числу перестановок, хотя он, как и оба ранее рассмотренных метода, в целом имеет трудоемкость  $O(n^2)$ . Его чуть более лучшие показатели связаны с тем, что в некоторых ситуациях выполняется перестановка не соседних элементов, а отстоящих на некотором расстоянии друг от друга.

Пусть имеется  $n$  элементов  $a_1, a_2, a_3, \dots, a_n$ , расположенных в ячейках массива. Сортировка выполняется за  $(n-1)$  шаг, пронумерованных от 1 до  $n-1$ . На каждом  $i$ -ом шаге обрабатываемый набор разбивается на 2 части:

- левую часть образуют уже упорядоченные на предыдущих шагах элементы  $a_1, a_2, a_3, \dots, a_{i-1}$
- правую часть образуют еще не обработанные элементы  $a_i, a_{i+1}, a_{i+2}, \dots, a_n$

Суть метода состоит в том, что в необработанном наборе отыскивается наименьший элемент, который меняется местами с элементом  $a_i$ . На первом шаге (при  $i = 1$ ), когда необработанным является весь исходный набор, это сводится к поиску наименьшего элемента в массиве и обмену его с первым элементом. Ясно, что поиск наименьшего элемента выполняется обычным попарным сравнением, но соседние элементы при этом не переставляются, что в целом уменьшает число пересылок.

**Пример.** Возьмем тот же исходный набор целых чисел: 15-33-42-07-12-19

|       | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | Выполняемые операции                                    |
|-------|-------|-------|-------|-------|-------|-------|---------------------------------------------------------|
| шаг 1 | 15    | 33    | 42    | 07    | 12    | 19    | сравнение 15 и 33, $\min = 15$                          |
|       | 15    | 33    | 42    | 07    | 12    | 19    | сравнение 15 и 42, $\min = 15$                          |
|       | 15    | 33    | 42    | 07    | 12    | 19    | сравнение 15 и 07, $\min = 07$                          |
|       | 15    | 33    | 42    | 07    | 12    | 19    | сравнение 07 и 12, $\min = 07$                          |
|       | 15    | 33    | 42    | 07    | 12    | 19    | сравнение 07 и 19, $\min = 07$ , обмен 15 и 07          |
| шаг 2 | 07    | 33    | 42    | 15    | 12    | 19    | сравнение 33 и 42, $\min = 33$                          |
|       | 07    | 33    | 42    | 15    | 12    | 19    | сравнение 33 и 15, $\min = 15$                          |
|       | 07    | 33    | 42    | 15    | 12    | 19    | сравнение 15 и 12, $\min = 12$                          |
|       | 07    | 33    | 42    | 15    | 12    | 19    | сравнение 12 и 19, $\min = 12$ , обмен 33 и 12          |
| шаг 3 | 07    | 12    | 42    | 15    | 33    | 19    | сравнение 42 и 15, $\min = 15$                          |
|       | 07    | 12    | 42    | 15    | 33    | 19    | сравнение 15 и 33, $\min = 15$                          |
|       | 07    | 12    | 42    | 15    | 33    | 19    | сравнение 15 и 19, $\min = 15$ , обмен 42 и 15          |
| шаг 4 | 07    | 12    | 15    | 42    | 33    | 19    | сравнение 42 и 33, $\min = 33$                          |
|       | 07    | 12    | 15    | 42    | 33    | 19    | сравнение 33 и 19, $\min = 19$ , обмен 42 и 19          |
| шаг 5 | 07    | 12    | 15    | 19    | 33    | 42    | сравнение 33 и 42, $\min = 33$ , обмена нет, все готово |

В данном примере сделано 15 сравнений (как и в методе пузырька), но всего 4 перестановки. Эта особенность сохраняется и в целом: по числу сравнений метод выбора близок к методу пузырька, но по числу перестановок существенно превосходит оба рассмотренные выше методы (оценка числа перестановок  $n \cdot \log_2 n$ )

Особенности программной реализации. Внешний цикл обрабатывает основные шаги и выполняет перестановку минимального элемента, а внутренний организует поиск наименьшего элемента в необработанной части массива

```
for i := 1 to n-1 do
 begin
 k := i; temp := a [i]; {устанавливаем начальный минимальный элемент}
 for j := i+1 to n do
 if a [j] < temp then
 begin {изменяем текущий минимальный элемент}
 k := j; temp := a [j];
 end;
 a [k] := a [i]; a [i] := temp;
 end;
```

**Общее заключение** по простейшим методам сортировки.

Метод обмена (пузырька) имеет единственное преимущество – нулевое число пересылок в случае, если исходный набор уже отсортирован в нужном порядке. В остальных случаях все его показатели пропорциональны  $n^2$ .

Метод вставок также дает хорошие результаты для упорядоченных входных данных (число сравнений и пересылок пропорционально  $n$ ). Во всех остальных случаях его показатели пропорциональны  $n^2$ , хотя что касается оценки среднего числа сравнений, то она чуть лучше, чем у других методов. Многочисленные эксперименты показывают, что метод вставок дает **наименьшее** время сортировки среди всех **простейших** методов.

Метод выбора, как это и следовало ожидать, имеет лучшие показатели по числу пересылок, особенно – для общего случая, где оценка  $O(n \cdot \log_2 n)$  заметно лучше оценки  $O(n^2)$ . Поэтому его можно рекомендовать к использованию из всех простейших методов в том случае, если именно число перестановок является наиболее важным.

### 1.6. Практическое задание

Реализовать программу, объединяющую простейшие методы сортировки массивов:

- сортировку обменом (метод пузырька)
- сортировку выбором
- сортировку вставками

Каждый метод реализуется своей подпрограммой, добавляемой в основную программу по мере разработки. Кроме того, необходима вспомогательная подпрограмма генерации исходного массива случайных целых чисел с заданным числом элементов (не более 10 000) и выводом этого массива на экран

Каждый исходный массив должен обрабатываться всеми подпрограммами сортировки с подсчетом и выводом фактического числа выполненных сравнений и пересылок. Поскольку каждый из универсальных методов выполняет сортировку “на месте”, т.е. изменяет исходный массив, то для наглядности работы можно передавать в подпрограмму сортировки копию исходного массива, объявив его как параметр-значение.

После завершения разработки программы необходимо выполнить всеми методами сортировку нескольких массивов с разным числом элементов (10, 100, 1.000, 10.000) и провести сравнительный анализ эффективности рассматриваемых методов.

Главная программа должна реализовать диалог с пользователем для выбора метода сортировки.

### 1.7. Контрольные вопросы по теме

1. В чем состоит задача выбора алгоритмов решения однотипных задач?
2. Какие критерии используются при выборе алгоритмов?
3. Как оценивается трудоемкость алгоритма?
4. Что такое O-нотация и для чего она используется?
5. Какие группы функций можно выделить с помощью O-нотации?
6. Какие рекомендации следует использовать при выборе алгоритмов с помощью O-нотации?
7. Что можно сказать о применимости алгоритмов класса  $O(2^n)$  и  $O(n!)$ ?
8. Как оценивается трудоемкость программы, использующей несколько взаимодействующих алгоритмов?
9. Как классифицируются методы сортировки?
10. Что такое внутренняя и внешняя сортировка и в чем состоят особенности этих задач?
11. В чем состоят особенности универсальных и специальных методов внутренней сортировки?
12. Какие основные методы сортировки относятся к универсальным и какую они имеют трудоемкость?
13. В чем состоит практическое значение изучения простейших методов сортировки?
14. Как классифицируются методы поиска?
15. В чем состоит суть метода сортировки обменом?
16. Какие шаги выполняет алгоритм сортировки обменом?
17. Как программно реализуется сортировка обменом?
18. В чем достоинства и недостатки метода сортировки обменом?
19. Приведите практический пример сортировки массива методом обмена.
20. В чем состоит суть метода сортировки вставками?
21. Какие шаги выполняет алгоритм сортировки вставками?
22. Как программно реализуется сортировка вставками?
23. В чем достоинства и недостатки метода сортировки вставками?

24. Приведите практический пример сортировки массива методом вставок.
25. В чем состоит суть метода сортировки выбором?
26. Какие шаги выполняет алгоритм сортировки выбором?
27. Как программно реализуется сортировка выбором?
28. В чем достоинства и недостатки метода сортировки выбором?
29. Приведите практический пример сортировки массива методом выбора.

## Тема 2. Улучшенные методы сортировки массивов

### 2.1. Метод Шелла

Метод Шелла является улучшенным вариантом метода вставок. Поскольку метод вставок дает хорошие показатели качества для небольших или почти упорядоченных наборов данных, метод Шелла использует эти свойства за счет многократного применения метода вставок.

Алгоритм метода Шелла состоит в многократном повторении двух основных действий:

- **объединение** нескольких элементов исходного массива по некоторому правилу
- **сортировка** этих элементов обычным методом вставок

Более подробно, на первом этапе **группируются** элементы входного набора с **достаточно большим шагом**. Например, выбираются все 1000-е элементы, т.е. создаются группы:

группа 1: 1, 1001, 2001, 3001 и т.д.

группа 2: 2, 1002, 2002, 3002 и т.д.

группа 3: 3, 1003, 2003, 3003 и т.д.

.....

группа 1000: 1000, 2000, 3000 и т.д.

Внутри каждой группы выполняется **обычная** сортировка вставками, что эффективно за счет **небольшого** числа элементов в группе.

На втором этапе выполняется группировка уже с **меньшим** шагом, например - все сотые элементы. В каждой группе опять выполняется обычная

сортировка вставками, которая эффективна за счет того, что после первого этапа в каждой группе набор данных будет уже **частично отсортирован**.

На третьем этапе элементы группируются с еще меньшим шагом, например – все десятые элементы. Выполняется сортировка, группировка с еще меньшим шагом и т.д.

На последнем этапе сначала выполняется группировка с **шагом 1**, создающая единственный набор данных размерности  $n$ , а затем - сортировка **практически отсортированного** набора.

**Пример.** Исходный набор: 15 – 33 – 42 – 07 – 12 – 19

Выполняем группировку с шагом 3, создаем три группы по 2 элемента и сортируем каждую из них отдельно:

группа 1: 15 – 07  $\Rightarrow$  07 – 15 (1 сравнение, 1 пересылка)

группа 2: 33 – 12  $\Rightarrow$  12 – 33 (1 сравнение, 1 пересылка)

группа 3: 42 – 19  $\Rightarrow$  19 – 42 (1 сравнение, 1 пересылка)

Новый набор чисел: 07 – 15 – 12 – 33 – 19 – 42

Группировка с меньшим шагом 2 дает 2 группы по 3 элемента, которые сортируются отдельно:

группа 1: 07 – 12 – 19  $\Rightarrow$  уже упорядочена (2 сравнения, 0 пересылок)

группа 2: 15 – 33 – 42  $\Rightarrow$  уже упорядочена (2 сравнения, 0 пересылок)

Новый набор чисел: 07 – 12 – 19 – 15 – 33 – 42

Последняя группировка с шагом 1 дает сам набор чисел; к нему применяется сортировка вставками с 5-ю сравнениями и только одной пересылкой, после чего получаем искомый результат.

Итого – 12 сравнений и 4 пересылки, что в общем-то не лучше чем у простых методов. Однако, здесь надо учесть два фактора.

Фактор 1 (общий). Улучшенные методы показывают свою эффективность именно для **больших** наборов данных (сотни, тысячи и т.д. элементов). Для очень малых наборов (как в примере) они могут давать даже худшие результаты.

Фактор 2 (специфический). Эффективность метода Шелла существенно зависит от выбора **последовательности шагов** группировки. Эта последовательность обязательно должна быть **убывающей**, а последний шаг обязательно **равен 1**. В настоящее время **неизвестна наилучшая** последовательность шагов, обеспечивающая наименьшую трудоемкость. На основе многочисленных экспериментов установлено, что число шагов группировки надо выбирать по формуле  $[(\log_2 n)] - 1$ , где скобки  $[ ]$  используются для обозначения целой части числа, а в качестве самих последовательностей рекомендуется один из следующих наборов (обращаю внимание: для удобства восприятия шаги даются в **обратном порядке**):

1, 3, 5, 9, 17, 33, ... (общая формула:  $t_k = (2 * t_{k-1}) - 1$  )

1, 3, 7, 15, 31, 63, 127, 255, 511, 1023, 2047, 4095, 8191, 16383, 32767 ...

(общая формула:  $t_k = (2 * t_{k-1}) + 1$ , а еще проще –  $(2^k - 1)$ ).

В соответствии с этими рекомендациями, в предыдущем примере надо взять лишь 2 шага группировки со значениями 3 и 1. В этом случае потребуется лишь 8 сравнений и 5 пересылок.

Что касается программной реализации, то по сравнению с методом вставок потребуется организовать еще один **самый внешний** цикл для выполнения группировок элементов с убывающими шагами. Сами шаги можно вычислять по приведенным выше формулам, а можно хранить в предварительно подготовленном вспомогательном массиве. Никакого выделения сгруппированных элементов в отдельные массивы не производится, вся работа выполняется за счет изменения индексов элементов.

```

for m := 1 to t do {t – число шагов группировки, m – номер шага}
begin
 k := h [m]; {выбор величины шага группировки из заданного массива}
 for i := k + 1 to n do {сортировка вставками внутри каждой группы}
 begin
 temp := a [i]; j := i – k;
 while (j > 0) and (temp < a [j]) do

```



**begin**

$a[j + k] := a[j]; \quad j := j - k;$

**end;**

$a[j + k] := \text{temp};$

**end;**

**end;**

Оценка трудоемкости метода Шелла выражается соотношением  $O(n^{1,2})$ , что лучше чем у простейших методов, особенно при больших  $n$ .

## 2.2. Метод быстрой сортировки

Данный метод в настоящее время считается наиболее быстрым универсальным методом сортировки. Как ни странно, он является обобщением самого плохого из простейших методов – обменного метода. Эффективность метода достигается тем, что перестановка применяется не для соседних элементов, а **отстоящих друг от друга на приличном расстоянии**.

Более конкретно, алгоритм быстрой сортировки заключается в следующем.

- пусть каким-то образом в исходном наборе выделен некий элемент  $x$ , который принято называть **опорным**. В простейшем случае в качестве опорного можно взять **серединный** элемент массива
- просматривается часть массива, расположенная **левее** опорного элемента и находится первый по порядку элемент  $a_i > x$
- после этого просматривается часть массива, расположенная **правее** опорного элемента, причем - **в обратном порядке**, и находится первый по порядку (с конца) элемент  $a_j < x$
- производится **перестановка** элементов  $a_i$  и  $a_j$
- после этого в левой части, начиная с  $a_i$  отыскивается еще один элемент, больший  $x$ , а в правой части, начиная с  $a_j$  отыскивается элемент, меньший  $x$
- эти два элемента меняются местами

- эти действия (поиск слева и справа с последующим обменом) продолжаются до тех пор, пока не будет достигнут опорный элемент **x**
- после этого слева от опорного элемента **x** будут находиться элементы, **меньшие опорного**, а справа – элементы, **большие опорного**. При этом обе половины скорее всего не будут отсортированными
- после этого массив разбивается на **правую** и **левую** части и каждая часть обрабатывается **отдельно** по той же самой схеме: определение опорного элемента, поиск слева и справа соответствующих элементов и их перестановка и т.д.

**Пример.** Пусть исходный набор включает 11 чисел: 13-42-28-17-09-25-47-31-39-15-20. Основные шаги сортировки:

1. Выбор срединного элемента 25 (индекс 6): 13 42 28 17 09 **25** 47 31 39 15 20
2. поиск слева первого элемента, большего 25: 42 (2 сравнения)
3. поиск справа от конца первого элемента, меньшего 25: 20 (1 сравнение)
4. перестановка элементов 42 и 20: 13 **20** 28 17 09 **25** 47 31 39 15 **42**
5. поиск слева от 25 еще одного элемента, большего 25: 28 (1 сравнение)
6. поиск справа от 25 еще одного элемента, меньшего 25: 15 (1 сравнение)
7. перестановка элементов 28 и 15: 13 20 **15** 17 09 **25** 47 31 39 **28** 42
8. поиск слева от 25 еще одного элемента, большего 25: нет (2 сравнения)
9. поиск справа от 25 еще одного элемента, меньшего 25: нет (3 сравнения)
10. теперь слева от 25 все элементы меньше 25, а справа – больше
11. выделяем отдельно левую часть: 13 20 15 17 09
12. выбираем срединный элемент 15 (индекс 3): 13 20 **15** 17 09
13. поиск слева от 15 элемента, большего 15: 20 (2 сравнения)
14. поиск справа от 15 элемента, меньшего 15: 09 (1 сравнение)
15. перестановка 20 и 09: 13 **09** **15** 17 **20**
16. поиск справа от 15 еще одного элемента, меньшего 15: нет (1 сравнение)
17. теперь слева от 15 все элементы меньше 15, а справа – больше

18. поскольку слева от 15 только 2 элемента, просто сравниваем их друг с другом и переставляем (09 и 13)
19. поскольку справа от 15 только 2 элемента, просто сравниваем их и не переставляем
20. получаем для левой части упорядоченный набор: **09 13 15 17 20**
21. возвращаемся к правой части: 47 31 39 28 42
22. выделяем срединный элемент 39 (индекс в данном поднаборе – 3): 47 31 **39** 28 42
23. поиск слева от 39 элемента, большего 39: 47 (1 сравнение)
24. поиск справа от 39 элемента, меньшего 39: 28 (2 сравнения)
25. переставляем 47 и 28: **28** 31 **39** **47** 42
26. поиск слева от 39 еще одного элемента, большего 39: нет (1 сравнение)
27. теперь слева от 39 все элементы меньше 39, а справа – больше
28. поскольку слева от 39 только 2 элемента, просто сравниваем их и не переставляем
29. поскольку справа от 39 только 2 элемента, просто сравниваем их и переставляем (42 и 47)
30. получаем для правой части упорядоченный набор: **28 31 39 42 47**
31. вместе с левой частью и срединным элементом 25 получаем окончательный результат

Итого для данного примера потребовалось 22 сравнения и 6 пересылок.

