Методы и средства хранения информации

ст. пр. Овчинников А.Л.

Структура курса

- Лекции – 18 часов
- Лабораторные занятия – 18 часов
- Самостоятельная работа студента – 90 часов

Виды контроля:

- Модульный контроль (тестирование)
- Защита лабораторных работ
- ЗАЧЕТ

Структура лекционного материала

- СТРУКТУРЫ ДАННЫХ ДЛЯ ХРАНЕНИЯ И ПОИСКА
- МЕТОДЫ ОРГАНИЗАЦИИ ХРАНЕНИЯ ДАННЫХ В СУБД
- РОЛЬ И ЗАДАЧИ ЗУ В СОВРЕМЕННЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМАХ И ЭВМ
- КЛАССИФИКАЦИЯ УСТРОЙСТВ ХРАНЕНИЯ ИНФОРМАЦИИ
- ОСНОВНЫЕ ПАРАМЕТРЫ ВЗУ
- ОСНОВНЫЕ ХАРАКТЕРИСТИКИ НОСИТЕЛЕЙ ИНФОРМАЦИИ
- МАГНИТНЫЕ НАКОПИТЕЛИ
- ОПТИЧЕСКИЕ НАКОПИТЕЛИ(НАКОПИТЕЛИ НА КОМПАКТ-ДИСКАХ, DVD)
- ОТКАЗОУСТОЙЧИВЫЕ СИСТЕМЫ ХРАНЕНИЯ ДАННЫХ. RAID-МАССИВЫ
- УСТРОЙСТВА ДЛЯ РЕЗЕРВНОГО КОПИРОВАНИЯ ДАННЫХ

Содержание лабораторного практикума

Лр1	Исследование линейных и нелинейных структур данных. Линейные списки и бинарные деревья поиска. Цель работы: Исследовать возможности применения линейных и нелинейных структур данных – линейных списков и бинарных деревьев поиска – для хранения, поиска и обработки информации. Приобрести практические навыки использования классов, реализующих списки и бинарные деревья поиска и исследовать их эффективность при выполнении операций добавления, удаления и поиска данных.
Лр2	Исследование нелинейных структур данных. AVL-деревья Цель работы: Исследовать возможности применения AVL-деревьев – для хранения, поиска и обработки информации. Приобрести практические навыки использования классов, реализующих AVL-деревья. Оценить эффективность использования AVL-деревьев в сравнении с бинарными деревьями поиска.
Лр3	Исследование Б-деревьев при доступе к данным во внешней памяти Цель работы: Исследовать возможности применения нелинейных структур данных – Б-деревьев, для хранения и поиска информации. Приобрести практические навыки использования Б-деревьев для реализации эффективного поиска и доступа к данным во внешней памяти. Произвести оценку эффективности использования Б-деревьев для организации хранения данных.
Лр4	Структуры данных, основанные на хеш-таблицах Цель работы: Исследовать возможности применения нелинейных структур данных – хеш-таблиц для хранения и обработки информации. Приобрести практические навыки использования хеш-таблиц для реализации быстрого доступа к данным.
Лр5	Исследование подсистемы WINAPI по работе с дисками Цель работы: Исследовать подсистему WINAPI по работе с дисковой подсистемой, приобрести практические навыки разработки программного обеспечения, использующего информацию о накопителях информации.
Лр6	Исследование главной загрузочной записи диска Цель работы: Изучение структуры главной загрузочной записи жесткого диска. Получение практических навыков работы с главной загрузочной записью.

Литература

1.	Савицкий, Н. И. Технологии организации, хранения и обработки данных [Текст] : учеб. пособие / Н. И. Савицкий. - М. : ИНФРА - М, 2001. (рус.).
2.	Гук, М. Аппаратные средства IBM PC [Текст] : энцикл. / М. Гук. - 2-е изд. - М. и др. : Питер, 2004. - 928 с. (рус.).
3.	Гасанов, Э. Э. Теория хранения и поиска информации [Текст] / Э. Э. Гасанов, В. Б. Кудрявцев. - М. : Физматлит, 2002. - 288 с. (рус.).
4.	Кузнецов, С. Д. СУБД(системы управления базами данных) и файловые системы [Текст] : научно-популярная литература / С. Д. Кузнецов; . - М. : Майор, 2001. - 175 с. (рус.).
5.	Гук, М. Дисковая подсистема ПК [Текст] / М. Гук. - СПб. и др. : Питер, 2001. - 336 с. (рус.).
6.	Побегайло, А. Системное программирование в Windows [Текст] / А. Побегайло. - СПб. : БХВ - Петербург, 2006. - 1056 с. (рус.).

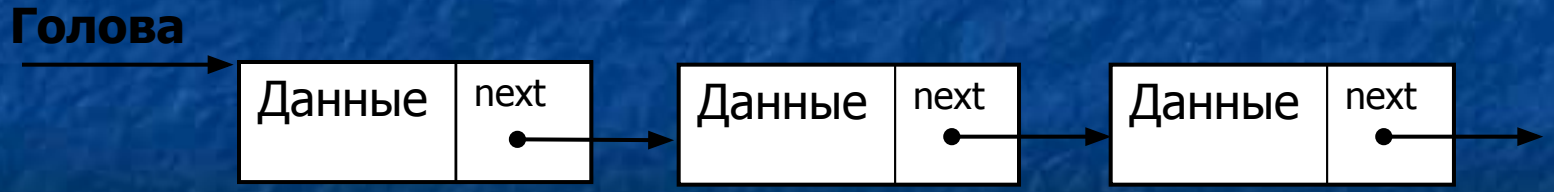
ЛИНЕЙНЫЕ И НЕЛИНЕЙНЫЕ СТРУКТУРЫ ДАННЫХ. ЛИНЕЙНЫЕ СПИСКИ И БИНАРНЫЕ ДЕРЕВЬЯ ПОИСКА.

- *Список* – это линейная динамическая структура данных. Каждый элемент такой структуры, в простейшем случае содержит два поля:

  поле данных;

  поле – указатель на следующий элемент.

С помощью указателя каждый из элементов списка связывается со следующим элементом (в случае однонаправленного списка)

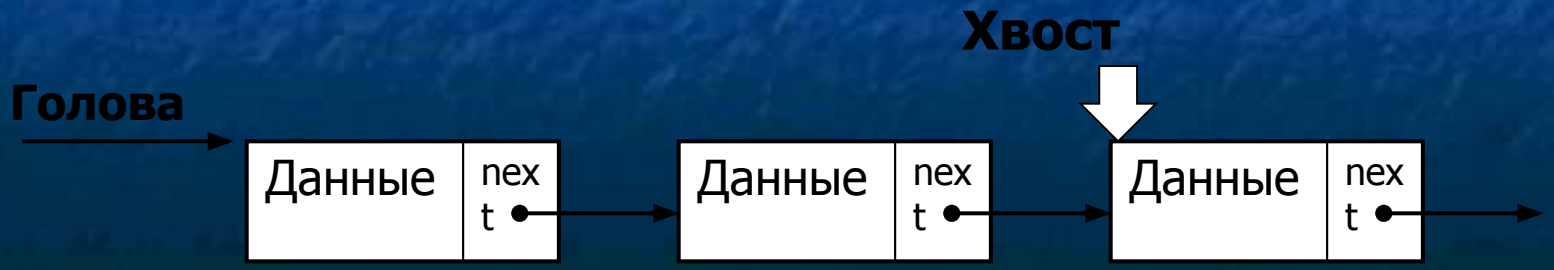


Для работы со списком обязательно хранится указатель на первый элемент(*голову*) списка.

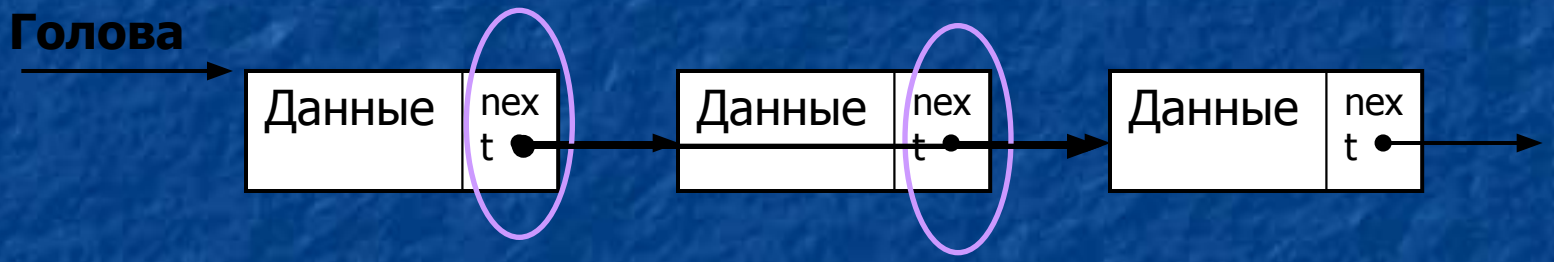
- Основным отличием списка от массива является то, что **список является динамической структурой**. Это позволяет:
  - создавать список, не резервируя лишнего места в памяти
  - создавать список той длины, которая необходима программе на данный момент времени, не опасаясь его переполнения.

Однако, в списке отсутствует возможность сразу обратиться к произвольному элементу, что существенно замедляет поиск (в массиве доступ к любому элементу может быть получен с использованием индекса).

- **Добавление** элементов, как правило, осуществляется в конец списка, поэтому желательно также всегда хранить указатель и на последний элемент – *хвост* списка.



- При **удалении** некоторого элемента списка необходимо найти предыдущий и следующий элементы и связать их (указатель предыдущего должен теперь указывать на следующий, а не на удаляемый). Только после этого можно освободить память от удаляемого элемента.



- В отличии от линейного списка, **дерево** – нелинейная динамическая структура данных.

- Древовидная структура характеризуется:
  - множеством **узлов** (nodes), происходящих от единственного начального узла, называемого **корнем** (root).
  - Каждый узел может быть **родителем** (parent), указывающим на 1 или более узлов, называемых **сыновьями** (children).
  - Каждый некорневой узел имеет только одного родителя, и каждый родитель имеет 0 или более сыновей.
  - Узел, не имеющий детей, называется **листом** (leaf).

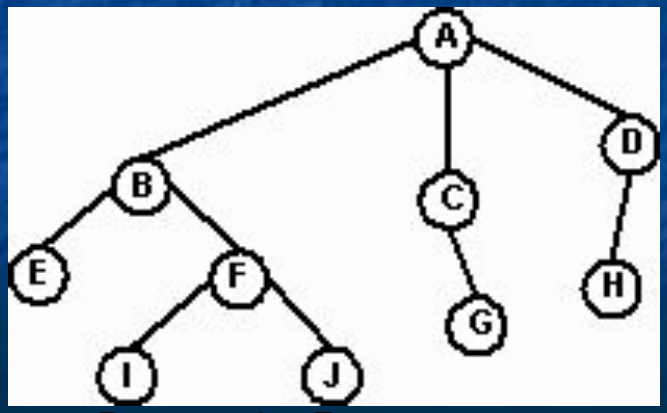


Рисунок 1 – Древовидная структура

- Длина пути от корня к какому-либо узлу есть **уровень узла**. Уровень корня равен 0. Каждый сын корня является узлом 1-го уровня, следующее поколение – узлами 2-го уровня и т.д. Например, на рисунке 2 узел F является узлом 2-го уровня (с длиной пути 2).
- **Глубина** (depth) дерева есть **его максимальный уровень**. Понятие глубины также может быть описано в терминах пути. Глубина дерева есть длина самого длинного пути от корня до узла. На рисунке 2 глубина дерева равна 3.

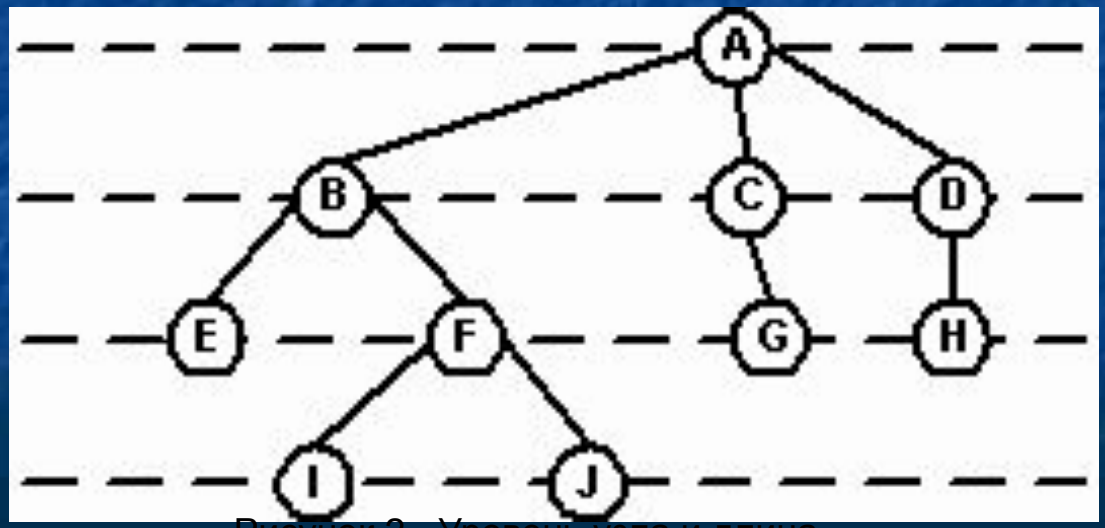


Рисунок 2 - Уровень узла и длина пути

- Сосредоточимся на ограниченном классе деревьев, где каждый родитель имеет не **более двух сыновей**.

