Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Институт информационных технологий

Кафедра информационных систем и технологий

А. И. Парамонов, Д. С. Потоцкий, А. Г. Савенко

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве учебно-методического пособия для специальности 6-05-0612-01 «Программная инженерия»

Минск БГУИР 2026

УДК 004.421(076)
ББК 32.973.3я73
П18

Р е ц е н з е н т ы:

кафедра «Программное обеспечение информационных систем и технологий»
Белорусского национального технического университета
(протокол № 7 от 12.02.2025);

заведующий кафедрой дискретной математики и алгоритмики
ФПМИ Белорусского государственного университета
доктор физико-математических наук, профессор В. М. Котов

Парамонов, А. И.

П18 Алгоритмы и структуры данных : учеб.-метод. пособие / А. И. Парамонов, Д. С. Потоцкий, А. Г. Савенко. – Минск : БГУИР, 2026. – 108 с. : ил.
ISBN 978-985-543-845-9.

Предназначено для студентов специальности «Программная инженерия». Содержит краткий теоретический материал, лабораторные и контрольные работы, а также примеры их выполнения. Можно использовать для самостоятельной работы студентов указанной специальности очной, заочной и дистанционной форм обучения. Будет полезно студентам всех специальностей профиля образования 06 «Информационно-коммуникационные технологии» и специалистам в области информационных технологий.

УДК 004.421(076)
ББК 32.973.3я73

ISBN 978-985-543-845-9

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Плохие программисты думают о коде.
Хорошие программисты думают
о структурах данных и их взаимосвязях.
Линус Торвальдс, создатель Linux

Учебно-методическое пособие написано согласно учебному плану специальности 6-05-0612-01 «Программная инженерия» и в соответствии с программой учебной дисциплины «Алгоритмы и структуры данных».

Оно направлено на формирование у студентов навыков алгоритмического мышления и представления о многообразии компьютерных структур данных, способов описания объектов и алгоритмизации процессов различных предметных областей.

Современные методы программирования включают в себя различные варианты структурирования данных и для эффективного их использования следует учитывать многие факторы. Поскольку освоение специальности «Программная инженерия» направлено на формирование компетенций и навыков, связанных с умелым использованием принципов информатики и компьютерных наук для их сочетания с инженерными подходами, разработанными для материального производства сложных корпоративных компьютерных систем, то одной из целей учебно-методического пособия обозначена выработка у студентов навыков выбора структур данных для повышения производительности их обработки и большей эффективности построенных на их основе программных средств.

В данном учебно-методическом пособии содержатся: краткое введение в многообразие структур данных, лабораторный практикум и контрольные работы. Каждый раздел содержит краткие теоретические сведения по теме, задания с вариантами и с подробным разбором примера, а также контрольные вопросы для самоподготовки.

Данное издание ставит целью ознакомить студентов с основными типами структур данных, основными алгоритмами обработки структур данных и возможностями современных языков программирования высокого уровня для эффективной организации данных, а также помочь им приобрести практические навыки оценивания эффективности алгоритмов обработки структур данных различных типов и владения методами построения статических и динамических структур данных.

Учебно-методическое пособие ориентировано в первую очередь на студентов заочной формы получения общего высшего образования, интегрированного с образовательными программами среднего специального образования, по специальности 6-05-0612-01 «Программная инженерия».

Оно может быть полезно студентам всех специальностей группы «Разработка программного обеспечения», а также может быть использовано для профессиональной ориентации в ходе подготовки к освоению квалификации инженера-программиста.

ВВЕДЕНИЕ В СТРУКТУРЫ ДАННЫХ

Структуры данных представляют собой некоторую абстракцию каких-либо типов данных языка программирования высокого уровня, выраженную в том числе в способе их организации в памяти вычислительной машины.

Между понятиями «типы данных» и «структуры данных» есть как определенная взаимосвязь, так и различия. Так, например, некоторые типы данных типизированных языков программирования высокого уровня одновременно являются и структурами данных (массивы, строки, записи и др.). Также тип данных, как и некоторые структуры данных, определяет способ хранения данных (выделение памяти, представление, множество допустимых значений) и допустимые операции над этими данными. Различие же заключается в допустимых операциях над значениями определенного типа данных и над структурами этих данных. Операции, выполняемые над данными определенных типов, конкретны, определяются самим типом (например, арифметические, логические, сравнения и т. д.) и имеют операторы для их выполнения. Операции, выполняемые над структурами данных, более абстрактны, как правило, не зависят от типа данных объектов и в основном связаны с модификацией организации данных в структуре (добавление, удаление, сортировка, поиск).

Структура данных – множество элементов данных и множество связей между ними.

Структура данных – программная единица, позволяющая хранить и обрабатывать множество однотипных и/или логически связанных данных с помощью вычислительной техники.

Таким образом, структура данных в общем случае взаимосвязана с архитектурой вычислительной машины и является ее отображением (рисунок 1).

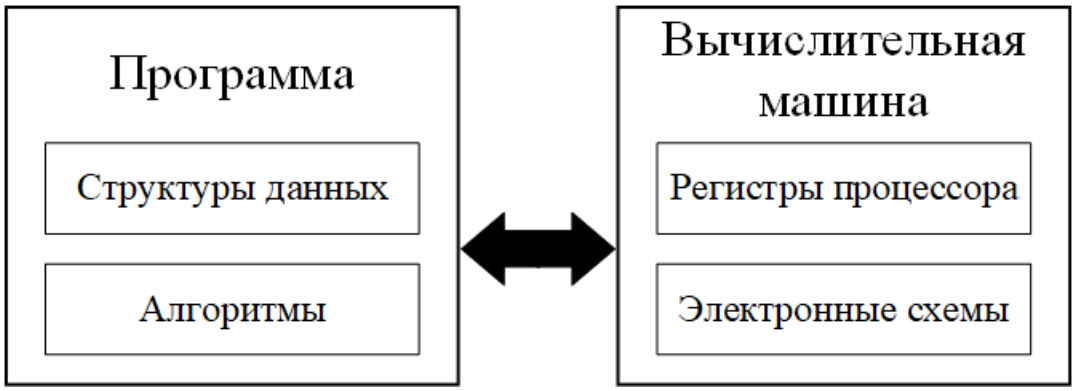


Рисунок 1 – Взаимосвязь архитектуры компьютера и структур данных

Кроме того, различают понятия физической и логической структур данных, которые также взаимосвязаны.

Физическая структура данных (внутренняя структура, структура хранения, структура памяти) – соответствует способу физического представления данных в памяти вычислительной машины.

Логическая (или абстрактная) структура данных – это структура данных, которая рассматривается без учета ее представления в машинной памяти.

Структуры данных предназначены для организации более эффективного хранения и обработки данных. Правильно выбранная для решения определенной задачи структура данных позволяет минимизировать вычислительные мощности (затраты памяти, процессорное время и т. д.).

Классификация структур данных

Структуры данных имеют несколько различных классификаций:

1 По сложности:

а) базовые структуры (простые) – неделимые на составные части;

б) интегрированные структуры (сложные) – состоят из других структур данных.

2 По связи между элементами:

а) связанные (например, списки);

б) несвязанные (например, массивы, векторы, строки, стеки, очереди).

3 По упорядоченности и расположению элементов в памяти:

а) линейные структуры:

– последовательные (например, массивы, векторы, строки, стеки, очереди);

– произвольные (например, односвязные и двусвязные списки);

б) нелинейные структуры (например, многосвязные списки, деревья, графы).

4 По изменчивости (количества элементов и связей):

а) статические;

б) полустатические;

в) динамические.

Обобщенная классификация структур данных представлена на рисунке 2.

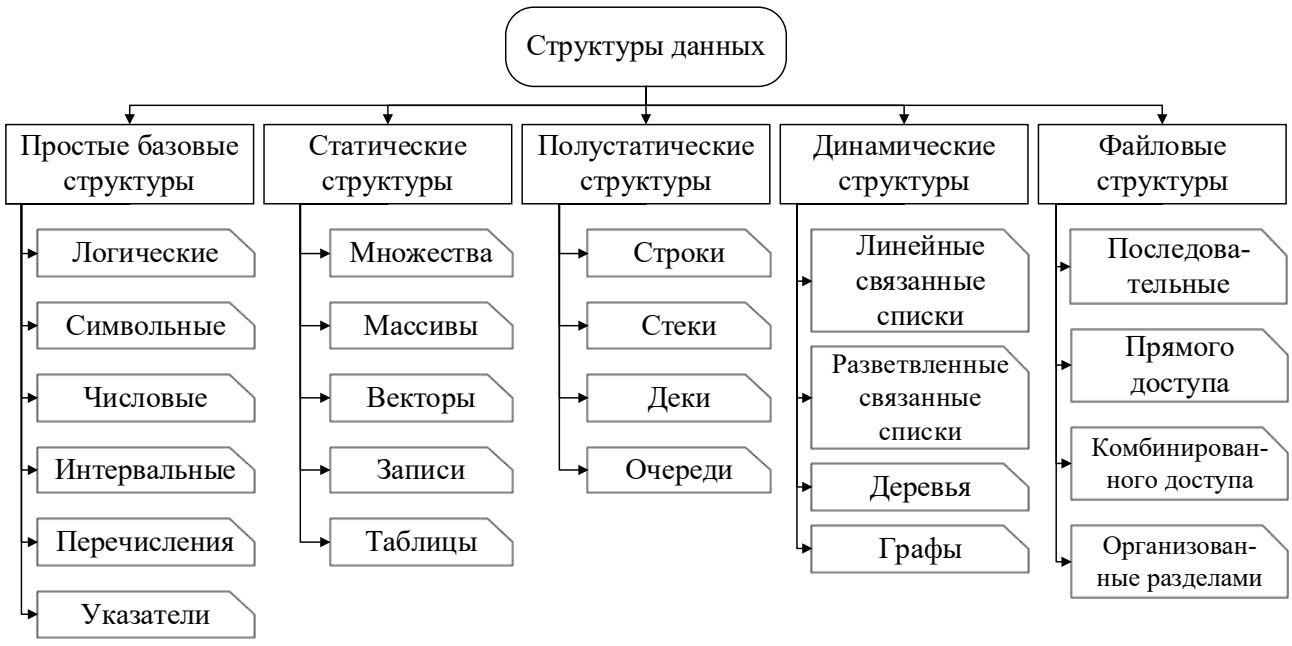


Рисунок 2 – Обобщенная классификация структур данных

Операции над структурами данных

Над структурами данных допустимы следующие операции:

1 Создание – выделяет память для структуры данных.

2 Уничтожение – освобождает память, выделенную для структуры данных.

3 Выбор (поиск) – обеспечивает доступ к данным внутри самой структуры.

4 Модификация (обновление) – позволяет изменять значения выбранных данных и их количество в структуре:

а) модификация значений данных;

б) добавление элементов данных в структуру;

в) удаление элементов данных из структуры;

г) изменение порядка (сортировка) элементов данных в структуре.

Основные структуры данных и их характеристики

1 Записи

Запись (record) – это составной тип данных (структура данных), состоящий из отдельных именованных элементов любых (и возможно разных) типов (за исключением функций), объединенных под одним идентификатором.

Поле записи – идентификатор отдельного элемента записи.

При разработке программ записи помогают в организации хранения и обработке сложных (разнородных) данных, не разобщая их по различным объектам, а группируя в одном.

Кроме того, элементы, входящие в запись, сами могут иметь тип записи, т. е. такие структуры могут быть вложенными.

Записи позволяют группу связанных между собой переменных использовать как множество отдельных элементов, а также как единое целое. Записи целесообразно использовать там, где необходимо объединить разнообразные данные, относящиеся к одному объекту.

В отличие от массива, который также является составным пользовательским типом данных, но при этом однородным объектом, запись может быть неоднородной, т. е. содержать данные разных типов.

Как и массив, запись представляет собой совокупность данных, но отличается способом обращения к элементам: в массиве обращение осуществляется по индексу элементов, а к полям (элементам) записи необходимо обращаться по имени (идентификатору поля).

Само по себе объявление записи не влечет за собой выделение памяти, по сути, это создание шаблона пользовательского типа данных. После объявления записи для дальнейшего использования такого созданного программистом типа данных необходимо определить переменную, имеющую тип созданной записи.

Описание структуры записи предоставляет компилятору необходимую информацию об элементах переменной-записи (структурной переменной) для резервирования места в оперативной памяти и организации доступа к ней при последующем определении переменной-записи и использовании отдельных элементов (полей) переменной-записи.

Операции над записями

Возможные операции над переменными-записями:

а) копирование. Копирование всех полей одной записи в другую может быть выполнено как поэлементно, так и целиком;

б) присваивание. Оператор присваивания выполняет то, что называется поверхностной копией в применении к переменным-записям.

Поверхностная копия представляет собой копирование бит за битом значений полей переменной-источника в соответствующие поля переменной-приемника. При этом может возникнуть проблема с такими полями, как указатели;

в) взятие адреса (оператор & в языках программирования C/C++);

г) осуществление доступа к элементам (полям) переменной-записи.

Над переменными-записями нельзя выполнять операцию сравнения (например, одной переменной-записи с другой).

2 Списки

Список (list) – это абстрактная структура данных, представляющая собой набор динамических элементов (чаще всего записей), связанных друг с другом каким-либо способом.

Динамические элементы, образующие список, как правило, имеют тип записи, поскольку помимо самого значения элемента (один тип данных), должны также иметь ссылку(-и) на другой(-ие) элемент(-ы) списка (другой тип данных – ссылочный). Кроме того, при работе со списками используются указатели на начало списка (первый элемент списка) и на текущий элемент списка.

Классификация списков представлена на рисунке 3.

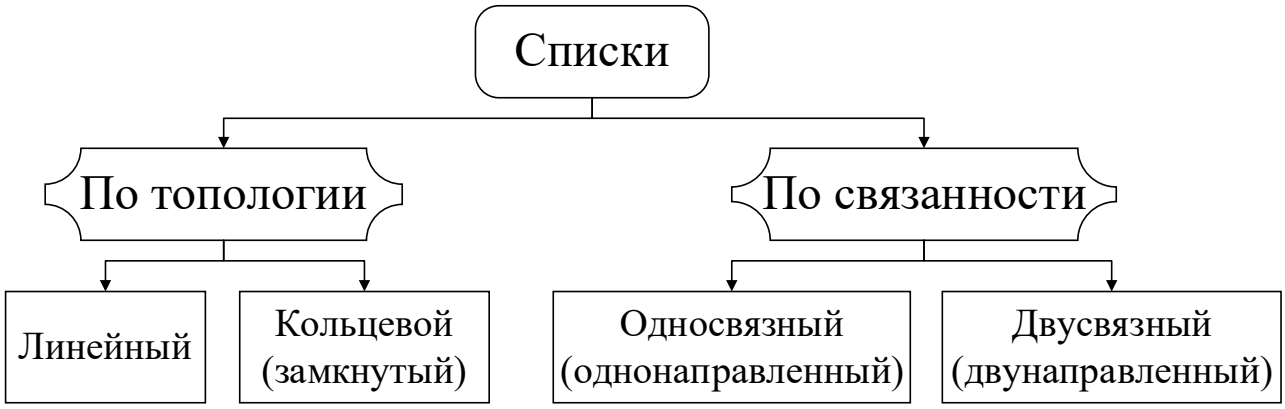


Рисунок 3 – Классификация списков

Операции, выполняемые над списками:

– поиск элемента в списке с заданным значением;

– добавление нового элемента (в начало или конец списка, до или после указанного узла);

– исключение определенного элемента из списка (не означает физическое удаление);

– упорядочивание (сортировка) элементов списка (для линейных списков);

– определение первого элемента (для линейных списков).

2.1 Односвязный (однонаправленный) линейный список – это структура данных, состоящая из однотипных элементов (узлов), каждый из которых содержит в себе значение элемента и ссылку на следующий элемент в списке.

Голова (head) односвязного списка – это самый первый элемент списка.

Хвост (tail) односвязного списка – это самый последний элемент списка, имеющий в качестве ссылки на следующий элемент значение, равное `NULL`.

Односвязный список проиллюстрирован на рисунке 4.

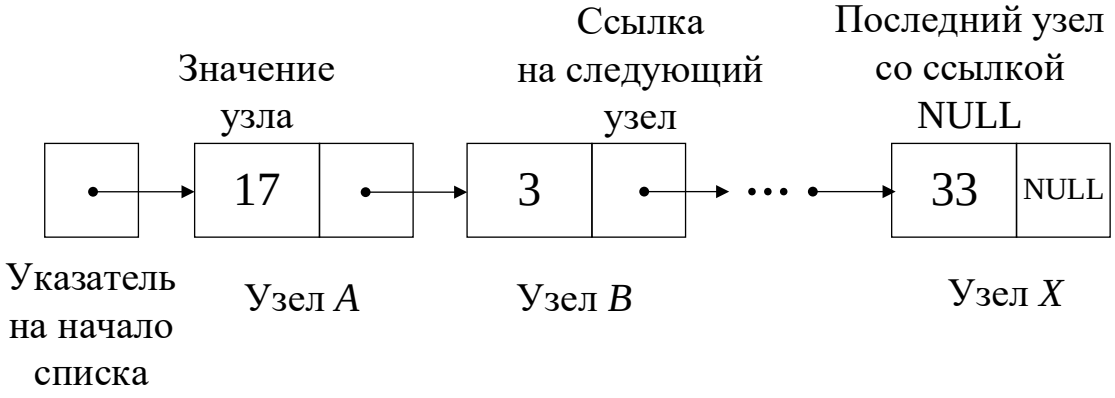


Рисунок 4 – Пример односвязного линейного списка

В памяти элементы односвязного линейного списка расположены хаотично (в отличие от последовательного расположения для массивов), поэтому в такой список можно добавлять произвольное число элементов. Однако доступ к элементам может осуществляться только последовательно (начиная с головы списка).

Добавление нового элемента в односвязный список и удаление элемента из него предполагает перезапись одной ссылки и проиллюстрировано на рисунке 5, *а* и *б* соответственно.

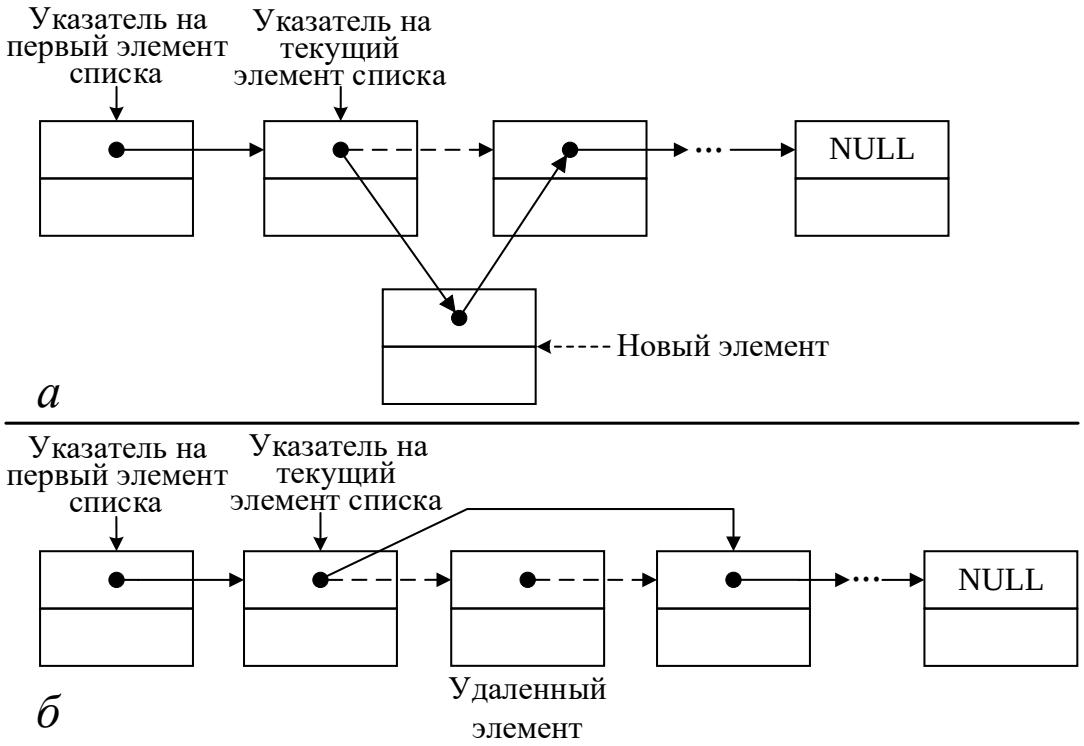


Рисунок 5 – Односвязный список: добавление (*а*) и удаление элемента (*б*)

2.2 Двусвязный (двунаправленный) линейный список – это структура данных, состоящая из однотипных элементов (узлов), каждый из которых содержит в себе значение элемента и ссылки на предыдущий и последующий элементы в списке.