В целом, оценка трудоемкости метода быстрой сортировки является типичной для улучшенных методов и выражается соотношением  $(n \cdot \log_2 n)/6$ . Отсюда следует, что данный метод неэффективен при малых  $n$  (десятки или сотни элементов), но с ростом  $n$  его эффективность резко растет, и при очень больших  $n$  метод дает **наилучшие показатели** среди всех универсальных методов сортировки.

К сожалению, есть одна ситуация, когда быстрая сортировка **теряет свою эффективность** и становится пропорциональной  $n^2$ , т.е. опускается до уровня простых методов. Эта ситуация связана с **правилом выбора опорного**

**элемента.** Эффективность метода сильно зависит от выбора опорного элемента, и использование простейшего способа выбора (серединный элемент массива) часто приводит к падению эффективности метода. Это связано с тем, что каждое разделение массива на две половины в идеале должно давать **примерно равное** число элементов слева и справа от опорного элемента (принцип дихотомии!). Если опорный элемент близок к минимальному или максимальному, после попарных перестановок будут получены **существенно неравномерные** наборы. Если подобная ситуация возникает на каждом шаге работы алгоритма, общая эффективность резко падает. Для устранения этого недостатка надо уметь **правильно выбирать опорный элемент**.

Наилучшее правило выбора опорного элемента – это так называемая **медиана**. Медиана – это средний элемент массива **не по расположению**, а по **значению**. В приведенном выше примере медианой является число 25, которое также было и серединным элементом (честно говоря, пример был подобран специально). К сожалению, поиск медианы в массиве является задачей, **сопоставимой по трудоемкости** с самой сортировкой, поэтому были предложены другие, более простые правила выбора опорного элемента.

На практике хорошо показал себя следующий способ: выбрать случайно в массиве **три элемента** и взять в качестве опорного **средний из них**. Этот способ очень прост в реализации, т.к. требует только двух сравнений, но, конечно, он может обеспечивать хорошие показатели только в среднем, и не гарантирует идеальное поведение алгоритма абсолютно для **ЛЮБЫХ** входных данных.

Есть и другие усовершенствования базового алгоритма быстрой сортировки. Среди них:

- Проверка каждого подмассива на возможную упорядоченность перед самой сортировкой
- Для малых подмассивов ( $n < 10$ ) разумно проводить сортировку с помощью более простых методов, например – Шелла

Комбинация всех этих методов известна как метод Синглтона [7].

Что касается программной реализации базового алгоритма, то можно заметить его принципиальную особенность: разделение массива на 2 половины, разделение каждой половины на свои половины и т.д. При каждом разделении приходится **запоминать** правую половину (конечно, не сами элементы, а лишь **индексы** левой и правой границы) и **возвращаться** к ней после полной обработки левой половины. Все это как нельзя лучше соответствует **рекурсивному** принципу обработки, и поэтому быстрая сортировка проще всего реализуется рекурсивно. Конечно, при необходимости можно включить явное использование стека для запоминания левой и правой границы подмассива, аналогично нерекурсивной реализации процедур обхода дерева.

Рекурсивная реализация с серединным опорным элементом схематично выглядит следующим образом:

**Procedure** QuickSort (left, right : **integer**);

{формальные параметры для запоминания границ}

**var** i, j : **integer**;

sred, temp : <описание элемента исходного массива>;

**begin**

i := left; j := right; {установка начальных значений границ подмассива}

sred := mas [(left + right) **div** 2]; {определение серединного элемента}

**repeat**

**while** (mas [i] < sred) **do** i := i + 1;

{поиск слева элемента, большего опорного}

**while** (mas [j] > sred) **do** j := j - 1;

{поиск справа элемента, меньшего опорного}

**if** i <= j **then**

**begin** {обмениваем элементы и изменяем индексы}

temp := mas [i]; mas [i] := mas [j]; mas [j] := temp;

i := i + 1; j := j - 1;

**end**;

**until** i > j;

```

if left < j then QuickSort (left , j); {обработка левой половины}
if i < right then QuickSort (i , right); {обработка правой половины}
end;

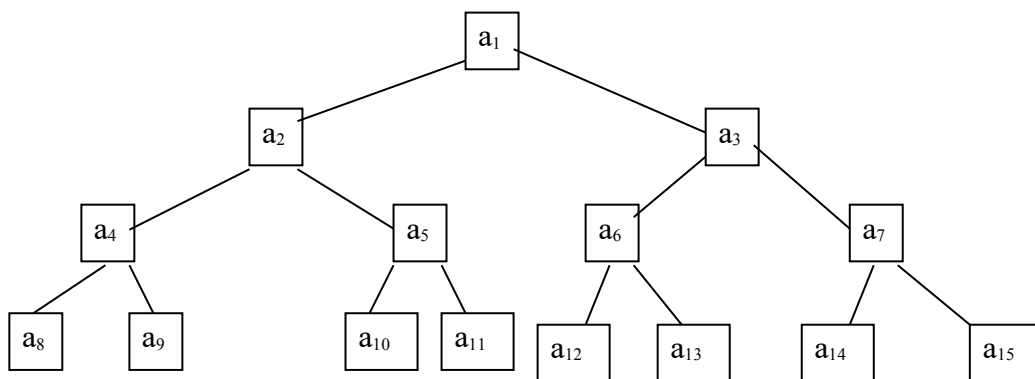
```

Первоначальный вызов этой процедуры производится в главной программе в виде QuickSort (1, n); здесь n – число элементов в массиве.

### 2.3. Пирамидальная сортировка

Пирамидальная сортировка является улучшенным вариантом сортировки выбором, в которой на каждом шаге должен определяться наименьший элемент в необработанном наборе данных. Поиск наименьшего элемента можно **совместить** с выполнением некоторых **дополнительных действий**, облегчающих поиск на последующих шагах. Для этого исходный набор данных представляется особым образом – в виде так называемой **пирамиды**. Пирамида – это специальная разновидность двоичного дерева, построенная по особым правилам и имеющая особые свойства.

Пусть имеется исходный массив n элементов  $a_1, a_2, a_3, \dots, a_n$ . Расположим эти элементы в виде двоичного дерева следующего вида (здесь важен порядок следования индексов элементов):



Подобное дерево называется пирамидой, если для всех элементов с индексами от 1 до  $n/2$  выполняются следующие условия:

$$a_i \leq a_{2i} \text{ и } a_i \leq a_{2i+1}$$

В частности, эти условия означают:  $a_1 \leq a_2$  и  $a_1 \leq a_3$ ;  $a_2 \leq a_4$  и  $a_2 \leq a_5$ ;  $a_3 \leq a_6$  и  $a_3 \leq a_7$ ;  $a_4 \leq a_8$  и  $a_4 \leq a_9$ , и т.д.

Другими словами, в каждом элементарном поддереве значение в **вершине** этого поддерева **меньше или равно** значений в **вершинах-потомках**.

**Пример** двоичного дерева-пирамиды с 15-ю элементами:

|                                                                                                 |                                                 |
|-------------------------------------------------------------------------------------------------|-------------------------------------------------|
|                                                                                                 |                                                 |
| <p>Это – пирамида. Соответствующий массив:<br/>15-25-20-30-40-22-33-50-60-45-47-44-28-55-66</p> | <p>Это – не пирамида (вершина 12 меньше 20)</p> |

Такие пирамиды иногда называют **минимальными**, в отличие от максимальных, где правило упорядоченности прямо противоположное.

Из примера видно, что пирамида **НЕ** является деревом поиска, т.к. строится по другим правилам.

Можно отметить следующие **полезные свойства** пирамиды:

- на **вершине** пирамиды всегда находится **наименьший** элемент во всем массиве (элемент  $a_1$ ), элемент  $a_2$  является наименьшим для левого поддерева, элемент  $a_3$  является наименьшим для правого поддерева и т.д.
- вершины, лежащие на самом нижнем уровне пирамиды **всегда** образуют из себя элементарные пирамиды, поскольку у них нет никаких потомков и их сравнивать не с кем

Пирамидальная сортировка включает два больших этапа:

- представление исходного массива в виде пирамиды
- последовательные удаления минимального элемента с вершины пирамиды с заменой его другим элементом

Реализация 1 этапа включает следующие шаги:

- условно разделяем исходный массив на две половины: левую с индексами от 1 до  $\lfloor n/2 \rfloor$ , и правую с индексами от  $\lfloor n/2 \rfloor + 1$  до  $n$  (здесь  $\lfloor \ ]$  обозначает

целую часть); считаем пока, что левая половина образует верхнюю часть строящейся пирамиды, а правая – нижний слой терминальных вершин

- выбираем в левой половине последний элемент (его индекс  $m = \lfloor n/2 \rfloor$ ), а в правой половине – его непосредственных потомков (одного или двух, но хотя бы один будет обязательно) с индексом  $2m$  и, возможно,  $2m+1$
- если потомков два, то выбираем из них наименьшего
- сравниваем элемент  $a_m$  с наименьшим из потомков: если он больше, то меняем эти элементы в массиве местами для получения фрагмента пирамиды; в противном случае оставляем все без изменений, поскольку эти элементы уже образуют фрагмент пирамиды
- повторяем все описанные действия последовательно для оставшихся в левой части элементов справа налево, т.е. для  $a_{m-1}, a_{m-2}, a_{m-3}, \dots, a_1$ , при этом если происходит обмен между родительской вершиной и одним из потомков, выполняется проверка для новой вершины-потомка, т.к. она может иметь своих потомков, с которыми возможно потребуется ее обменять для выполнения условия пирамиды.

Тем самым, для каждого элемента массива находится его новое расположение в массиве таким образом, чтобы выполнялись условия пирамиды. Процесс определения нужного положения элемента в массиве-пирамиде называют **просеиванием** элемента через пирамиду. Построение пирамиды заканчивается после просеивания первого элемента  $a_1$ . **Пример** для 15 элементов приведен в таблице (символ  $\sim$  обозначает перестановку элементов)

| $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | $a_7$ | $a_8$ | $a_9$ | $a_{10}$ | $a_{11}$ | $a_{12}$ | $a_{13}$ | $a_{14}$ | $a_{15}$ | сравнение и обмен                  |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|----------|----------|----------|----------|----------|------------------------------------|
| 45    | 40    | 28    | 25    | 30    | 44    | 33    | 22    | 60    | 15       | 55       | 47       | 66       | 20       | 50       | $33 > \min(20, 50)$ , $33 \sim 20$ |
| 45    | 40    | 28    | 25    | 30    | 44    | 20    | 22    | 60    | 15       | 55       | 47       | 66       | 33       | 50       | $44 < \min(47, 66)$ , нет          |
| 45    | 40    | 28    | 25    | 30    | 44    | 20    | 22    | 60    | 15       | 55       | 47       | 66       | 33       | 50       | $30 > \min(15, 55)$ , $30 \sim 15$ |
| 45    | 40    | 28    | 25    | 15    | 44    | 20    | 22    | 60    | 30       | 55       | 47       | 66       | 33       | 50       | $25 > \min(22, 60)$ , $25 \sim 22$ |
| 45    | 40    | 28    | 22    | 15    | 44    | 20    | 25    | 60    | 30       | 55       | 47       | 66       | 33       | 50       | $28 > \min(44, 20)$ , $28 \sim 20$ |
| 45    | 40    | 20    | 22    | 15    | 44    | 28    | 25    | 60    | 30       | 55       | 47       | 66       | 33       | 50       | $28 < \min(33, 50)$ , нет          |
| 45    | 40    | 20    | 22    | 15    | 44    | 28    | 25    | 60    | 30       | 55       | 47       | 66       | 33       | 50       | $40 > \min(22, 15)$ , $40 \sim 15$ |
| 45    | 15    | 20    | 22    | 40    | 44    | 28    | 25    | 60    | 30       | 55       | 47       | 66       | 33       | 50       | $40 > \min(30, 55)$ , $40 \sim 30$ |
| 45    | 15    | 20    | 22    | 30    | 44    | 28    | 25    | 60    | 40       | 55       | 47       | 66       | 33       | 50       | $45 > \min(15, 20)$ , $45 \sim 15$ |
| 15    | 45    | 20    | 22    | 30    | 44    | 28    | 25    | 60    | 40       | 55       | 47       | 66       | 33       | 50       | $45 > \min(22, 30)$ , $45 \sim 22$ |



|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |                         |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|-------------------------|
| 15 | 22 | 20 | 45 | 30 | 44 | 28 | 25 | 60 | 40 | 55 | 47 | 66 | 33 | 50 | 45 > min(25, 60), 45~25 |
| 15 | 22 | 20 | 25 | 30 | 44 | 28 | 45 | 60 | 40 | 55 | 47 | 66 | 33 | 50 | Пирамида построена      |

В худшем случае каждый шаг в просеивании очередного элемента требует двух сравнений: сначала сравниваются два потомка текущего элемента, а потом наименьший из них сравнивается с самим элементом. В примере для построения пирамиды потребовалось  $11 \cdot 2 = 22$  сравнения и 9 пересылок.

Реализация второго этапа состоит в  $(n-1)$ -кратном повторении следующих действий:

- с вершины пирамиды забирается минимальный элемент  $a_1$
- на его место в вершину пирамиды помещается последний элемент в массиве, причем индекс этого последнего элемента на каждом шаге будет уменьшаться от  $a_n$  до  $a_2$
- помещенный в вершину элемент просеивается через пирамиду обычным образом, при этом он встает на свое место, а в вершину пирамиды выталкивается минимальный из оставшихся в массиве элементов
- на последнем шаге в пирамиде останется один элемент (самый большой) и сортировка будет закончена

При этом возникает вопрос – куда девать снимаемые с вершины пирамиды элементы? Можно просто помещать их в конец массива на место элемента, размещаемого в вершине. В результате на месте исходного массива создается упорядоченный ПО УБЫВАНИЮ набор данных. При необходимости, алгоритм легко можно изменить для получения возрастающих последовательностей, если вместо минимальных использовать максимальные пирамиды.

| $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ | $a_7$ | $a_8$ | $a_9$ | $a_{10}$ | $a_{11}$ | $a_{12}$ | $a_{13}$ | $a_{14}$ | $a_{15}$ | сравнение и обмен       |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|----------|----------|----------|----------|----------|-------------------------|
| 15    | 22    | 20    | 25    | 30    | 44    | 28    | 45    | 60    | 40       | 55       | 47       | 66       | 33       | 50       | 15 = min, 15~50         |
| 50    | 22    | 20    | 25    | 30    | 44    | 28    | 45    | 60    | 40       | 55       | 47       | 66       | 33       | 15       | 50 > min(22, 20), 50~20 |
| 20    | 22    | 50    | 25    | 30    | 44    | 28    | 45    | 60    | 40       | 55       | 47       | 66       | 33       | 15       | 50 > min(44, 28), 50~28 |
| 20    | 22    | 28    | 25    | 30    | 44    | 50    | 45    | 60    | 40       | 55       | 47       | 66       | 33       | 15       | 50 > 33, 50~33          |
| 20    | 22    | 28    | 25    | 30    | 44    | 33    | 45    | 60    | 40       | 55       | 47       | 66       | 50       | 15       | 20 = min, 20~50         |
| 50    | 22    | 28    | 25    | 30    | 44    | 33    | 45    | 60    | 40       | 55       | 47       | 66       | 20       | 15       | 50 > min(22, 28), 50~22 |
| 22    | 50    | 28    | 25    | 30    | 44    | 33    | 45    | 60    | 40       | 55       | 47       | 66       | 20       | 15       | 50 > min(25, 30), 50~25 |
| 22    | 25    | 28    | 50    | 30    | 44    | 33    | 45    | 60    | 40       | 55       | 47       | 66       | 20       | 15       | 50 > min(45, 60), 50~45 |
| 22    | 25    | 28    | 45    | 30    | 44    | 33    | 50    | 60    | 40       | 55       | 47       | 66       | 20       | 15       | 22 = min, 22~66         |
| 66    | 25    | 28    | 45    | 30    | 44    | 33    | 50    | 60    | 40       | 55       | 47       | 22       | 20       | 15       | 66 > min(25, 28), 66~25 |