  Такие деревья называются **бинарными** деревьями. **Бинарные деревья** (binary trees) имеют унифицированную структуру, допускающую разнообразные алгоритмы прохождения и **эффективный доступ к элементам**.

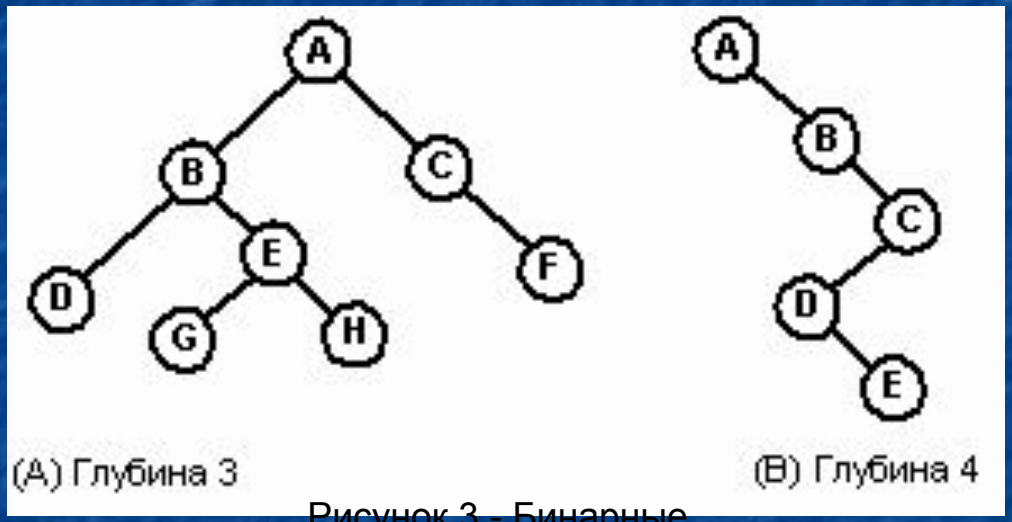
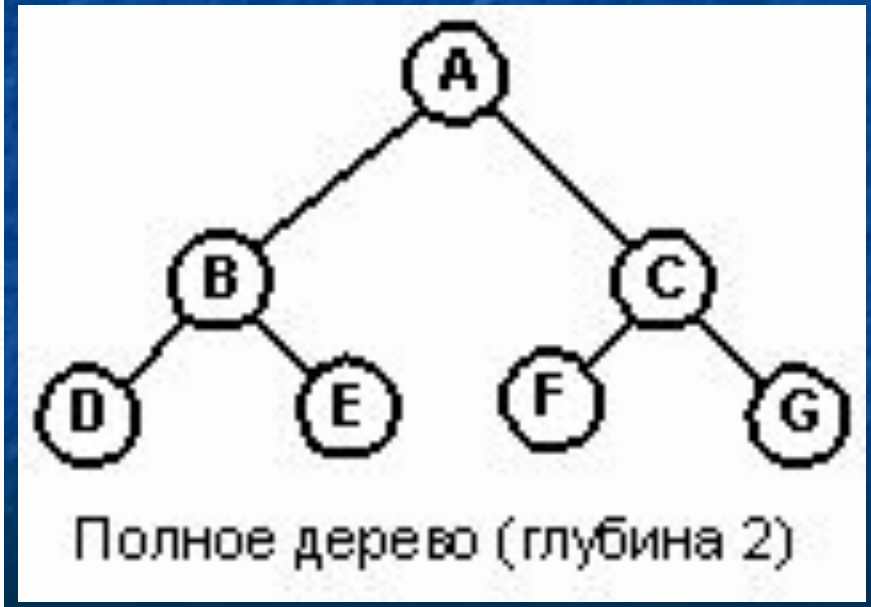


Рисунок 3 - Бинарные деревья

На уровне ***n*** бинарное дерево может содержать от 1 до 2^n узлов. **Число узлов, приходящееся на уровень, является показателем плотности дерева**. На рисунке 3 дерево A содержит 8 узлов при глубине 3, в то время как дерево B содержит 5 узлов при глубине 4. Последний случай является особой формой, называемой **вырожденным (degenerate) деревом**, у которого есть единственный лист (E) и каждый нелистовой узел имеет только одного сына.

- **Законченные** бинарные деревья (complete binary tree) - деревья глубины N, где каждый уровень 0...N-1 имеет полный набор узлов, и все листья уровня N расположены слева.
- Законченное бинарное дерево, содержащее 2^N узлов на уровне N, является **полным**.
- На рисунке показаны **законченное** и **полное** бинарные деревья:

Законченное дерево (глубина 3)



Полное дерево (глубина 2)

- Законченные и полные бинарные деревья обладают следующими свойствами:
  - На нулевом уровне имеется 2^0 узлов, на первом - 2^1, на втором - 2^2 и т.д. На первых **k-1** уровнях имеется 2^k-1 узлов: $\mathbf{1 + 2 + 4 + ... + 2^{k1} = 2^k - 1}$
  - На уровне **k** количество дополнительных узлов колеблется от 1 до 2^k (полное). В полном дереве число узлов равно:$\mathbf{1 + 2 + 4 + ... + 2^{k-1} + 2^k = 2^{k+1} - 1}$
  - Число узлов законченного бинарного дерева удовлетворяет неравенству:$\mathbf{2k \leq N \leq 2k-1 - 1 \leq 2k-1}$; решая его относительно k, имеем: $\mathbf{k \leq \log_2 (N) \leq k+1}$

В случае, **если бинарное дерево упорядочено**, т.е. для каждого поддерева выполняется условие – значение каждого узла левого поддерева меньше значения корневого узла, а значение любого узла правого поддерева больше или равно значения корневого узла – такое дерево называется **бинарным деревом поиска**.
Упорядоченность дерева накладывает свои особенности на процедуры создания дерева, добавления и удаления элементов (узлов), а также поиска.

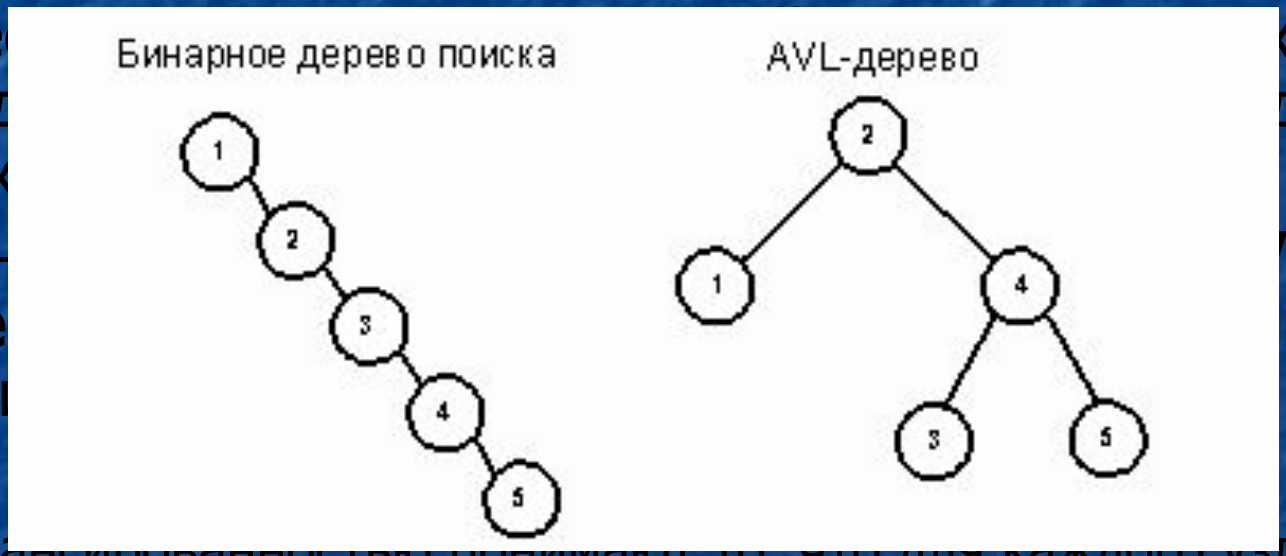
- Очевидно, что бинарное дерево поиска будет иметь существенные преимущества перед линейным списком по времени поиска данных.
- Действительно, если для поиска в линейном списке, содержащем **N** элементов, в худшем случае нужно выполнить **N** операций сравнения, то в случае полного бинарного дерева поиска, содержащего такое же количество элементов, наибольшее количество сравнений - $\mathbf{log_2(N)}$.
- Очевидно, что чем больше **N**, тем более выгодно использование при поиске бинарного дерева поиска по сравнению с линейным списком.

Однако, при сравнении двух рассматриваемых структур по времени **выполнения операций добавления** элементов следует отметить **преимущество линейного списка**.
Действительно, при добавлении в бинарное дерево поиска сначала придется найти подходящий лист(выполнив при этом в случае полного дерева $\mathbf{log_2(N)}$ операций сравнения), тогда как добавление к списку, в случае хранения хвоста происходит сразу же.

- Как известно, при **удалении** узла из бинарного дерева поиска, **необходимо сначала найти** требуемый элемент, а затем может возникнуть необходимость модификации дерева (замещения удаляемого узла), при этом требуется время и на поиск замещающего узла.
- В списке же, удаление узла также сводится к его поиску (причем, в однонаправленном списке желательно сразу же найти и предыдущий элемент), после чего переопределением указателя предыдущего элемента текущий элемент может быть удален из списка.
- Таким образом, в **случае небольшого числа элементов, возможно и преимущество линейного списка**, однако, **с ростом числа элементов все существеннее будет преимущество бинарного дерева поиска**.

Сбалансированные деревья. AVL-деревья.

- Как известно, при некотором стечении обстоятельств бинарное дерево поиска может оказаться вырожденным. Тогда высота его будет N, и доступ к данным существенно замедлится.
- Суще… …к деревьев, облад… …деревьев поиск…
- Они н… …**VL-дере**… …ьского и Ланд…



Под сбалансированностью понимают то, что для каждого узла дерева **высоты обоих его поддеревьев различаются не более чем на 1**.

Строго говоря, этот критерий нужно называть **AVL-сбалансированностью** в отличие от идеальной сбалансированности, когда для каждого узла дерева количества узлов в левом и правом поддеревьях различаются не более чем на 1.

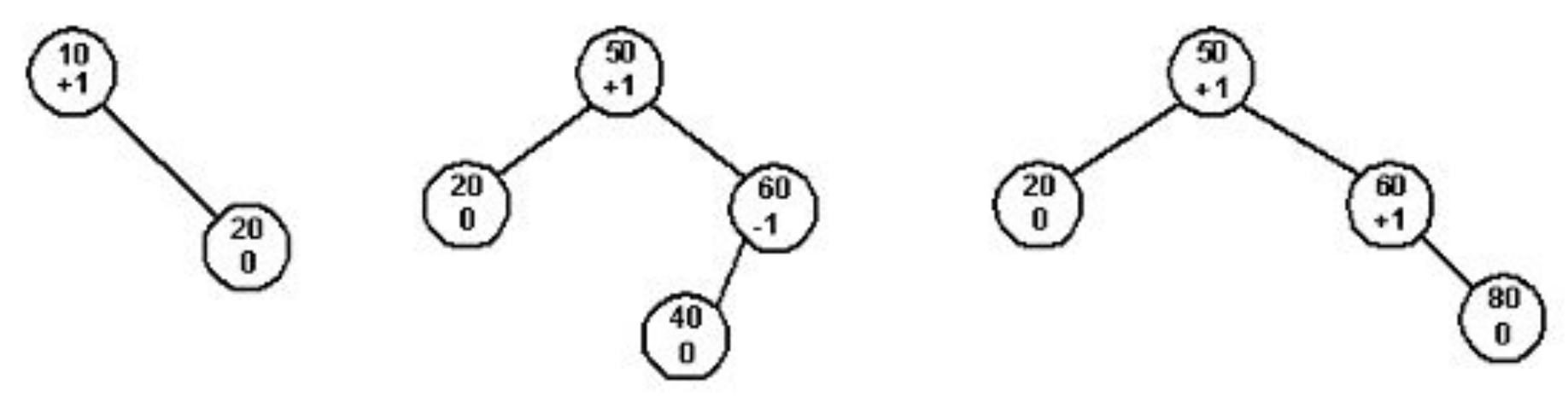
Сбалансированные деревья. AVL-деревья.