Ссылка на предыдущий элемент головы и ссылка на последующий элемент хвоста двусвязного списка имеет значение, равное `NULL`.

Двусвязный список проиллюстрирован на рисунке 6.

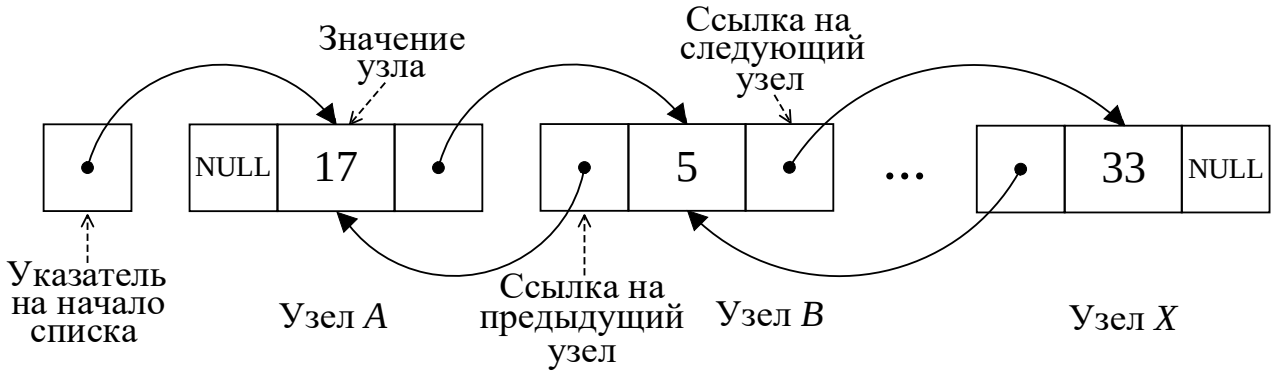


Рисунок 6 – Пример двусвязного линейного списка

Добавление нового элемента в двусвязный список и удаление элемента из него предполагает перезапись двух ссылок и проиллюстрировано на рисунке 7, *а* и *б* соответственно.

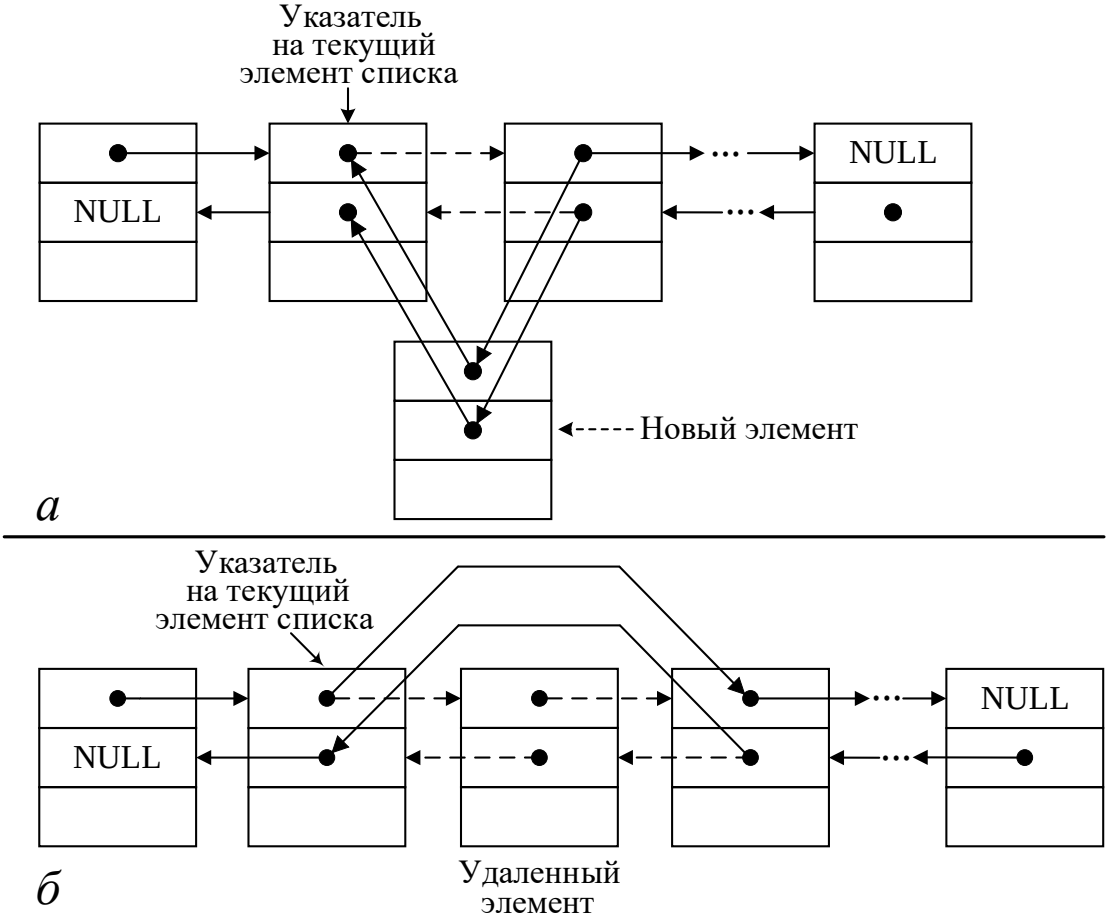


Рисунок 7 – Двусвязный список: добавление (*а*) и удаление (*б*) элемента

Перемещение по элементам двусвязного списка возможно в обе стороны: от головы к хвосту и наоборот.

2.3 Кольцевой (циклический) список (или кольцо) – это структура данных, базирующаяся на основе связанного списка (односвязного или двусвязного), в которой хвост списка зациклен на голову.

В односвязном кольцевом списке последний элемент имеет ссылку на первый (рисунок 8), а в двусвязном также и первый элемент, в качестве ссылки на предыдущий элемент, имеет связь с последним (хвостом).

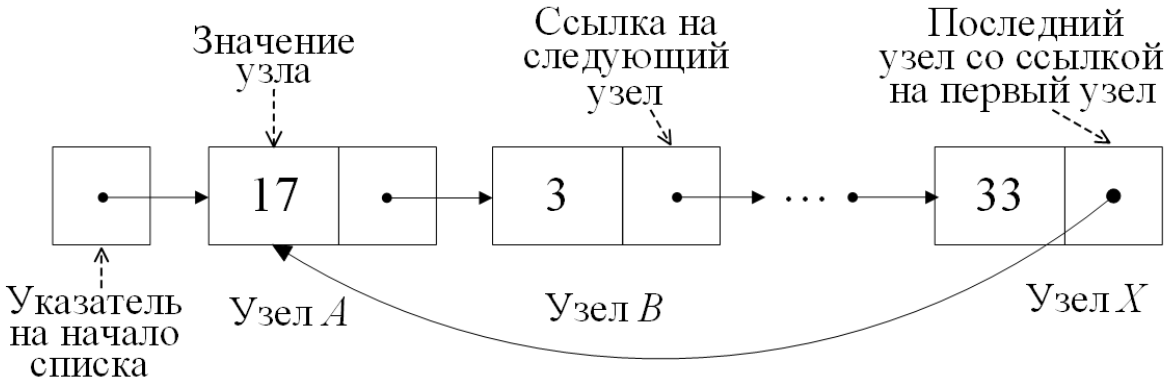


Рисунок 8 – Пример односвязного кольцевого списка

Добавление и удаление элементов в кольцевом списке в зависимости от степени связанности (одно- и двусвязном) выполняется аналогично соответствующим линейным спискам, рассмотренным ранее.

3 Очередь

Очередь (queue) – это структура данных, как правило, базирующаяся на связанном списке и реализующая принцип «первый пришел – первый вышел» (first in – first out, FIFO): добавление элементов осуществляется только в конец (хвост) очереди, а извлечение – только из начала (головы) очереди.

Очередь имеет два указателя: на хвост и голову (рисунок 9).

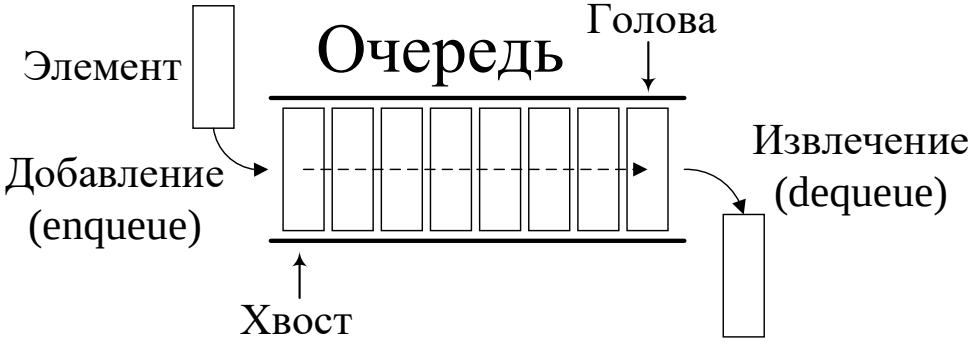


Рисунок 9 – Иллюстрация очереди

Для очереди определены следующие операции:

– проверка очереди на пустоту (`empty`);

– добавление нового элемента в конец (хвост) очереди (`push`);

– удаление элемента из начала (головы) очереди (`pop`);

– получение количества элементов в очереди (`size`).

Очереди могут быть реализованы различными способами:
– на массиве;
– на связанном списке;
– на стеках (как правило, на двух или шести).

4 Стек

Стек (stack) – это структура данных, представляющая собой упорядоченный набор элементов, в которой и добавление новых элементов, и удаление существующих производится только с одного конца (вершины) стека, реализуя таким образом принцип «последним пришел – первым вышел» (last in – first out, LIFO).

Стек имеет один указатель – на вершину стека (рисунок 10).

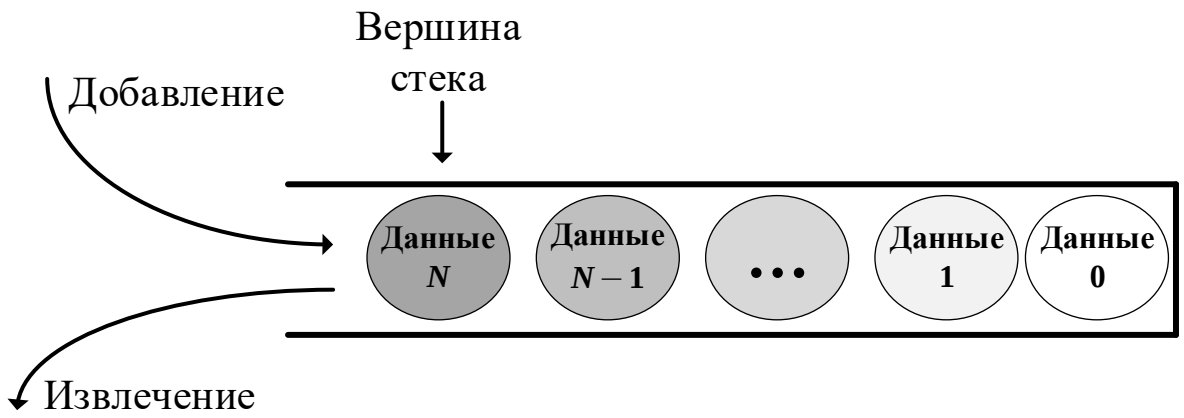


Рисунок 10 – Иллюстрация стека

Для стека определены следующие операции:
– проверка стека на пустоту (`empty`);
– добавление нового элемента в вершину стек (`push`);
– удаление элемента из вершины стека (`pop`);
– возврат элемента из вершины стека без его удаления (`peek`).

Стек также может быть реализован различными способами:
– на массиве (статическом или динамическом);
– на связанном списке (односвязном или двусвязном).

5 Дек

Дек (deque, double-ended queue) – это структура данных, представляющая собой комбинацию стека и очереди: добавление и извлечение элементов осуществляется как с хвоста, так и с головы дека.

Дек имеет два указателя: на хвост и голову (рисунок 11). Перемещение по деку возможно в обе стороны: от хвоста к голове и наоборот.

Для дека определены следующие операции:
– проверка дека на пустоту (`empty`);
– добавление нового элемента в конец (хвост) дека (`pushBack`);
– удаление элемента из конца (хвоста) дека (`popBack`);
– добавление нового элемента в начало (голову) дека (`pushFront`);
– удаление элемента из начала (головы) дека (`popFront`).

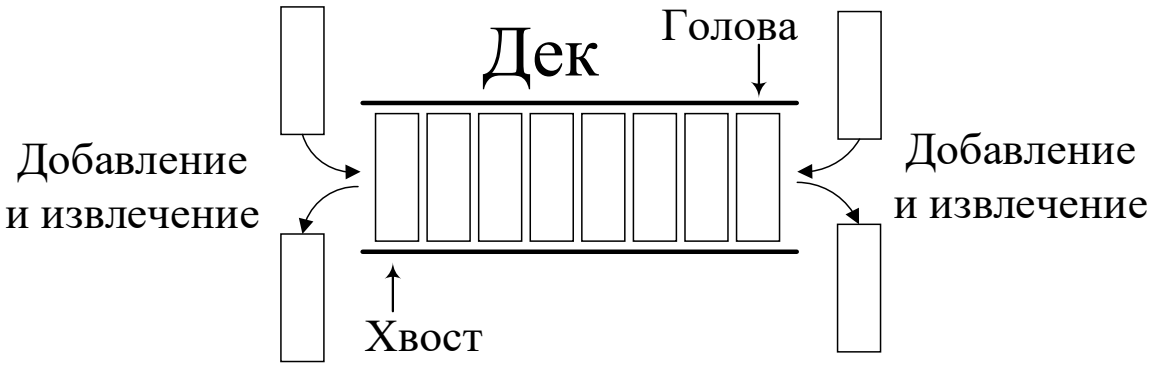


Рисунок 11 – Иллюстрация дека

Дек может быть реализован различными способами:
– на массиве (статическом или динамическом);
– на двусвязном списке;
– на двух стеках.

6 Словарь

Словарь (map, dictionary) – это структура данных, представляющая собой ассоциативный массив и реализующая интерфейс «ключ – значение». Ключом является некоторый уникальный идентификатор любого типа, а значением может быть любая объектная переменная (в том числе и другие структуры данных).

Неуникальный словарь (multimap) – это словарь с неуникальными ключами, т. е. содержащий более одной записи (пары «ключ – значение») с одним и тем же ключом.

Для словаря определены следующие операции:
– добавление нового элемента с уникальным ключом в коллекцию;
– удаление элемента по ключу из коллекции;
– изменение значения элемента по ключу;
– поиск (получение) значения элемента по ключу.

7 Дерево

Дерево (tree) – это динамическая нелинейная рекурсивная структура данных, содержащая непустое конечное множество элементов (вершин или узлов) связанных друг с другом, образуя иерархическую древовидную структуру.

Вершина (узел) – это элемент дерева, содержащий значение (ключ) и ссылки на сыновьи узлы (или значение `NULL` при их отсутствии).

Корень (корневой узел) – это начальный, самый верхний узел дерева (имеющий первый уровень).

Ветвь (дуга) – это связь между узлами (вершинами) дерева.

Предок (отец) – узел более высокого уровня (меньшего по значению), имеющий связь с текущим (узел, из которого выходит ветвь).

Потомок (ребенок, сын) – узел более низкого уровня (большего по значению), имеющий связь с текущим (узел, в который входит ветвь).

Лист – это терминальный узел, в который входит одна ветвь и не выходит ни одной, т. е. он не имеет потомков (детей).

Внутренние узлы (узлы ветвления) – узлы, не являющиеся листьями.

Высота (глубина) дерева – это максимальный уровень листа дерева (длина максимального пути от корня к листу).

Степень узла – это количество потомков (ветвей) внутреннего узла.

Степень дерева – это максимальная степень всех узлов дерева.

Поддерево – это часть древообразной структуры данных, которая может быть представлена в виде отдельного дерева.

Упорядоченное дерево – это дерево, у которого ветви, исходящие из каждой вершины, упорядочены определенным образом.

Сильноветвящееся дерево – дерево с произвольной степенью дерева.

Пример сильноветвящегося дерева представлен на рисунке 12.

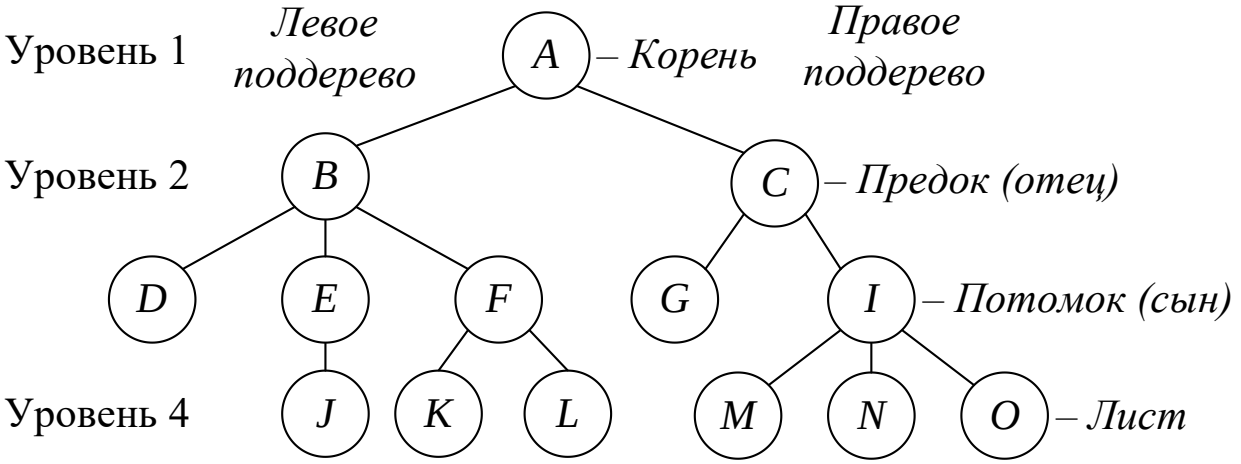


Рисунок 12 – Пример сильноветвящегося дерева

Обход дерева – это упорядоченная определенным образом последовательность вершин дерева, в которой каждая вершина будет встречаться только один раз.

Наиболее популярные способы обхода деревьев (прямой, симметричный и обратный) представлены на рисунке 13.

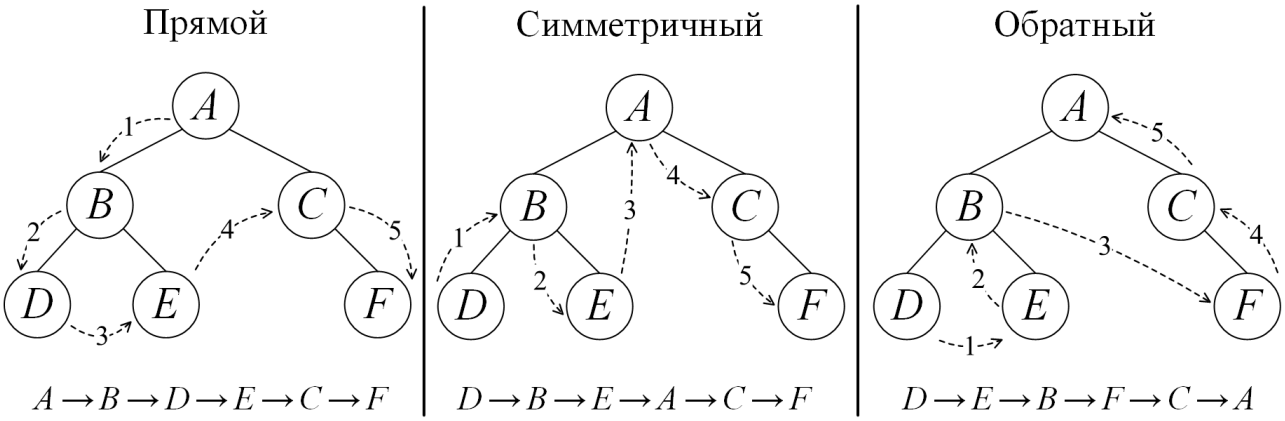


Рисунок 13 – Способы обхода деревьев

Различают следующие основные виды деревьев:
– бинарное (двоичное) дерево;
– бинарное (двоичное) дерево поиска;
– АВЛ-деревья (AVL-деревья);
– красно-черные деревья (КЧ-деревья);
– *B*-деревья.

7.1 Бинарное (двоичное) дерево

Бинарное (двоичное) дерево – это дерево, в котором каждая вершина имеет не более двух потомков, т. е. степень такого дерева не превышает двух. Каждый узел содержит ключ узла и не более двух указателей на различные бинарные поддеревья (левое и правое).