|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |                        |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|------------------------|
| 25 | 66 | 28 | 45 | 30 | 44 | 33 | 50 | 60 | 40 | 55 | 47 | 22 | 20 | 15 | 66>min(45, 30), 66~30  |
| 25 | 30 | 28 | 45 | 66 | 44 | 33 | 50 | 60 | 40 | 55 | 47 | 22 | 20 | 15 | 66>min(40, 55), 66~40  |
| 25 | 30 | 28 | 45 | 40 | 44 | 33 | 50 | 60 | 66 | 55 | 47 | 22 | 20 | 15 | 25 = min, 25~47        |
| 47 | 30 | 28 | 45 | 40 | 44 | 33 | 50 | 60 | 66 | 55 | 25 | 22 | 20 | 15 | 47>min(30, 28), 47~28  |
| 28 | 30 | 47 | 45 | 40 | 44 | 33 | 50 | 60 | 66 | 55 | 25 | 22 | 20 | 15 | 47>min(44, 33), 47~33  |
| 28 | 30 | 33 | 45 | 40 | 44 | 47 | 50 | 60 | 66 | 55 | 25 | 22 | 20 | 15 | 28 = min, 28~55        |
| 55 | 30 | 33 | 45 | 40 | 44 | 47 | 50 | 60 | 66 | 28 | 25 | 22 | 20 | 15 | 55>min(30, 33), 55~30  |
| 30 | 55 | 33 | 45 | 40 | 44 | 47 | 50 | 60 | 66 | 28 | 25 | 22 | 20 | 15 | 55>min(45, 40), 55~40  |
| 30 | 40 | 33 | 45 | 55 | 44 | 47 | 50 | 60 | 66 | 28 | 25 | 22 | 20 | 15 | 55<66, 30 = min, 30~66 |
| 66 | 40 | 33 | 45 | 55 | 44 | 47 | 50 | 60 | 30 | 28 | 25 | 22 | 20 | 15 | 66>min(40, 33), 66~33  |
| 33 | 40 | 66 | 45 | 55 | 44 | 47 | 50 | 60 | 30 | 28 | 25 | 22 | 20 | 15 | 66>min(44, 47), 66~44  |
| 33 | 40 | 44 | 45 | 55 | 66 | 47 | 50 | 60 | 30 | 28 | 25 | 22 | 20 | 15 | 33 = min, 33~60        |
| 60 | 40 | 44 | 45 | 55 | 66 | 47 | 50 | 33 | 30 | 28 | 25 | 22 | 20 | 15 | 60>min(40, 44), 60~40  |
| 40 | 60 | 44 | 45 | 55 | 66 | 47 | 50 | 33 | 30 | 28 | 25 | 22 | 20 | 15 | 60>min(45, 55), 60~45  |
| 40 | 45 | 44 | 60 | 55 | 66 | 47 | 50 | 33 | 30 | 28 | 25 | 22 | 20 | 15 | 60>50, 60~50           |
| 40 | 45 | 44 | 50 | 55 | 66 | 47 | 60 | 33 | 30 | 28 | 25 | 22 | 20 | 15 | 40 = min, 40~60        |
| 60 | 45 | 44 | 50 | 55 | 66 | 47 | 40 | 33 | 30 | 28 | 25 | 22 | 20 | 15 | 60>min(45, 44), 60~44  |
| 44 | 45 | 60 | 50 | 55 | 66 | 47 | 40 | 33 | 30 | 28 | 25 | 22 | 20 | 15 | 60>min(66, 47), 60~47  |
| 44 | 45 | 47 | 50 | 55 | 66 | 60 | 40 | 33 | 30 | 28 | 25 | 22 | 20 | 15 | 44 = min, 44~60        |
| 60 | 45 | 47 | 50 | 55 | 66 | 44 | 40 | 33 | 30 | 28 | 25 | 22 | 20 | 15 | 60>min(45, 47), 60~45  |
| 45 | 60 | 47 | 50 | 55 | 66 | 44 | 40 | 33 | 30 | 28 | 25 | 22 | 20 | 15 | 60>min(50, 55), 60~50  |
| 45 | 50 | 47 | 60 | 55 | 66 | 44 | 40 | 33 | 30 | 28 | 25 | 22 | 20 | 15 | 45 = min, 45~66        |
| 66 | 50 | 47 | 60 | 55 | 45 | 44 | 40 | 33 | 30 | 28 | 25 | 22 | 20 | 15 | 66>min(50, 47), 66~47  |
| 47 | 50 | 66 | 60 | 55 | 45 | 44 | 40 | 33 | 30 | 28 | 25 | 22 | 20 | 15 | 47 = min, 47~55        |
| 55 | 50 | 66 | 60 | 47 | 45 | 44 | 40 | 33 | 30 | 28 | 25 | 22 | 20 | 15 | 55>min(50, 66), 55~50  |
| 50 | 55 | 66 | 60 | 47 | 45 | 44 | 40 | 33 | 30 | 28 | 25 | 22 | 20 | 15 | 55<60, 50 = min, 50~60 |
| 60 | 55 | 66 | 50 | 47 | 45 | 44 | 40 | 33 | 30 | 28 | 25 | 22 | 20 | 15 | 60>min(55, 66), 60~55  |
| 55 | 60 | 66 | 50 | 47 | 45 | 44 | 40 | 33 | 30 | 28 | 25 | 22 | 20 | 15 | 55 = min, 55~66        |
| 66 | 60 | 55 | 50 | 47 | 45 | 44 | 40 | 33 | 30 | 28 | 25 | 22 | 20 | 15 | 66>60, 66~60           |
| 60 | 66 | 55 | 50 | 47 | 45 | 44 | 40 | 33 | 30 | 28 | 25 | 22 | 20 | 15 | 60 = min, 60~66        |
| 66 | 60 | 55 | 50 | 47 | 45 | 44 | 40 | 33 | 30 | 28 | 25 | 22 | 20 | 15 | сортировка закончена   |

В данном примере выполнено 51 сравнение и 40 пересылок, что вместе с этапом построения пирамиды дает 73 сравнения и 49 пересылок.

В целом, данный метод с точки зрения трудоемкости имеет типичное для улучшенных методов поведение: довольно высокая трудоемкость для небольших  $n$ , но с ростом  $n$  эффективность метода растет. При больших  $n$  метод в среднем немного уступает быстрой сортировке и имеет оценку порядка  $(n \cdot \log_2 n)/2$ . Единственное, в чем он превосходит быструю сортировку, так это поведение на **аномальных** входных данных, когда быстрая сортировка перестает быть “быстрой”, а пирамидальная сохраняет свою трудоемкость порядка  $O(n \cdot \log_2 n)$ . В [4] указывается, что пирамидальную сортировку выгодно использовать в том случае, когда требуется не провести полную

сортировку большого набора данных, а лишь найти несколько (причем – немного) первых наименьших элементов.

Необходимо отметить, что понятие пирамидального дерева имеет и самостоятельное значение, и часто используется для решения других задач, не связанных с сортировкой массивов, например – для эффективной реализации приоритетных очередей [4].

В заключение рассмотрим вопросы программной реализации пирамидальной сортировки. Поскольку оба этапа алгоритма основаны на повторении одинаковых действий по просеиванию элементов, удобно оформить просеивание в виде вспомогательной процедуры.

```
Procedure Sito (al, ar : word);
var i, j : word;
 x : <описание элемента массива>;
begin
 i := al; j := 2*al; x := mas[al];
 if (j < ar) and (mas[j + 1] < mas[j]) then j := j + 1;
 while (j <= ar) and (mas[j] < x) do
 begin
 mas[i] := mas[j]; i := j; j := 2*j;
 if (j < ar) and (mas[j + 1] < mas[j]) then j := j + 1;
 end;
 mas[i] := x;
 end;
```

Тогда основная программа будет лишь включать два последовательных цикла – один для построения пирамиды, второй – для снятия с вершины наименьшего элемента и просеивания нового вершинного элемента.

```
left := (n div 2) + 1; right := n;
 {определение границ правой половины массива}

while left > 1 do
 {цикл построения пирамиды}
 begin
```

```

 left := left - 1; Sito (left, right);
end;
while right > 1 do (цикл сортировки)
begin
 temp := mas[1]; mas[1] := mas[right]; mas[right] := temp;
 right := right - 1; Sito (left, right);
end;

```

## 2.4. Практическое задание

Добавить в ранее созданную программу с простейшими методами сортировки подпрограммы, реализующие все три улучшенных метода сортировки массивов.

Каждый метод реализуется своей подпрограммой, добавляемой в основную программу по мере разработки. Каждый исходный массив должен обрабатываться всеми программами сортировки с подсчетом и выводом фактического числа выполненных сравнений и пересылок. После завершения разработки программы необходимо выполнить всеми методами сортировку нескольких массивов с разным числом элементов (10, 100, 1.000, 10.000) и провести сравнительный анализ эффективности рассматриваемых методов.

Главная программа должна реализовать диалог с пользователем для выбора метода сортировки.

Рекомендации по реализации метода Шелла. Для хранения шагов группировки можно ввести вспомогательный массив. После генерации исходного массива надо организовать цикл, в котором запросить и ввести количество шагов и величины самих шагов. Выполнить сортировку исходного массива для нескольких **разных наборов шагов** и найти среди этих наборов наилучший по числу выполненных сравнений.

## 2.5. Контрольные вопросы по теме

1. В чем состоит суть сортировки методом Шелла?
2. За счет чего метод Шелла дает лучшие показатели по сравнению с простейшими методами?
3. Приведите практический пример сортировки массива методом Шелла.
4. Какой фактор оказывает наибольшее влияние на эффективность сортировки методом Шелла?
5. Какие последовательности шагов группировки рекомендуются для практического использования в методе Шелла?
6. Как программно реализуется сортировка методом Шелла?
7. В чем состоит суть метода быстрой сортировки?
8. За счет чего метод быстрой сортировки дает лучшие показатели по сравнению с простейшими методами?
9. Что такое опорный элемент в методе быстрой сортировки и как он используется?
10. Приведите практический пример быстрой сортировки массива.
11. Что можно сказать о применимости метода быстрой сортировки с точки зрения его эффективности?
12. Какой фактор оказывает решающее влияние на эффективность метода быстрой сортировки?
13. Почему выбор срединного элемента в качестве опорного в методе быстрой сортировки может резко ухудшать эффективность метода?
14. Какое правило выбора опорного элемента в методе быстрой сортировки является наилучшим и почему его сложно использовать?
15. Какое простое правило выбора опорного элемента в методе быстрой сортировки рекомендуется использовать на практике?
16. Какие усовершенствования имеет базовый алгоритм метода быстрой сортировки?
17. Почему быстрая сортировка проще всего программно реализуется с помощью рекурсии?

18. Как программно реализуется рекурсивный вариант метода быстрой сортировки?
19. Какие особенности имеет не рекурсивная программная реализация метода быстрой сортировки?
20. В чем состоит суть метода пирамидальной сортировки?
21. Какой набор данных имеет пирамидальную организацию?
22. Чем отличаются друг от друга дерево поиска и пирамидальное дерево?
23. Приведите пример пирамидального дерева с целочисленными ключами.
24. Какие полезные свойства имеет пирамидальное дерево?
25. Какие шаги выполняются при построении пирамидального дерева?
26. Что такое просеивание элемента через пирамиду?
27. Приведите практический пример построения пирамидального дерева.
28. Какие шаги выполняются на втором этапе пирамидальной сортировки?
29. Приведите практический пример реализации второго этапа пирамидальной сортировки.
30. Что можно сказать о трудоемкости метода пирамидальной сортировки?

### Тема 3. Специальные методы сортировки

#### 3.1. Простейшая карманная сортировка.

Как было отмечено ранее, специальные методы сортировки основаны либо на использовании некоторой **дополнительной информации** об исходном наборе данных, либо на использовании **дополнительной памяти**, сопоставимой по объему с самим сортируемым массивом. Отсюда следует, что они имеют ограниченную область применения, но их огромное преимущество – более высокая эффективность, оцениваемая как  $O(n)$ .

Самая простая разновидность специальных методов – так называемая карманная сортировка.

Пусть как обычно исходный набор данных включает  $n$  элементов, причем относительно их ключей известно следующее:

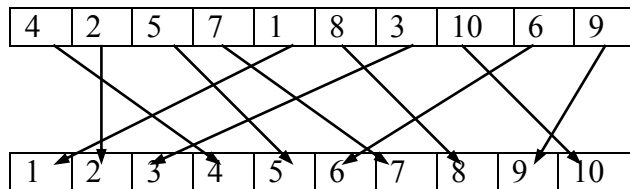
- ключи являются целыми числами, изменяющимися только от 1 до  $n$

- все эти ключи различные

В этом случае ключи можно рассматривать как индексы элементов в массиве. Если в исходном массиве значение ключа может НЕ совпадать с индексом элемента, то после сортировки соответствие должно быть полным: ключ 1 – в ячейке 1, ключ 2 - в ячейке 2 и т.д., ключ  $n$  – в ячейке  $n$ .

Для реализации метода в простейшем случае можно ввести **второй** массив для хранения отсортированного набора. Тогда сортировка сводится к последовательному просмотру элементов исходного массива и **копированию** каждого элемента на его нужное место в результирующем массиве.

**Например**, для массива из 10 элементов с неповторяющимися ключами от 1 до 10 получим:



Вся сортировка сводится лишь к 10 пересылкам и не требует ни одного сравнения! Реализация - одним единственным циклом (здесь Mas – исходный массив, RezMas - результирующий):

**for**  $i := 1$  **to**  $n$  **do**

RezMas[Mas[i].key] := Mas[i];

Если есть проблемы со свободной памятью, то метод можно изменить так, чтобы не использовать дополнительный массив, а проводить сортировку “**на месте**”, правда – за счет небольшого увеличения числа операций. Алгоритм состоит в повторении следующих шагов:

- берем первый элемент в исходном массиве, пусть его ключ равен  $i$
- перемещаем в массиве первый элемент в ячейку  $i$ , а  $i$ -ый элемент – в первую ячейку; после этой операции  $i$ -ый элемент окажется на своем месте

- если ключ первого элемента не равен 1 (например –  $j$ ), то обмениваем его с  $j$ -ым элементом, после чего и  $j$ -ый элемент оказывается на своем месте
- повторяем эти действия до тех пор, пока на первом месте не окажется элемент с ключом 1, причем каждое повторение обязательно размещает один элемент массива на своем “законном” месте
- при необходимости повторяем эти действия и для других элементов массива

**Пример** работы алгоритма – в следующей таблице.

|   |   |   |   |   |    |   |    |   |    |                                                         |
|---|---|---|---|---|----|---|----|---|----|---------------------------------------------------------|
| 4 | 2 | 5 | 7 | 1 | 8  | 3 | 10 | 6 | 9  | 1.key = 4, меняем элементы 4 и 1                        |
| 7 | 2 | 5 | 4 | 1 | 8  | 3 | 10 | 6 | 9  | 1.key = 7, меняем элементы 7 и 1                        |
| 3 | 2 | 5 | 4 | 1 | 8  | 7 | 10 | 6 | 9  | 1.key = 3, меняем элементы 3 и 1                        |
| 5 | 2 | 3 | 4 | 1 | 8  | 7 | 10 | 6 | 9  | 1.key = 5, меняем элементы 5 и 1                        |
| 1 | 2 | 3 | 4 | 5 | 8  | 7 | 10 | 6 | 9  | 1.key = 1, также для 2, 3, 4, 5; 6.key = 8, обмен 6 и 8 |
| 1 | 2 | 3 | 4 | 5 | 10 | 7 | 8  | 6 | 9  | 6.key = 10, меняем 6 и 10                               |
| 1 | 2 | 3 | 4 | 5 | 9  | 7 | 8  | 6 | 10 | 6.key = 9, меняем 6 и 9                                 |
| 1 | 2 | 3 | 4 | 5 | 6  | 7 | 8  | 9 | 10 | 6.key = 6, также для остальных; все готово              |

В этом примере потребовалось 17 сравнений и 7 пересылок. В общем случае, трудоемкость метода пропорциональна  $n$ .

Схема программной реализации:

```

for $i := 1$ to n do
 while $Mas[i].key \neq i$ do
 “переставить $Mas[i]$ и $Mas[Mas[i].key]$ ”;

```

### 3.2. Карманная сортировка для случая повторяющихся ключей

Рассмотренный метод можно обобщить следующим образом. Пусть по-прежнему ключи могут иметь значения только от 1 до  $n$ , но во входном наборе каждое значение **может повторяться** любое число раз, в том числе и ноль. Пусть исходный массив содержит  $m$  элементов, причем  $m > n$ . В этом случае с одним и тем же ключом может быть связано несколько элементов, и их нельзя поместить в одну ячейку результирующего массива.

Очевидным решением является использование **вспомогательных списков** для хранения элементов с **одинаковыми** ключами. Тем самым, приходим к



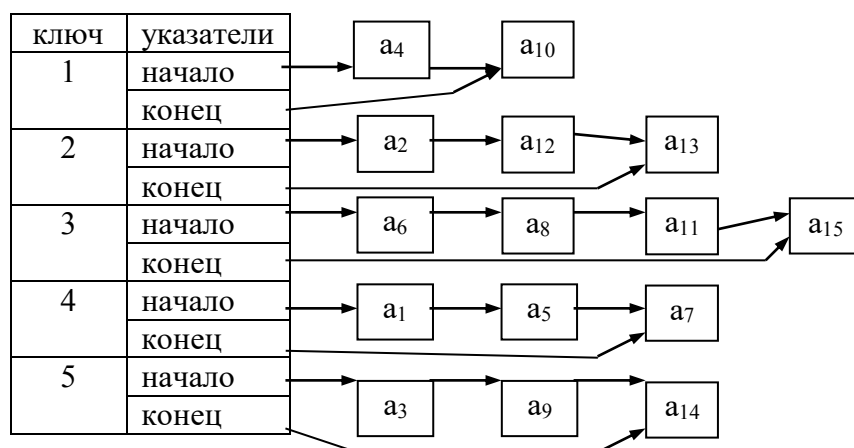
использованию комбинированной структуры данных, а именно – **массиву списков**. Каждая ячейка массива соответствует одному конкретному значению ключа в диапазоне от 1 до  $n$  и хранит указатель на связанный список **одноключевых** элементов.

Тогда прежде всего надо распределить элементы входного набора по ячейкам результирующего массива в соответствии со значениями ключей, создавая и дополняя при необходимости вспомогательные списки. После этого, если требуется получить общий результирующий отсортированный набор данных, можно объединить все списки, добавляя в конец одного списка начало другого. Это легко реализуется, если в каждой ячейке результирующего массива хранить не только адрес первого элемента списка, но и адрес последнего элемента.