- В общем случае **высота сбалансированного дерева не превышает $\log_2 N$**.
- Таким образом, AVL-дерево является структурой хранения, обеспечивающей быстрый доступ к данным, независимо от порядка поступления ключей при его построении.

Очевидно, что AVL-деревья имеют структуру, аналогичную бинарным деревьям поиска.
Все операции идентичны описанным для бинарных деревьев, за исключением операций **добавления и удаления** узлов.

В случае AVL-деревьев, после выполнения каждой из этих операций **проверяется соотношение высот левого и правого поддеревьев** тех узлов, которые затронула операция(от добавленного/удаленного узла до корня).
Значение, содержащее разность высот правого и левого поддеревьев - **показатель сбалансированности -** хранится в дополнительном поле каждого узла AVL-дерева.

Сбалансированные деревья. AVL-деревья.



AVL-деревья со значениями показателя сбалансированности

- Если показатель сбалансированности отрицателен, то узел «перевешивает влево», так как высота левого поддерева больше, чем высота правого поддерева.
- При положительном показателе сбалансированности узел «перевешивает вправо».
- Сбалансированный по высоте узел имеет показатель сбалансированности равный 0.
- **В AVL-дереве показатель сбалансированности каждого узла должен принимать значения из диапазона [-1, 1].**

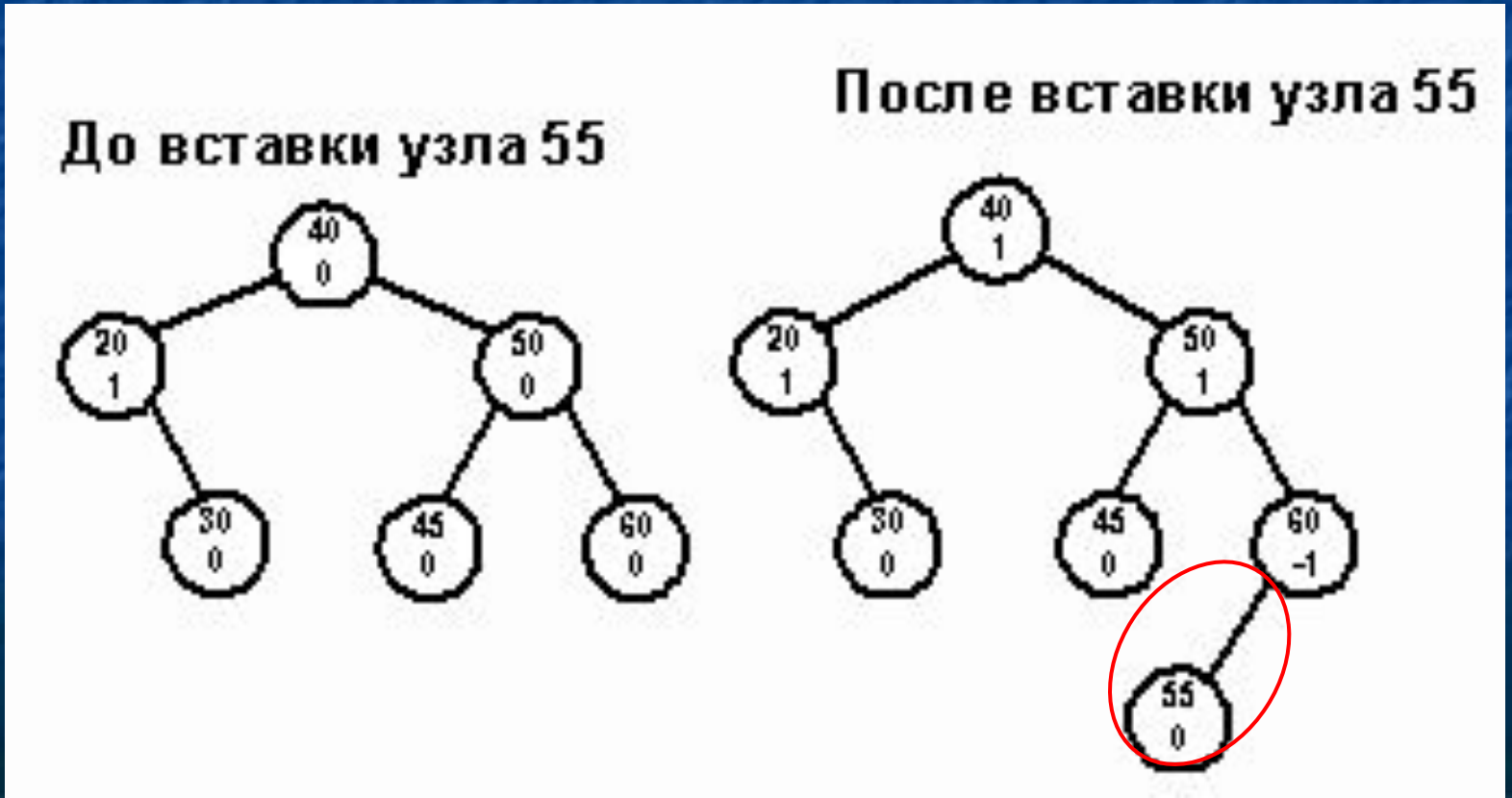
Алгоритм AVL-вставки

- Процесс поиска места для вставки нового узла в AVL-дерево такой же, как и в случае бинарного дерева поиска.
- Осуществляется спуск по левым и правым сыновьям, пока не встретится пустое поддерево, а затем производится предварительная вставка нового узла в этом месте(возможно использование рекурсивного спуска).
- Обновление показателей сбалансированности узлов, затронутых при добавлении ведется в обратном порядке(при возвращении из рекурсии). При этом показатель сбалансированности родительского узла можно скорректировать после изучения эффекта от добавления нового элемента в одно из его поддеревьев. Необходимость корректировки определяется для каждого узла, входящего в поисковый маршрут.

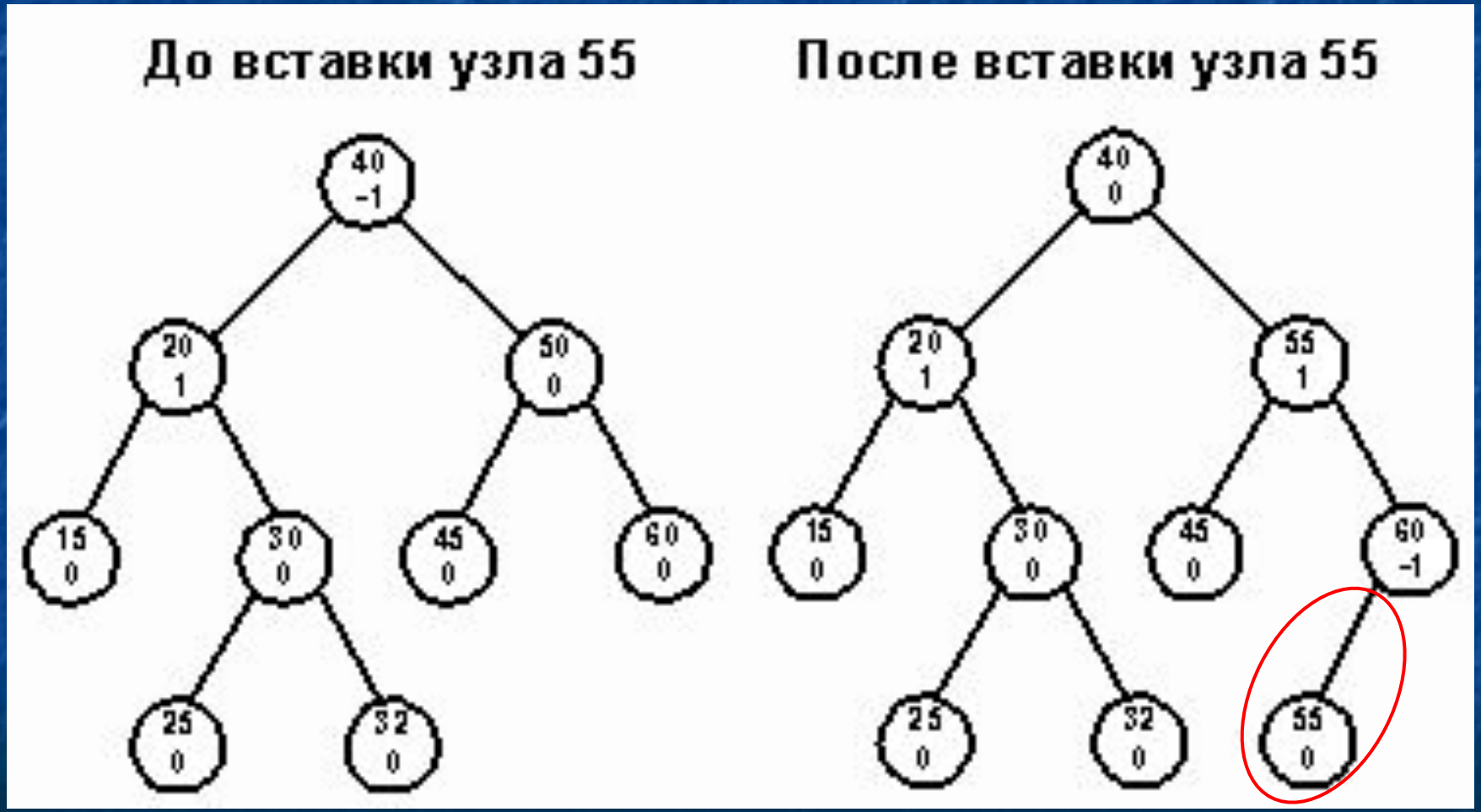
Алгоритм AVL-вставки

- Есть три возможных ситуации.
- В двух первых случаях узел сохраняет сбалансированность, и реорганизация поддеревьев не требуется. Нужно лишь скорректировать показатель сбалансированности данного узла.
- В третьем случае разбалансировка дерева требует **модификации** дерева – выполнения процедур **одинарного** или **двойного** поворотов узлов.

- ***Случай 1.*** Узел на поисковом маршруте изначально является сбалансированным (показатель сбалансированности равен 0). После вставки в поддерево нового элемента узел стал перевешивать влево или вправо в зависимости от того, в какое поддерево была произведена вставка:



- ***Случай 2.*** Одно из поддеревьев узла перевешивает, и новый узел вставляется в более легкое поддерево. Узел становится сбалансированным:



- ***Случай 3.*** Одно из поддеревьев узла перевешивает, и новый узел помещается в **более тяжелое поддерево**. Очевидно, что при этом нарушается условие сбалансированности дерева, так как показатель сбалансированности выходит за пределы -1..1.

  Чтобы восстановить равновесие, нужно выполнить модификацию дерева – одну из процедур **поворота** дерева.