Пример бинарного дерева и организации его структуры представлен на рисунке 14.

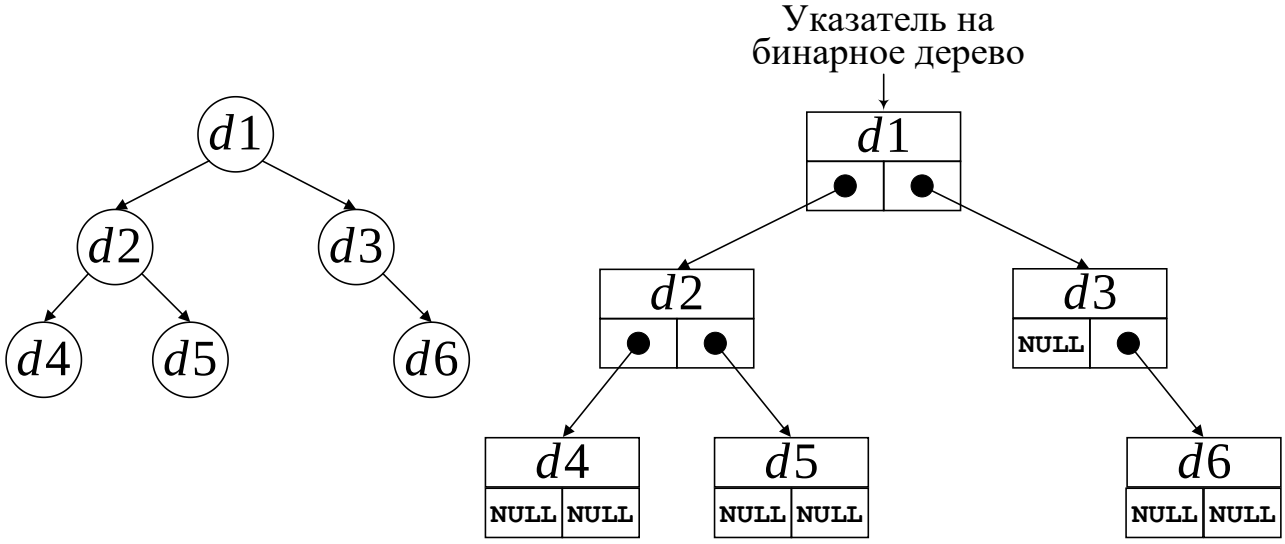


Рисунок 14 – Пример бинарного дерева

Классификация бинарных деревьев по степени вершин представлена на рисунке 15.

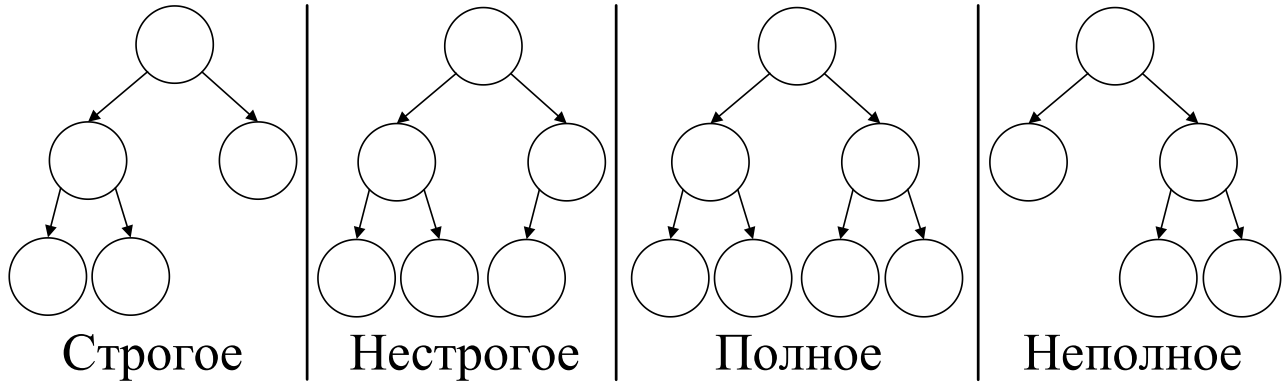


Рисунок 15 – Классификация бинарных деревьев

Строгое бинарное дерево – вершины дерева имеют степень только нуль (у листьев) или два (у узлов). Строгое бинарное дерево с N листьями всегда содержит $2N - 1$ узлов.

Нестрогое бинарное дерево – вершины дерева имеют степень нуль (у листьев) и один или два (у узлов).

Полное бинарное дерево уровня N – это дерево, в котором каждый узел уровня N является листом и каждый узел уровня меньше N имеет непустые левое и правое поддеревья.

7.2 Бинарное дерево поиска

Бинарное дерево поиска (БДП) – это несбалансированное двоичное дерево, в котором элементы больше корневого (и далее относительно узлов в поддеревьях) размещаются в правом поддереве, а элементы, которые меньше корневого (и далее относительно узлов в поддеревьях), размещаются в левом поддереве.

Более подробно БДП рассмотрено в лабораторной работе № 1 (с. 35).

7.3 АВЛ-дерево

АВЛ-дерево (AVL tree) – это сбалансированное по высоте двоичное дерево поиска, у которого для каждого его узла высота его двух поддеревьев различается не более чем на единицу по модулю.

Высота узла – это количество уровней, на которых расположены потомки данного узла.

Узел без детей (лист) имеет значение высоты, равное нулю, а несуществующий узел – минус единице.

Нарушение баланса АВЛ-дерева (изменение высоты узлов) может произойти после вставки нового узла или удаления существующего.

При добавлении/удалении узлов их высоту необходимо рекурсивно обновлять, начиная с листа в поддереве, где произошли изменения. Этот лист будет иметь высоту, равную нулю. Для определения высоты его родителя необходимо взять высоту его левого и правого ребенка, выбрать большую из них (это может быть нуль или минус единица) и рекурсивно добавлять единицу по пути к корню в данном поддереве (рисунок 16, *а*). Изменение высоты происходит, только если новый узел вставляется как лист на новый уровень, т. к. если он вставляется как второй ребенок родительского узла, высота не изменится (рисунок 16, *б*).

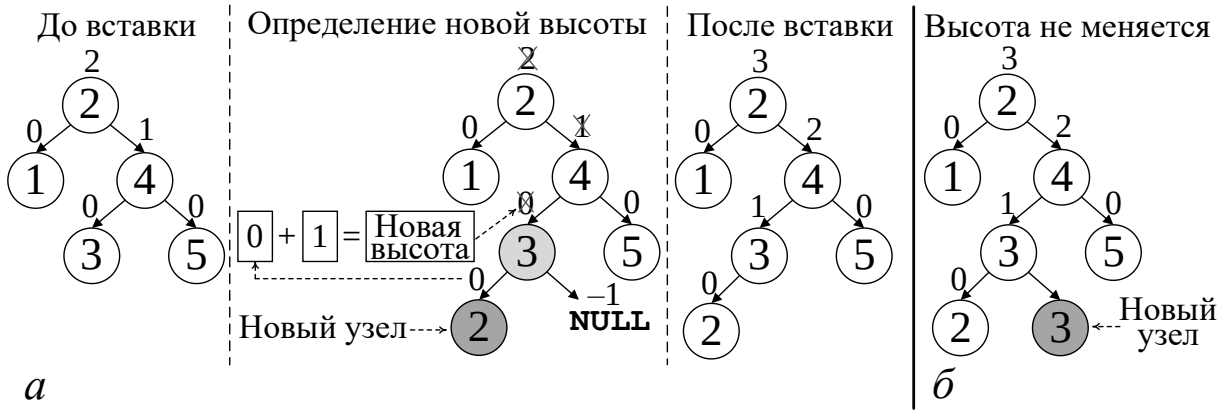


Рисунок 16 – Примеры вставки и определения высоты АВЛ-дерева

Балансировка АВЛ-дерева осуществляется после операций вставки/удаления новых узлов в случае разницы высот левого и правого поддеревьев, равной двум по модулю для отдельного узла. В какую сторону перегружено дерево показывает знак разницы высот:

– если знак отрицательный, значит слева высота больше и необходимо балансировать дерево вправо;

– если знак положительный, значит справа высота больше и необходимо балансировать дерево влево.

Балансировка означает поворот дерева в правую или левую сторону (относительно определенного несбалансированного узла) без изменения порядка следования узлов (рисунок 17).

Для АВЛ-деревьев различают следующие повороты:

– малый левый – если разница высот *L*-поддерева и *B*-поддерева равна двум, а высота *C* <= *R* (см. рисунок 17, *а*). Необходим после вставки нового узла в правое поддерево правой дочерней вершины. Чтобы сбалансировать дерево, необходимо заменить вершину *A* вершиной *B* и сделать поддерево *C* правым поддеревом вершины *A*;

– большой левый (правый левый, или *RL*) – если разница высот *L*-поддерева и *B*-поддерева равна двум, а высота *C* > *R* (см. рисунок 17, *б*). Необходим после вставки нового узла в левое поддерево правой дочерней вершины. Чтобы сбалансировать дерево, необходимо заменить вершину *A* вершиной *C* и сделать поддерево *M* правым поддеревом вершины *A*, а поддерево *N* – левым поддеревом вершины *B*. При большом левом повороте сперва выполняется правый поворот относительно правого потомка (вершина *B*) текущего (небалансного) узла (вершина *A*), т. е. вершина *B* заменяется вершиной *C*, правое поддерево *N* становится левым поддеревом вершины *B*, а затем левый поворот относительно текущего (вершины *A*);

– малый правый – если разница высот *R*-поддерева и *B*-поддерева равна минус двум, а высота *C* <= *L* (см. рисунок 17, *в*). Необходим после вставки нового узла в левое поддерево левой дочерней вершины. Чтобы сбалансировать дерево, необходимо заменить вершину *A* вершиной *B* и сделать поддерево *C* левым поддеревом вершины *A*;

– большой правый (левый правый или *LR*) – если разница высот *R*-поддерева и *B*-поддерева равна минус двум, а высота *C* > *L* (см. рисунок 17, *г*). Необходим после вставки нового узла в правое поддерево левой дочерней вершины. Чтобы сбалансировать дерево, необходимо заменить вершину *A* вершиной *C* и сделать поддерево *M* правым поддеревом вершины *B*, а поддерево *N* – левым поддеревом вершины *A*. При большом правом повороте сперва выполняется левый поворот относительно левого потомка (вершина *B*) текущего (небалансного) узла (вершина *A*), т. е. вершина *B* заменяется вершиной *C*, левое поддерево *M* становится правым поддеревом вершины *B*, а затем правый поворот относительно текущего (вершины *A*).

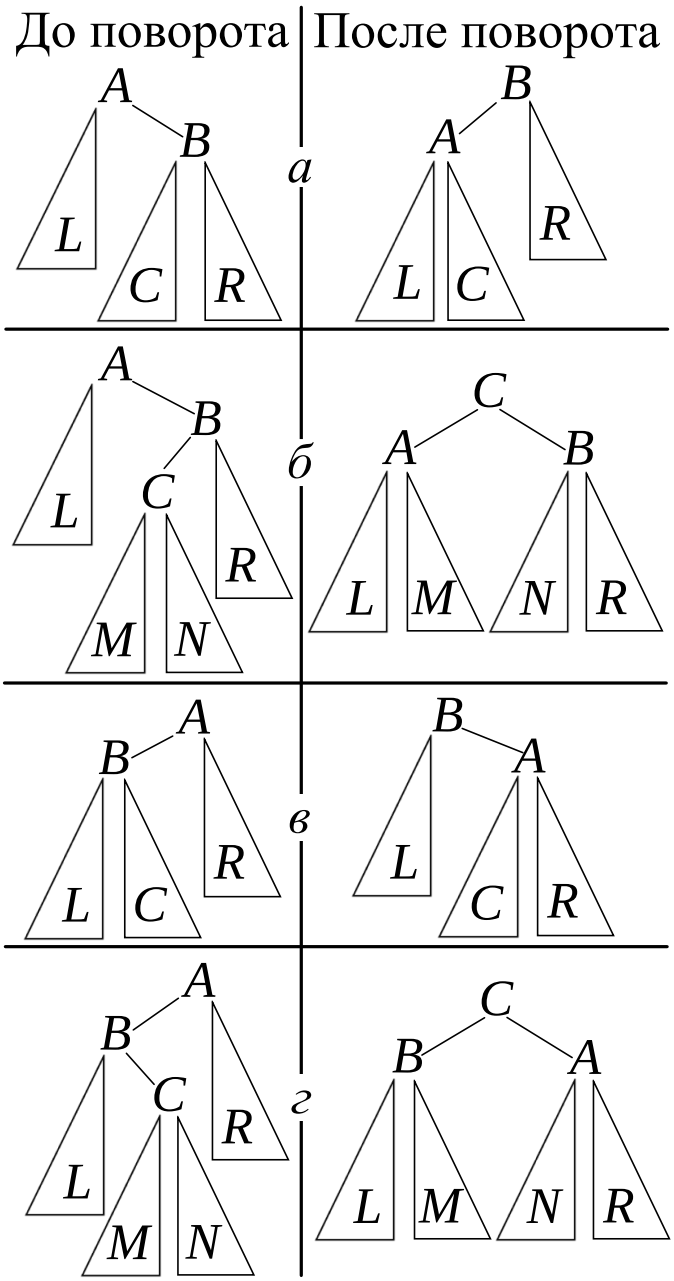
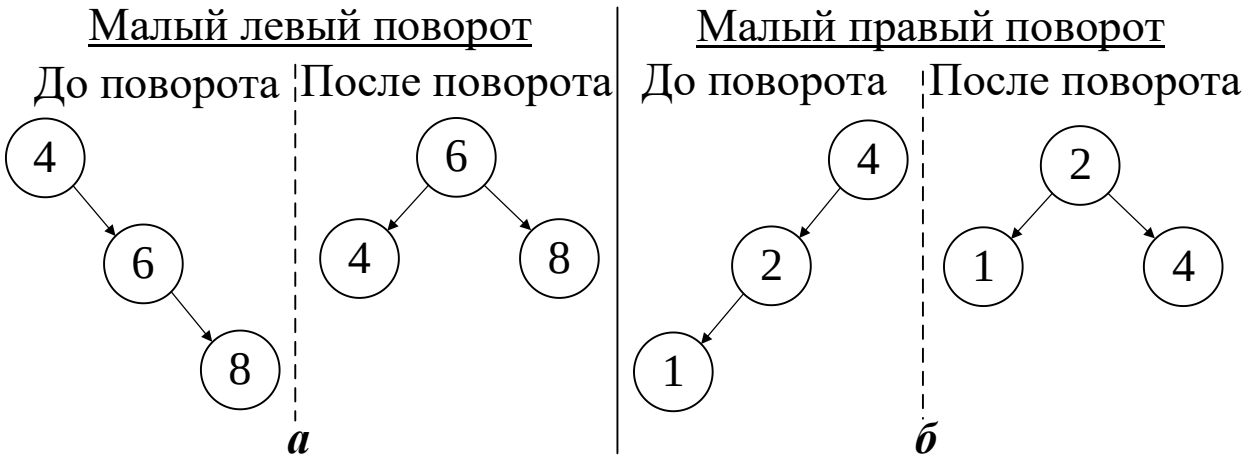


Рисунок 17 – Виды балансировки (поворотов) АВЛ-дерева

Как видно из рисунка 17, большие повороты – это соответствующие комбинации правого и левого малого поворотов.

Примеры самых простых случаев малого левого и правого поворотов представлены на рисунке 18, *а* и *б* соответственно.

Как видно из рисунка 18, *а*, суть малого левого поворота заключается в том, что узел, относительно которого осуществляется поворот (балансировка), меняет свои ссылки: вначале он *4* указывал на своего правого ребенка *6*, а при повороте – его прежний правый ребенок *6* указывает на него *4* как на левого ребенка. Как видно из рисунка 18, *б*, суть малого правого поворота заключается в том, что узел, относительно которого осуществляется поворот (балансировка), меняет свои ссылки: вначале он *4* указывал на своего левого ребенка *2*, а при повороте – его прежний левый ребенок *2* указывает на него *4* как на правого ребенка.



а – малый левый поворот;
б – малый правый поворот
Рисунок 18 – Примеры простейших малых поворотов АВЛ-дерева

Пример более сложного малого правого поворота представлен на рисунке 19.

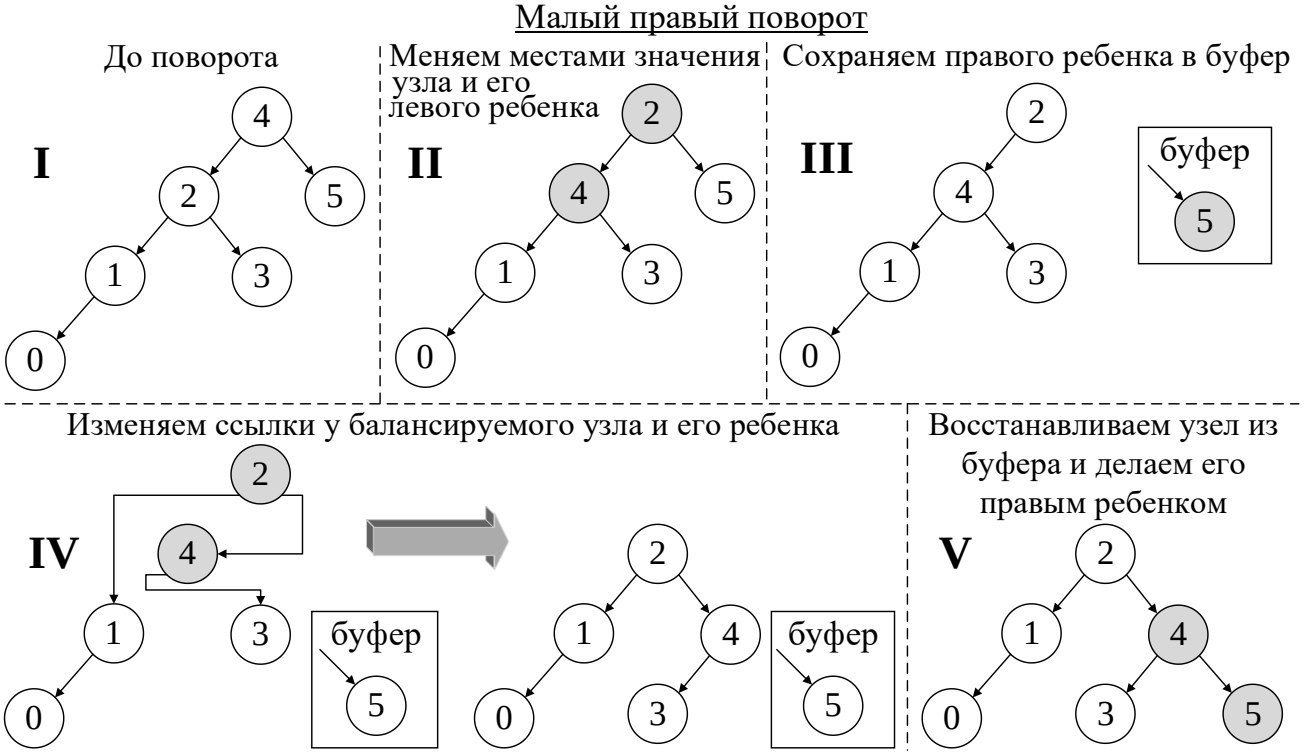


Рисунок 19 – Пример малого правого поворота

Как видно из рисунка 19, для выполнения балансировки необходимо:

– поменять местами значения узла *4*, относительно которого выполняется поворот (текущего узла) и его левого ребенка *2*;

– значение правого ребенка *5* текущего узла (уже *2*) сохранить в буфер;

– поменять ссылки текущего узла *2*: правый указатель на левого ребенка *4*, а левый указатель – на левого потомка *1* уже правого ребенка *4*;

– правую ссылку потомка *4* текущего узла *2* сделать левой;

– восстановить из буфера значение *5* и добавить его в правое поддерево текущего узла *2*.