**Пример.** Пусть задан следующий входной набор из 15 элементов с диапазоном изменения ключей от 1 до 5 (в скобках указан ключ элемента):

$a_1(4)$ ,  $a_2(2)$ ,  $a_3(5)$ ,  $a_4(1)$ ,  $a_5(4)$ ,  $a_6(3)$ ,  $a_7(4)$ ,  $a_8(3)$ ,  $a_9(5)$ ,  $a_{10}(1)$ ,  $a_{11}(3)$ ,  $a_{12}(2)$ ,  $a_{13}(2)$ ,  $a_{14}(5)$ ,  $a_{15}(3)$

Тогда эти элементы будут распределены следующим образом:



После объединения списков получим:

$a_4, a_{10}, a_2, a_{12}, a_{13}, a_6, a_8, a_{11}, a_{15}, a_1, a_5, a_7, a_3, a_9, a_{14}$

Для программной реализации необходимо объявить массив указателей и ссылочный тип для поддержки списков. Дополнительные затраты памяти связаны в первую очередь с реализацией дополнительных списков, т.к. общее

число элементов в этих списках равно  $m$ , а кроме того для каждого элемента надо 4 байта для адресных полей. Трудоемкость данного метода оценивается соотношением  $O(m+n)$ , а при  $m \gg n$  становится пропорциональной числу элементов  $m$ .

### 3.3. Поразрядная сортировка

Данный метод является обобщением карманной сортировки. Пусть известно, что каждый ключ является  **$k$ -разрядным целым числом**. Например, если  $k = 4$ , то все ключи находятся в диапазоне 0000 – 9999. Смысл поразрядной сортировки заключается в том, что  $k$  раз повторяется карманная сортировка. На первом шаге все ключи группируются по **младшей цифре** (разряд единиц). Для этого в каждом ключе выделяется младшая цифра и элемент помещается в соответствующий список-карман для данной цифры. Потом все списки **объединяются** и создается новый массив, в котором элементы упорядочены по младшей цифре ключа. К этому массиву опять применяется карманная сортировка, но уже по **более старшей** цифре (разряд десятков): в каждом ключе выделяется вторая справа цифра и элементы распределяются по соответствующим спискам. Потом списки объединяются в массив, где элементы будут упорядочены уже **по двум младшим** цифрам. Процесс распределения по все более старшим цифрам с последующим объединением повторяется до старшей цифры (разряд  $k$ ).

Поскольку цифр всего 10, то для реализации метода необходим вспомогательный массив из 10 ячеек для хранения адресов соответствующих списков.

**Пример.** Пусть имеется исходный набор из 15-ти двухразрядных ключей ( $k=2$ ):  
56, 17, 83, 09, 11, 27, 33, 02, 16, 45, 08, 37, 66, 99, 90

**Первый шаг:** выделяем младшую цифру и распределяем ключи по десяти спискам:

ключ 56 в список для цифры 6, ключ 17 в список для цифры 7, ....., ключ 16 опять в список для цифры 6 и т.д.

| 0  | 1  | 2  | 3  | 4 | 5  | 6  | 7  | 8  | 9  |
|----|----|----|----|---|----|----|----|----|----|
| 90 | 11 | 02 | 83 |   | 45 | 56 | 17 | 08 | 09 |
|    |    |    | 33 |   |    | 16 | 27 |    | 99 |
|    |    |    |    |   |    | 66 | 37 |    |    |

Объединяем списки: 90, 11, 02, 83, 33, 45, 56, 16, 66, 17, 27, 37, 08, 09, 99

В этом наборе ключи упорядочены по младшей цифре

**Второй шаг:** выделяем старшую цифру (десятки) и распределяем ключи по своим спискам:

ключ 90 в список для 9, ключ 11 в список для 1, ключ 02 в список для 0 и т.д.

| 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7 | 8  | 9  |
|----|----|----|----|----|----|----|---|----|----|
| 02 | 11 | 27 | 33 | 45 | 56 | 66 |   | 83 | 90 |
| 08 | 16 |    | 37 |    |    |    |   |    | 99 |
| 09 | 17 |    |    |    |    |    |   |    |    |

Объединение этих списков дает отсортированный набор:

02, 08, 09, 11, 16, 17, 27, 33, 37, 45, 56, 66, 83, 90, 99

Для программной реализации необходимо объявить десятиэлементный массив и ссылочный тип для организации списков. Удобно вместе с началом каждого списка хранить указатель на его последний элемент, что упрощает как добавление новых элементов в конец списка, так и объединение отдельных списков. Внешний цикл алгоритма повторяется  $k$  раз (разрядность ключа). Каждый раз внутри этого цикла необходимо:

- обнулить все 20 указателей
- циклом по числу элементов ( $m$ ) распределить элементы по своим спискам, выделяя в ключе необходимую цифру
- циклом по 10 объединить списки в новый набор данных

В целом, поразрядная сортировка не эффективна при малых объемах входных данных, но чем больше этот объем, тем выше эффективность метода. Трудоемкость пропорциональна объему входных данных, что лучше чем у быстрой сортировки. Платой за это являются дополнительные затраты памяти для поддержки вспомогательных списков.

Интересно отметить, что данный метод очень хорошо подходит для современных архитектур вычислительных систем с конвейерной обработкой данных и использованием нескольких процессоров.

### **3.4. Практическое задание**

Реализовать специальные методы сортировки:

- Простейшую карманную с использованием второго массива и без него
- Обобщенную карманную сортировку с повторяющимися ключами и дополнительными списками
- Поразрядную сортировку

Все методы реализуются как подпрограммы и поэтапно добавляются в главную программу.

### **3.5. Контрольные вопросы по теме**

1. В чем состоят особенности специальных методов сортировки?
2. Какие условия должны выполняться для применимости простейшего метода карманной сортировки?
3. Как в простейшем случае выполняется карманная сортировка?
4. Как программно реализуется простейшая карманная сортировка?
5. Как реализуется простейшая карманная сортировка без использования второго массива?
6. Приведите практический пример простейшей карманной сортировки массива “на месте”.
7. Как программно реализуется простейшая карманная сортировка с использованием только одного массива?
8. Какое обобщение имеет простейшая карманная сортировка?
9. Какие структуры данных необходимы для реализации карманной сортировки с повторяющимися ключами?
10. Как выполняется карманная сортировка для случая повторяющихся ключей?

11. Приведите практический пример использования карманной сортировки с повторяющимися ключами.
12. Какие достоинства и недостатки имеет карманная сортировка массивов?
13. Какие условия необходимы для применения метода поразрядной сортировки?
14. В чем состоит смысл метода поразрядной сортировки массивов?
15. Какие структуры данных необходимы для реализации метода поразрядной сортировки?
16. Приведите практический пример использования поразрядной сортировки.
17. Какие шаги необходимы для реализации метода поразрядной сортировки?
18. Что можно сказать об эффективности метода поразрядной сортировки?
19. Какие достоинства и недостатки имеет поразрядная сортировка?
20. Как программно выполняется поразрядная сортировка?

## Тема 4. Поиск с использованием хеш-функций

### 4.1. Основные понятия

Пусть имеется набор из  $n$  элементов  $a_1, a_2, a_3, \dots, a_n$  с некоторыми ключами (как и раньше, для простоты будем считать, что сам элемент совпадает с его ключом). Требуется этот набор организовать в виде некоторой структуры данных с возможностью многократного поиска в нем элементов с заданным ключом. Эта задача может решаться различными способами:

- если набор элементов никак не упорядочен, то поиск выполняется прямым сравнением всех элементов в массиве или списке с трудоемкостью  $O(n)$
- если элементы упорядочены в массиве или в дереве поиска, поиск более эффективно выполняется как двоичный, с трудоемкостью  $O(\log_2 n)$

Возникает вопрос: существуют ли еще более эффективные методы поиска? Оказывается, при выполнении некоторых дополнительных условий можно организовать исходный набор ключей в виде специальной структуры данных, называемой **хеш-таблицей**, поиск в которой **ЛЮБОГО** элемента в идеале

выполняется за **ОДНО** сравнение и **НЕ** зависит от размерности входного набора. Другими словами, трудоемкость такого метода поиска, называемого **хеш-поиском**, пропорциональна  $O(1)$ , что является абсолютным рекордом!

Метод хеш-поиска заключается в следующем. Исходные элементы  $a_1, a_2, a_3, \dots, a_n$  распределяются некоторым специальным образом по ячейкам массива. Пока будем считать, что число ячеек массива  $m > n$ . Идеальным поиском можно считать такой, когда **по любому входному ключу сразу вычисляется индекс ячейки с этим ключом**, без проверки содержимого остальных ячеек. Для вычисления индекса ячейки по входному ключу используется специальная функция, называемая **хеш-функцией**. Эта функция ставит в соответствие каждому ключу индекс ячейки массива, где должен располагаться элемент с этим ключом:

$$h(a_i) = j, \quad j = (1, m);$$

Массив, заполненный элементами исходного набора в порядке, определяемом хеш-функцией, называется хеш-таблицей. Отсюда следует, что решение задачи поиска данным методом во многом зависит от используемой хеш-функции. Предложено довольно много различных хеш-функций. Самой простой, но не самой лучшей хеш-функцией является функция взятия остатка от деления ключа нацело на  $m$ :

$$h(a_i) = (a_i \bmod m) + 1;$$

Ясно, что каждое значение этой функции лежит в пределах от 1 до  $m$  и может приниматься в качестве индекса ячейки массива.

Принято считать, что хорошей является хеш-функция, которая удовлетворяет следующим условиям:

- функция должна быть очень простой с вычислительной точки зрения
- функция должна распределять ключи в хеш-таблице как можно более равномерно

Использование данного метода включает два этапа:

- **построение** хеш-таблицы для заданного набора ключей с помощью выбранной хеш-функции, т.е. определение для каждого ключа его местоположения в таблице
- **использование** построенной таблицы для поиска элементов с помощью той же самой хеш-функции

Рассмотрим два примера с целыми и строковыми ключами.

**Пример 1.** Пусть задан набор из 8 целочисленных ключей:

35, 19, 07, 14, 26, 40, 51, 72.

Требуется распределить эти ключи в массиве из 10 ячеек с помощью простейшей хеш-функции.

Для этого каждый ключ делим нацело на 10 и используем остаток в качестве индекса размещения ключа в массиве:

$35 \bmod 10 = 5$ , индекс размещения ключа 35 равен 5  
 $19 \bmod 10 = 9$ , индекс размещения ключа 19 равен 9  
 $07 \bmod 10 = 7$ , индекс размещения ключа 07 равен 7  
 $14 \bmod 10 = 4$ , индекс размещения ключа 14 равен 4  
 $26 \bmod 10 = 6$ , индекс размещения ключа 26 равен 6  
 $40 \bmod 10 = 0$ , индекс размещения ключа 40 равен 0  
 $51 \bmod 10 = 1$ , индекс размещения ключа 51 равен 1  
 $72 \bmod 10 = 2$ , индекс размещения ключа 72 равен 2

Получаем следующую хеш-таблицу:

| индекс | 1  | 2  | 3  | 4 | 5  | 6  | 7  | 8  | 9 | 10 |
|--------|----|----|----|---|----|----|----|----|---|----|
| ключ   | 40 | 51 | 72 |   | 14 | 35 | 26 | 07 |   | 19 |

Если требуется найти в этой хеш-таблице ключ со значением 26, то этот поиск выполняется ровно за одно сравнение: делим 26 на 10, берем остаток 6, входим в ячейку с индексом 6 и сравниваем находящееся там значение с заданным ключом.

**Пример 2.** Пусть ключи являются строковыми. В этом случае предварительно текстовый ключ надо преобразовать в числовой. Например, можно сложить ASCII-коды всех символов, входящих в этот текстовый ключ.

Например, если строковый ключ имеет значение END, то его целочисленный эквивалент будет равен сумме кодов всех трех символов:  $\text{ord}(E) + \text{ord}(N) + \text{ord}(D) = 69 + 78 + 68 = 215$

Тогда для четырех строковых ключей, являющихся служебными словами языка Паскаль, получим следующие значения простейшей хеш-функции, определяющие размещение этих ключей в десятиэлементной хеш-таблице:

$$h(\text{END}) = (215 \bmod 10) + 1 = 6$$

$$h(\text{VAR}) = (233 \bmod 10) + 1 = 4$$

$$h(\text{AND}) = (211 \bmod 10) + 1 = 2$$

$$h(\text{NIL}) = (227 \bmod 10) + 1 = 8$$

В результате для этих четырех строковых ключей получаем следующую хеш-таблицу:

| индекс | 1 | 2   | 3 | 4   | 5 | 6   | 7 | 8   | 9 | 10 |
|--------|---|-----|---|-----|---|-----|---|-----|---|----|
| ключ   |   | AND |   | VAR |   | END |   | NIL |   |    |

Поиск в этой таблице некоторого ключа выполняется очень просто: находится целочисленный эквивалент строкового ключа, вычисляется значение хеш-функции и сравнивается содержимое полученной ячейки с заданным ключом. Например,  $h(\text{VAR}) = 4$ , сравниваем содержимое ячейки 4 с ключом VAR, фиксируем совпадение и завершаем поиск с признаком успеха.

Приведенные выше примеры носят несколько искусственный характер, поскольку они описывают идеальный случай, когда хеш-функция для **всех различных ключей** дает **РАЗЛИЧНЫЕ значения индексов в массиве**. В этом случае каждый ключ имеет свое уникальное расположение в массиве, не конфликтуя с другими ключами. Подобная ситуация возможна, если исходный набор ключей **известен заранее** и после построения хеш-таблицы **не изменяется**, т.е. ключи НЕ добавляются и НЕ удаляются из хеш-таблицы. В этом случае за счет подбора хеш-функции и, возможно, небольшого изменения самих ключей можно построить **бесконфликтную** хеш-таблицу. Важным практическим примером такой ситуации является построение таблицы ключевых слов в программах-трансляторах с языков программирования. Здесь набор ключевых слов является постоянным, изменяясь только при изменении версии транслятора, а с другой стороны, обработка транслятором входного



текста на языке программирования требует многократного и очень быстрого распознавания в этом тексте ключевых слов языка.

К сожалению, идеальный случай возможен весьма редко, и ограничивать применение хеш-поиска только данным случаем было бы неразумно, учитывая выдающиеся потенциальные скоростные возможности метода. Поэтому были предложены различные **усовершенствования** хеш-поиска, существенно расширившие область его использования. Эти усовершенствования так или иначе связаны с **обработкой конфликтных ситуаций**, когда два **РАЗНЫХ** ключа претендуют на **ОДНО** и то же место в хеш-таблице, т.е. хеш-функция дает для этих разных ключей  $a_i$  и  $a_k$  одно и то же значение:

$$h(a_i) = h(a_k) = j$$

Например, в приведенном выше примере со строковыми ключами простая перестановка символов в ключе приводит к конфликту:

$$h(VAR) = h(RAV) = h(AVR) = (233 \bmod 10) + 1 = 4$$

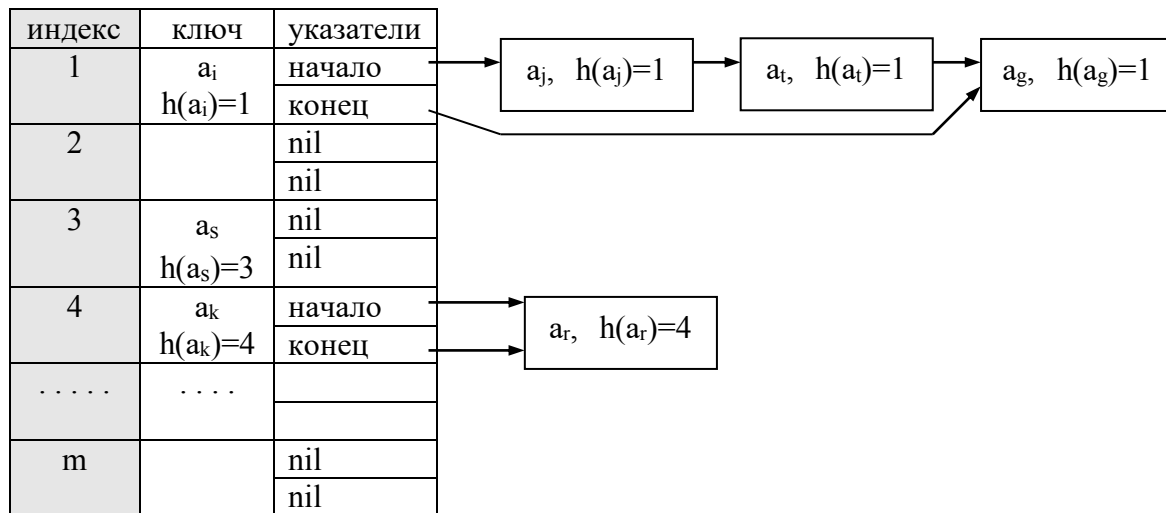
Для разрешения конфликтов были предложены разные методы, которые можно сгруппировать в две основные группы – **открытое хеширование** и **внутреннее хеширование** (необходимо отметить, что данная терминология не является общепринятой и допускает разночтения, поэтому в первую очередь надо обращать внимание на сущность метода разрешения конфликтов).

#### 4.2. Разрешение конфликтов: открытое хеширование

Пусть имеется  $n$  элементов  $a_1, a_2, a_3, \dots, a_n$ , на основе которых требуется построить хеш-таблицу, причем некоторые ключи **могут конфликтовать** между собой, претендуя на одну и ту же ячейку таблицы. Идея открытого хеширования совершенно прозрачна: **связать все элементы с одним и тем же значением хеш-функции во вспомогательный линейный список**. Данный метод иногда называют **методом цепочек**.