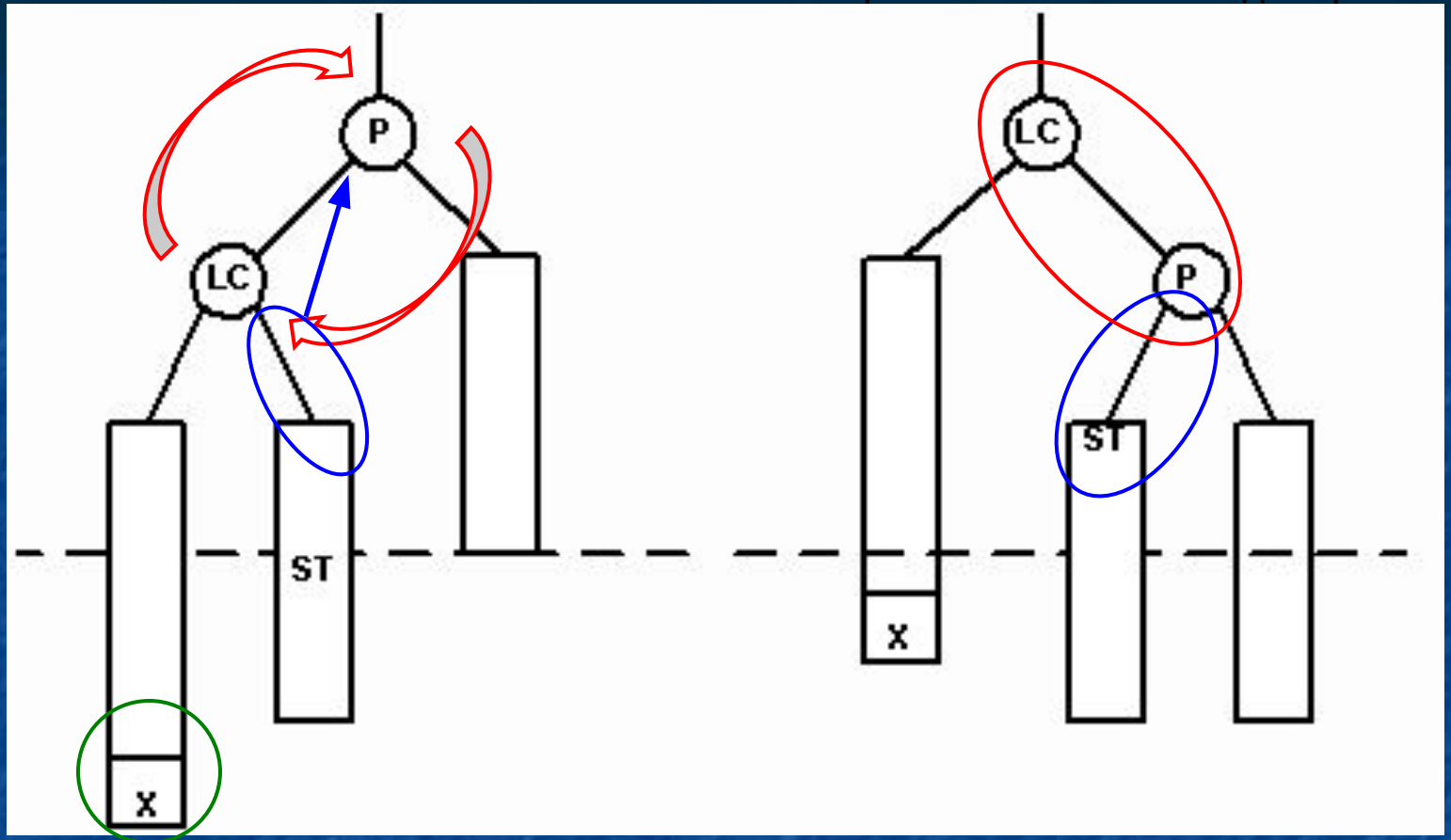
Повороты AVL-дерева

Повороты необходимы, когда родительский узел (обозначим его P) становится разбалансированным - баланс фактор становится равен 2 или -2.

Одинарный поворот вправо (single right rotation) происходит тогда, **когда родительский узел** P и **его левый сын** (обозначим его LC) **начинают перевешивать влево** после вставки узла в позицию X.

Сбалансированные деревья. AVL-деревья.

Алгоритм AVL-вставки. Одинарный поворот.



- В процессе одинарного поворота:
  - LC замещает своего родителя, который становится правым сыном.
  - Бывшее правое поддерево узла LC (ST) присоединяется к P в качестве левого поддерева.

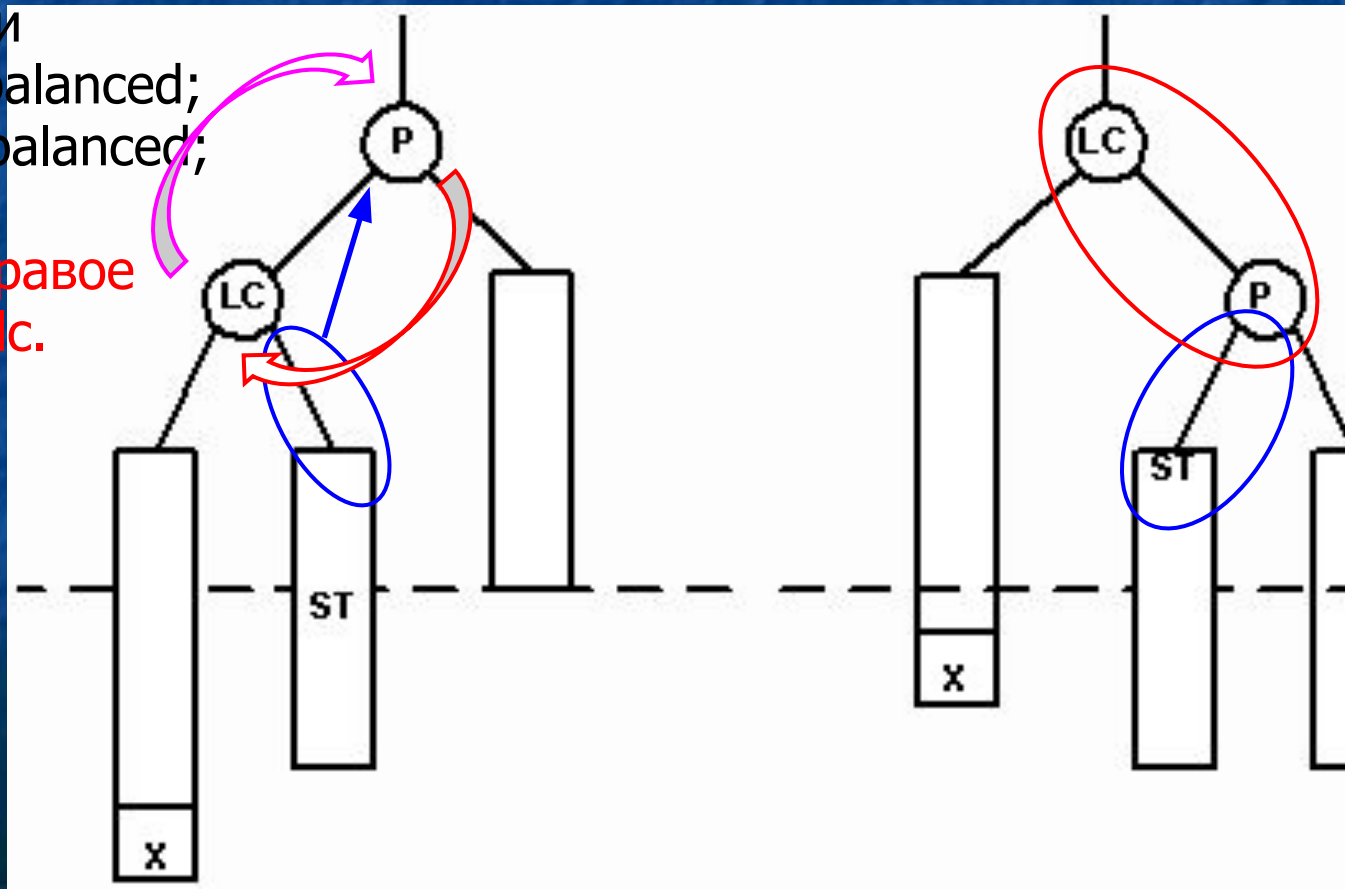
Одинарный поворот уравновешивает как родителя, так и его левого сына.


Исходное дерево
30
-1
15
0
40
0
10
0
20
0

Сбалансированные деревья. AVL-деревья.

Алгоритм AVL-вставки. Одинарный поворот.

```
template <class T>
void AVLTree<T>::SingleRotateRight
    (AVLTreeNode<T>* &p)
{
      AVLTreeNode<T> *lc;
  // установить lc на левое поддерево
  lc = p->Left();
  // скорректировать показатели
    сбалансированности
  p->balanceFactor = balanced;
  lc->balanceFactor = balanced;
  p->left = lc->Right();
  // переместить p в правое
      //поддерево узла lc.
      lc->right = p;
 // сделать lc новым
      //корнем.
  p = lc;
}
```



Алгоритм AVL-вставки. Двойной поворот.

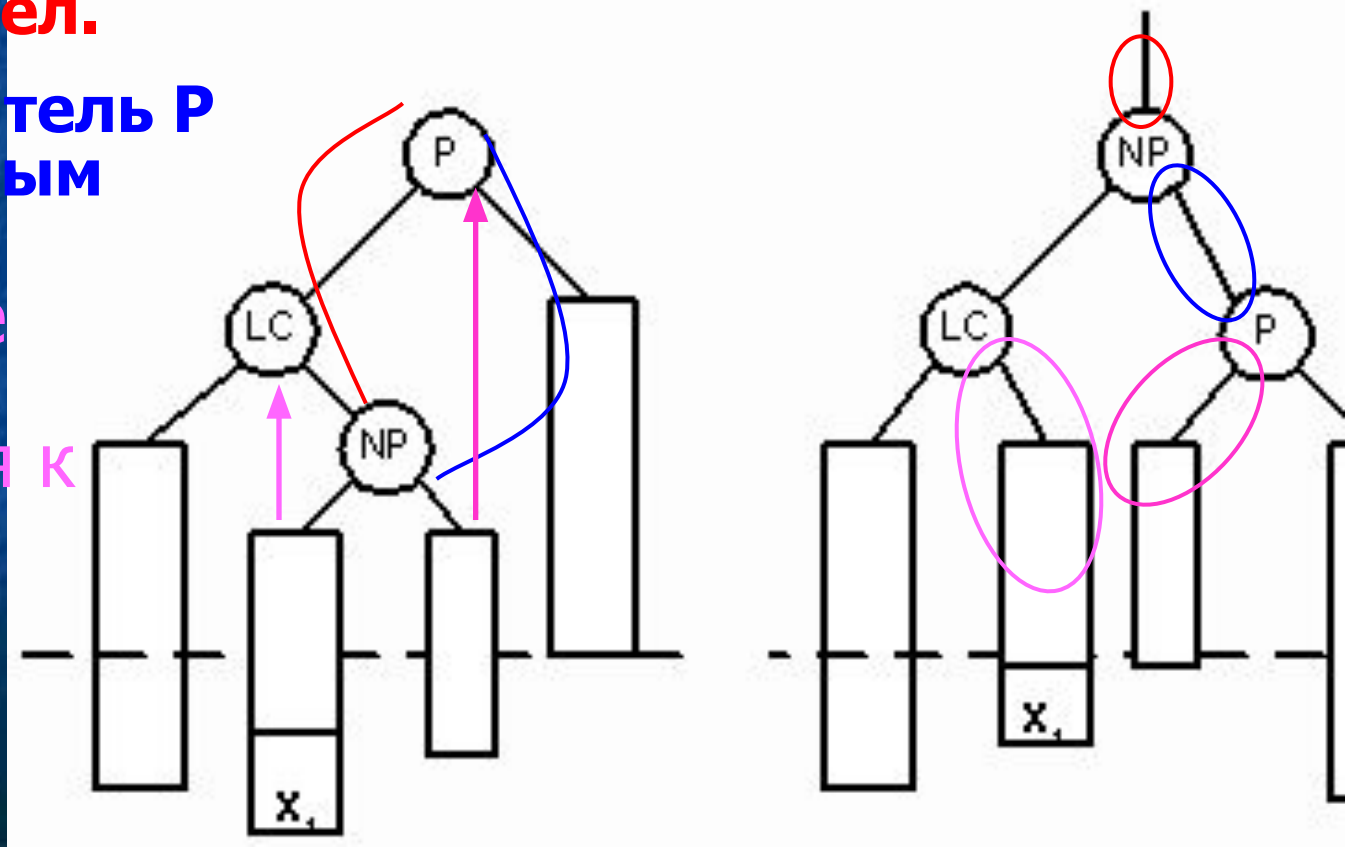
Двойной поворот вправо (double right rotation) нужен тогда, **когда родительский узел** (P) становится **перевешивающим влево**, а его **левый сын** (LC) **перевешивающим вправо**.

NP – корень правого перевешивающего поддерева узла LC. Тогда в результате поворота:

1. Узел NP замещает родительский узел.