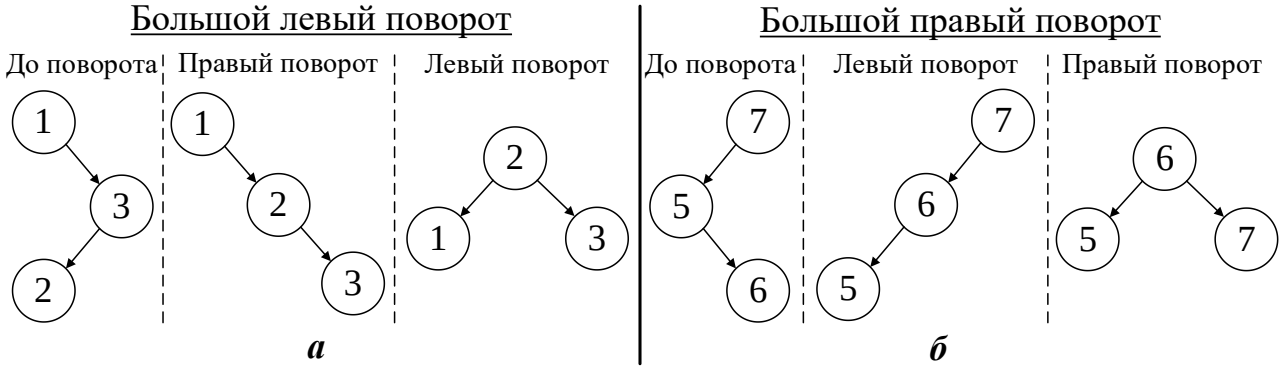
Иными словами:

– текущий корень *4* поддерева заменяется на левый дочерний узел *2*;

– предыдущий корень *4* становится правым дочерним узлом для *2*;

– предыдущее правое поддерево узла *2* становится левым поддеревом для узла *4*.

Пример простейших больших поворотов представлен на рисунке 20.



а – большой левый поворот;
б – большой правый поворот

Рисунок 20 – Примеры простейших больших поворотов АВЛ-дерева

Пример более сложного большого левого поворота представлен на рисунке 21.

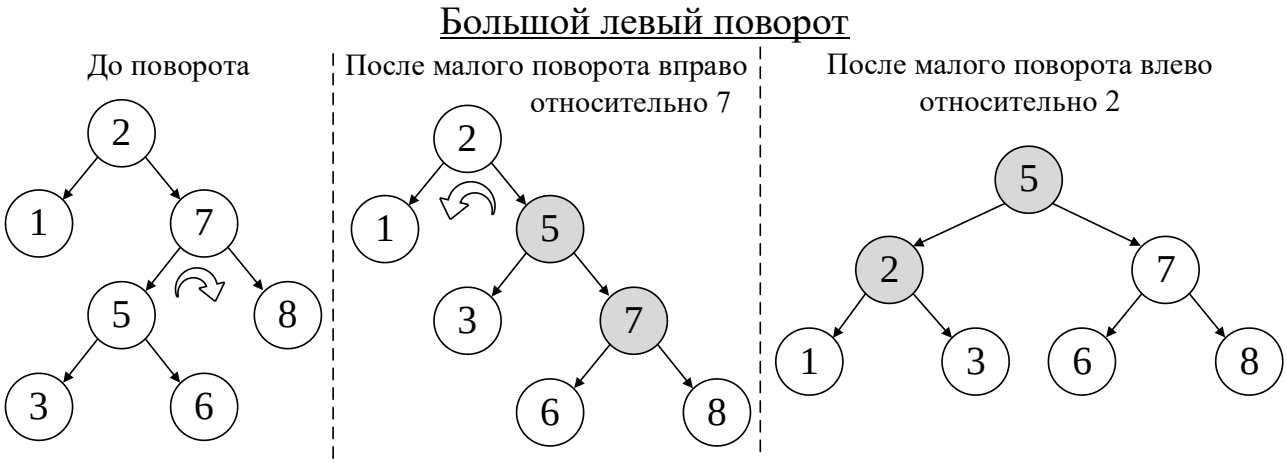


Рисунок 21 – Пример большого поворота влево АВЛ-дерева

7.4 Красно-черное дерево

Красно-черное дерево (red-black tree, КЧ-дерево) – это сбалансированное по высоте двоичное дерево поиска, у которого для осуществления балансировки для каждого узла вводится атрибут цвета (бит цвета), который может принимать только значение «красный» или «черный».

Правила окраски красно-черного дерева:

– корень и листья дерева всегда имеют только черный цвет;

– все листья дерева имеют значение (ключ), равное `NULL`;

– красные узлы могут иметь только черных предков;

– черный узел может иметь черного предка;

– все простые пути из любого узла до листьев содержат одинаковое количество черных узлов.

Балансировка красно-черного дерева осуществляется после операций вставки/удаления новых узлов. При балансировке появляются новые понятия.

Дед – узел, являющийся предком отца (на один уровень выше).

Дядя – узел, находящийся в другом поддереве относительно деда, но на одном уровне с отцом.

Брат – узел, находящийся в другом поддереве относительно отца, но на одном уровне с текущим узлом.

Черная высота вершины – это количество черных вершин на пути из нее (текущей вершины) в лист, без учета текущей вершины.

Пример красно-черного дерева представлен на рисунке 22.

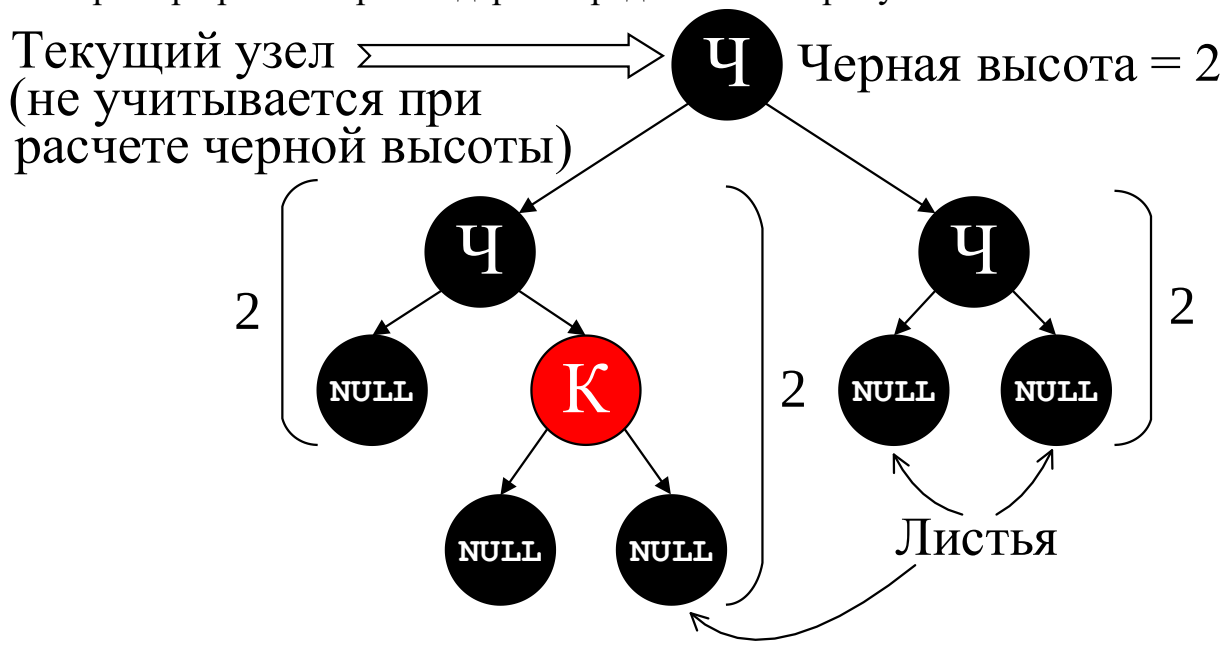


Рисунок 22 – Пример красно-черного дерева

При балансировке красно-черного дерева выполняются не только повороты, рассмотренные для АВЛ-деревьев (малый/большой левый/правый), но и перекрашивание узлов в соответствии с правилами, описанными выше, и с учетом необходимости равенства черной высоты для всех узлов одного уровня.

Вставка нового узла в красно-черное дерево

Вставляемый (новый) узел может быть как красного, так и черного цвета, однако, рациональнее вставлять узел именно красного цвета, т. к. его вставка в среднем случае ломает меньшее количество правил окраски дерева и, соответственно, требует меньше действий для перекрашивания. Вставка же черного узла гарантированно изменит черную высоту и потребует балансировки дерева.

Рассмотрим примеры вставки нового узла красного цвета. В зависимости от места вставки нового узла красного цвета могут сложиться следующие варианты балансировки (перекрашивания) красно-черного дерева:

А) красный узел вставляется в качестве корня КЧ-дерева (рисунок 23).

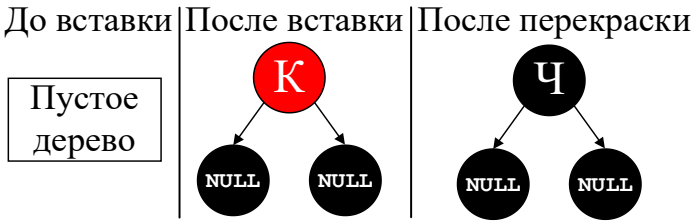


Рисунок 23 – Пример А: вставка нового узла в качестве корня КЧ-дерева

После вставки такого красного узла дерево будет не сбалансировано, поскольку корень обязан быть черным. Для балансировки необходимо просто перекрасить вставленный узел в черный цвет;

Б) красный узел вставляется после красного, причем дед не является корнем, а дядя тоже красный (рисунок 24).

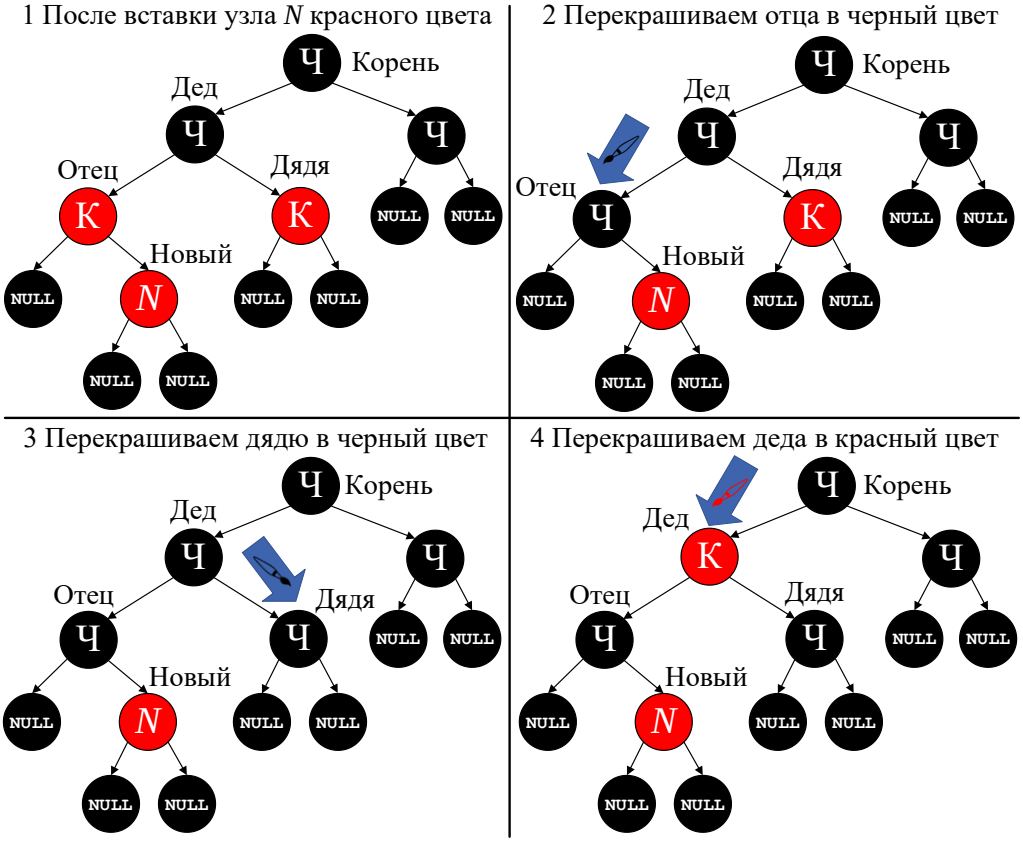


Рисунок 24 – Пример Б: вставка нового красного узла после красного узла в КЧ-дереве

Вставка нового красного узла *N* после красного узла нарушает правило о том, что у красного узла не может быть красного родителя, поэтому перекрашиваем отца в черный цвет. Это в свою очередь приводит к тому, что черная высота деда в левом поддереве не совпадает с черной высотой деда справа (слева стало больше на единицу), поэтому для выравнивания черной высоты перекрашиваем дядю в черный цвет. После этого черная высота деда увеличивается на единицу по сравнению с тем, что было до балансировки. Чтобы это исправить, необходимо перекрасить деда в красный цвет, что допустимо, поскольку его предок черного цвета. Теперь дерево будет сбалансировано;

В) красный узел вставляется после красного, причем дед является корнем, а дядя тоже красный (рисунок 25).

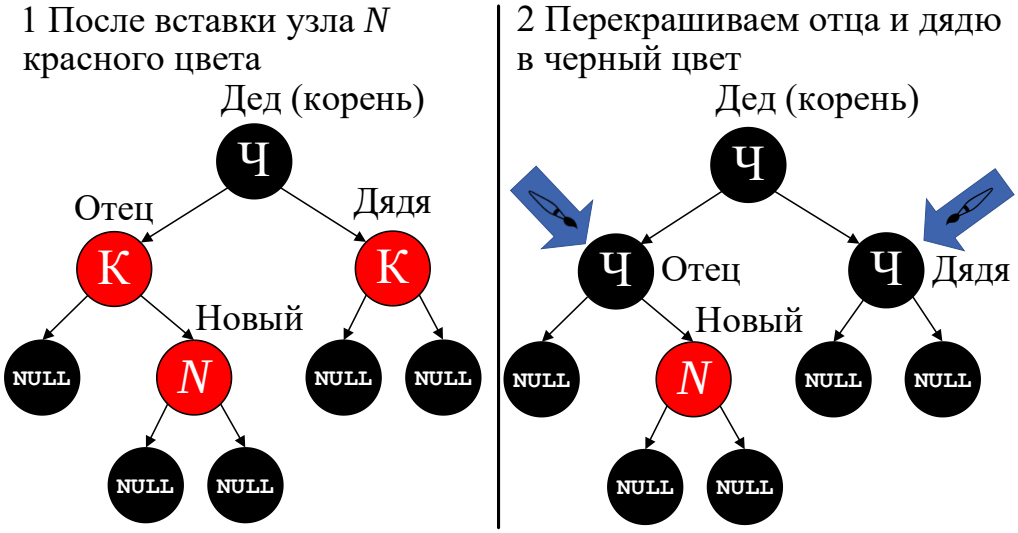


Рисунок 25 – Пример В: вставка нового красного узла после красного узла в КЧ-дереве

Вставка нового красного узла *N* после красного узла также нарушает правило о том, что у красного узла не может быть красного родителя, поэтому перекрашиваем отца в черный цвет. Это в свою очередь также приводит к тому, что черная высота деда в левом поддереве не совпадает с черной высотой деда справа (слева стало больше на единицу), поэтому для выравнивания черной высоты перекрашиваем дядю в черный цвет. На этом балансировка заканчивается, поскольку черная высота всего дерева (относительно корня) одинакова, а дед является корнем и может быть черного цвета;

Г) красный узел вставляется после красного, образуя зигзаг (зубец), а дядя черного цвета (рисунок 26).

Вставка нового красного узла *N* после красного узла также нарушает правило о том, что у красного узла не может быть красного родителя, кроме того, образуется так называемый зигзаг, нарушающий баланс. Чтобы избавиться от зигзага, необходимо выполнить левый поворот относительно родителя вставленного узла. В результате образуется так называемая прямая линия из двух красных узлов. Чтобы избавиться от нее, перекрашиваем узел *N* в черный цвет. Это нарушает черную высоту относительно корня (родителя узла *N*). Делаем правый поворот относительно корня (родителя узла *N*). Теперь дерево будет сбалансировано.

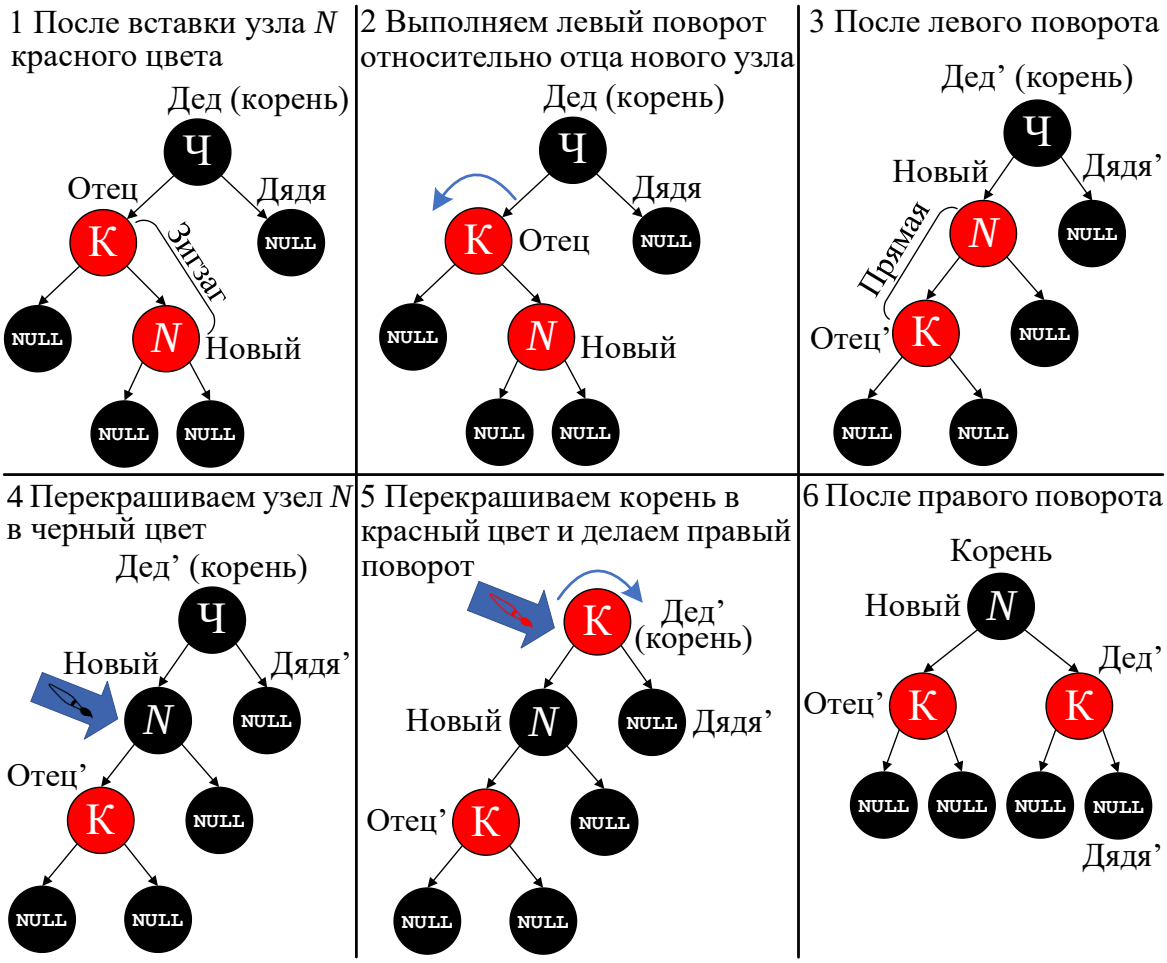


Рисунок 26 – Пример Г: вставка нового красного узла после красного узла в КЧ-дереве

Д) красный узел вставляется после красного, образуя прямую линию, а дядя черного цвета (рисунок 27).

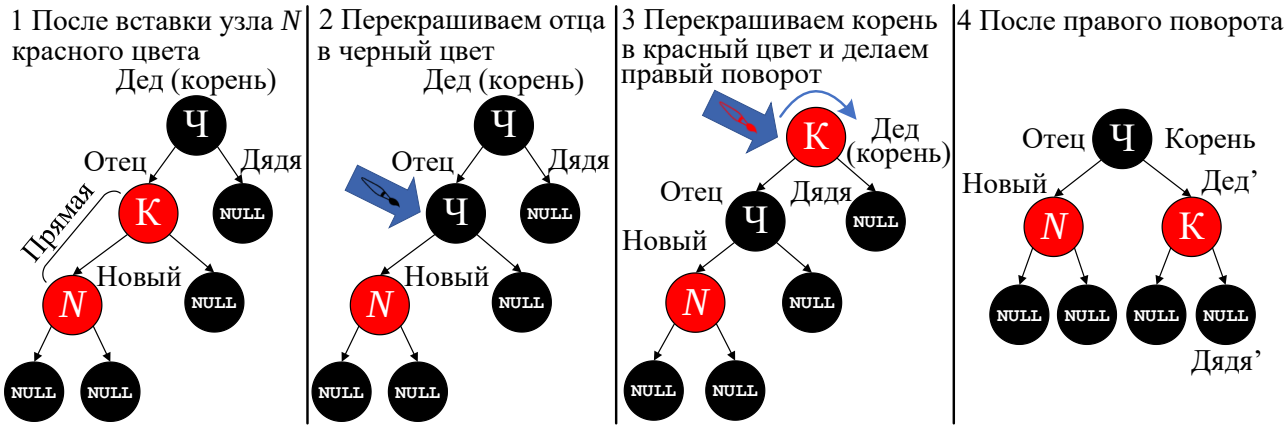


Рисунок 27 – Пример Д: вставка нового красного узла после красного узла в КЧ-дереве

Вставка нового красного узла *N* после красного узла также нарушает правило о том, что у красного узла не может быть красного родителя и образуется так называемая прямая линия из двух красных узлов. Чтобы избавиться от нее, перекрашиваем отца в черный цвет. Это нарушает черную высоту относительно корня (деда). Делаем правый поворот относительно корня (деда). Теперь дерево будет сбалансировано;

Е) красный узел вставляется после черного. Данный случай не требует балансировки, поскольку такая вставка не нарушает черной высоты дерева.