Обращаю внимание, что мы еще раз приходим к необходимости использования комбинированной структуры данных – массива указателей.

Хеш-таблица как массив записей должна хранить не только ключи элементов, но и по два указателя на начало и конец вспомогательного списка.



### Алгоритм построения хеш-таблицы:

- находим значение хеш-функции для очередного ключа и по этому значению как индексу входим в таблицу
- если данная клетка таблицы пустая, то записываем в нее соответствующий ключ
- если ячейка занята, то сравниваем хранящийся там ключ с заданным ключом:
  - если ключи совпадают, то каким-то образом обрабатываем повторный ключ (например, просто ничего не выполняем)
  - если ключи не совпадают, то добавляем новый ключ в конец списка

### Алгоритм поиска в построенной таблице:

- находим значение хеш-функции для искомого ключа и по этому значению как индексу входим в таблицу
- если ячейка с найденным индексом пустая, то поиск заканчивается неудачей
- если ячейка не пустая, то выполняем сравнение ключей:
  - если ключи совпадают, то поиск заканчивается за одно сравнение

- если ключи не совпадают, то организуем просмотр линейного вспомогательного списка с положительным или отрицательным результатом

**Пример.** Задано 10 целочисленных ключей, на основе которых надо построить хеш-таблицу размерности 5, используя для разрешения конфликтов метод открытого хеширования. Поскольку число исходных элементов  $n=10$  больше размерности таблицы ( $m=5$ ), то без использования вспомогательных списков таблицу построить нельзя. Набор входных ключей с соответствующими значениями хеш-функции приведены в следующей таблице (использована простейшая хеш-функция):

|                      |    |    |    |    |    |    |    |    |    |    |
|----------------------|----|----|----|----|----|----|----|----|----|----|
| ключ                 | 33 | 17 | 09 | 04 | 22 | 19 | 42 | 53 | 64 | 25 |
| значение хеш-функции | 4  | 3  | 5  | 5  | 3  | 5  | 3  | 4  | 5  | 1  |

Тогда хеш-таблица будет иметь следующий вид:

| индекс | ключ | указатели |
|--------|------|-----------|
| 1      | 25   | nil       |
|        |      | nil       |
| 2      |      | nil       |
|        |      | nil       |
| 3      | 17   | начало    |
|        |      | конец     |
| 4      | 33   | начало    |
|        |      | конец     |
| 5      | 09   | начало    |
|        |      | конец     |

Подсчитаем для данного примера среднее число сравнений, которые необходимо сделать для поиска любого из 10 исходных ключей:

- ключ 33 – одно сравнение, т.к. он непосредственно находится в ячейке таблицы
- ключи 17 и 09 – тоже по одному сравнению
- ключ 04 – два сравнения (в ячейке 5 находится ключ 09, идем по списку, совпадение на первом элементе)
- ключ 22 – 2 сравнения
- ключи 19 и 42 – по 3 сравнения (вторые элементы в списках)

- ключ 53 – 2 сравнения
- ключ 64 – 4 сравнения
- ключ 25 – 1 сравнение

Итого – 20 сравнений, т.е. в среднем 2 сравнения на один ключ.

Из примера видно, что для данного метода большое значение имеет **равномерность** распределения ключей по хеш-таблице, что гарантирует **короткие** вспомогательные списки и тем самым уменьшает число сравнений при поиске. Наихудшим является случай, когда для всех ключей хеш-функция дает одно и то же значение, и все элементы выстраиваются в один длинный линейный список.

Другим фактором, влияющим на эффективность открытого хеширования, является **размер** хеш-таблицы по отношению к числу входных данных. Если эти величины равны, то теоретически можно обойтись без линейных списков, если между ключами нет конфликтов. На практике рекомендуют выбирать размер хеш-таблицы равным  $n/2$ .

#### 4.3. Разрешение конфликтов: внутреннее хеширование

Пусть имеется  $n$  элементов  $a_1, a_2, a_3, \dots, a_n$ , на основе которых требуется построить хеш-таблицу, причем некоторые ключи могут конфликтовать между собой, претендуя на одну и ту же ячейку таблицы. Внутреннее хеширование для размещения конфликтующих ключей использует не вспомогательные списки, а **свободные ячейки** хеш-таблицы. Для этого **размер таблицы** обязательно должен быть **больше числа элементов** ( $m > n$ ).

Если при построении хеш-таблицы для очередного ключа хеш-функция определяет некоторую ячейку  $j$ , и эта ячейка **оказывается занятой** другим ключом, то предпринимается попытка найти в таблице **свободную** ячейку для размещения конфликтующего ключа. Если хотя бы одна свободная ячейка в таблице есть, то в ней и размещается конфликтующий ключ, в противном случае добавление считается невозможным.

Существуют различные правила поиска свободных мест для конфликтующих ключей. Необходимо подчеркнуть, что правило выбора свободной ячейки должно быть **одним и тем же** как при построении таблицы, так и при использовании ее для поиска.

Самое простое правило заключается в **последовательном круговом просмотре** всех ячеек, начиная с индекса  $j$ , т. е. ячеек с номерами  $j+1, j+2, \dots, m-1, m, 1, 2, \dots, j-1$ .

**Пример.** Пусть задано 6 целочисленных ключей 15, 17, 19, 35, 05, 28. Построим на их основе хеш-таблицу размерности 10 с помощью простейшего правила определения свободных ячеек.

$h(15) = 6$ , размещаем ключ 15 в ячейке 6

$h(17) = 8$ , размещаем ключ 17 в ячейке 8

$h(19) = 10$ , размещаем ключ 19 в ячейке 10

$h(35) = 6$ , но ячейка 6 уже занята, проверяем следующую: ячейка 7 свободна, размещаем ключ 35 в ячейке 7

$h(05) = 6$ , ячейка 6 занята, следующая ячейка 7 тоже занята, ячейка 8 – тоже, поэтому размещаем ключ 05 в ячейке 9

$h(28) = 9$ , ячейка 9 занята, ячейка 10 занята и является последней, проверяем ячейку 1 и размещаем ключ 28 именно в ней.

Получим следующую хеш-таблицу:

| индекс | 1            | 2 | 3 | 4 | 5 | 6            | 7            | 8            | 9            | 10            |
|--------|--------------|---|---|---|---|--------------|--------------|--------------|--------------|---------------|
| ключ   | 28 ( $h=9$ ) |   |   |   |   | 15 ( $h=6$ ) | 35 ( $h=6$ ) | 17 ( $h=8$ ) | 05 ( $h=6$ ) | 19 ( $h=10$ ) |

Для поиска любого из шести ключей в этой таблице потребуется в среднем  $(1+2+1+4+1+6 = 15)/6 = 2,5$  сравнений.

Для циклического вычисления индекса ключа  $a_k$  в простейшем случае можно использовать следующую простую формулу:

$$j = (h(a_k) + i) \bmod m + 1, \text{ где } i = 0, 1, 2, \dots, m-2$$

Здесь для целочисленных ключей  $h(a_k) = a_k$ .

Недостатком простейшей формулы определения свободных ячеек является эффект “кучкования” ключей с одинаковыми значениями хеш-функции: эти

ключи размещаются в **соседних** свободных ячейках, тогда как “хорошее” правило должно распределять эти ключи **равномерно** по таблице.

Поэтому более эффективным является правило, которое позволяет выполнять “перепрыгивание” через несколько соседних ячеек. Например, можно воспользоваться следующей функцией вычисления индексов размещения ключей:

$$j = ((h(a_k) + i^2) \bmod m) + 1, \text{ где } i = 0, 1, 2, \dots, m-2$$

Эта функция дает не постоянный шаг приращения индекса (+1), а переменный: +1, +4, +9, +16 и т.д.

#### **Алгоритм построения хеш-таблицы:**

- находим значение хеш-функции для очередного ключа и по этому значению как индексу входим в таблицу
- если данная клетка таблицы пустая, то записываем в нее соответствующий ключ
- если ячейка занята, то сравниваем хранящийся там ключ с заданным ключом:
  - если ключи совпадают, то каким-то образом обрабатываем повторный ключ (например, просто ничего не выполняем)
  - если ключи не совпадают, то с помощью выбранного правила организуем циклический поиск свободной ячейки, который завершается либо при обнаружении свободной ячейки, либо по достижению конца цикла, что говорит об отсутствии в таблице свободных ячеек и невозможности добавления нового ключа

#### **Алгоритм поиска:**

- находим значение хеш-функции для искомого ключа и по этому значению как индексу входим в таблицу
- если ячейка с найденным индексом пустая, то поиск заканчивается неудачей
- если ячейка не пустая, то выполняем сравнение ключей:
  - если ключи совпадают, то поиск заканчивается за одно сравнение

- если ключи не совпадают, то с помощью выбранного правила организуем циклический просмотр ячеек, который завершается либо при обнаружении свободной ячейки (поиск неудачен), либо по совпадению ключей (поиск удачен)

Эффективность внутреннего хеширования существенно зависит от наличия в хеш-таблице пустых ячеек, поэтому на практике идут на искусственное **увеличение размерности таблицы** на. 10-20% для обеспечения достаточного количества свободных клеток.

Многочисленные эксперименты показывают, что для заполненной на 50% таблицы (половина ячеек пустые) для поиска любого ключа в среднем требуется лишь 1,5 сравнения, причем это число не зависит от количества элементов. Это еще раз подтверждает высочайшую эффективность хеш-поиска!

В заключение дадим **общие рекомендации** по применению хеш-поиска.

1. Наиболее целесообразно использовать данный метод для постоянного и не меняющегося набора ключей, т.к. в этом случае за счет подбора хеш-функции и самих ключей можно построить бесконфликтную таблицу и достичь максимально возможной скорости поиска – ровно 1 сравнение для любого ключа независимо от их количества
2. Если набор ключей может меняться и заранее неизвестен, то важным становится выбор хеш-функции для равномерного распределения ключей по таблице. Одной из лучших хеш-функций считается следующая: исходный целочисленный ключ возводится в квадрат, преобразуется в двоичное представление в виде набора битов и из середины этого набора извлекается необходимое число битов в соответствии с размерностью таблицы (например, для таблицы размерности 1024 необходимо извлечь 10 битов)
3. При использовании открытого хеширования рекомендуется размер таблицы выбирать равным  $n/2$ , что в среднем должно обеспечивать короткие вспомогательные списки (1-2 элемента), которые могут просматриваться только последовательно

4. При использовании внутреннего хеширования рекомендуется размер таблицы выбирать равным  $1,2 \cdot n$  для обеспечения достаточного количества свободных ячеек.
5. В обоих случаях, если в процессе использования хеш-поиска происходит добавление в таблицу новых элементов и из-за этого нарушаются приведенные выше рекомендации, целесообразно выполнить **реструктуризацию** хеш-таблицы, добавив в базовый массив необходимое число ячеек
6. Некоторые сложности возникают при удалении элементов из хеш-таблицы, особенно – при использовании внутреннего хеширования, т.к. при этом за счет появления “незапланированных” пустых ячеек может нарушиться работа алгоритма поиска
7. Ну и, наконец – ложка дегтя в бочке меда: метод **совершенно непригоден** для обработки **упорядоченных** наборов.

#### 4.4. Практические задания

**Задание 1.** Построить бесконфликтную хеш-таблицу для заданного набора текстовых ключей. Число ключей (и размер таблицы) равен 10. В качестве ключей взять 10 любых служебных слов языка Паскаль. Для преобразования текстовых ключей в числовые значения использовать суммирование кодов символов текстового ключа: код (End) = код (E) + код (n) + код (d). Преобразование числового кода ключа в значение индекса выполнить с помощью простейшей хеш-функции, которая берет остаток от целочисленного деления кода на размер хеш-таблицы (в задании – 10).

Для подсчета индексов ключей написать вспомогательную программу, которая в диалоге многократно запрашивает исходный текстовый ключ (служебное слово языка Паскаль) и подсчитывает значение хеш-функции. Программа должна вычислять длину очередного текстового ключа с помощью функции Length, в цикле вычислять сумму кодов всех символов ключа и брать остаток от деления этой суммы на 10.



Если некоторые исходные ключи будут конфликтовать друг с другом, можно изменить исходное слово, например – сменить регистр начальной буквы или всех букв в слове, полностью заменить слово на близкое по значению (End на Stop и т.д.), ввести какие-либо спецсимволы или придумать другие способы

После подбора неконфликтующих ключей написать основную программу, которая должна:

- ввести подобранные ключи и расположить их в ячейках хеш-таблицы в соответствии со значением хеш-функции
- вывести хеш-таблицу на экран
- организовать циклический поиск разных ключей, как имеющихся в таблице (с выводом местоположения), так и отсутствующих:  
вычислить для ключа значение хеш-функции и сравнить содержимое соответствующей ячейки таблицы с исходным ключом

**Задание 2.** Реализовать метод внутреннего хеширования. Исходные ключи – любые слова (например – фамилии). Размер хеш-таблицы должен задаваться в программе с помощью константы  $m$ . Хеш-функция – такая же, что и в задании 1, но делить надо на константу  $m$ . В случае возникновения конфликта при попытке размещения в таблице нового ключа, для него ищется первое свободное по порядку место по формуле

$$j = ((h(\text{ключ}) + i) \bmod m) + 1, \text{ где } i = 0, 1, 2, \dots, m-2$$

Программа должна выполнять следующие действия:

- добавление нового ключа в таблицу с подсчетом сделанных при этом сравнений
- поиск заданного ключа в таблице с подсчетом сделанных при этом сравнений
- вывод текущего состояния таблицы на экран

После отладки программы необходимо выполнить ее для разных соотношений числа исходных ключей и размерности таблицы: взять 10 ключей и разместить их поочередно в таблице размерности 11, 13 и 17. Для каждого случая найти суммарное число сравнений, необходимое для размещения

ключей и их поиска. Сделать вывод о влиянии количества пустых мест в таблице на эффективность поиска.

**Задание 3.** Реализовать метод открытого хеширования. Исходные ключи – любые слова (например – фамилии). Размер хеш-таблицы должен задаваться в программе с помощью константы  $m$ . Хеш-функция – такая же, что и в задании 1, но делить надо на константу  $m$ . В случае возникновения конфликта при попытке размещения в таблице нового ключа этот ключ добавляется в конец вспомогательного списка. Это требует включения в каждую ячейку хеш-таблицы двух указателей на начало и конец вспомогательного списка.

Программа должна выполнять следующие действия:

- добавление нового ключа в таблицу с подсчетом сделанных при этом сравнений
- поиск заданного ключа в таблице с подсчетом сделанных при этом сравнений
- вывод текущего состояния таблицы на экран
- удаление заданного ключа из таблицы

Алгоритм удаления:

- вычислить хеш-функцию и организовать поиск удаляемого элемента в таблице
- если удаляемый элемент найден в ячейке таблицы, то эта ячейка либо становится пустой (если связанный с ней список пуст), либо в нее записывается значение из первого элемента списка с соответствующим изменением указателей
- если удаляемый элемент найден в списке, то производится его удаление с изменением указателей

После отладки программы необходимо выполнить ее для разных соотношений числа исходных ключей и размерности таблицы: взять 20 ключей и разместить их поочередно в таблице размерности 9, 17 и 23. Для каждого случая найти суммарное число сравнений, необходимое для размещения

ключей и их поиска. Сделать вывод о влиянии размерности таблицы на эффективность поиска.

#### **4.5. Контрольные вопросы по теме**

1. В чем заключается метод хеш-поиска?
2. Для чего используется хеш-функция и какие к ней предъявляются требования?
3. Что такое хеш-таблица и как она используется?
4. Как по трудоемкости соотносятся между собой основные методы поиска (полный перебор, двоичный поиск, хеш-поиск)?
5. Как с помощью простейшей хеш-функции находится расположение в таблице строковых ключей?
6. Какие проблемы могут возникать при построении хеш-таблиц с произвольными наборами ключей?
7. В каких ситуациях можно построить бесконфликтную хеш-таблицу?
8. Где на практике и почему можно использовать бесконфликтные хеш-таблицы?
9. Что такое открытое хеширование и для чего оно применяется?
10. Какие структуры данных используются для реализации открытого хеширования?
11. Какие шаги выполняет алгоритм построения хеш-таблицы при открытом хешировании?
12. Какие шаги выполняет алгоритм поиска в хеш-таблице при открытом хешировании?
13. Какие проблемы могут возникать при использовании открытого хеширования?
14. Как влияет размер хеш-таблицы на эффективность открытого хеширования?
15. Что такое внутреннее хеширование и для чего оно применяется?

16. Какие правила можно использовать для поиска свободных ячеек при внутреннем хешировании?
17. Какие шаги выполняет алгоритм построения хеш-таблицы при внутреннем хешировании?
18. Какие шаги выполняет алгоритм поиска в хеш-таблице при внутреннем хешировании?
19. Как влияет размер хеш-таблицы на эффективность внутреннего хеширования?
20. В каких задачах НЕ следует применять метод хеш-поиска?

## **Тема 5. Внешний поиск и внешняя сортировка**

### **5.1. Особенности обработки больших наборов данных**

Задачи внешнего поиска и сортировки возникают в тех случаях, когда обрабатываемый набор данных является слишком большим и для его размещения в оперативной памяти (ОП) нет достаточного места. Подобные задачи всегда встречаются при использовании баз данных с большими объемами информации. В этом случае в ОП считывается **только часть** данных, а остальные данные хранятся в файлах на диске.