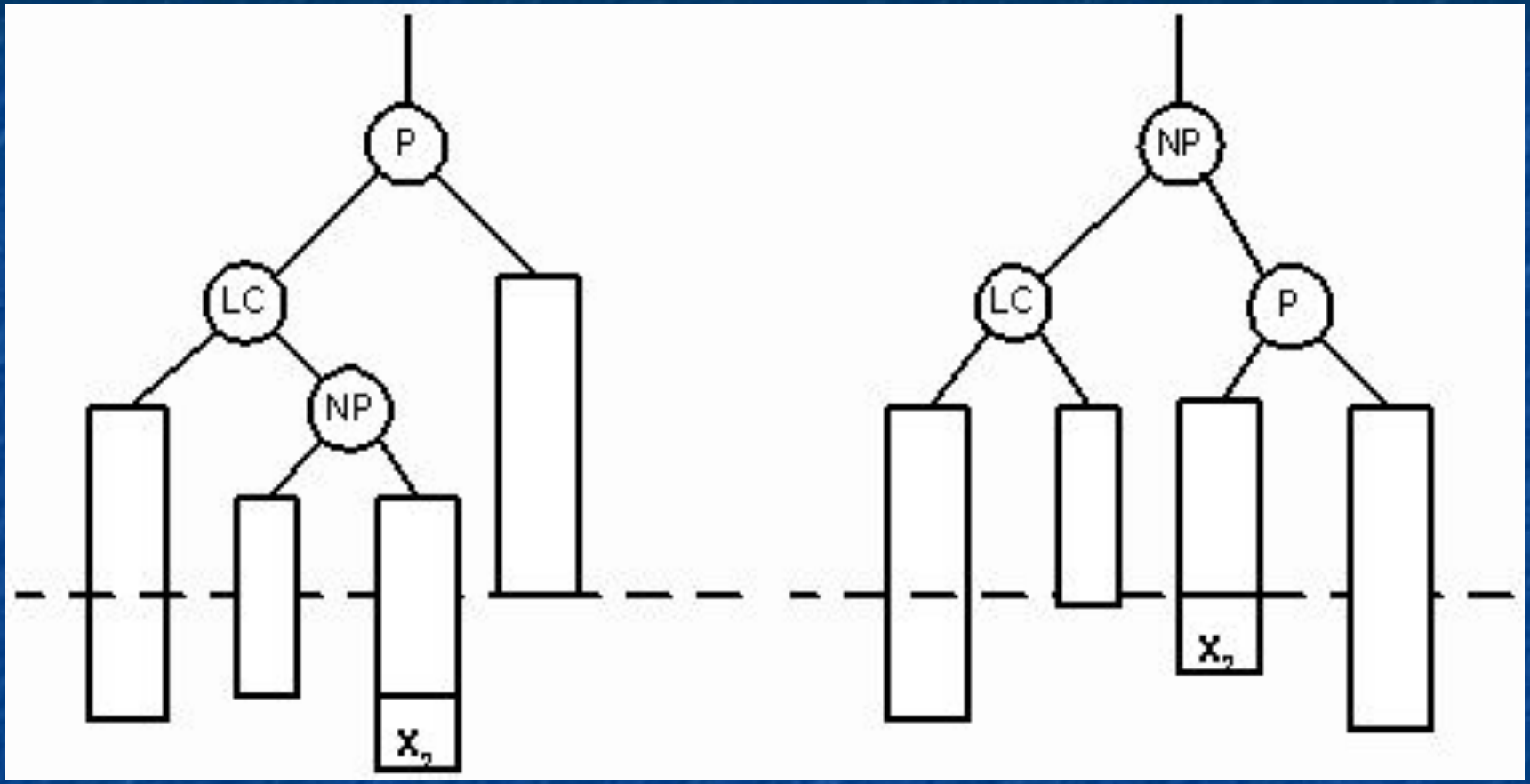
2. Бывший родитель P становится правым сыном NP .

3. Осиротевшие поддеревья NP распределяются к **LC** и **P** .

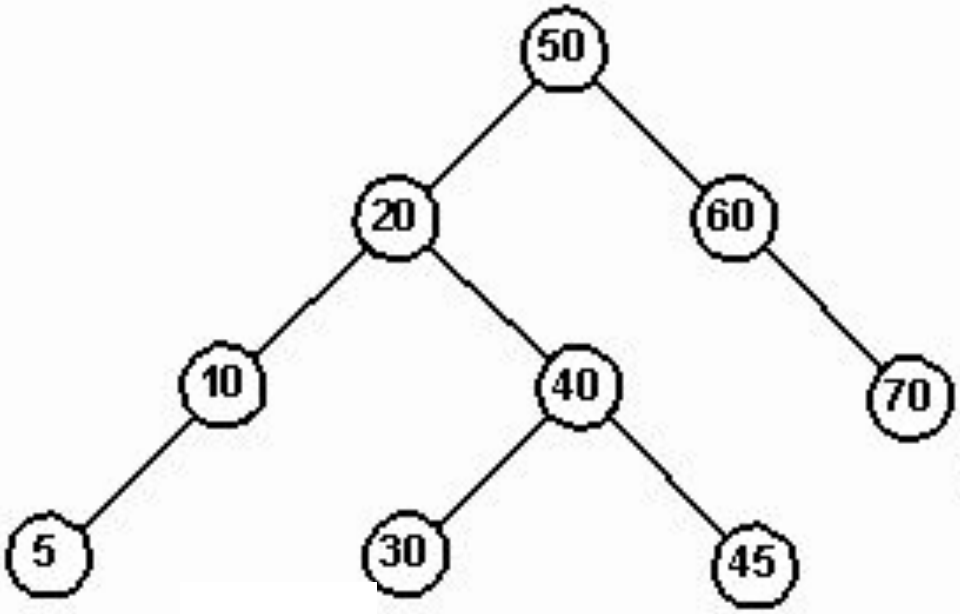


Сбалансированные деревья. AVL-деревья.

Алгоритм AVL-вставки. Двойной поворот.



До включения узла 25



Оценка сбалансированных деревьев

AVL-деревья требуют **дополнительных затрат** на поддержание сбалансированности при вставке или удалении узлов(повороты требуются примерно в половине случаев вставок и удалений).

Если в дереве **постоянно происходят вставки и удаления** элементов, эти операции могут **значительно снизить** быстродействие.

С другой стороны, если данные превращают **бинарное дерево поиска в вырожденное**, то **теряется поисковая эффективность**.

Таким образом**, применение AVL-деревьев целесообразно в тех случаях, когда поиск является доминирующей операцией.**

В таких случаях сначала происходит построение дерева, а потом производится поиск по этому дереву с небольшим количеством изменений.

Доступ к данным во внешней памяти. B-деревья.

До сих пор задача поиска решалась в предположении, что данные хранятся во внутренней(оперативной) памяти

Если структуры данных, например, **бинарное дерево поиска** или **AVL-дерево**, **расположены** не в оперативной памяти, а **на диске**, то **они становятся малоэффективными**.

При работе с данными, хранящимися на внешних носителях, целесообразно сокращать количество обращений к носителю.

Если предположить, что в одном узле дерева, хранящегося на диске, будет расположен не один, а несколько элементов – узел в этом случае называется **страницей дерева** и за одно обращение к диску будет считываться целая страница, то число обращений к диску при поиске существенно уменьшится.

Например, при $N = 1$ млн. число обращений к обычному дереву равно примерно 20 ($\log_2 10^6 \approx 20$), а при постраничном чтении, например по 7 элементов, число обращений сократится в 3 раза, что ускорит поиск в 3 раза.

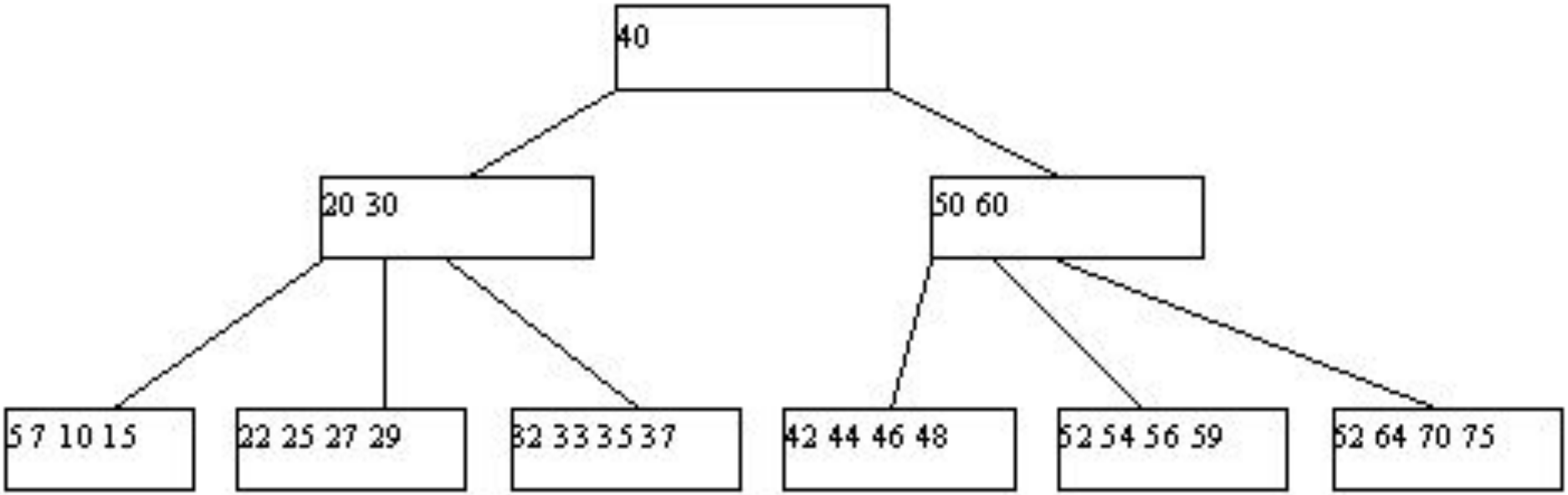
Доступ к данным во внешней памяти. B-деревья.

В 1970 году Р. Байер и Маккрейт предложили структуру данных, названную **B-деревьями**, позволяющую производить поиск по большому дереву, расположенному во внешней памяти.

Определение: Б-деревом **порядка n** называется динамическая структура обладающая следующими свойствами:

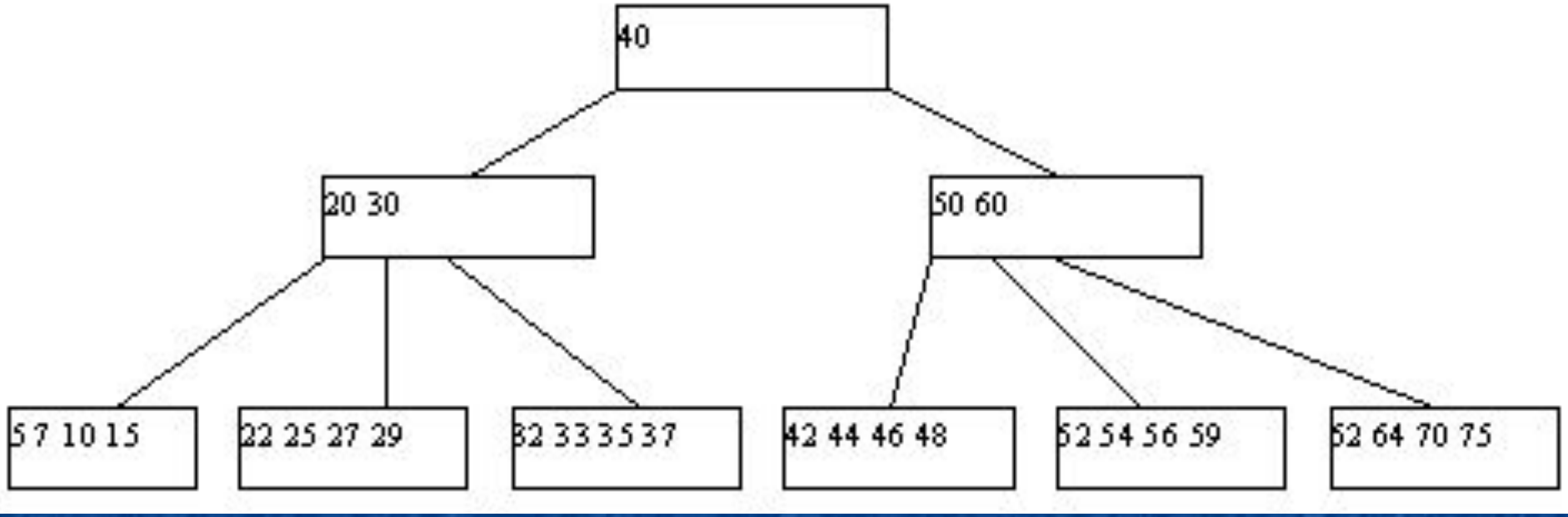
1. Каждая страница имеет **не более 2n** элементов
2. Каждая страница, кроме корневой, имеет **не менее n** элементов. Корневая страница может иметь от 1 до 2n элементов
3. Каждая страница является либо листовой, либо содержит m+1 потомков, где m - число элементов на этой странице
4. Все листья находятся на одном уровне

Доступ к данным во внешней памяти. B-деревья.



1. Каждая страница имеет **не более 2n** элементов
2. Каждая страница, кроме корневой, имеет **не менее n** элементов. Корневая страница может иметь от 1 до 2n элементов
3. Каждая страница является либо листовой, либо содержит m+1 потомков, где m - число элементов на этой странице
4. Все листья находятся на одном уровне

Доступ к данным во внешней памяти. B-деревья.