Удаление узла из красно-черного дерева

Удаление узла из красно-черного дерева происходит аналогично удалению из обычного дерева, но только до момента балансировки.

Также следует отметить, что сразу физическое удаление узла происходит в следующих случаях:

– если у удаляемого узла нет детей (кроме значений NULL), т. е. это лист;

– если у удаляемого узла только один ребенок.

Если у удаляемого узла два ребенка, то он физически не удаляется, а заменяется (копируется значение) на одного из его детей: максимального по левой ветке (max) или минимального по правой ветке (min). Только после этого происходит физическое удаление одного из выбранных детей (max или min) как листа.

При удалении могут возникнуть следующие варианты балансировки:

А) если удаляемый узел имеет красный цвет. Этот случай гарантирует, что у него нет детей, т. к. в противном случае дерево было бы не сбалансированным. При удалении такого красного узла он просто заменяется на значение NULL и балансировка не требуется;

Б) если удаляемый узел имеет черный цвет и ребенка красного цвета. После удаления черного узла его красный потомок займет место удаленного узла и тем самым уменьшит черную высоту на единицу. Для балансировки достаточно перекрасить этот красный узел в черный цвет;

В) если удаляемый узел имеет черный цвет и ребенка черного цвета. В этом случае удаляемый черный узел заменяется на значение NULL, а для определения балансировки нужно обратить внимание на брата удаляемого узла. Возможны следующие варианты:

В.1) брат удаляемого узла имеет красный цвет. Тогда:

– дети брата будут точно черного цвета;

– отец будет точно черного цвета.

Для балансировки необходимо (рисунок 28):

– перекрасить брата в черный цвет;

– перекрасить отца в красный цвет;

– выполнить левый поворот относительно отца.

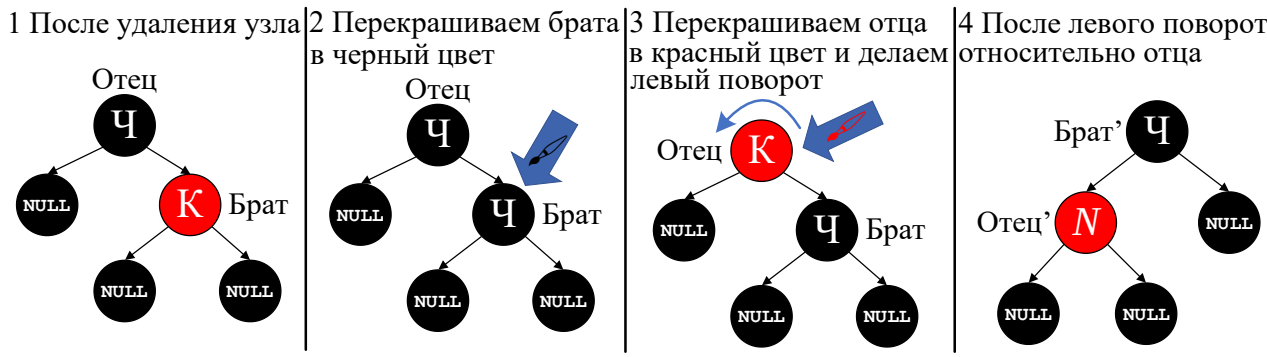


Рисунок 28 – Пример В.1: балансировка после удаления узла КЧ-дерева

В.2) брат удаляемого узла черный с детьми красного цвета или черным ребенком слева и красным справа.

Для балансировки необходимо (рисунок 29):

– перекрасить брата в цвет отца;

– перекрасить отца и правого красного ребенка брата в черный цвет;

– выполнить левый поворот относительно отца.

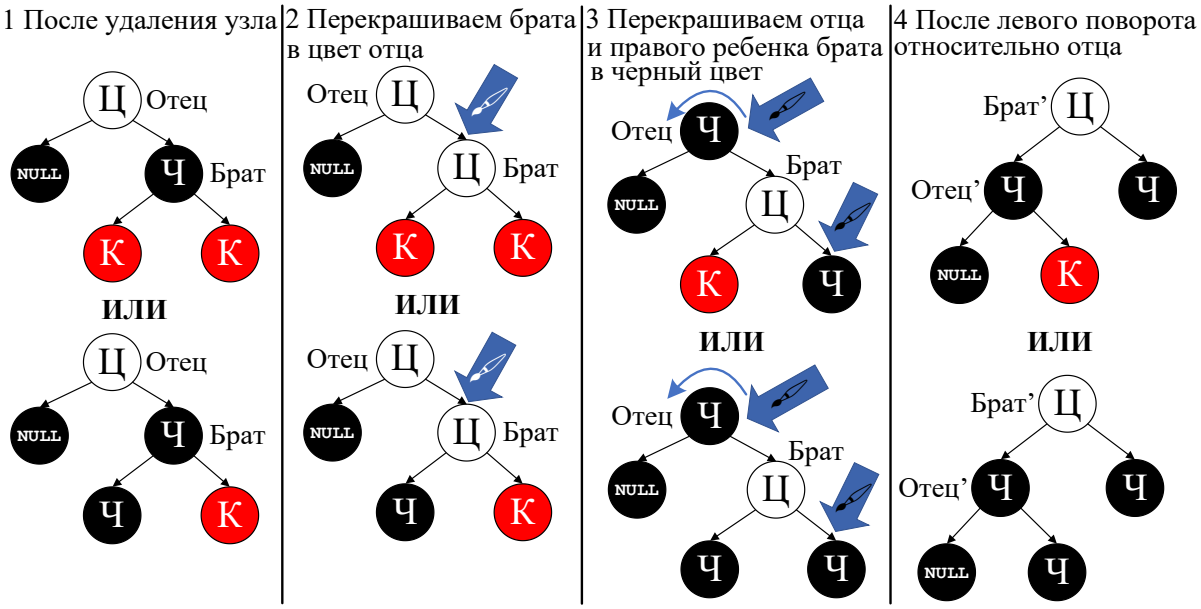


Рисунок 29 – Пример В.2: балансировка после удаления узла КЧ-дерева

В.3) брат удаляемого узла черный с красным ребенком слева и черным справа.

Для балансировки необходимо (рисунок 30):

– перекрасить брата в красный цвет;

– перекрасить левого ребенка брата в черный цвет;

– выполнить правый поворот относительно брата;

– выполнить балансировку из случая В.2.

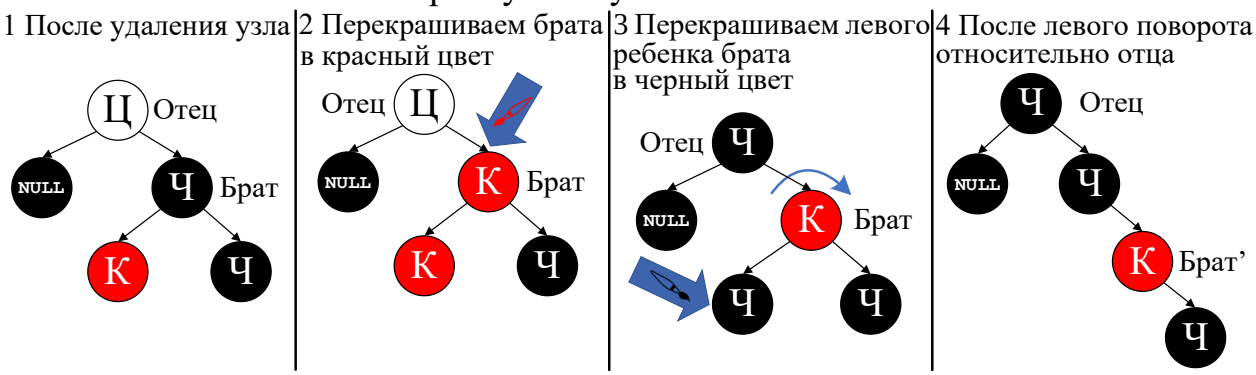


Рисунок 30 – Пример В.3: балансировка после удаления узла КЧ-дерева

В.4) брат удаляемого узла черный с черными детьми слева и справа. В данной ситуации цвет и уровень иерархии отца неизвестен и могут сложиться следующие варианты:

В.4.1) отец является корнем (и соответственно имеет черный цвет). Для балансировки необходимо перекрасить брата в красный цвет (рисунок 31).

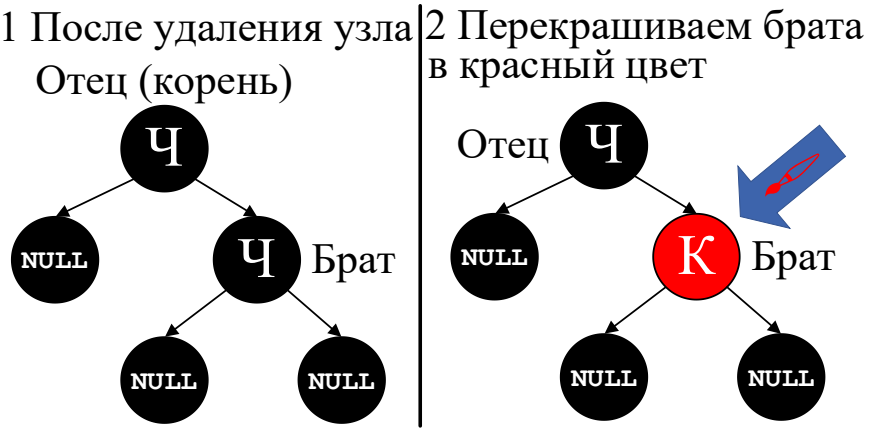


Рисунок 31 – Пример В.4.1: балансировка после удаления узла КЧ-дерева

В.4.2) отец имеет красный цвет (и соответственно не является корнем).
Для балансировки необходимо (рисунок 32):
– перекрасить брата в красный цвет;
– перекрасить отца в черный цвет.

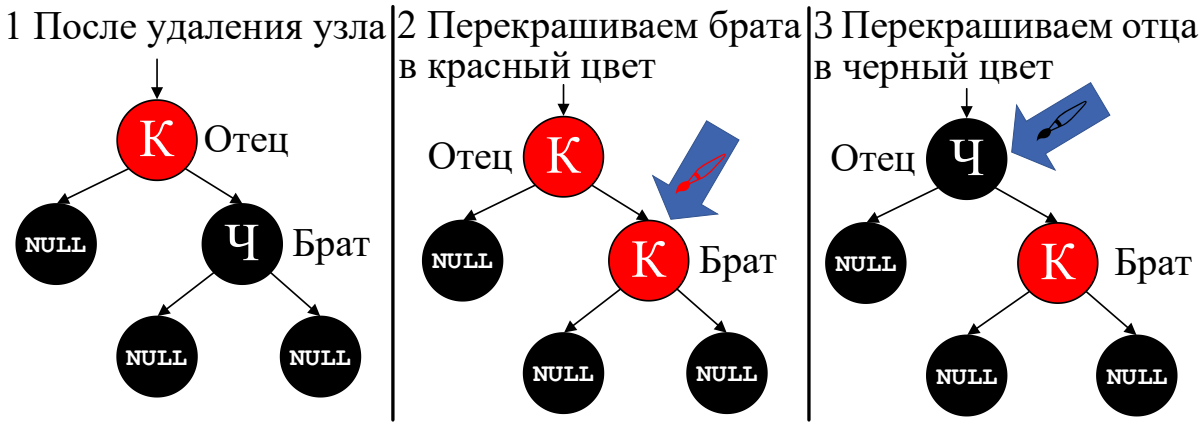


Рисунок 32 – Пример В.4.2: балансировка после удаления узла КЧ-дерева

В.4.3) отец имеет черный цвет и не является корнем.
Для балансировки необходимо (рисунок 33):
– перекрасить отца в красный цвет;
– выполнить левый поворот относительно отца.

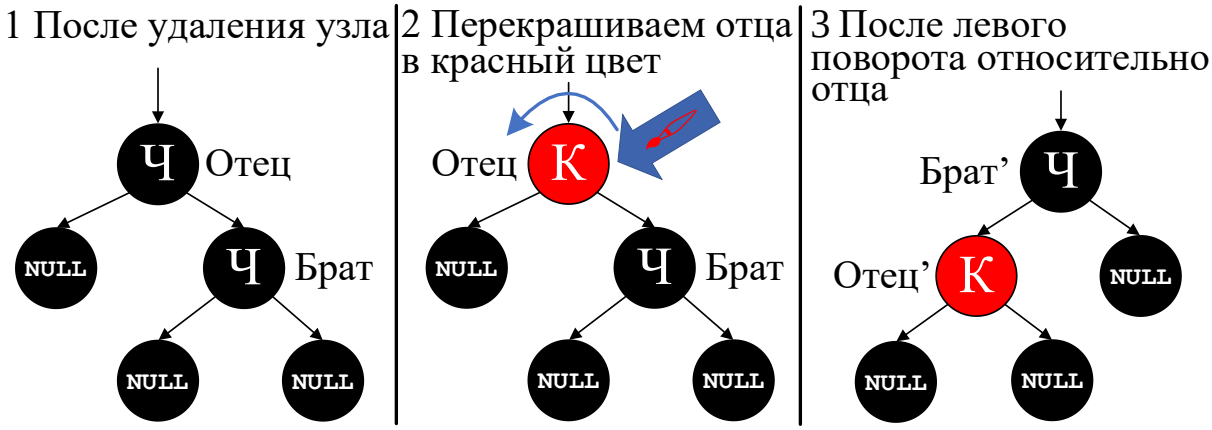


Рисунок 33 – Пример В.4.3: балансировка после удаления узла КЧ-дерева

7.5 B-дерево

B-дерево – это сбалансированное по высоте дерево поиска порядка *m*, в котором каждый узел содержит множество значений (ключей) и имеет более двух потомков.

Порядок B-дерева – определяет количество хранимых значений (ключей) в узле и количество его потомков.

B-дерево порядка *m* обладает следующими свойствами:

– глубина всех листьев одинакова;

– все узлы, кроме корня, должны иметь как минимум $m / 2 - 1$ ключей и максимум $m - 1$ ключей;

– все узлы без листьев, кроме корня (т. е. все внутренние узлы), должны иметь минимум $m / 2$ потомков;

– если корень – это узел, не содержащий листьев, он должен иметь минимум два потомка;

– узел без листьев с $n - 1$ ключами должен иметь n потомков;

– все ключи в узле должны располагаться в порядке возрастания их значений.

Над *B*-деревом определены следующие операции:

– поиск (выполняется аналогично с бинарным деревом поиска);

– вставка нового узла;

– удаление узла из *B*-дерева.

В *B*-дереве новый узел может быть вставлен только в узел-лист. Это значит, что новая пара «ключ – значение» всегда добавляется только к узлу-листу. Вставка выполняется по следующему алгоритму:

– шаг 1. Проверить дерево на пустоту;

– шаг 2. Если дерево пустое, создать новый узел с новым значением ключа и его принять за корневой узел;

– шаг 3. Если дерево не пустое, найти подходящий узел-лист, к которому будет добавлено новое значение (аналогично бинарному дереву поиска);

– шаг 4. Если в текущем узле-листе есть незанятая ячейка, добавить новую пару «ключ – значение» к текущему узлу-листу, следуя возрастающему порядку значений ключей внутри узла;

– шаг 5. Если текущий узел полон и не имеет свободных ячеек, необходимо разделить узел-лист, отправив среднее значение родительскому узлу. Шаг повторяется, пока отправляемое значение не будет зафиксировано в узле;

– шаг 6. Если разделение происходит с корнем дерева, тогда среднее значение становится новым корнем дерева и высота дерева увеличивается на единицу.

B-дерево как структура данных применяется во внешней памяти и более подробно рассмотрено в лабораторной работе № 4 (с. 72).

8 Графы

Граф – это топологическая модель данных, состоящая из множества объектов данных (вершин) и множества линий связи (отношений) этих объектов данных друг с другом (дуг).

Вершина – точка в графе, отдельный объект данных.

Ребро – неупорядоченная пара двух вершин, которые связаны друг с другом. Эти вершины называются концевыми точками или концами ребра.

Инцидентность вершины ребру – если вершина является для этого ребра концевой.

Смежные вершины – это две вершины, инцидентные одному ребру.

Смежные ребра – это два ребра, инцидентные одной вершине.

Петля – это ребро, инцидентное одной вершине, т. е. такое, которое замыкается на одной вершине.

Псевдограф – это граф с петлями.

Кратные (параллельные) ребра – это ребра, имеющие одинаковые концевые вершины.

Мультиграф – это граф с кратными ребрами.

Псевдомультиграф – это граф с петлями и кратными ребрами.

Степень вершины – это количество ребер, инцидентных указанной вершине (исходящих из вершины). Петля увеличивает степень вершины на два.

Изолированная вершина – это вершина с нулевой степенью.

Висячая вершина – это вершина со степенью, равной единице.

Подграф – это выделенные из исходного графа несколько вершин и несколько ребер (между выбранными вершинами).

Полный граф – это граф, в котором каждые две вершины соединены одним ребром.

Регулярный граф – это граф, в котором степени всех вершин одинаковые.

Двудольный граф – это граф, в котором все вершины можно разделить на два множества таким образом, что каждое ребро соединяет вершины из разных множеств.

Планарный (плоский) граф – это граф, который можно разместить на плоскости таким образом, чтобы ребра не пересекались.

Путь (маршрут) – это последовательность смежных ребер, задающаяся перечислением вершин, по которым он пролегает.

Длина пути – это количество ребер на пути.

Цепь – это путь без повторяющихся ребер.

Простая цепь – это цепь без повторяющихся вершин.

Цикл (контур) – это цепь, в котором последняя вершина совпадает с первой.

Длина цикла – это количество ребер в цикле. Самым коротким циклом является петля.

Цикл Эйлера – это цикл, проходящий по каждому ребру ровно один раз (такой цикл существует, когда все вершины в связанном графе имеют четную степень).

Цикл Гамильтона – это цикл, проходящий через все вершины графа по одному разу (простой цикл, в который входят все вершины графа).

Взвешенный граф – это граф, в котором у каждого ребра и/или каждой вершины есть «вес», т. е. некоторое число, обозначающее длину пути, его стоимость и т. п.

Связный граф – это граф, в котором существует путь между любыми двумя вершинами.

Дерево – это связный граф без циклов.

Лес – это граф, в котором несколько деревьев.

Ориентированный граф (орграф) – это граф, в котором ребра имеют направления.

Дуга – это ребро в ориентированном графе, имеющее направление.

Полустепень захода вершины – это количество дуг, заходящих в эту вершину.

Исток – это вершина с нулевой полустепенью захода.

Полустепень исхода вершины – это количество дуг, исходящих из этой вершины.

Сток – это вершина с нулевой полустепенью исхода.

Компонента связности – это множество таких вершин графа, что между любыми двумя вершинами существует путь.

Компонента сильной связности – это максимальное множество вершин орграфа, между любыми двумя вершинами которого существует путь по дугам.

Компонента слабой связности – это максимальное множество вершин орграфа, между любыми двумя вершинами которого существует путь по дугам без учета направления (т. е. по дугам можно двигаться в любом направлении).

Мост – это ребро, при удалении которого количество связанных компонент графа увеличивается.

Обход графа – это перебор всех вершин или ребер графа в поисках некоторой вершины, удовлетворяющей определенным условиям.

Идея обхода графа заключается в том, чтобы пометить каждую вершину при первом ее посещении и хранить информацию о тех вершинах, не все ребра из которых просмотрены.

Различают следующие способы обхода графов:

1 Обход (поиск) в глубину (depth first search, DFS) – это способ перебора вершин графа ветвями, т. е. когда возможные пути по ребрам, выходящим из вершин, разветвляются, то сначала полностью проходится одна ветка и только потом осуществляется переход к другим веткам (если они останутся непройденными).