Решение подобных задач неизбежно связано с учетом особенностей взаимодействия ОП и внешней памяти. Главное их отличие – **время доступа**. Поскольку доступ к внешней памяти выполняется **значительно медленнее**, то главным критерием при разработке алгоритмов становится не количество элементарных операций с расположенными в ОП данными, а **число обращений** к внешней памяти. Методы внешнего поиска и сортировки должны быть такими, чтобы время обращения к внешней памяти было как можно меньше.

В свою очередь, это требует четкого понимания особенностей хранения информации во внешней памяти. Как известно, для хранения данных на диске создаются элементарные физические единицы (секторы), которые на логическом уровне объединяются в более крупные кластеры. Размер кластера

определяется типом операционной системы. За одно обращение к диску считывается содержимое сразу всего кластера. Это связано с тем, что при таком подходе минимизируется время поиска необходимых секторов на диске, включающее в себя время на подвод считывающей головки к нужной дорожке и время на поворот диска к нужному сектору. Считываемые с диска наборы байтов помещаются в буферную область памяти. Отсюда можно сделать вывод, что методы поиска и сортировки должны быть построены таким образом, чтобы можно было обрабатывать **сразу целую группу элементов**. Нельзя читать данные из внешней памяти в ОП по одному элементу – должна быть считана целая группа логически связанных элементов, которая потом обрабатывается алгоритмом и, возможно, приводит к необходимости чтения другой группы элементов.

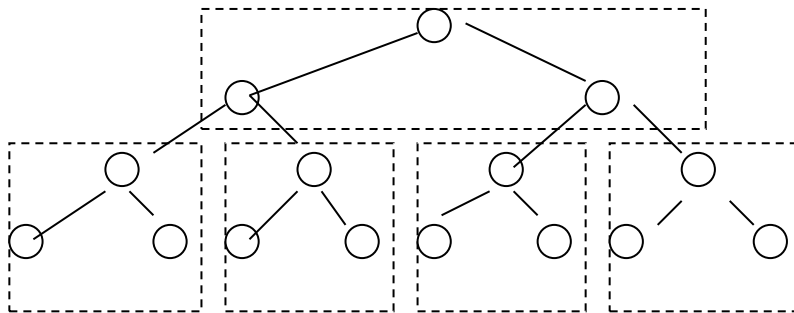
На этих принципах построен ряд методов внешнего поиска и сортировки. Среди них одним из наиболее известных методов поиска является метод Б-деревьев (B-tree).

## 5.2. Организация внешнего поиска с помощью Б-деревьев.

Данный метод использует понятие двоичного дерева поиска, но модифицирует это понятие таким образом, что дерево получает **страничную организацию**. Поэтому Б-деревья часто называют деревьями со страничной организацией.

Предположим, что мы хотим организовать в оперативной памяти дерево поиска с очень большим числом вершин (миллионы и более). Пусть при этом свободной ОП не достаточно для одновременного хранения сразу всех вершин. Часть (и при том весьма существенная) вершин должна оставаться на диске. Возникает задача такой организации дерева, чтобы можно было считывать и обрабатывать сразу группу вершин. Такая группа вершин получила название **“страница”**. Тем самым, все сверхбольшое дерево поиска разбивается на ряд страниц и чтение вершин с диска выполняется **постранично**. Это позволяет существенно уменьшить количество обращений к внешней памяти.

В качестве примера рассмотрим обычное дерево поиска из 15 вершин. Пусть для простоты оно является идеально сбалансированным. Выделим в нем 5 элементарных поддеревьев по 3 вершины в каждом и логически объединим эти вершины вместе в виде страниц. Если в дереве необходимо найти какую-либо терминальную вершину, и вершины считываются в ОП по одной, то для этого в обычном дереве потребуется 4 обращения к внешней памяти. А в дереве со страничной организацией потребуется считать из внешней памяти лишь 2 страницы – корневую и одну из терминальных. Даже в этом простейшем примере страничная организация дерева позволяет уменьшить число обращений к диску в два раза. Для сверхбольших деревьев этот эффект становится еще более заметным.



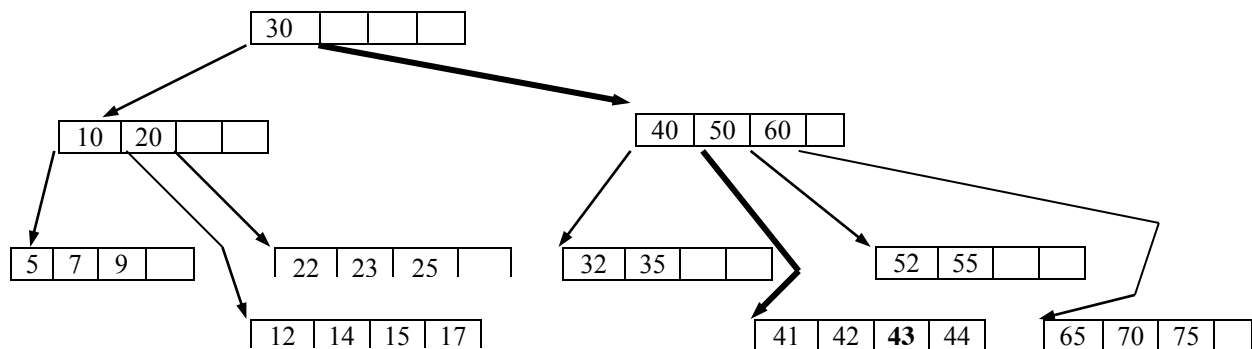
Для эффективной страничной организации сверхбольшого дерева поиска очень важным условием является **равномерность его роста**, максимальное соответствие идеальной сбалансированности. К сожалению, условие идеальной сбалансированности приводит к очень сложным алгоритмам, поэтому Байером и было введено понятие дерева со страничной организацией (Б-дерево).

**Б-дерево порядка  $m$  со страничной организацией** - это структура данных, удовлетворяющая следующим условиям:

- весь набор из  $n$  элементов разбивается на отдельные страницы, причем каждая страница может содержать от  $m$  до  $2m$  вершин, за исключением корневой страницы, для которой число вершин может изменяться от 1 до  $2m$

- каждая страница является либо терминальной (потомков нет), либо имеет ровно на одного потомка больше по сравнению с текущим числом вершин на этой странице
- все терминальные страницы находятся на одном уровне

**Пример.** Рассмотрим простейшее Б-дерево порядка  $m=2$  с ключами целого типа. В соответствии с приведенными выше правилами, каждая страница такого дерева содержит от 2 до 4 вершин, а корневая – от 1 до 4. Каждая нетерминальная страница может иметь от 3 до 5 потомков. Пример дерева – на рисунке (к сожалению, терминальные страницы приходится располагать на разных уровнях).



Поскольку Б-дерево является обобщением классического дерева поиска, то оно обладает следующими свойствами:

- на каждой странице элементы **упорядочены** по возрастанию ключей
- каждая страница–потомок содержит ключи в **строго определенном диапазоне**, который задается родительскими ключами

Последнее свойство является обобщением основного правила дерева поиска. Например, страница с ключами 40, 50 и 60 имеет четырех потомков, причем самый левый потомок содержит ключи меньше 40 (но больше 30), более правая страница содержит ключи в диапазоне от 40 до 50, еще более правая – в диапазоне от 50 до 60, а самая правая – больше 60.

Эффективность использования Б-дерева для поиска ключей в сверхбольших наборах данных можно оценить следующим образом.. Если Б-дерево имеет порядок  $m$ , а число элементов –  $n$ , то самым наихудшим случаем

будет случай, когда на каждой странице находится минимально возможное число элементов, а именно -  $m$ . В этом случае высота Б-дерева определяется величиной  $\log_m n$ , а именно высота и определяет количество обращений к внешней памяти.

Например, если число элементов в наборе данных  $n = 1$  миллион, а порядок Б-дерева  $m = 100$ , то  $\log_{100} 1000000 = 3$ , т.е. максимум может потребоваться 3 обращения к внешней памяти. С другой стороны, даже в этом наихудшем случае оперативная память используется достаточно эффективно (примерно половина памяти от заявленного объема).

### 5.3. Б-дерево как структура данных

Базовым элементом Б-дерева является **страница**, поскольку именно она содержит всю основную информацию. Поэтому, прежде всего, необходимо описать структуру страницы Б-дерева.

Каждая страница должна содержать следующие данные:

- текущее количество элементов на странице (оно изменяется от  $m$  до  $2m$ )
- указатель на страницу, являющуюся самым левым потомком данной страницы
- основной массив элементов страницы, размерность массива  $2m$ , каждый элемент – это запись со следующими полями:
  - ключ некоторого типа
  - указатель на страницу, являющуюся потомком текущей страницы и содержащую ключи, большие ключа данного элемента
  - информационная часть (как некоторая структура или указатель на область памяти)

Соответствующие описания типов можно сделать следующим образом:

```

type pPage = ^Page; {ссылочный тип для адресации страниц}
 TItem = record {описание структуры элемента массива}
 key : integer; {ключ вершины дерева}
 cPage : pPage; {указатель на страницу-потомка}
 end;

```



```

inf : <описание информационной части>;

end;

Page = record {описание структуры страницы}

 nPage : word; {число вершин на странице}

 leftPage : pPage; {указатель на самого левого потомка}

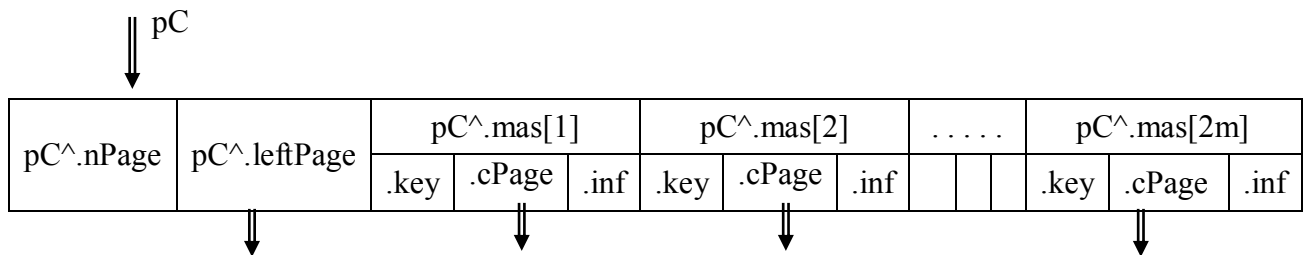
 mas : array [1 .. 2m] of TItem; {основной массив}

end;

```

Из переменных достаточно ввести два указателя: на корневую страницу (pRoot) и текущую обрабатываемую в данный момент страницу (pCurrent).

Схематично структуру страницы можно представить следующим образом (для краткости указатель pCurrent заменен идентификатором pC):



Видно, что Б-дерево является достаточно сложной комбинированной структурой, представляя собой набор связанных указателями записей. Для использования Б-дерева необходим стандартный набор операций: поиск заданного элемента, добавление вершины, удаление вершины.

#### 5.4. Поиск элемента в Б-дереве.

Пусть имеется Б-дерево порядка  $m$ , в котором требуется найти вершину с ключом  $x$ . Алгоритм поиска:

- считываем в ОП корневую страницу и организуем на этой странице поиск заданного элемента; если порядок Б-дерева небольшой, то можно применить обычный поиск прямым перебором; для деревьев большого порядка можно использовать метод двоичного поиска в упорядоченном массиве

- если заданный элемент на корневой странице найден, то обрабатываем его и завершаем поиск
- если заданного элемента на корневой странице нет, то возможно появление одной из трех следующих ситуаций:
  - искомый элемент меньше самого левого ключа ( $x < pRoot^.mas[1].key$ ); в этом случае поиск надо продолжить на странице, определяемой самой левой ссылкой ( $pRoot^.leftPage$ ), для чего в ОП загружается вторая страница и поиск на ней повторяется описанным выше образом
  - искомый элемент находится между двумя элементами ( $pRoot^.mas[j].key < x < pRoot^.mas[j+1].key$ ); в этом случае поиск надо продолжить на странице, которая определяется ссылкой  $pRoot^.mas[j].cPage$
  - искомый элемент больше самого правого элемента ( $x > pRoot^.mas[nPage].key$ ); поиск продолжаем на странице, определяемой ссылкой в последнем элементе массива ( $pRoot^.mas[nPage].cPage$ )

Если при этом какая-либо ссылка на страницу оказывается пустой, то это является признаком того, что искомого элемента в Б-дереве нет.

Для ускорения обработки Б-деревьев рекомендуется корневую страницу постоянно держать в ОП. Поскольку одновременно в ОП может находиться несколько страниц, то это накладывает ограничения на размер страницы и на порядок Б-дерева.

Как обычно, поиск можно реализовать как рекурсивно, так и итеративно (с помощью цикла). Программная реализация приводится как составляющая процедуры добавления новой вершины в Б-дерево.

**Например,** для рассмотренного выше Б-дерева поиск элемента с ключом 43 будет выполнен следующим образом:

- заходим на корневую страницу и устанавливаем, что ключ 43 больше всех ключей на корневой странице
- загружаем в ОП новую страницу, определяемую самой правой ссылкой на корневой странице

- организуем поиск ключа 43 среди элементов новой страницы (40, 50, 60) и фиксируем ситуацию расположения ключа 43 между ключами 40 и 50
- загружаем в ОП третью страницу, определяемую ссылкой у элемента 40 и организуем ее просмотр
- на этой странице (41, 42, 43, 44) находим искомый элемент

Наоборот, поиск ключа 45 приведет на последнем шаге к необходимости перехода на страницу, являющуюся самым правым потомком третьей страницы, но соответствующая ссылка является пустой и поиск заканчивается неудачно.

### 5.5. Добавление вершины в Б-дерево

Пусть имеется Б-дерево порядка  $m$ , в которое требуется добавить новый элемент с ключом  $x$ . Как обычно, прежде всего организуется поиск элемента с этим ключом.

Пусть такого элемента в дереве нет. Признаком данной ситуации является попытка загрузки новой страницы с пустой ссылкой. Именно в данный момент и необходимо начать добавление вершины на текущую терминальную страницу.

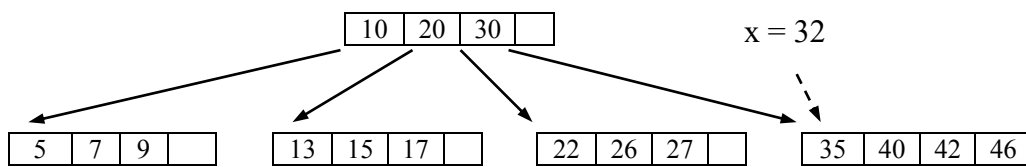
При этом возможны следующие две ситуации:

- текущая страница, в которую должен быть вставлен новый элемент, **заполнена не полностью** ( $pCurrent^{nPage} < 2m$ ); в этом случае новый элемент **просто вставляется** в данную страницу в нужное место, с увеличением счетчика числа элементов на странице
- пусть текущая страница **заполнена полностью**, т.е.  $pCurrent^{nPage} = 2m$ ; добавление элемента на полностью заполненную страницу реализуется за счет **разделения** этой страницы **на две страницы**; создается новая страница, и после этого старая и новые страницы заполняются равномерно:
  - левые  $m$  элементов передаются на новую страницу
  - правые  $m$  элементов остаются на старой странице

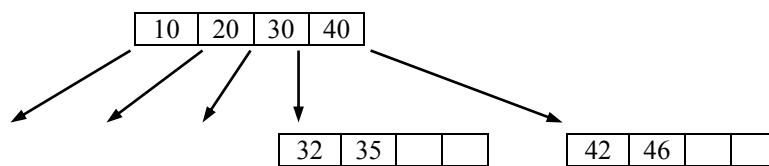
- центральный (серединный) элемент передается на родительскую страницу

При этом вытолкнутый наверх элемент вставляется на родительской странице в нужное место, если для этого есть свободное место. В противном случае, происходит **расщепление родительской страницы** на две, с **выталкиванием** центрального элемента еще выше и т.д. Такой процесс расщепления и выталкивания может дойти до корневой страницы. Если корневая страница заполнена полностью, то она расщепляется на две, с выталкиванием центрального элемента наверх, т.е. **создается новая корневая страница**, которая будет содержать единственный вытолкнутый снизу элемент. В этом случае высота Б-дерева **увеличивается** на 1.

**Пример.** Пусть в Б-дерево порядка 2 надо добавить вершину с ключом  $x = 32$

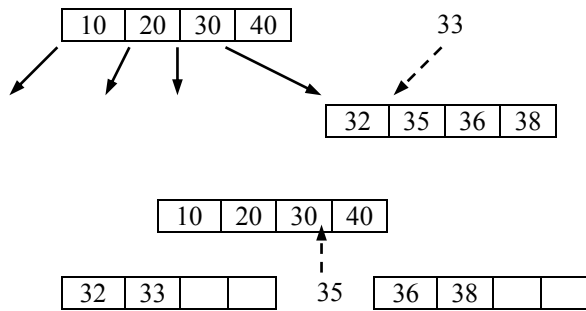


Страница, на которую надо добавить ключ 32, заполнена полностью, поэтому создается новая страница с элементами 32 и 35, на старой остаются ключи 42 и 46, а элемент 40 выталкивается на родительскую страницу, занимая в ней крайнее правое место

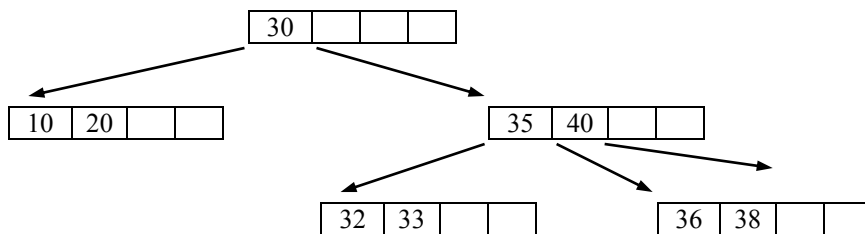


Такой алгоритм добавления сохраняет структуру Б-дерева, т.к. синхронно на 1 увеличивается число элементов на родительской странице и число ее страниц-потомков.