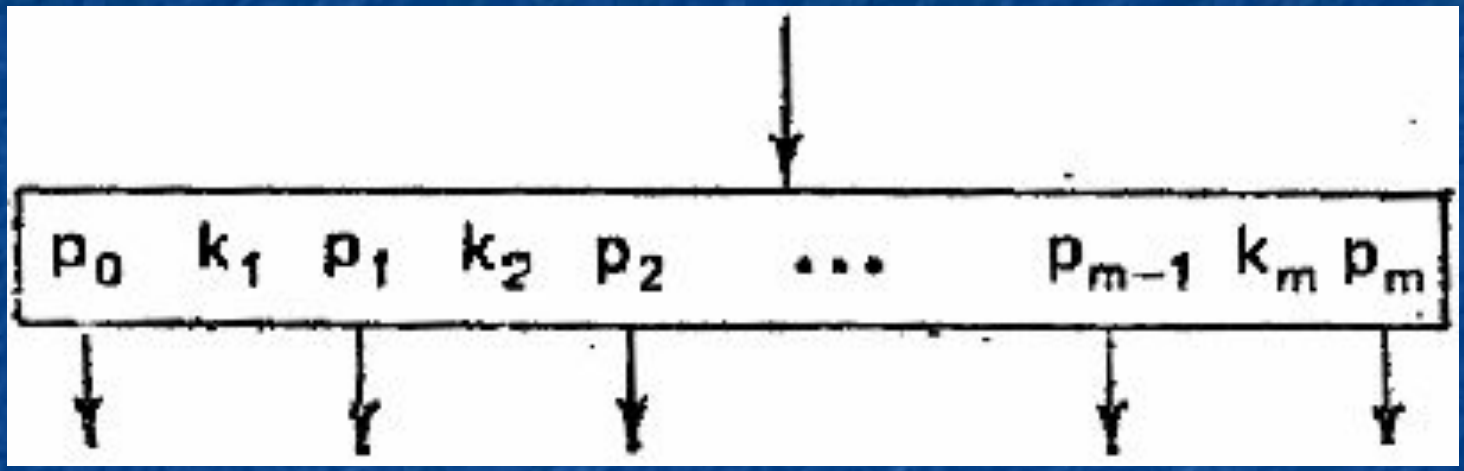


Элементы на страницах B-деревьев располагаются в возрастающем порядке. Если спроецировать Б-дерево на один-единственный уровень, включая потомков между ключами их родительской страницы, то ключи идут в возрастающем порядке слева направо.

Память в B-дереве всегда используется минимум на 50%, так как страницы всегда заполнены как минимум наполовину!

Алгоритм поиска элемента в B-дереве

Пусть задан некоторый аргумент поиска **x** и страница имеет вид:



Если страница уже считана в оперативную память и можно воспользоваться обычными методами поиска среди ключей k_i ... k_m.

Если поиск элемента **x** на странице неудачен, то мы попадаем в одну из следующих ситуаций:

1. $k_i < x < k_{i+1}$. Поиск продолжается на странице p_i.
2. $k_m < x$. Поиск продолжается на странице p_m.
3. $x < k_1$. Поиск продолжается на странице p_0.

Если m достаточно большое, то это может быть двоичный поиск, если же оно мало, то можно воспользоваться простым последовательным перебором.

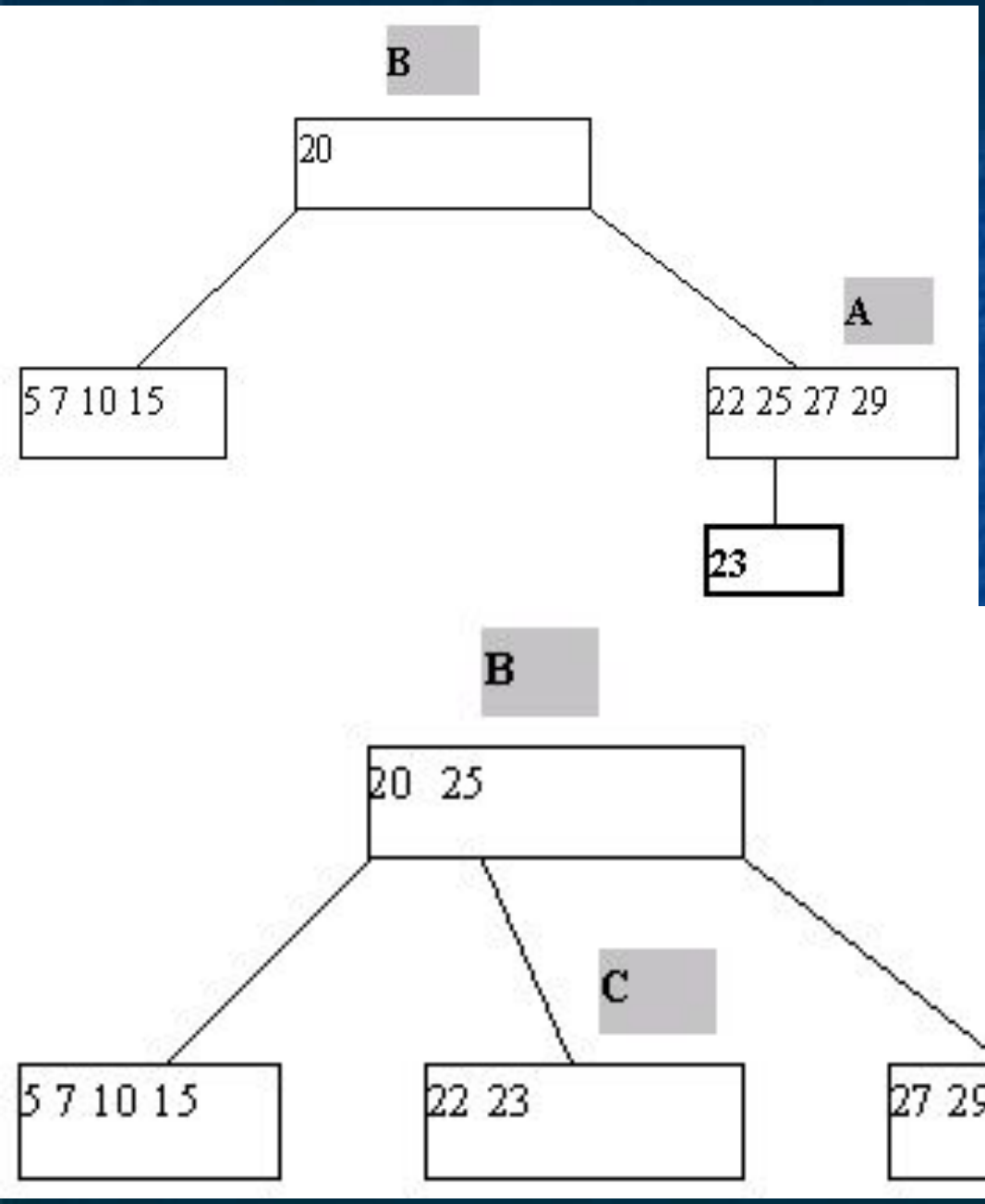
Включение элемента в B-дерево

Включение в B-дерево проводится **только в листовые страницы**. При этом поиск листовой страницы для вставки элемента происходит согласно алгоритма поиска в B-дереве.

Если новый элемент нужно поместить на страницу с $m < 2n$ элементами, то процесс включения элемента затрагивает лишь одну страницу B-дерева.

Включение в заполненную страницу затрагивает структуру дерева и может привести к появлению новых страниц.

Доступ к данным во внешней памяти. В-деревья.

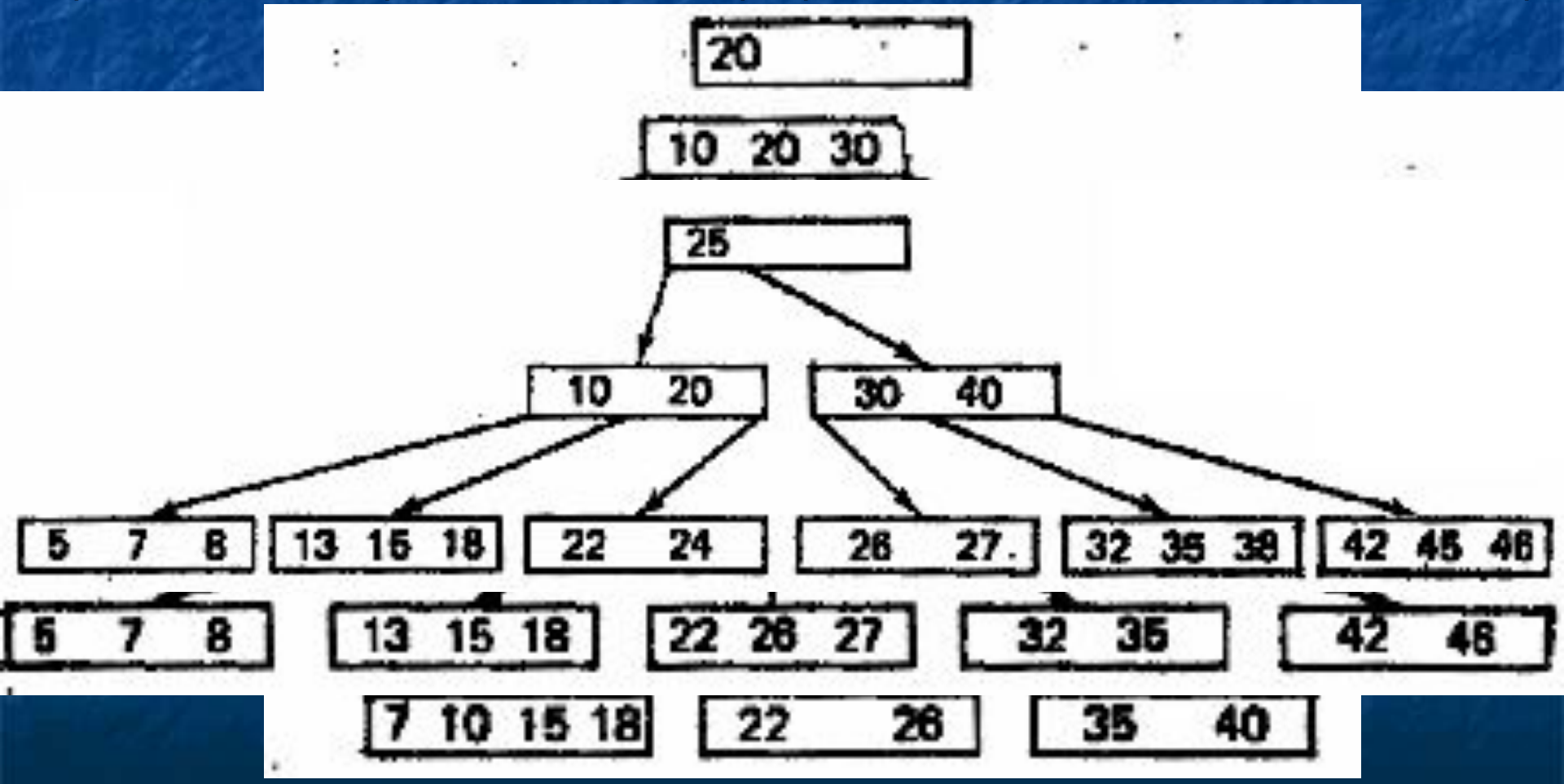


Включение элемента в В-дерево

1. Обнаруживается, что ключ 23 отсутствует. Включение в страницу А невозможно, поскольку она уже заполнена.
2. Страница А **разделяется** на две страницы (вводится новая страница С).
3. Ключи — их всего 2n+1— поровну распределяются в С и А, а **серединный** ключ переносится наодин уровень вверх на «родительскую» страницу.

Включение элемента в B-дерево

Процесс добавления элементов в Б-дерево представлен на рисунке, причем включаемые ключи идут в таком порядке: 20; 40 10 30 15; 35 7 26 18 22; 5; 42 13 46 27 8 32 38 24 45 25;



Исключение элемента из B-дерева

При **исключении** элемента из B-дерева можно выделить два случая:

1. Исключаемый элемент находится на листовой странице.
2. Элемент находится не на листовой странице.

В первом случае, удаление элемента с листовой страницы не вызовет затруднений, если эта страница содержит более **n** элементов.

Если же **количество элементов на странице после удаления станет меньше n**, то **необходимо проделать некоторые действия, чтобы предотвратить нарушение второго условия B-дерева (**m>=n).