Для реализации поиска в глубину обычно используются такие структуры данных, как стек (для хранения вершин, смежных с текущей) и множество (для хранения посещенных вершин).

Пример обхода графа в глубину представлен на рисунке 34.

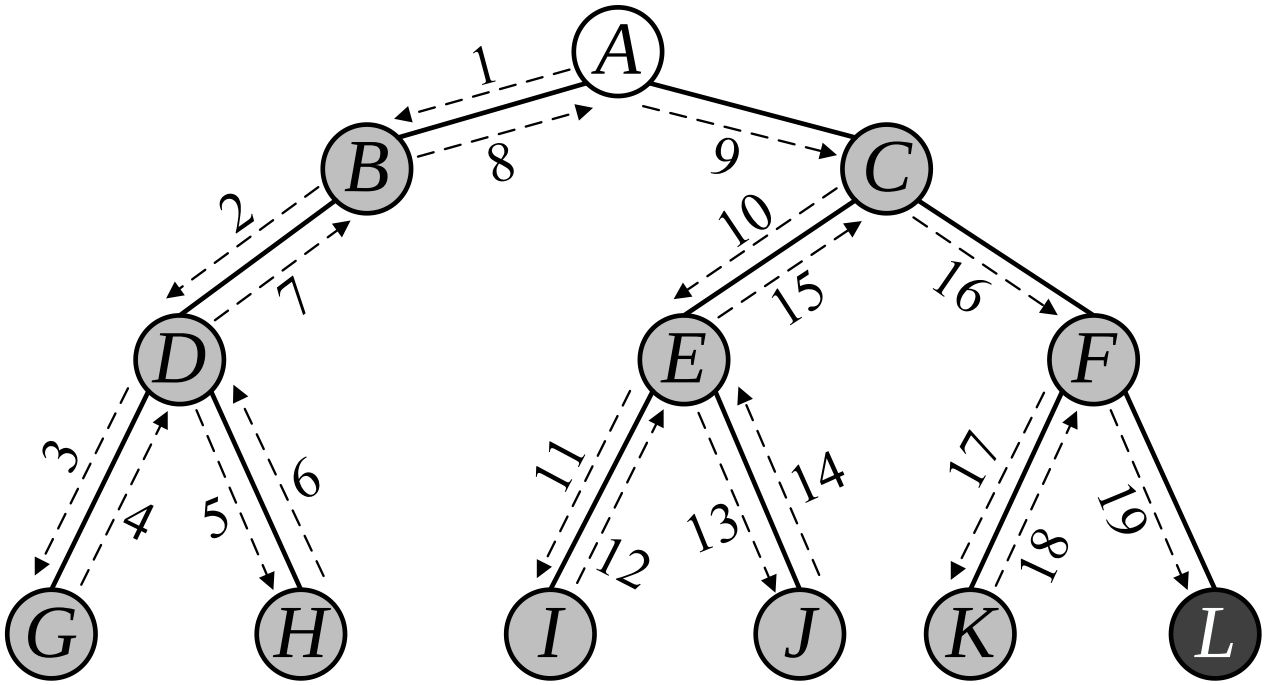


Рисунок 34 – Пример обхода графа в глубину

Алгоритм поиска в глубину:

– шаг 1. Выбранную начальную вершину пометить как посещенную;

– шаг 2. Исследовать первую непосещенную соседнюю вершину;

– шаг 3. Повторить шаги 1 и 2 для этой соседней вершины;

– шаг 4. Если все соседи текущей вершины посещены, возвратиться назад к предыдущей вершине и продолжить процесс;

– шаг 5. Процесс повторяется до тех пор, пока не будут посещены все вершины, достижимые из начальной вершины.

2 Обход (поиск) в ширину (breadth-first search, BFS) – это способ перебора вершин графа, при котором сначала проходятся смежные вершины, а уже потом вершины на следующем уровне. То есть сначала проходятся все вершины, смежные с начальной вершиной (вершина, с которой начинается обход). Затем исследуются все вершины на расстоянии двух ребер от начальной, затем все на расстоянии трех ребер и т. д.

Для реализации поиска в ширину обычно используются такие структуры данных, как очередь (для хранения вершин, смежных с текущей) и множество (для хранения посещенных вершин).

Пример обхода графа в ширину представлен на рисунке 35.

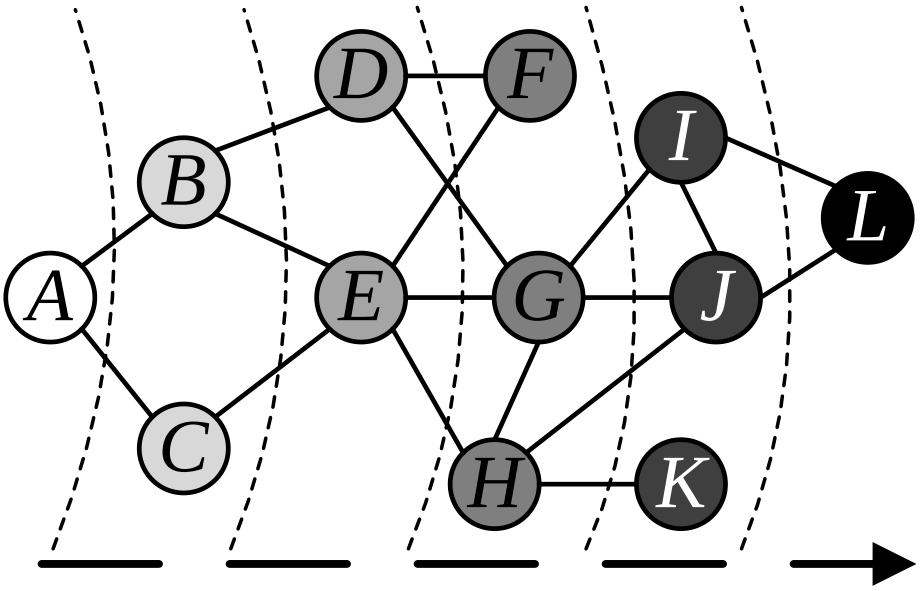


Рисунок 35 – Пример обхода графа в ширину

Алгоритм поиска в ширину:

– шаг 1. Выбранную начальную вершину поместить в очередь;

– шаг 2. Пока очередь не пуста, выполнить следующие действия:

а) извлечь вершину из головы очереди и пометить ее как посещенную;

б) каждую непосещенную соседнюю вершину добавить в очередь.

– шаг 3. Повторять шаг 2, пока не будут посещены все вершины, достижимые из начальной вершины.

Важнейшей задачей в теории графов является поиск кратчайшего пути.

Различают следующие алгоритмы поиска кратчайшего пути на графах:

1 Алгоритм Дейкстры. Это алгоритм поиска кратчайшего пути в графе из заданной вершины во все остальные (single-source shortest path problem).

Алгоритм Дейкстры предназначен для поиска кратчайшего пути от одной вершины ко всем остальным. Работает только для графов без ребер отрицательного веса.

Описание алгоритма:

– шаг 1. Задать граф с вершинами V и матрицей смежности, где `graph[i][j]` хранит вес ребра между вершинами i и j (если ребро отсутствует, записывается значение ∞);

– шаг 2. Создать массив `dist[V]`, в котором `dist[i]` обозначает кратчайшее расстояние от исходной вершины до i-й. Изначально `dist[i] =` ∞, кроме начальной вершины (`dist[start] = 0`). Использовать множество или приоритетную очередь для хранения необработанных вершин;

– шаг 3. Из множества необработанных вершин выбрать вершину u с минимальным значением `dist[u]`;

– шаг 4. Для каждой смежной вершины v проверить, можно ли улучшить ее текущее расстояние `dist[v]` через вершину u. После обработки вершину u пометить как посещенную;

– шаг 5. Алгоритм продолжается, пока не будут обработаны все вершины или не найдены кратчайшие пути до нужных вершин. По завершении `dist[i]` содержит кратчайшее расстояние от исходной вершины до каждой другой.

Временная сложность алгоритма:

– $O(V^2)$ при использовании обычного массива для хранения расстояний;

– $O((V + E) \log V)$ при использовании приоритетной очереди (например, двоичной кучи).

Пространственная сложность:

– $O(V^2)$ при использовании матрицы смежности;

– $O(V + E)$ при использовании списка смежности.

2 Алгоритм Флойда – Уоршелла (алгоритм Флойда). Решает задачу нахождения кратчайших путей между всеми парами вершин во взвешенном графе с положительным или отрицательным весом ребер (но без отрицательных циклов). Этот алгоритм постепенно улучшает известные пути, проверяя, можно ли добраться из вершины *i* в вершину *j* через какую-то промежуточную вершину *k*.

Описание алгоритма:

– шаг 1. Создать матрицу расстояний `dist[V][V]` (где *V* – количество вершин в графе), где элемент `dist[i][j]` обозначает расстояние между вершинами *i* и *j* и равен *w*(*i*, *j*) (весу ребра, если такое существует) или ∞ (если ребра нет). Элемент `dist[i][i]` равен 0 (расстояние от вершины до самой себя);

– шаг 2. Последовательно перебрать все вершины *k* и проверить, можно ли улучшить путь из вершины *i* в вершину *j* через промежуточную вершину *k*:

```
dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
```

– шаг 3. После выполнения алгоритма `dist[i][j]` содержит длину кратчайшего пути между любыми двумя вершинами.

3 Алгоритм Беллмана – Форда. Это алгоритм поиска кратчайшего пути во взвешенном графе. Алгоритм находит кратчайшие пути от одной вершины графа до всех остальных. В отличие от алгоритма Дейкстры, алгоритм Беллмана – Форда допускает ребра с отрицательным весом.

Описание алгоритма:

– шаг 1. Инициализация. Выбрать начальную вершину. Установить расстояние до источника, равное нулю (`distance[source]=0`). Установить расстояние до всех остальных вершин, равное ∞ (`distance[v] = ∞`);

– шаг 2. Релаксация ребер. Повторить *V* – 1 раз (*V* – количество вершин в графе) для каждого ребра (*u*, *v*) с весом *w*. Если расстояние до вершины *u* плюс вес ребра *w* меньше расстояния до вершины *v*, то `distance[v]=distance[u]+w`;

– шаг 3. Проверка на наличие отрицательных циклов. После выполнения *V* – 1 итераций выполнить еще одну итерацию для каждого ребра (*u*, *v*) с весом *w*. Если `(\text{distance[u]}+w<\text{distance[v]}`, то это означает, что существует отрицательный цикл в графе;

– шаг 4. Результат. Если отрицательные циклы не обнаружены, массив расстояний будет содержать кратчайшие расстояния от источника до всех остальных вершин.

ОБЩИЕ РЕКОМЕНДАЦИИ ПО ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ

Лабораторные работы рекомендуется выполнять по следующему алгоритму:

1 Изучить материалы лекций, рекомендованную литературу и теоретическую часть данного издания.

2 Ответить на контрольные вопросы для самоподготовки по теме.

3 Выполнить индивидуальные задания в соответствии с целями работы, применяя обязательный учебный элемент.

4 Подготовить отчет по выполненному заданию.

5 Предоставить отчет преподавателю и защитить свою работу.

Отчет по каждой лабораторной работе подготавливается отдельно и предоставляется преподавателю до защиты. Отчет оформляется в текстовом редакторе (например, в MS Word) в соответствии с типовой формой оформления отчета по лабораторной работе.

Титульный лист должен выполняться по стандарту БГУИР и содержать:

– название университета и структурного подразделения (кафедры);

– наименование дисциплины;

– заголовок лабораторной работы (номер и название);

– номер группы и Ф. И. О. студента;

– дату выполнения (представления к защите) лабораторной работы.

Каждый отчет должен содержать:

– формулировку общей задачи и индивидуального задания;

– описание и/или графическую схему решения задачи;

– листинг кода решения задачи;

– экранные формы с демонстрацией работы программы;

– контрольные примеры (тестовые наборы), которые демонстрируют работу программы с предусмотренными и непредусмотренными входными данными, сообщения о критических ситуациях;

– дополнительные сведения, которые указаны в задании к лабораторной работе.

Защита проводится по каждой работе индивидуально (если иное не указано в задании) в форме беседы студента с преподавателем. По результатам защиты лабораторной работы студенту выставляется оценка по 10-балльной шкале. При получении студентом оценки 4 балла и выше работа считается защищенной. Оценка за лабораторную работу формируется по совокупности критериев выполнения требований к отчету и защите.

Все лабораторные работы должны быть выполнены и защищены до промежуточной аттестации по дисциплине.

1 ЛАБОРАТОРНАЯ РАБОТА № 1. БИНАРНОЕ ДЕРЕВО ПОИСКА

1.1 Краткие теоретические сведения

В программировании и вычислительной технике при решении многих прикладных задач большую популярность получили бинарные (двоичные) деревья. Это деревья, у которых степень вершин может быть не более двух, т. е. вершина дерева может иметь не более двух потомков, называемых левым и правым сыновьями. В отдельный подкласс бинарных деревьев выделены деревья поиска.

Бинарное дерево поиска (БДП) – это несбалансированное двоичное дерево, в котором элементы больше корневого (и далее относительно узлов в поддеревьях) размещаются в правом поддереве, а элементы, которые меньше корневого (и далее относительно узлов в поддеревьях), размещаются в левом поддереве.

К основным операциям, выполняемым с БДП, относятся следующие:

– вставка нового узла в дерево;

– удаление узла из дерева;

– обход дерева.

Создание бинарного древа можно реализовать на основе операции вставки элемента.

Построение БДП (вставка нового элемента) осуществляется по правилам:

– если БДП является пустым, то первый добавляемый элемент становится корнем БДП;

– если новый добавляемый элемент меньше текущего узла, то он рекурсивно добавляется в левое поддерево, т. е. когда будет найден родитель левого поддерева, у которого нет ребенка (ссылка на левое поддерево родителя равна `NULL`);

– если новый добавляемый элемент больше текущего узла либо равен ему, то он рекурсивно добавляется в правое поддерево, т. е. когда будет найден родитель, у которого нет ребенка (ссылка на правое поддерево родителя равна `NULL`).

Пример пошагового построения БДП представлен на рисунке 1.1.

Благодаря рекурсии все элементы, по которым мы движемся, пока не найдем нужное место, складываются в стек. После добавления нового узла рекурсия начнет скручиваться, и мы начнем подниматься к каждому вышестоящему узлу на пути вверх, пока не дойдем до корня.

После окончания построения БДП все узлы окажутся отсортированными: все узлы с меньшими значениями – всегда слева, а все узлы с большими значениями – всегда справа. За счет такой упорядоченности поиск и вставка в БДП осуществляется за $O(\log N)$.

Псевдокод построения БДП представлен функцией `insert` в листинге 1.1.

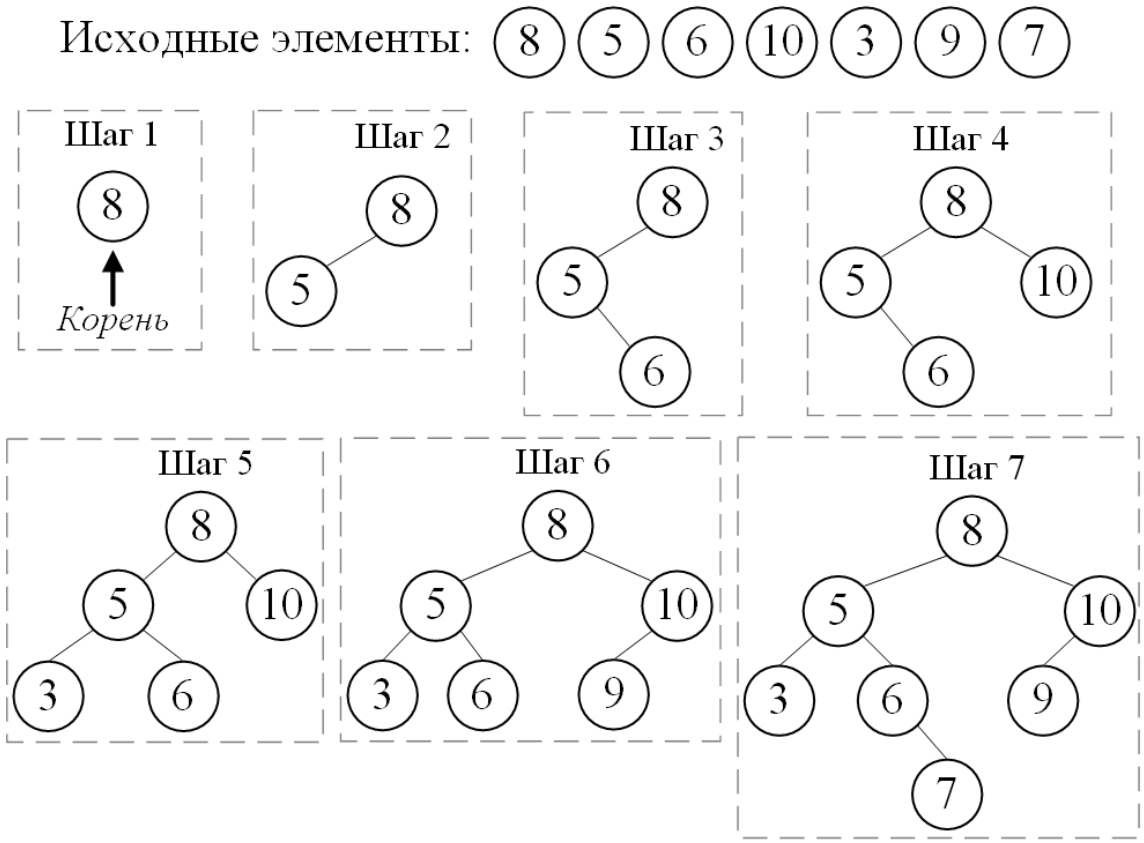


Рисунок 1.1 – Пошаговое построение бинарного дерева поиска

Листинг 1.1 – Пример построения бинарного дерева поиска

```
1   class Node {
2     int key;
3     int value;
4     Node left = null;
5     Node right = null;
6     Node(int key, int value) {this.key = key; this.value = value;}
7   }
8   void insert(Node node, int key, int value) {
9     if(key < node.key) {
10      if(node.left == null) node.left = new Node(key, value);
11      else insert(node.left, key, value);
12    }
13    else if(key >= node.key) {
14      if(node.right == null) node.right = new Node(key, value);
15      else insert(node.right, key, value);
16    }
17  }
```

В ходе решения прикладных задач с применением структур в виде деревьев выполняются различные обходы дерева. Существует несколько способов прохождения всех узлов дерева. Наиболее популярны три следующих способа обхода: прямой (сверху вниз), обратный (снизу вверх), симметричный (слева направо).

Поиск элемента в БДП осуществляется рекурсивно, путем сравнения искомого значения сперва с корневым узлом, а затем в нужном поддереве: левом, если текущий узел больше искомого, или правом, если текущий узел меньше искомого.

Поиск минимального и максимального узлов в БДП представляет еще более тривиальную задачу. Минимальным элементом будет являться самый нижний элемент слева, а максимальным – самый нижний элемент справа.

Для поиска минимального элемента БДП необходимо рекурсивно спускаться влево, пока не будет найден узел со ссылкой, равной `NULL` для левого потомка.

Для поиска максимального элемента БДП необходимо рекурсивно спускаться вправо, пока не будет найден узел со ссылкой, равной `NULL` для правого потомка.

Псевдокод поиска минимального и максимального элементов БДП представлен в листинге 1.2 функциями `getMin` и `getMax` соответственно.

Листинг 1.2 – Поиск по бинарному дереву поиска

```
1   Node search(Node node, int key) {
2     if(node == null) return null;
3     if(node.key == key) return node;
4     return (key < node.key) ? search(node.left, key):
5     search(node.right, key);
6   }
7   Node getMin(Node node) {
8     if(node == null) return null;
9     if(node.left == null) return node;
10    return getMin(node.left);
11  }
12  Node getMax(Node node) {
13    if(node == null) return null;
14    if(node.right == null) return node;
15    return getMax(node.right);
16  }
```

Удаление элемента из БДП осуществляется по следующим правилам:

– если удаляемый узел является листом, то он заменяется на значение `NULL` (рисунок 1.2, *а*);

– если удаляемый узел имеет только одного ребенка, то он заменяется на этого ребенка (рисунок 1.2, *б*).

В обоих этих случаях удаляемый узел, по сути, заменяется на своего ребенка (существующего или значение `NULL`);

– если удаляемый узел имеет двух детей, то на место удаляемого узла должен встать узел, который будет большим в левом поддереве удаляемого или меньшим в правом поддереве относительно удаляемого. Таким образом, необходимо найти либо максимальный элемент левого поддерева, либо минимальный

элемент правого поддерева. Найденный элемент перезаписывается на место удаляемого и затем рекурсивно удаляется по одному из первых двух правил (рисунок 1.2, *в*).