Если в данном примере добавить еще элементы 36 и 38, то при попытке добавления элемента 33 опять придем к необходимости расщепления страницы:



Поскольку корневая страница заполнена полностью, происходит ее расщепление на две страницы, выталкивание наверх элемента 30 и создание новой корневой страницы



Рассмотрим вопрос программной реализации алгоритма добавления. Поскольку добавление элемента на терминальной странице может привести к необходимости добавления и на ее родительской странице, а там, в свою очередь – на своей родительской странице и т.д., вплоть до корневой, в процессе поиска надо **запоминать пройденный путь**, т.е. пройденные на каждой странице точки разветвления. Как обычно, подобное запоминание можно реализовать либо неявно с помощью механизма **рекурсии**, либо явно с помощью **стековой структуры**.

Для рекурсивной реализации надо ввести подпрограмму Add, которая должна использовать три параметра:

- адрес новой текущей страницы, используемый как в процессе движения от корневой страницы к терминальной, так и обратно; поскольку этот адрес автоматически должен запоминаться и восстанавливаться, ссылочный параметр надо объявить как **параметр-значение**

- **выходной** логический параметр, который определяет факт расщепления текущей страницы и, следовательно, необходимость добавления выталкиваемого элемента на родительскую страницу
- **выходной** параметр-переменная, через который на родительскую страницу передается сам выталкиваемый элемент.

Схематично подпрограмма Add описывается следующим образом

```

procedure Add (pCurrent : pPage; var IsUp : boolean; var Item : TItem);
begin
 if pCurrent = nil then {элемента в дереве нет}
 begin
 “формирование полей нового элемента Item”;
 IsUp := true;
 end
 else begin {продолжаем поиск на странице pCurrent}
 “загрузка новой страницы”;
 “поиск на странице элемента с заданным ключом”;
 if “элемент найден” then “обработка элемента”
 else begin
 “определение адреса страницы для продолжения поиска”;
 Add (“адрес страницы”, IsUp, Item);
 if IsUp then {добавить элемент на текущую страницу }
 if pCurrent^.nPage < 2m then
 begin
 pCurrent^.nPage := pCurrent^.nPage + 1;
 IsUp := false;
 “добавление элемента Item в страничный массив”
 end
 else begin
 “создание новой страницы”;
 “формирование полей новой страницы”;

```

“корректировка полей старой страницы”;

“формирование выталкиваемого вверх элемента”

**end;**

**end;**

**end;**

**end;**

Как обычно, начальный вызов подпрограммы **Add** выполняется из главной программы, причем после возврата обратно в главную программу, возможно, придется создавать новую корневую страницу:

**begin**

**Add** (pRoot, IsUp, Item);

**if** IsUp **then**

**begin**

pTemp := pRoot; **New**(pRoot);

pRoot^.nPage := 1; pRoot^.LeftPage := pTemp;

pRoot^.mas[1] := Item;

**end;**

**end;**

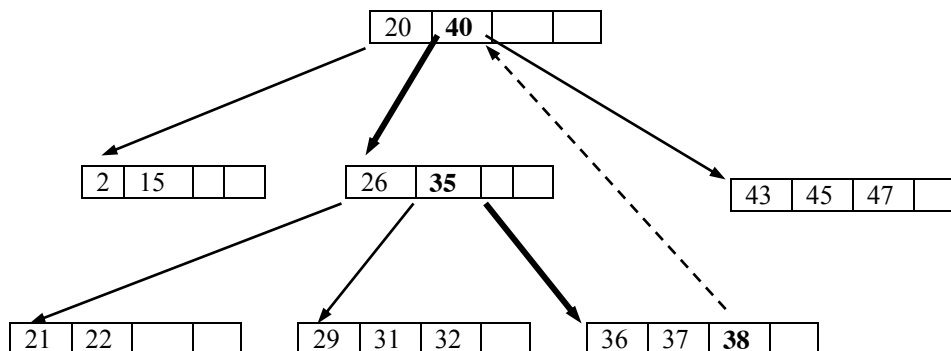
## 5.6. Удаление вершины из Б-дерева

Пусть имеется Б-дерево порядка  $m$ , из которого требуется удалить элемент с ключом  $x$ . Прежде всего, организуется поиск удаляемого элемента последовательным просмотром страниц, начиная с корневой. Пусть удаляемый элемент найден на некоторой странице. При этом возможны следующие две ситуации:

- если найденная страница является **терминальной**, то элемент просто удаляется с нее с последующей проверкой количества оставшихся на этой странице элементов; если после удаления на странице остается **слишком мало** элементов, предпринимаются **некоторые дополнительные действия**, описываемые немного позже

- если страница **нетерминальная**, то на место удаляемого элемента надо поставить **элемент-заменитель**, который находится также, как и в обычном дереве: входим в левое (правое) поддереву и спускаемся как можно ниже, придерживаясь максимально правых (левых) поддеревьев; после этого надо проверить страницу, с которой был взят элемент-заменитель и при необходимости предпринять **корректирующие действия** для восстановления структуры Б-дерева

**Например**, пусть в простейшем Б-дереве порядка 2 на рисунке ниже требуется удалить элемент с ключом  $x = 40$ . Левое поддерево для ключа 40 определяется ссылкой, связанной с его левым соседом, т.е. элементом 20. Просматриваем новую страницу (26, 35), выбираем самый правый элемент 35 и по его ссылке выходим на страницу (36, 37, 38). На этой странице крайний правый элемент 38 не имеет потомка и поэтому именно он должен быть элементом-заменителем. Ясно, что он является ближайшим меньшим элементом по отношению к удаляемому. Путь к элементу-заменителю показан жирной линией.



В обоих случаях после удаления элемента с некоторой страницы надо эту страницу **проверить на число оставшихся элементов**. По правилам построения Б-деревьев на каждой странице (кроме корневой) не может быть меньше  $m$  элементов. Если это условие после удаления не выполняется, т.е. на странице остается лишь  $m-1$  элемент, необходима ее **корректировка** за счет привлечения одной из соседних страниц. Для этого с помощью родительской страницы определяется **адрес соседней страницы** и выполняется ее **загрузка** в



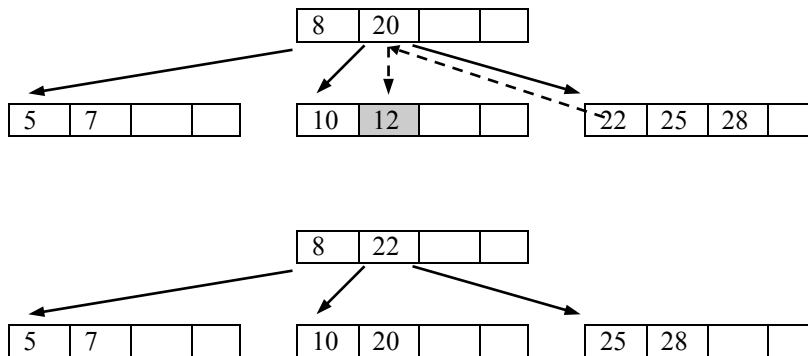
ОП. Совместная обработка двух соседних страниц выполняется следующим образом.

1. Если на соседней странице имеется **более  $m$**  элементов (хотя бы  $m+1$ ), то выполняется **перераспределение** элементов: с более “богатой” соседней страницы некоторое число элементов передается на “бедную” текущую страницу. В простейшем случае достаточно передать один элемент, но на практике чаще всего передается столько элементов, чтобы на этих двух страницах их было **почти поровну**. Здесь есть один тонкий момент – элементы с одной страницы на другую передаются не прямо, а **через родительскую страницу**, т.е. происходит как бы **перетекание** элементов с дочерней страницы на родительскую с вытеснением с нее элементов на текущую страницу.
2. Если соседняя страница сама является “**небогатой**”, т.е. содержит **ровно  $m$**  элементов, используется другой прием. В этом случае происходит **объединение** двух “бедных” страниц с созданием **одной полной** страницы. Полная страница создается следующим образом:  $m-1$  элемент берется с текущей страницы,  $m$  - с соседней и один элемент - с родительской страницы. Интересно, что этот прием заимствования одного элемента с родительской страницы позволяет **сохранить** необходимую структуру Б-дерева, т.к. синхронно на 1 уменьшается число элементов на родительской странице и число страниц-потомков.

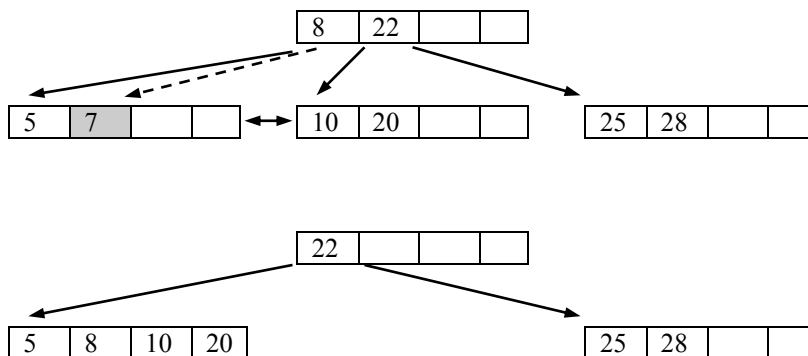
Поскольку число элементов на родительской странице уменьшилось, то надо и эту страницу проверить на “бедность”, и, при необходимости, выполнить ее корректировку. Это, в свою очередь, может привести к заимствованию элемента с еще более верхней страницы и т.д. В худшем случае этот процесс распространится **до корневой страницы**, что может потребовать корректировки самой корневой страницы. Правда, эта корректировка выполняется чуть по-другому, т.к. корневая страница может содержать и меньше  $m$  элементов. Особенность возникает лишь тогда, когда до удаления на корневой странице был **лишь один элемент**, и после удаления она

становится пустой. В этом случае эта пустая страница **просто удаляется**, и тем самым на 1 **уменьшается высота** Б-дерева.

**Пример.** Пусть в простейшем Б-дереве порядка 2 удаляется вершина 12 и страница становится “бедной”. Соседняя страница (22, 25, 28) может отдать один элемент, и поэтому ключ 22 передается на родительскую страницу, вытесняя с нее ключ 20.



Если в этом дереве удалить, например, вершину 7, то придется объединять две страницы (5) и (10, 20) вместе и добавлять элемент 8 с родительской страницы для создания одной полной страницы.



Программная реализация удаления имеет те же особенности, что и добавление, т.е. необходимо запоминать траекторию движения от корневой странице к текущей. Для этого можно ввести рекурсивную подпрограмму Delete с тремя параметрами:

- $x$  – ключ удаляемого элемента (параметр-значение)
- $pCurrent$  – ссылка на текущую страницу (параметр-значение)

- IsPure – признак того, что текущая страница после удаления содержит слишком мало элементов (параметр-переменная).

Псевдокод подпрограммы Delete:

```

procedure Delete (x : <тип ключа>; pCurrent : pPage; var IsPure : boolean);
begin
 if pCurrent = nil then “элемента x в дереве нет”
 else begin
 “поиск x на странице pCurrent”;
 if “x найден” then
 begin
 if “страница pCurrent^ терминальная”
 then begin
 “удалить элемент”;
 “уменьшить счетчик числа элементов”;
 “установить признак IsPure”;
 end
 else begin
 “найти элемент-заменитель”;
 “вставить элемент-заменитель на место
 удаленного x”;
 “уменьшить счетчик числа элементов”;
 “установить признак IsPure”;
 end
 end
 end
 else begin
 Delete (x, “адрес дочерней страницы”, IsPure);
 if IsPure then “вызов вспом. подпрограммы Pager()”;
 end;
end;
end;

```

Здесь используется вспомогательная подпрограмма `Pager`, выполняющая корректировку страницы после удаления с нее элемента `x` в случае ее “обеднения”. Эта подпрограмма имеет 4 параметра:

- адрес текущей “бедной” страницы (`pCurrent`)
- адрес родительской страницы (`pParent`)
- номер элемента на родительской странице, который является непосредственным предком “бедной” страницы (`nParent`)
- логический признак (`IsPure`)

Псевдокод подпрограммы `Pager`:

```
procedure Pager (pCurrent, pParent : pPage; nParent : integer; var IsPure :
 boolean);
```

```
var pTemp : pPage; {адрес соседней страницы}
```

```
begin
```

```
 “определение адреса pTemp соседней вершины”;
```

```
 “определение числа элементов, удаляемых с соседней страницы”;
```

```
 “пересылка одного элемента с родительской страницы на текущую”;
```

```
if “со страницы pTemp пересылается более одного элемента” then
```

```
 begin
```

```
 “пересылка элементов со страницы pTemp на pCurrent”;
```

```
 “пересылка одного элемента на родительскую страницу”;
```

```
 “сдвиг влево элементов в массиве на странице pTemp”;
```

```
 “установка счетчиков числа элементов на соседних страницах”;
```

```
 IsPure := false;
```

```
 end
```

```
else begin {объединение страниц}
```

```
 “пересылка всех эл-тов со страницы pTemp на страницу pCurrent”;
```

```
 “сдвиг влево элементов в массиве на родительской странице”;
```

```
 “изменение счетчиков числа эл-тов на тек. странице и ее родителе”;
```

```
 “удаление пустой страницы pTemp”;
```

```
 “установка признака IsPure для родительской страницы”;
```

**end;**

**end;**

Главная программа должна вызвать подпрограмму Delete, а после возврата из цепочки рекурсивных вызовов – удалить опустевшую корневую страницу.

### 5.7. Внешняя сортировка

Пусть требуется выполнить сортировку сверхбольшого набора данных, который целиком в ОП не помещается. В этом случае рассмотренные ранее методы сортировки массивов не работают и приходится использовать внешние файлы. Исходный набор данных хранится во внешнем файле и, очевидно, многократно должен считываться в ОП. В каждый момент времени в ОП находится **лишь часть** полного набора. Главным критерием при разработке методов сортировки становится **минимизация числа обращений** к внешней памяти.

Основой большинства алгоритмов внешней сортировки является **принцип слияния** двух упорядоченных последовательностей в единый упорядоченный набор.

**Например,** пусть имеются две упорядоченные числовые последовательности:

(5, 11, 17) и (8, 9, 19, 25)

Их слияние выполняется следующим образом:

- сравниваются первые числа 5 и 8, наименьшее (5) помещается в выходной набор: (5)
- сравниваются 11 и 8, наименьшее (8) – в выходной набор: (5, 8)
- сравниваются 11 и 9, наименьшее (9) – в выходной набор: (5, 8, 9)
- сравниваются 11 и 19, наименьшее (11) – в выходной набор: (5, 8, 9, 11)
- сравниваются 17 и 19, наименьшее (17) – в вых. набор: (5, 8, 9, 11, 17)
- остаток второй последовательности (19, 25) просто копируется в выходной набор с получением окончательного результата: (5, 8, 9, 11, 17, 19, 25)

В общем виде алгоритм слияния можно представить следующим образом.

Пусть  $A=\{a_i\}$  и  $B=\{b_j\}$  – две исходные упорядоченные последовательности,  $R$  – результирующая последовательность.

1. сравниваем  $a_1$  и  $b_1$ , если  $a_1 < b_1$ , то записываем  $a_1$  в  $R$ , иначе -  $b_1$  в  $R$
2. если в уменьшившемся наборе остались числа, то сравниваем его минимальный элемент с минимальным элементом другого набора и записываем наименьший из них в  $R$
3. повторяем шаг 2 до тех пор, пока не закончится какой-нибудь из наборов
4. как только заканчивается какой-нибудь набор, все оставшиеся элементы другой последовательности копируются в результирующий набор

Алгоритм слияния можно использовать и для сортировки массивов, если последовательно применить его несколько раз **ко все более длинным** упорядоченным последовательностям. Для этого:

1. в исходном наборе выделяются две подряд идущие возрастающие подпоследовательности (**серии**)
2. эти подпоследовательности (серии) сливаются в одну более длинную упорядоченную последовательность так, как описано выше
3. шаги 1 и 2 повторяются до тех пор, пока не будет достигнут конец входного набора
4. шаги 1 –3 применяются к новому полученному набору, т.е. выделяются пары серий, которые сливаются в еще более длинные наборы, и.т.д. до тех пор, пока не будет получена единая упорядоченная последовательность.