В этом случае для текущей страницы **необходимо позаимствовать элемент с одной из соседних страниц**.

Если соседняя страница содержит более n элементов, то происходит перемещение крайнего элемента (ближнего к текущей странице) в родительскую страницу, а элемент из родительской страницы в текущую.

Исключение элемента из B-дерева

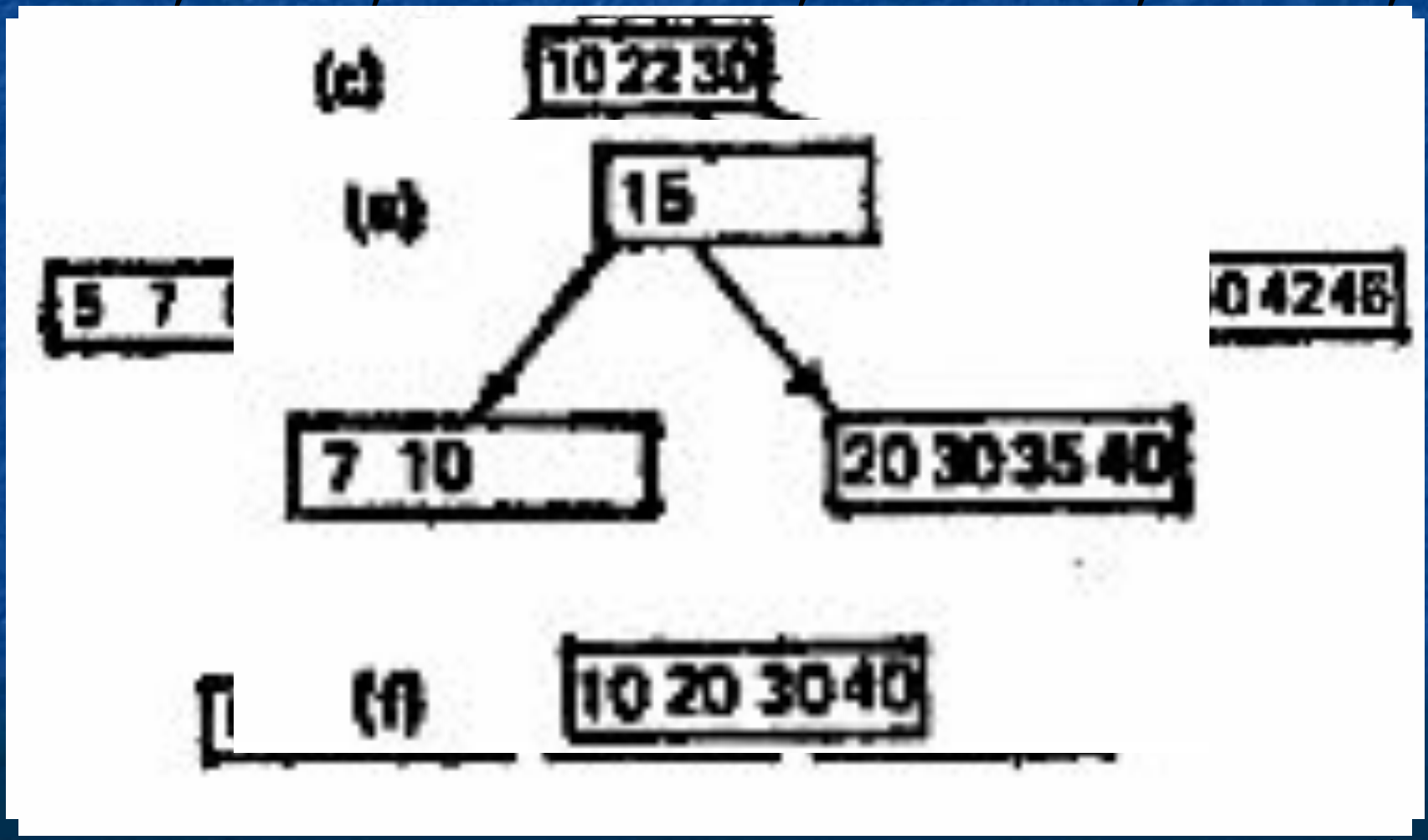
В том случае, если занять элемент на соседних страницах невозможно (они содержат по n элементов), выполняется процедура **слияния** двух страниц.

Так как общее число элементов на двух соседних страницах после удаления становится равным 2n-1, то, забирая элемент с родительской страницы, получим одну страницу содержащую 2n элементов. При этом вторая освободившаяся страница уничтожается. Процесс слияния страниц в точности обратен процессу разделения страницы.

Исключение элемента из B-дерева

Рассмотрим деградацию Б-дерева в случае удаления ключей в последовательности:

25 45 24; 38 32; 8 27 46 13 42; 5 22 18 26; 7 35 15;



Программная реализация B-дерева.

```
//задаем порядок дерева
#define LEVEL 3
//количество элементов на странице
#define ITEM_ON_LEVEL 2*LEVEL
//задаем максимальное количество страниц
#define MAX_PAGES 2000
//описываем структуру хранящую информацию о
//занятости страниц в файле
struct file_info{
    //отводим по 1 байту для описания состояния страниц
    //пока 1 -занята 0 - свободна
    unsigned char m_state[MAX_PAGES];
    //храния смещение относительно начала файла для
корневой страницы
    int m_irootoffset;
    //смещение начала записи этой структуры в файл
    int offset;
};
```

Программная реализация B-дерева.

```
template <class DATA>
class CBTreePage
{
public:
    //массив элементов на странице
    DATA m_arkeys[ITEM_ON_LEVEL];
    //массив смещений на доцерние страницы
    int m_arpages[ITEM_ON_LEVEL+1];
    //текущее кол-во элементов на странице
    int m_ikeys;
    //конструктор по умолчанию
    CBTreePage();
    //метод записи страницы в файл
    bool SavePage(fstream &file,int offset);
    //метод загрузки страницы из файл
    bool LoadPage(fstream &file,int offset);
    friend class CBTree<DATA>;
};
```

Программная реализация B-дерева.

```
template <class DATA>
class CBTree
{
private:
    //смещение корневой страницы
    int m_iroot;
    //максимальное смещение занимаемое страницами дерева
    int max_offset;
    //структура описывающая файл текущего дерева
    file_info m_finfo;
    // колличество страниц
    int NumberOfPages;
    //используются public методом AddItem
    bool Add(int &root,DATA &item, int &up_page, DATA &up_item,bool
&split);
    void SplitPage(CBTreePage<DATA> &page, int spot, DATA &up_item, int
&up_page);
    //используются public методом DeleteItem
    bool DeleteFromPage(int page,DATA &item,bool &too_small);
    void SwapItem(CBTreePage<DATA> &root,int key_pos, int child,
bool&too_small);
    void TooSmall(CBTreePage<DATA> &parent, int child, int child_num
bool &too_small);
```

Хеширование данных. Хеш-таблицы.

Для ускорения доступа к данным в таблицах баз данных можно использовать предварительное упорядочивание таблицы в соответствии со значениями ключей.

Потери времени на повторное упорядочивание таблицы могут значительно превышать выигрыш от сокращения времени поиска. Поэтому для сокращения времени доступа к данным в таблицах часто используется так называемое **случайное упорядочивание или *хеширование***.

При этом данные организуются в виде таблицы при помощи ***хеш*-функции** h, используемой для "вычисления" адреса по значению ключа.

Хеш-таблицы - один из наиболее эффективных способов реализации словарей (словарь - абстрактный тип данных (множеств) с операторами вставки, удаления и проверки членства INSERT, DELETE и MEMBER).

Идеальной хеш-функцией является такая хеш-функция, которая для любых двух неодинаковых ключей дает неодинаковые адреса:

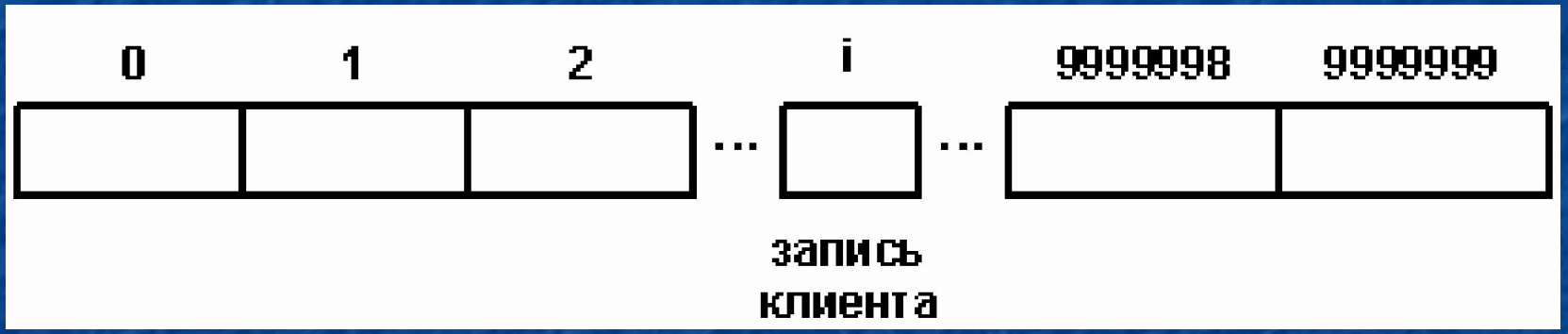
$$k1 \neq k2 \Rightarrow h(k1) \neq h(k2)$$

Такая организация данных носит название "совершенное хеширование".

В случае заранее неопределенного множества значений ключей и ограниченной длины таблицы подбор совершенной функции затруднителен.

Ключи и хеш-функция

В большинстве приложений ключ обеспечивает косвенную ссылку на данные. Ключ отображается во множество целых чисел посредством **хеш-функции** (hash function). Полученное в результате значение затем используется для доступа к данным.



Предположим, Key – положительное целое, а HF(Key) – значение младшей цифры числа Key. Тогда диапазон индексов равен 0-9.
Например, если Key = 49, HF(Key) = HF(49) = 9.
Эта хеш-функция в качестве возвращаемого значения использует остаток от деления на 10.

```
// Хеш-функция, возвращающая младшую цифру ключа
int HF(int key){  return key % 10;}
```

Коллизии и методы разрешения коллизий

Очевидно, что при заполнении хеш-таблицы могут возникать ситуации, когда для двух неодинаковых ключей функция вычисляет один и тот же адрес. Данный случай носит название "**коллизия**", а такие ключи называются "**ключи-синонимы**".

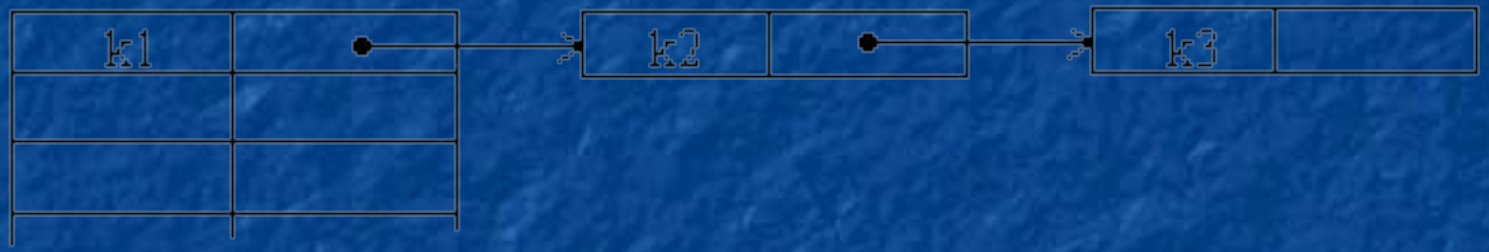
Для разрешения коллизий используются различные методы, которые в основном сводятся к методам "**цепочек**" и "**открытой адресации**".

Коллизии и методы разрешения коллизий

Коллизии и методы разрешения коллизий

Поиск в хеш-таблице с цепочками переполнения:

1. вычисляется адрес по значению ключа.
2. осуществляется последовательный поиск в списке, связанном с вычисленным адресом.