Псевдокод удаления элемента из БДП представлен функцией `delete` в листинге 1.3.

Листинг 1.3 – Удаление элементов из бинарного дерева поиска

```
1   Node delete(Node node, int key) {
2     if(node == null) return null;
3     else if(key < node.key) node.left = delete(node.left, key);
4     else if(key > node.key) node.right = delete(node.right, key);
5     else {
6       if(node.left == null || node.right == null)
7         node = (node.left == null) ? node.right : node.left;
8       else {
9         Node maxinLeft = getMax(node.left);
10        node.key = maxinLeft.key;
11        node.value = maxinLeft.value;
12        node.right = delete(node.right, maxinLeft.key);
13      }
14    }
15    return node;
16  }
```

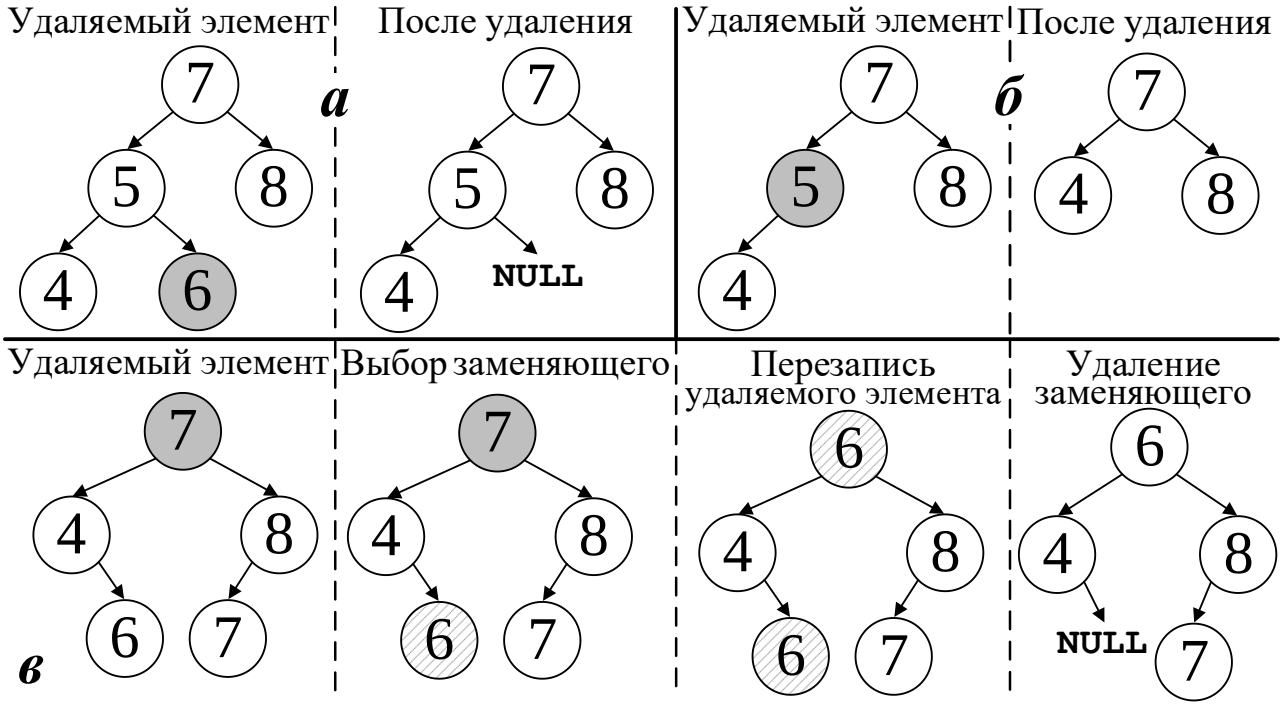


Рисунок 1.2 – Примеры удаления элементов из бинарного дерева поиска

Для проверки правильности удаления узла из БДП можно использовать один из видов обхода дерева: симметричный, обратный или прямой.

При симметричном обходе БДП все узлы будут выведены в порядке их возрастания (функция `printTree` в листинге 1.4). Если узлы будут выведены в одинаковом порядке до и после удаления, значит удаление прошло правильно.

При обратном обходе сперва будет выведен левый ребенок, затем правый, затем родитель (функция `deleteTree` в листинге 1.4).

При прямом обходе сперва будет выведен левый ребенок, затем родитель, затем правый ребенок (функция `copyTree` в листинге 1.4). Такой обход может быть использован при копировании БДП в память, поскольку дерево проходится в том порядке, в котором они были размещены (сверху вниз).

Примеры обходов представлены на рисунке 1.3.

Листинг 1.4 – Примеры реализаций обходов

```
1   void printTree(Node node) {
2      if(node == null) return;
3      printTree(node.left);
4      print(node.value);
5      printTree(node.right);
6   }
7   void deleteTree (Node node) {
8      if(node == null) return;
9      deleteTree(node.left);
10     deleteTree(node.right);
11     print(node.value);
12  }
13  void copyTree(Node node) {
14     if(node == null) return;
15     copyTree(node.left);
16     copyTree(node.right);
17     print(node.value);
18  }
```

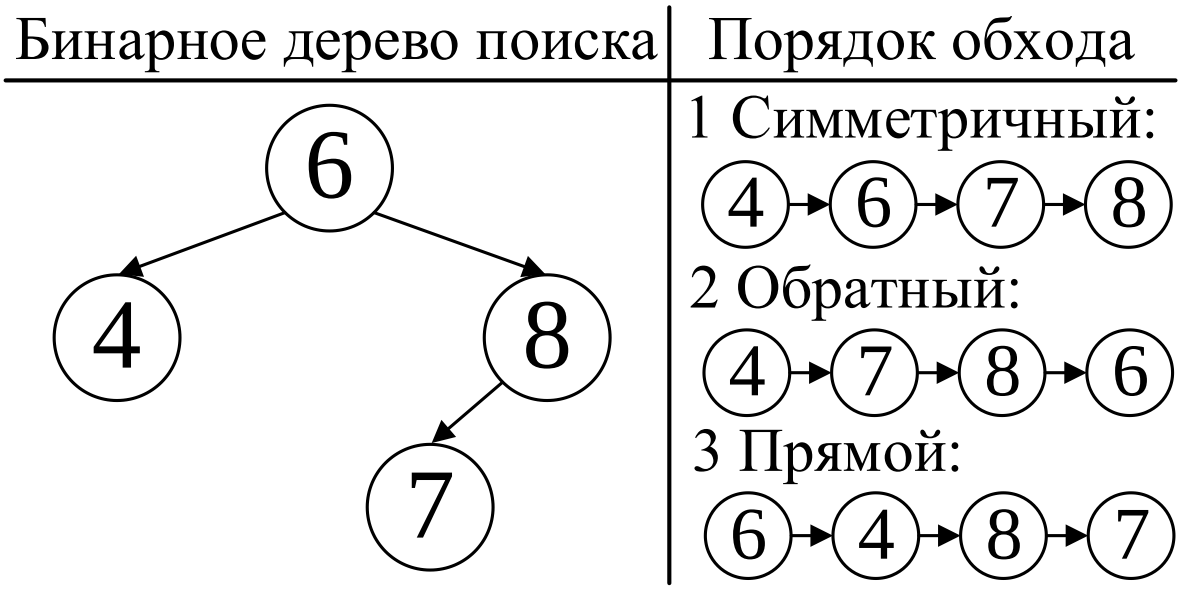


Рисунок 1.3 – Пример различных обходов бинарного дерева поиска

1.2 Задание

1 Согласовать индивидуальный вариант задания с преподавателем (таблица 1.1), внимательно изучить формулировку задачи и пример.

2 Разработать алгоритм и написать программный код для решения следующей задачи.

Организовать ввод от 10 до 15 целых чисел, из которых построить бинарное дерево поиска с использованием указателей.

Реализовать обход дерева прямым, симметричным и обратным способами. Вывести результаты обходов (порядок обхода вершин).

Реализовать процедуры поиска, вставки и удаления элементов бинарного дерева поиска. Вывести результаты работы процедур.

В соответствии с индивидуальным заданием реализовать пользовательскую функцию обработки дерева. Организовать вывод результатов работы.

3 Организовать текстовый пользовательский интерфейс в программе. При выполнении задания реализовать ввод исходных данных пользователем с клавиатуры, а вывод результатов выполнения программы в консоль (на экран). Предусмотреть возможность автоматического заполнения случайными данными в заданном диапазоне.

4 Проверить правильность вычислений на тестовых примерах, выполнив серию контрольных прогонов программы.

5 ***Дополнительно*** реализовать визуальное отображение построенного бинарного дерева поиска в консоли.

Таблица 1.1 − Варианты индивидуальных заданий для лабораторной работы № 1

Вариант	**Задание**
0	Реализуйте прямой, симметричный и обратный обходы дерева. Реализуйте функции поиска, вставки и удаления элементов
1	Реализуйте функцию поиска минимума и максимума в дереве
2	Реализуйте функцию, которая ищет заданный узел и, если он найден, предлагает заменить его значение на новое, введенное пользователем
3	Реализуйте функцию удаления всех листовых узлов дерева
4	Реализуйте функцию нахождения ближайшего большего элемента для заданного значения
5	Реализуйте функцию определения уровня узла с заданным значением
6	Реализуйте функцию вывода всех листовых узлов дерева

Вариант	Задание
7	Реализуйте функцию определения количества узлов, имеющих только одного потомка
8	Реализуйте функцию подсчета узлов с четными значениями
9	Реализуйте функцию вывода самого длинного пути в дереве
10	Реализуйте функцию определения уровня самого глубокого узла
11	Реализуйте функцию нахождения всех узлов на заданном уровне
12	Реализуйте функцию подсчета количества полных узлов (имеющих двух потомков)
13	Вывести среднее значение всех элементов
14	Реализуйте поиск предшественника и последователя для заданного узла
15	Реализуйте функцию, вычисляющую произведение значений всех узлов дерева
16	Реализуйте функцию, которая выводит номер уровня, на котором сумма значений узлов четная
17	Реализуйте функцию поиска, которая также выводит путь поиска от корня до нужного узла
18	Реализуйте операцию удаления всех узлов, значение которых превышает заданный порог
19	Реализуйте функцию вычисления среднего арифметического значений всех листьев дерева
20	Реализуйте функцию, вычисляющую сумму всех элементов дерева
21	Подсчитайте количество узлов, имеющих два ребенка
22	Реализуйте функцию определения количества узлов, не имеющих левого потомка
23	Реализуйте функцию нахождения суммы всех значений на нечетных уровнях дерева
24	Реализуйте функцию вывода узлов, находящихся только на четных уровнях
25	Реализуйте функцию вычисления среднего арифметического значений всех узлов
26	Реализуйте функцию вычисления разницы между максимальным и минимальным значениями

Вариант	Задание
27	Реализуйте функцию поиска элемента с последующим выводом его родителя
28	Реализуйте функцию нахождения суммы всех значений на четных уровнях дерева
29	Реализуйте функцию, которая выводит номер уровня, на котором сумма значений узлов нечетная
30	Реализуйте функцию подсчета узлов с нечетными значениями
31	Реализуйте функцию подсчета количества неполных узлов (имеющих менее двух потомков)

1.3 Пример выполнения задания

Для выполнения задания (вариант 0) сначала определим структуру узла бинарного дерева поиска. Каждый узел содержит целочисленное значение, а также указатели на левый и правый дочерние узлы.

Функция `createNode` выделяет память для нового узла, инициализирует его значением и устанавливает указатели на дочерние узлы в `nullptr`.

Функция `insert` рекурсивно реализует вставку нового элемента в дерево. Если дерево пустое, создается новый узел; если значение меньше значения текущего узла, происходит переход в левое поддерево, иначе – в правое.

Обход дерева реализован тремя функциями:

1 Прямой (`preorder`) – сначала выводится значение текущего узла, затем происходит обход левого и правого поддеревьев.

2 Симметричный (`inorder`) – сначала выполняется обход левого поддерева, затем выводится значение узла, далее – обход правого поддерева.

3 Обратный (`postorder`) – сначала происходит обход левого и правого поддеревьев, а затем выводится значение текущего узла.

Функция `find` рекурсивно ищет заданное значение. Если текущий узел равен `nullptr` или значение найдено, возвращается этот узел, иначе – осуществляется переход влево или вправо в зависимости от сравнения.

Процедура удаления элемента реализована функцией `remove`, которая обрабатывает следующие случаи:

– узел не найден;

– узел имеет один или не имеет дочерних элементов;

– узел имеет два дочерних элемента – в этом случае значение узла заменяется на минимальное значение в правом поддереве (определяемое функцией `findMinNode`), после чего удаляется найденный минимальный узел.

Функция `inputNumber` обеспечивает корректный ввод целочисленных значений с проверкой ввода.

В функции `main` запрашивается количество элементов и их ввод пользователем. Далее на основе введенных чисел строится бинарное дерево поиска с помощью функции `insert`. Затем выводятся результаты обхода дерева тремя способами (прямым, симметричным и обратным). Также реализуются операции поиска, вставки и удаления элементов с последующим выводом состояния дерева.

Программная реализация алгоритма решения задания для варианта 0 представлена в листинге 1.5.

Листинг 1.5 – Программная реализация индивидуального задания по лабораторной работе № 1

```
#include <iostream>
#include <vector>
using namespace std;

struct node {
        int value;
        node* left;
        node* right;
};

node* createNode(int value) {
        node* newNode = new node;
        newNode->value = value;
        newNode->left = nullptr;
        newNode->right = nullptr;
        return newNode;
}

node* insert(node* root, int value) {
        if (root == nullptr) {
                return createNode(value);
        }
        if (value < root->value)
                root->left = insert(root->left, value);
        else
                root->right = insert(root->right, value);
        return root;
}

void directTraversal(node* root) {
        if (root == nullptr) {
                return;
        cout << root->value << " ";
        directTraversal(root->left);
        directTraversal(root->right);
}
```

```
37  void reverseTraversal(node* root) {
38          if (root == nullptr) {
39                  return;
40          reverseTraversal(root->left);
41          reverseTraversal(root->right);
42          cout << root->value << " ";
43  }
44  void symmetricTraversal(node* root) {
45          if (root == nullptr) {
46                  return;
47          symmetricTraversal(root->left);
48          cout << root->value << " ";
49          symmetricTraversal(root->right);
50  }
51
52  node* find(node* root, int value) {
53          if (root == nullptr || root->value == value) {
54                  return root;
55          }
56          return (value < root->value) ? find(root->left, value) :
57          find(root->right, value);
58  }
59
60  node* findMinNode(node* root) {
61          while (root->left != nullptr) {
62                  root = root->left;
63          }
64          return root;
65  }
66  node* remove(node* root, int value) {
67          if (root == nullptr) {
68                  return nullptr;
69          if (value < root->value) {
70                  root->left = remove(root->left, value);
71          }
72          else if (value > root->value) {
73                  root->right = remove(root->right, value);
74          }
75          else {
76                  if (root->left == nullptr) {
77                          node* temp = root->right;
78                          delete root;
79                          return temp;
80                  }
81                  else if (root->right == nullptr) {
82                          node* temp = root->left;
83                          delete root;
84                          return temp;
85                  }
86                  else
87                  {
88                    node* temp = findMinNode(root->right);
```

```
89                      root->value = temp->value;
90                      root->right = remove(root->right, temp->value);
91                      }
92          }
93          return root;
94  }
95  void inputNumber(int* num){
96        while (!(cin >> *num)) {
97          cout <<"Некорректное значение. Введите целое число:";
98          cin.clear();
99          cin.ignore( numeric_limits<streamsize>::max(), '\n');
100         }
101 }
102
103 int main() {
104         setlocale(LC_ALL, "ru");
105         int n;
106         cout <<"Введите количество элементов (~ 10-15): ";
107         inputNumber(&n);
108
109         vector<int> values;
110         cout << "Введите " << n << " целых чисел: ";
111         for (int i = 0; i < n; i++) {
112                 int temp;
113                 inputNumber(&temp);
114                 values.push_back(temp);
115         }
116
117         node* root = nullptr;
118         for (int i = 0; i < n; i++) {
119                 root = insert(root, values[i]);
120         }
121         cout << "Прямой обход: ";
122         directTraversal(root);
123         cout << endl;
124         cout << "Симметричный обход: ";
125         symmetricTraversal(root);
126         cout << endl;
127         cout << "Обратный обход: ";
128         reverseTraversal(root);
129         cout << endl;
130         int searchValue;
131         cout << "Введите значение для поиска: ";
132         inputNumber(&searchValue);
133
134         node* foundNode = find(root, searchValue);
135         cout << "Элемент " << searchValue;
136         if (foundNode != nullptr) {
137                 cout << " найден." << endl;
138         }
```

```
139         else {
140                 cout << " не найден." << endl;
141         }
142         int insertValue;
143         cout << "Введите значение для вставки: ";
144         inputNumber(&insertValue);
145
146         root = insert(root, insertValue);
147         cout << "Дерево после вставки: ";
148         directTraversal(root);
149         cout << endl;
150
151         int removeValue;
152         cout << "Введите значение для удаления: ";
153         inputNumber(&removeValue);
154
155         root = remove(root, removeValue);
156         cout << "Дерево после удаления: ";
157         directTraversal(root);
158         cout << endl;
159         return 0;
160 }
```

Результаты контрольных прогонов программы, демонстрирующие работу алгоритма на различных входных данных, представлены на рисунках 1.4–1.8.

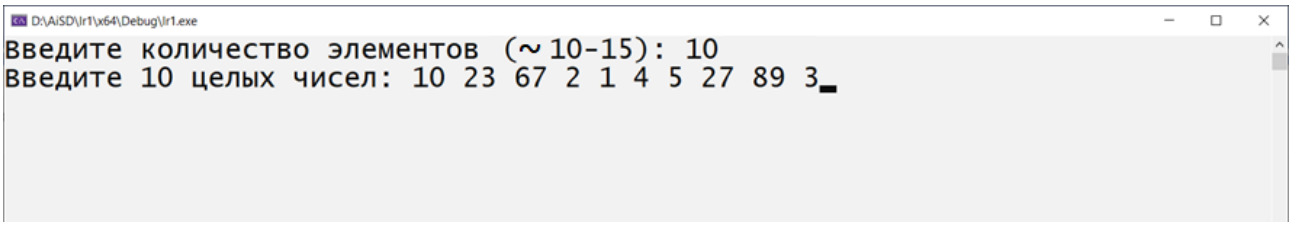


Рисунок 1.4 – Демонстрация ввода чисел для построения дерева

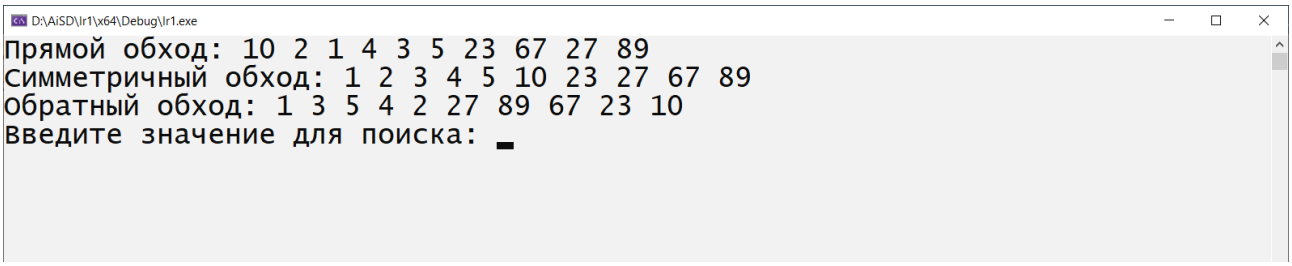


Рисунок 1.5 – Результат работы функции вывода обходов дерев

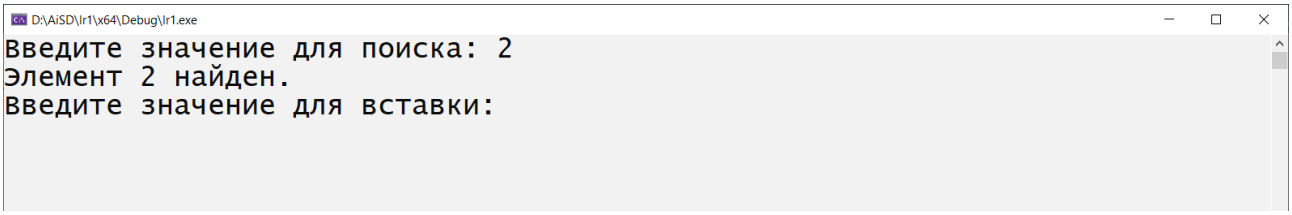


Рисунок 1.6 – Результат работы функции поиска узла

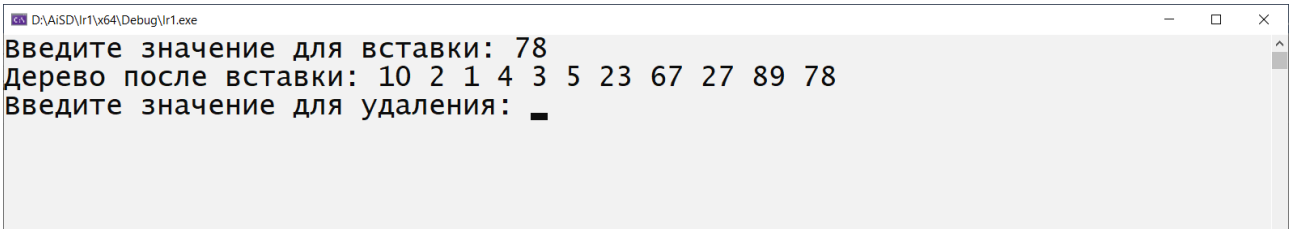


Рисунок 1.7 – Результат работы функции вставки узла

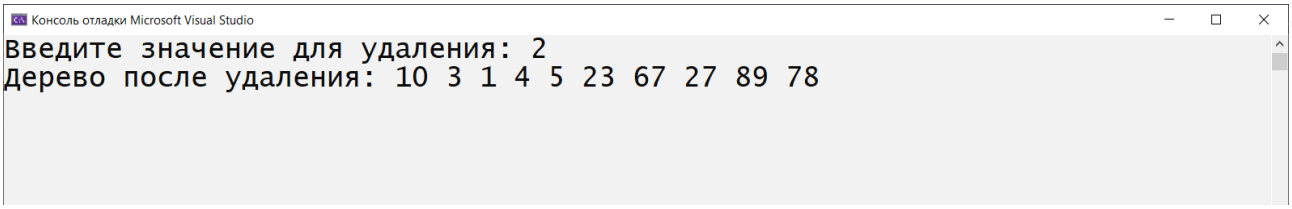


Рисунок 1.8 – Результат работы функции удаления узла

1.4 Контрольные вопросы

1 Что такое бинарное дерево поиска?