**Пример.** Пусть имеется входной неупорядоченный набор чисел:

24, 08, 13, 17, 06, 19, 20, 11, 07, 55, 33, 22, 18, 02, 40

**Этап 1.** Выделяем в нем серии (показаны скобками) и попарно сливаем их:

(24), (08, 13, 17), (06, 19, 20), (11), (07, 55), (33), (22), (18), (02, 40)

(24) + (08, 13, 17) = (08, 13, 17, 24)

(06, 19, 20) + (11) = (06, 11, 19, 20)

(07, 55) + (33) = (07, 33, 55)

$$(22) + (18) = (18, 22)$$

Новый набор чисел:

08, 13, 17, 24, 06, 11, 19, 20, 07, 33, 55, 18, 22, 02, 40

**Этап 2.** Выделяем в новом наборе серии и попарно сливаем их (число серий уменьшилось):

(08, 13, 17, 24), (06, 11, 19, 20), (07, 33, 55), (18, 22), (02, 40)

(08, 13, 17, 24) + (06, 11, 19, 20) = (06, 08, 11, 13, 17, 19, 20, 24)

(07, 33, 55) + (18, 22) = (07, 18, 22, 33, 55)

Новый набор:

06, 08, 11, 13, 17, 19, 20, 24, 07, 18, 22, 33, 55, 02, 40

**Этап 3.** Выделяем в новом наборе серии (всего три) и сливаем их:

(06, 08, 11, 13, 17, 19, 20, 24), (07, 18, 22, 33, 55), (02, 40)

(06, 08, 11, 13, 17, 19, 20, 24) + (07, 18, 22, 33, 55) = (06, 07, 08, 11, 13, 17, 18, 19, 20, 22, 24, 33, 55)

Новый набор и его две серии:

(06, 07, 08, 11, 13, 17, 18, 19, 20, 22, 24, 33, 55), (02, 40)

**Этап 4.** После слияния этих двух серий получим полностью упорядоченный набор:

02, 06, 07, 08, 11, 13, 17, 18, 19, 20, 22, 24, 33, 40, 55

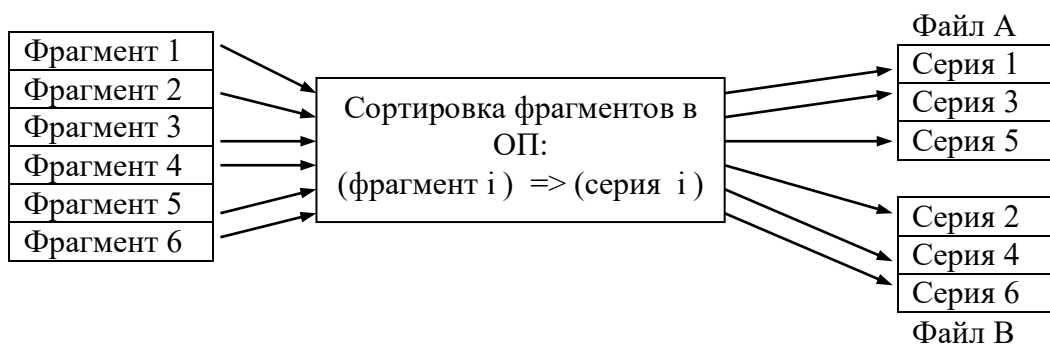
Данный метод сортировки называется **естественным слиянием**. Его особенностью является то, что для его реализации на каждом шаге достаточно сравнивать **только два элемента**. Именно это позволяет хранить все основные элементы в дисковых файлах, загружая в ОП только небольшое число элементов из каждой серии.

Алгоритм слияния эффективно работает на **длинных** сериях и неэффективно на коротких. Поэтому его **нецелесообразно** применять с самого начала сортировки, поскольку для случайного входного набора на первых шагах будут получаться **очень короткие** серии, что приведет к неоправданно большому числу обращений к дисковой памяти.

Для устранения этого недостатка можно поступить следующим образом:

- исходный сверхбольшой набор данных разделяется на отдельные фрагменты такого размера, чтобы каждый фрагмент мог **целиком** разместиться в ОП
- каждый фрагмент **отдельно** загружается в ОП и **сортируется** каким-нибудь классическим улучшенным методом (например – алгоритмом быстрой сортировки)
- каждый отсортированный фрагмент рассматривается как серия и сохраняется во вспомогательном файле; в простейшем случае достаточно двух вспомогательных файлов, в которые серии сбрасываются поочередно (нечетные серии – в один файл (А), четные – в другой (В)).

Эти действия носят подготовительный характер.



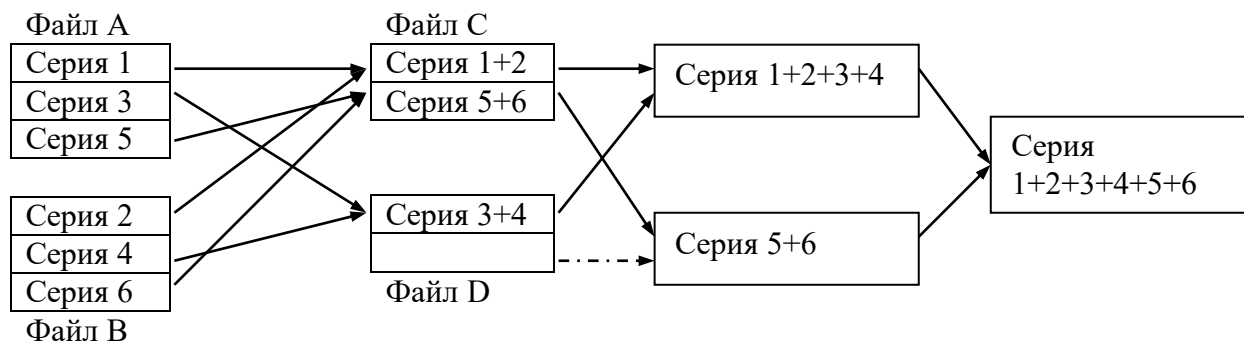
После этого начинается 2-ой этап сортировки:

- выполняется объединение первых серий из разных файлов, т.е. серий 1 и 2, и результат записывается в третий вспомогательный файл С
- выполняется объединение вторых серий из разных файлов, т.е. серий 3 и 4, и результат записывается во вспомогательный файл D
- выполняется объединение третьих серий из разных файлов, т.е. серий 5 и 6, и результат записывается опять в файл С
- серии 7 и 8 после объединения записываются в файл D и т.д.

В итоге, в файлах С и D получатся объединенные серии, которые потом между собой начинают попарно объединяться с записью результата во вспомогательные файлы А и В и т. д. Процесс укрупнения серий продолжается



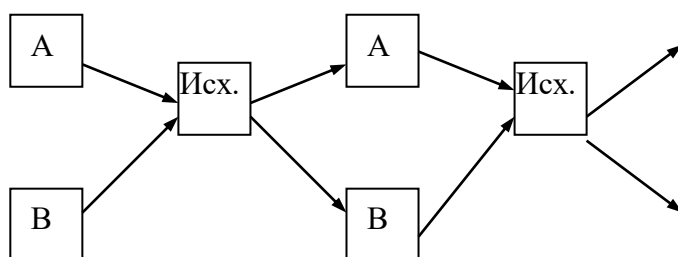
до тех пор, пока не будет получена одна единственная серия, представляющая полученный результат.



Для программной реализации данного метода сортировки необходимы:

- достаточно большой массив в ОП для реализации 1-го этапа
- пять файлов, из которых один исходный и 4 вспомогательных.

Если необходимой для четырех вспомогательных файлов дисковой памяти не хватает, то можно обойтись двумя вспомогательными файлами, но за счет замедления процесса сортировки. В этом случае объединенные серии из А и В помещаются в исходный файл, а потом распределяются опять в А и В. Тем самым появляется дополнительный шаг распределения серий из исходного файла по двум вспомогательным.



Наоборот, если требуется ускорить сортировку за счет дополнительной дисковой памяти, то вместо 4-х вспомогательных файлов можно использовать 6,8,10 и т.д. Ускорение работы происходит за счет того, что объединяются не 2 серии, а 3, 4 или 5 и т.д.

### 5.8. Практические задания

**Задание 1.** Разработать программу для имитации обработки простейшего Б-дерева второго порядка. Программа должна выполнять следующие действия:

- построение Б-дерева для заданного набора вершин, начиная с пустого дерева
- добавление отдельной вершины
- поиск заданной вершины
- удаление заданной вершины
- вывод Б-дерева в наглядном виде

Имитация обработки определяется тем, что файлы для хранения элементов дерева не используются.

**Задание 2.** Разработать программу сортировки файлов методом естественного слияния. Для простоты этап предварительной сортировки фрагментов для получения начальных серий не реализуется. Должны использоваться 5 файлов – входной и 4 вспомогательных. Исходный набор заполняется случайными числами в диапазоне от 1 до 1 миллиона.

### 5.9. Контрольные вопросы по теме

1. В чем состоят особенности задач внешнего поиска и сортировки?
2. Какой критерий является основным для алгоритмов внешнего поиска и сортировки?
3. Для решения каких задач используется структура данных Б-дерево?
4. Что называется страницей Б-дерева?
5. Почему страничная организация Б-дерева является эффективной для решения задачи внешнего поиска?
6. Каким условиям должно удовлетворять Б-дерево?
7. Какими свойствами обладает Б-дерево как дерево поиска?
8. Как оценивается эффективность использования Б-деревьев для задач внешнего поиска?
9. Какие данные должна содержать страница Б-дерева?

10. Какие описания необходимы для определения Б-дерева как структуры данных?
11. Какие шаги включает алгоритм поиска в Б-дереве?
12. Какие основные шаги включает алгоритм добавления вершины в Б-дерево?
13. Что происходит при попытке добавления новой вершины на полностью заполненную страницу Б-дерева?
14. К чему может привести процесс расщепления полной страницы Б-дерева при добавлении новой вершины?
15. В каких случаях высота Б-дерева может увеличиться на единицу?
16. Приведите практический пример добавления вершины в простейшее Б-дерево.
17. Какие особенности возникают при программной реализации алгоритма добавления новой вершины в Б-дерево?
18. Почему при добавлении новой вершины в Б-дерево необходимо запоминать путь от корневой страницы к терминальной?
19. Какие параметры использует рекурсивная подпрограмма добавления новой вершины в Б-дерево?
20. Приведите схему рекурсивной подпрограммы добавления новой вершины в Б-дерево.
21. Что должна делать главная программа при добавлении новой вершины в Б-дерево?
22. Какие основные шаги включает алгоритм удаления вершины из Б-дерева?
23. Как обрабатывается удаление вершины из Б-дерева для терминальной и нетерминальной страницы?
24. Как находится элемент-заменитель для удаляемой из Б-дерева вершины?
25. Какая ситуация может возникнуть после удаления вершины с одной из страниц Б-дерева?
26. За счет чего сохраняется необходимая структура Б-дерева после удаления вершины?

27. Зачем и как выполняется перераспределение вершин между двумя соседними страницами при удалении вершины из Б-дерева?
28. Зачем и как выполняется объединение двух страниц при удалении вершины в Б-дереве?
29. К каким последствиям может привести процесс слияния соседних страниц при удалении вершин из Б-дерева?
30. Приведите практический пример удаления вершины из простейшего Б-дерева.
31. Какие особенности имеет программная реализация удаления вершины из Б-дерева?
32. Какие параметры должна иметь рекурсивная подпрограмма удаления вершины из Б-дерева?
33. Приведите схему рекурсивной подпрограммы удаления вершины из Б-дерева.
34. В чем состоит принцип слияния двух упорядоченных последовательностей?
35. Какие шаги выполняет алгоритм слияния двух упорядоченных последовательностей?
36. Как используется алгоритм слияния последовательностей для сортировки данных?
37. Что такое серия и как это понятие используется при сортировке данных?
38. Как можно улучшить сортировку файлов методом естественного слияния?
39. В чем состоит первый этап внешней сортировки?
40. В чем состоит второй этап внешней сортировки?
41. Какие необходимы вспомогательные файлы при внешней сортировке и как они используются?
42. Как можно уменьшить число вспомогательных файлов, используемых при внешней сортировке?
43. За счет чего можно ускорить сортировку файлов?

44. Как влияет число вспомогательных файлов на эффективность внешней сортировки?

### **Основные термины и понятия**

**АВЛ-сбалансированное дерево** – разновидность дерева поиска, в котором для каждой вершины высота ее левого и правого поддеревья отличаются не более чем на единицу

**Балансировка дерева** – перестановка вершин с целью сохранения “хорошей” структуры дерева

**Б-дерево** – разновидность дерева поиска с очень большим числом ключей, основанная на группировании соседних ключей в виде страниц и хранении этих страниц на дисковой памяти

**Быстрая сортировка** - улучшенный метод сортировки массивов, основанный на разбиении набора данных на все меньшие подмассивы с последующей сортировкой каждого из них

**Внешняя сортировка** – упорядочивание больших наборов данных с преимущественным хранением на дисковой памяти

**Внутреннее хеширование** – метод разрешения конфликта ключей при хеш-поиске, основанный на размещении конфликтующих ключей в свободных ячейках таблицы

**Внутренняя сортировка** – упорядочивание элементов массива, полностью располагающихся в оперативной памяти

**Высота дерева** – наиболее длинный путь от корневой вершины к терминальной

**Граф** – нелинейная структура данных, состоящая из элементов-вершин, между некоторыми из которых установлены определенные связи

**Двоичное дерево** – разновидность дерева, в котором каждая вершина может иметь не более двух связанных с нею поддеревьев

**Двунаправленный линейный список** – разновидность списковой структуры, в которой каждый элемент имеет два указателя на каждого из своих соседей

**Дерево** – нелинейная структура данных, рекурсивно представимая в виде отдельных поддеревьев

**Дерево поиска** – разновидность двоичного дерева, в котором для каждой вершины ключи в левом поддереве меньше, а в правом поддереве больше ключа этой вершины

**Динамическое распределение памяти** – механизм, с помощью которого при выполнении программы можно получать и освобождать области памяти для хранения каких-либо данных

**Естественное слияние** – один из методов внешней сортировки, основанный на попарном сравнении и объединении двух упорядоченных последовательностей в одну последовательность

**Заголовок списка** – дополнительный необязательный элемент в начале списка, который никогда не удаляется из списка

**Идеально сбалансированное дерево** – разновидность двоичного дерева, для каждой вершины которого число вершин в левом и правом поддеревьях отличаются не более чем на единицу

**Карманная сортировка** – один из специальных методов сортировки, применяемый для ключей с различными значениями в пределах от 1 до  $n$

**Конфликт ключей** – ситуация, когда при использовании хеш-поиска два различных ключа претендуют на одно и то же место в массиве

**Матрица смежности** – способ описания графа с помощью двумерного массива  $N \times N$ , ненулевые элементы которого соответствуют связанным вершинам

**Метод Шелла** – улучшенный метод сортировки массивов, основанный на многократном группировании элементов массива с уменьшающимся шагом и последующей сортировкой методом вставок

**О-нотация** – способ оценивания трудоемкости алгоритма в зависимости от объема обрабатываемых данных

**Открытое хеширование** – метод разрешения конфликта ключей при хеш-поиске, основанный на использовании вспомогательных списков для конфликтующих ключей

**Очередь** – линейная структура данных, в которую добавление производится с одного конца, а удаление – с другого

**Переменные-указатели** – специальные переменные, значениями которых являются адреса областей памяти

**Пирамида** – разновидность двоичного дерева, в котором ключ каждой вершины не больше ключей всех ее потомков

**Пирамидальная сортировка** – улучшенный метод сортировки массивов, основанный на специальном представлении исходного массива в виде так называемой пирамиды

**Поразрядная сортировка** - один из специальных методов сортировки, применяемый для целочисленных ключей с известным числом разрядов и требующий использования дополнительных списков

**Простейшие методы сортировки** – алгоритмически простые методы сортировки массивов с трудоемкостью порядка  $n^2$

**Сортировка вставками** – простейший метод сортировки массивов, основанный на поиске для каждого очередного элемента подходящего места в уже обработанной последовательности

**Сортировка выбором** - простейший метод сортировки массивов, основанный на поиске в необработанном подмножестве наименьшего элемента

**Сортировка обменом** – простейший метод сортировки массивов, основанный на попарном сравнении и перестановке соседних элементов

**Специальные методы сортировки** – группа методов сортировки массивов, основанных на использовании дополнительной информации о сортируемом наборе, требующих большой дополнительной памяти, но имеющих линейную трудоемкость

**Список** – линейная структура данных с возможностью добавления и удаления элементов в любом месте

**Список смежности** – способ описания графа в виде массива или главного списка вершин, с каждым элементом которого связан список смежных с ней вершин

**Стек** – линейная структура данных с односторонним добавлением и удалением элементов

**Улучшенные методы сортировки** – алгоритмически достаточно сложные методы сортировки массивов с трудоемкостью порядка  $n \cdot \log n$

**Универсальные методы сортировки** – группа методов сортировки массивов, не требующих никакой дополнительной информации о сортируемых наборах и никакой дополнительной памяти

**Хеш-поиск** – метод поиска, позволяющий по значению входного ключа сразу определять его положение в массиве, организованном в виде специальной таблицы (хеш-таблицы)

**Хеш-таблица** – массив ключей, размещенных в ячейках, индексы которых определяются с помощью специального преобразования (хеш-функции)

**Хеш-функция** – специальный алгоритм (в частном случае – функция), применяемый для преобразования входного ключа в индекс его размещения в массиве (хеш-таблице)

### Литература

1. Кнут Д. Искусство программирования, том 1. Основные алгоритмы, 3-е изд. – М.: Изд. дом “Вильямс”, 2000 г.
2. Кнут Д. Искусство программирования, том 3. Сортировка и поиск, 2-е изд. – М.: Изд. дом “Вильямс”, 2000 г.
3. Вирт Н. Алгоритмы и структуры данных
4. Ахо А., Хопкрофт Д., Ульман Д. Структуры данных и алгоритмы. – М.: Изд. дом “Вильямс”, 2001 г.
5. Кормен Т. и др. Алгоритмы: построение и анализ. – МЦНМО, 2000 г.
6. Топп У., Форд У. Структуры данных в C++. – М.: ЗАО “Издательство БИНОМ”, 2000 г.



7. Хэзфилд Р., Кирби Л. и др. Искусство программирования на С. Фундаментальные алгоритмы, структуры данных и примеры приложений. – К.: Издательство “ДиаСофт”, 2001 г.