$k1 \neq k2$

$h(k1)=h(k2)$

$k3 \neq k1$

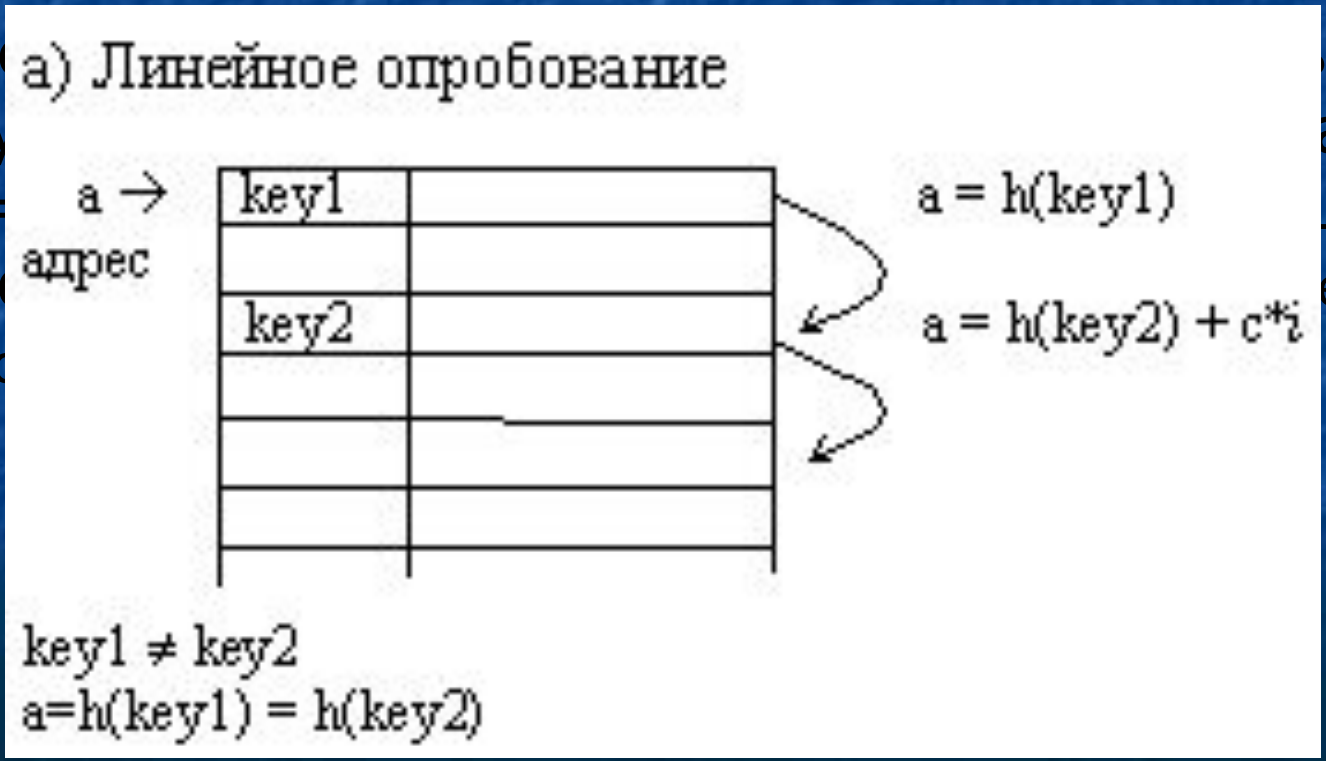
$h(k3)=h(k1)$

Процедура удаления из таблицы сводится к поиску элемента и его удалению из цепочки переполнения.

Коллизии и методы разрешения коллизий

Методы открытой адресации состоят в том, чтобы, пользуясь каким-либо алгоритмом, обеспечивающим перебор элементов таблицы, просматривать их в поисках свободного места для новой записи. Иногда данный метод хеширования называют ***закрытым, внутренним,*** или **прямым *хешированием***.

Линейнному переборуданным шагом a=шить коллизиельный перебор во



Коллизии и методы разрешения коллизий

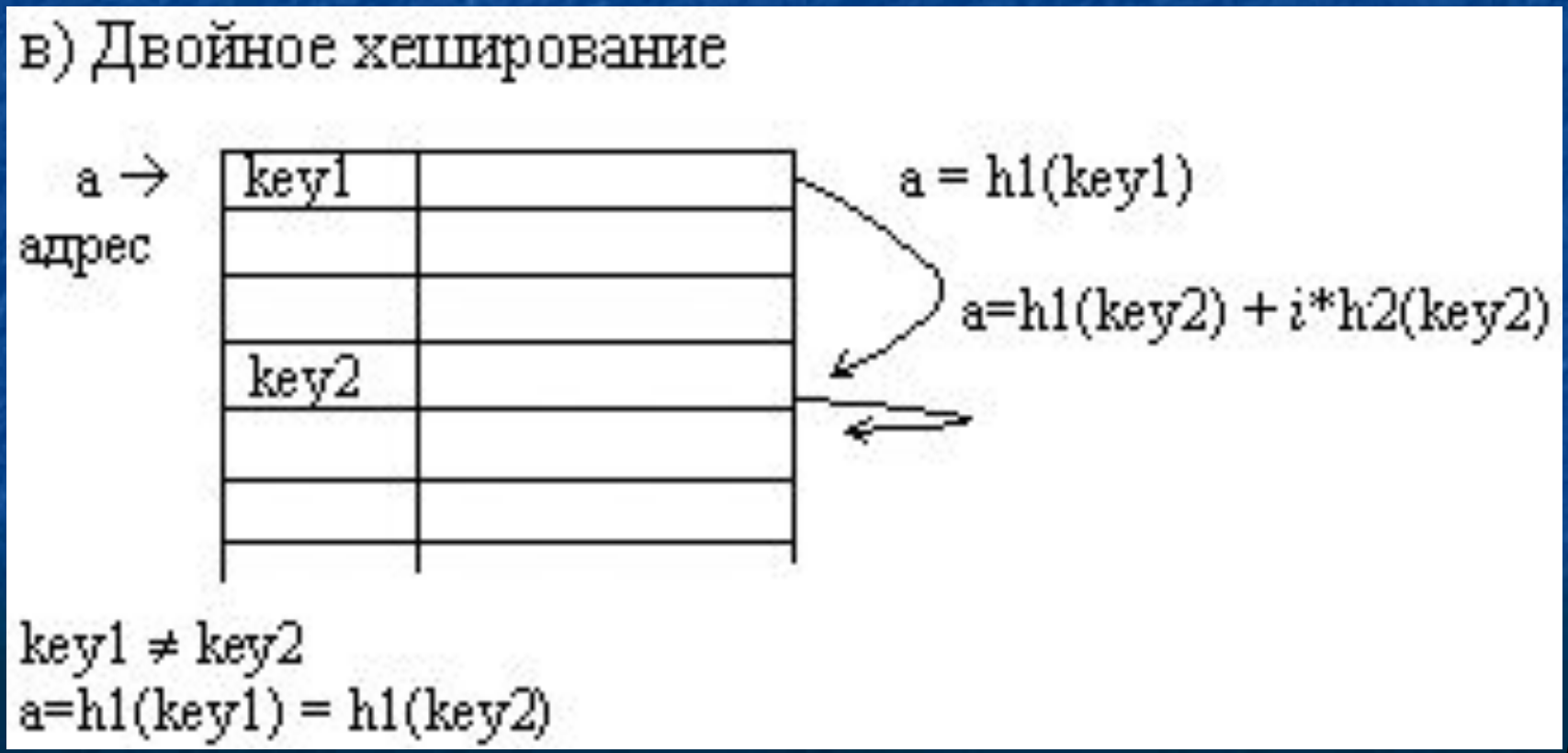
Квадратичное опробование отличается от линейного тем, что шаг перебора элементов не линейно зависит от номера попытки найти свободный элемент:

$$a = h(key2) + c*\mathbf{i} + d*\mathbf{i}^2.$$

Благодаря нелинейности такой адресации уменьшается число проб при большом числе ключей-синонимов.

Коллизии и методы разрешения коллизий

Еще одна разновидность метода открытой адресации, которая называется **двойным хешированием**, основана на нелинейной адресации, достигаемой за счет суммирования значений основной и дополнительной хеш-функций:
a=h1(key) + **i***h2(key).



Коллизии и методы разрешения коллизий

Алгоритмы вставки и поиска для метода линейного опробования:

Вставка:

i = 0

a = h(key) + i*c

Если t(a) = свободно, то t(a) = key, записать элемент, *стоп элемент добавлен*

i = i + 1, перейти к шагу 2

Поиск:

i = 0

a = h(key) + i*c

Если t(a) = key, то *стоп элемент найден*

Если t(a) = свободно, то *стоп элемент не найден*

i = i + 1, перейти к шагу 2

Коллизии и методы разрешения коллизий

С процедурой **удаления** дело обстоит не так просто, так как она в данном случае не будет являться обратной процедуре вставки.

В том случае, **если за удаляемым элементом** в результате разрешения коллизий **были размещены элементы** с другими ключами, **то поиск этих элементов после удаления** всегда **будет давать отрицательный результат**, так как алгоритм поиска останавливается на первом элементе, находящемся в состоянии свободно.

Скорректировать эту ситуацию можно различными способами. Наилучший из них состоит в том, что для элементов хеш-таблицы **добавляется состояние "удалено"**. Данное состояние **в процессе поиска интерпретируется, как занято**, а в **процессе добавления - как свободно**.

Коллизии и методы разрешения коллизий

Сформулируем алгоритмы вставки, поиска и удаления для хеш-таблицы, имеющей три состояния элементов.

Вставка:

i = 0
a = h(key) + i*c
Если t(a) = свободно или t(a) = удалено, то t(a) = key, записать элемент, *стоп элемент добавлен*
i = i + 1, перейти к шагу 2

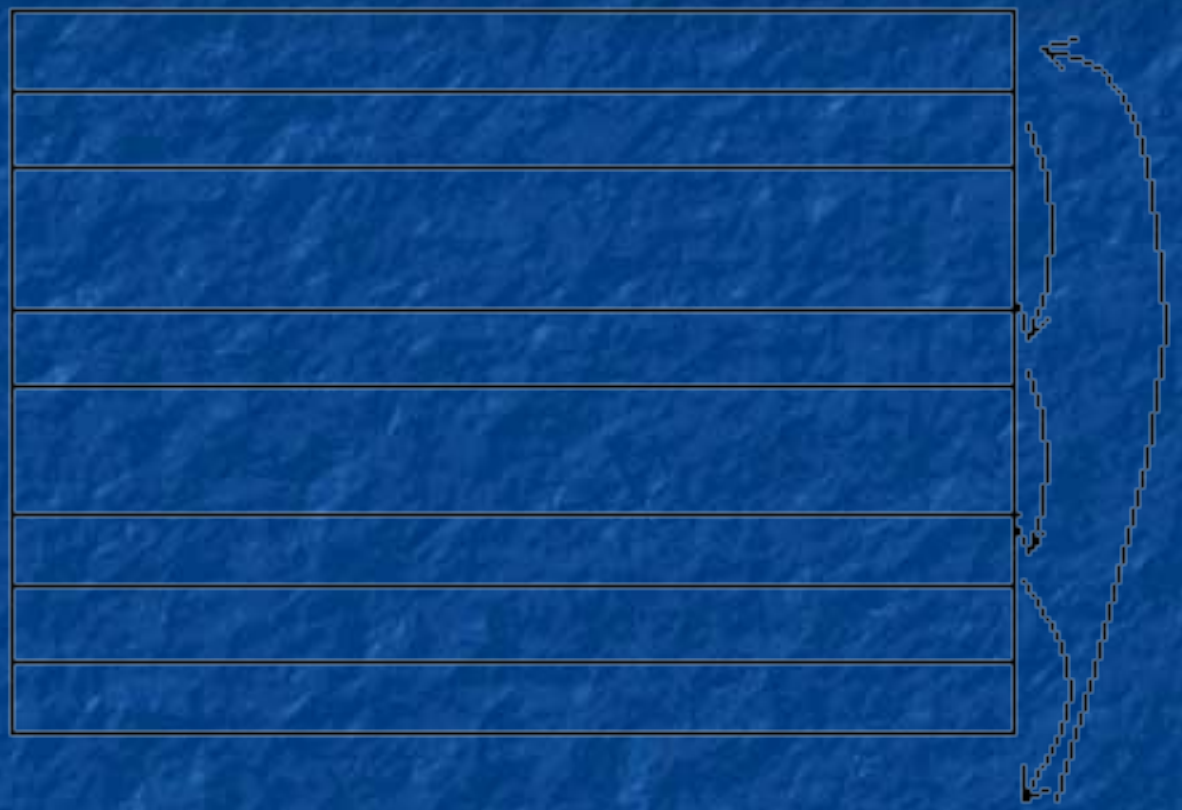
Удаление:

i = 0
a = h(key) + i*c
Если t(a) = key, то t(a) = удалено, *стоп элемент удален*
Если t(a) = свободно, то *стоп элемент не найден*
i = i + 1, перейти к шагу 2

Поиск:

i = 0
a = h(key) + i*c
Если t(a) = key, то *стоп элемент найден*
Если t(a) = свободно, то *стоп элемент не найден*
i = i + 1, перейти к шагу 2

Переполнение таблицы и рехеширование



Оценка качества хеш-функции