2 Какие свойства есть у любого узла бинарного дерева поиска?

3 Как выполняется вставка нового узла в бинарное дерево поиска?

4 Какие способы обхода бинарного дерева существуют? В чем их различия?

5 Как реализовать поиск элемента в бинарном дереве поиска?

6 Как удалить узел из бинарного дерева поиска? Какие случаи нужно учитывать при удалении?

7 Какая временная сложность у основных операций (вставка, удаление, поиск) для бинарного дерева поиска в лучшем и худшем случае?

8 Что такое сбалансированность дерева поиска? В каких случаях дерево вырождается в «бамбук»?

9 Можно ли вставить в бинарное дерево поиска несколько одинаковых элементов?

10 Какие есть способы программной реализации бинарного дерева поиска?

11 Какие структуры данных могут быть альтернативой бинарному дереву поиска? В каких случаях?

12 Как с помощью бинарного дерева поиска можно реализовать сортировку?

2 ЛАБОРАТОРНАЯ РАБОТА № 2. АЛГОРИТМЫ ПОИСКА НА ОРИЕНТИРОВАННЫХ ГРАФАХ

2.1 Краткие теоретические сведения

Ориентированный граф (орграф) – это топологическая модель данных, состоящая из множества объектов данных (вершин) и множества линий связи (отношений) этих объектов данных друг с другом (дуг), причем эти линии связи имеют направление (начало и конец).

Ориентированный граф – граф, в котором каждое ребро имеет направление от одной вершины к другой.

Дуга – это ребро в ориентированном графе, имеющее направление.

Полустепень захода вершины – количество дуг, заходящих в эту вершину.

Исток – вершина с нулевой полустепенью захода.

Полустепень исхода вершины – количество дуг, исходящих из этой вершины.

Сток – вершина с нулевой полустепенью исхода.

Алгоритмы на графах решают различные задачи, такие как:

– нахождение маршрута от одного объекта к другому;

– поиск связанных компонент;

– вычисление кратчайших путей;

– поиск сети максимального потока и др.

Структуры данных для работы с графами

Поскольку граф – это пара множеств вершин и связей между ними, то можно использовать двумерный массив (матрицу) для представления графа. Также для представления графа может использоваться список и другие более сложные структуры данных. Выбор структуры данных для графа зависит от решаемой задачи и является определяющим эффективность работы алгоритма.

Граф может быть задан следующими структурами данных:

1 Матрицей смежности. Матрица смежности отражает наличие связи (ребер) между вершинами (элементы матрицы), а сами вершины являются индексами матрицы.

Для невзвешенных графов значения ячеек матрицы смежности следующие:

0 – отсутствие связи (ребра);

1 – наличие связи (ребра).

Для взвешенных графов значения ячеек матрицы смежности следующие:

0 – отсутствие связи (ребра);

1…n – вес ребра.

Если у графа нет петель, то все ячейки матрицы, расположенные на главной диагонали, всегда равны нулю, т. к. ни у одной вершины нет ребра, которое и начинается, и заканчивается в ней.

Для графа с количеством вершин N матрица смежности будет иметь размерность $N \times N$ (т. е. необходима память N^2).

Для неориентированных графов без петель матрица смежности симметрична относительно главной диагонали. Благодаря этому можно уменьшить объем памяти (восстановив одну из половин относительно диагонали), необходимый для ее хранения, вдвое ($N^2 / 2$).

2 Списком смежности. Список смежности представляет собой списки всех вершин и ссылки на каждую смежную с ними.

Не имеет значения порядок расположения ссылок, т. к. смежность рассматривается относительно первой ячейки, все остальные ссылки указывают лишь на связь с ней, а не между собой. Это неизбежно приводит к дублированию вершин.

Для неориентированного графа сумма длин всех списков определяется как удвоенная мощность множества ребер ($2 \cdot |E_{\text{неориент}}|$), а объем необходимой памяти равен сумме всех ребер (ссылок) и вершин списка ($E_{\text{неориент}} + V$).

Для ориентированного графа сумма длин всех списков определяется как мощность множества ребер ($|E_{\text{ориент}}|$), и объем необходимой памяти соответственно тоже будет меньшим ($E_{\text{ориент}} + V$).

Пример матриц и списков смежности для неориентированного графа представлен на рисунке 2.1, *а*, а для ориентированного – на рисунке 2.1, *б*.

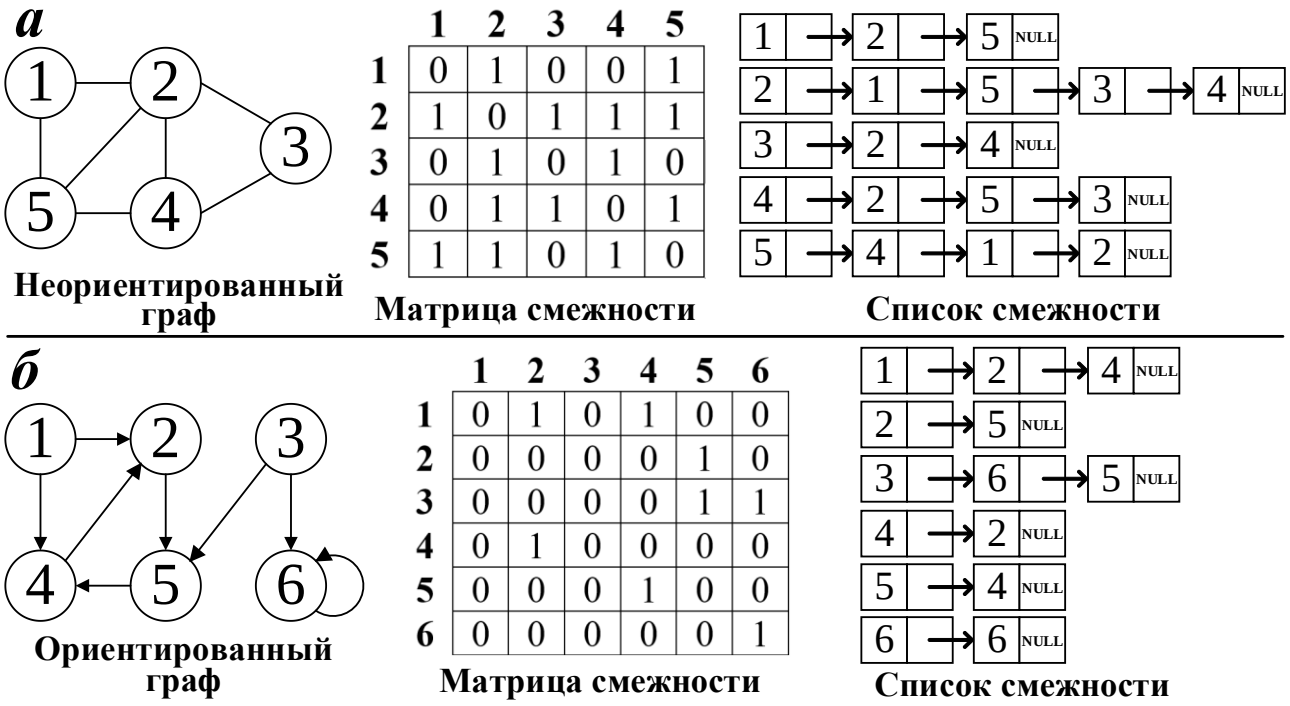


Рисунок 2.1 – Примеры матриц и списков смежности графов

3 Матрицей инцидентности. Матрица инцидентности представляет собой двумерный массив размерностью $N \times M$, где N – число строк, равное числу вершин графа, а M – число столбцов, равное числу ребер графа.

Индекс i обозначает порядковый номер вершины ($i = \overline{1, N}$), а индекс j обозначает порядковый номер ребра ($j = \overline{1, M}$).

Для неориентированных графов значения ячеек матрицы инцидентности:

0 – если вершина N_i не инцидентна ребру M_j;

1 – если вершина N_i инцидентна ребру M_j.

Для ориентированных графов значения ячеек матрицы инцидентности:

0 – если вершина N_i не инцидентна ребру M_j;

1 – если вершина N_i начало дуги M_j;

–1 – если вершина N_i конец дуги M_j;

2 – если вершина N_i является и началом, и концом петли M_j.

Матрица инцидентности имеет свои особенности:

– в столбце может быть только две ненулевых ячейки, т. к. у ребра всегда есть начало и конец;

– при суммировании строки ячейки со значением –1 могут складываться только с ячейками, также имеющими значение –1;

– при суммировании строки ячейки со значением 1 могут складываться только с ячейками, также имеющими значение 1.

Последние две особенности позволяют узнать степень входа и степень выхода из вершины (например, при сложении для первой вершины получается сумма, сигнализирующая, что из нее исходит одно ребро и входят два других ребра).

Объем необходимой памяти для матрицы смежности равен произведению мощностей множеств вершин и ребер ($|V| \cdot |E|$).

4 Списком инцидентности. Список инцидентности представляет собой списки всех ребер и ссылки на каждую связанную с ними вершину.

Пример матриц и списков инцидентности для неориентированного графа представлен на рисунке 2.2, *а*, а для ориентированного – на рисунке 2.2, *б*.

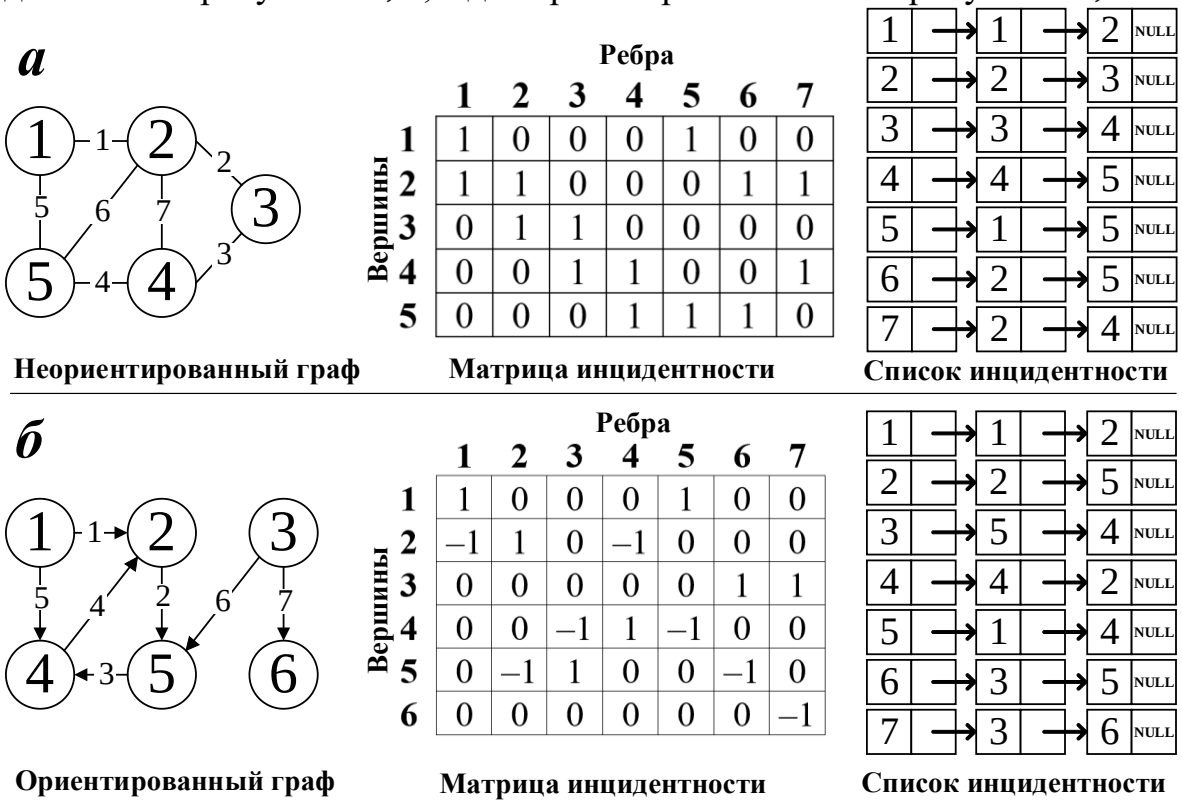


Рисунок 2.2 – Примеры матриц и списков инцидентности графов

Недостатком списка инцидентности является сложность определения наличия конкретного ребра (для этого требуется выполнить поиск по списку).

5 Классом и хеш-таблицей. Отдельно взятый узел представляет собой класс, который хранит его значение и ссылки на его ребра и его родителей (класс `Node` в листинге 2.1).

Ребро в свою очередь тоже представляет собой отдельный класс, который хранит значение веса ребра, и ссылку на узел, к которому ведет (класс `Edge` в листинге 2.1).

Сам граф будет реализован хеш-таблицей (парой «ключ – значение»), в котором ключом выступает значение узла, а значением – сам узел (рисунок 2.3).

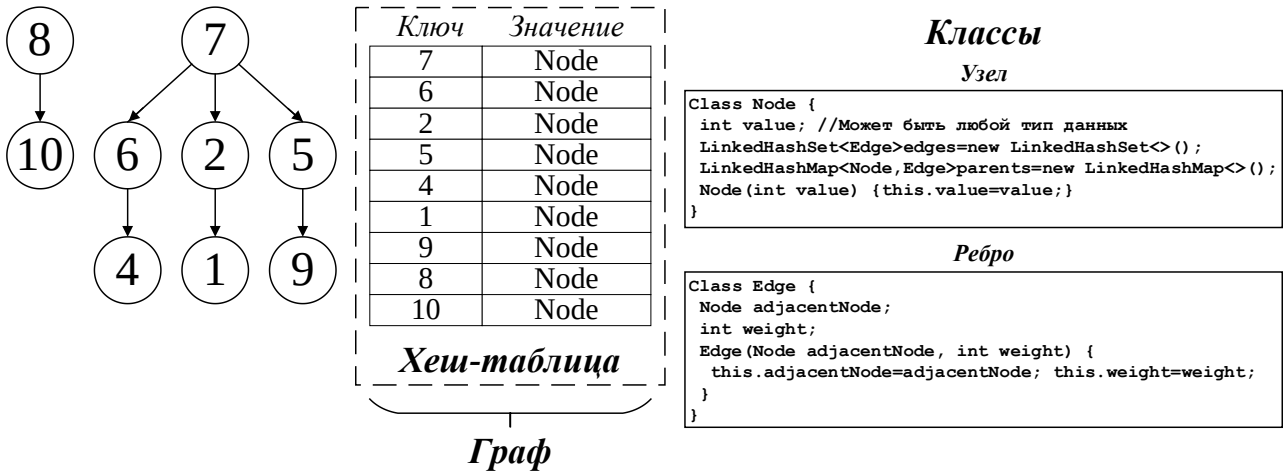


Рисунок 2.3 – Пример реализации графа с помощью классов и хеш-таблицы

Преимуществом такой организации является то, что значением узла уже может являться любой тип данных и, кроме этого, можно легко получить значения как всех его детей, так и родителей.

Метод добавления нового узла в такую структуру данных можно сразу объединить с методом поиска уже существующего узла. Проверяется переданное значение и если им оказывается –1, то значит узла в графе не существует, возвращается значение `NULL` и создается новый узел. Если переданное методу значение существует, то возвращается этот узел (метод `addOrGetNode` в листинге 2.1).

Для генерации графа (хеш-таблицы) перебираются переданные значения и вызывается метод `addOrGetNode`, который добавляет их в хеш-таблицу. Далее в объект класса `Edge` записываются необходимые ребра между узлами (в объект узла, от которого идет ребро, а в объект узла, в который идет ребро, запишется родитель, из которого оно идет) (метод `createGraph` в листинге 2.1).

Листинг 2.1 – Работа с графом через классы и хеш-таблицы

```
1   class Node {
2     int value; //значение узла
3     LinkedHashSet<Edge> edges=new LinkedHashSet<>0;//список ребер
4     //список родителей:
5     LinkedHashMap< Node, Edge> parents = new LinkedHashMap<>0;
6     Node(int value) {this.value = value;}
7   }
8   class Edge {
9     Node adjacentNode; //узел, к которому ведет ребро
10    int weight; //вес ребра
11    Edge(Node adjacentNode, int weight) {
12    this.adjacentNode = adjacentNode; this.weight = weight;
13    }
14  }
15  //Метод добавления или поиска узла
16  Node addOrGetNode(HashMap<Integer, Node> graph, int value) {
17    if(value == -1) retum null;
18    if(graph.containsKey(value)) return graph.get(value);
19    Node node = new Node(value);
20    graph.put(value, node);
21    return node;
22  }
23  //Метод генерации графа в хеш-таблицу
24  HashMap<Integer,Node> createGraph(int[][] graphData) {
25    HashMap<Integer,Node> graph = new HashMap<>();
26    for(int[]row: graphData) {
27      Node node = addOrGetNode(graph, row[0]);
28      Node adjacentNode = addOrGetNode(graph, row[1]);
29      if(adjacentNode == null) continue;
30      Edge edge = new Edge(adjacentNode, row[2]);
31      node.edges.add(edge);
32      adjacentNode.parents.put(node, edge);
33    }
34    return graph;
35  }
```

2.2 Задание

1 Согласовать индивидуальный вариант задания с преподавателем (таблица 2.1), внимательно изучить формулировку задачи и пример.

2 Разработать алгоритм и написать программный код для решения следующей задачи.

Представить ориентированный граф, состоящий из 7–10 вершин, с помощью матрицы смежности.

Указать вершину *v* и определить список вершин, смежных с вершиной *v*.

Указать вершину *v* и определить список вершин, из которых можно попасть в вершину *v*.

В соответствии с индивидуальным заданием реализовать пользовательскую функцию обработки графа. Организовать вывод результатов работы.

Внимание! При необходимости следует задать ребрам вес.

3 Организовать текстовый пользовательский интерфейс в программе. При выполнении задания реализовать ввод исходных данных пользователем с клавиатуры, а вывод результатов выполнения программы в консоль (на экран). Предусмотреть возможность автоматического заполнения случайными данными в заданном диапазоне.

4 Проверить правильность вычислений на тестовых примерах, выполнив серию контрольных прогонов программы.

5 ***Дополнительно*** реализовать визуальное отображение построенного графа в консоли.

Таблица 2.1 − Варианты индивидуальных заданий для лабораторной работы № 2

Вариант	**Задание**
0	Определить кратчайшие пути от вершины-источника до всех вершин орграфа на основе алгоритма Дейкстры
1	Определить кратчайшие пути от вершины-источника до всех вершин орграфа на основе алгоритма Дейкстры. Определить число вершин, имеющих только исходящие ребра
2	Определить кратчайшие расстояния между каждой парой вершин орграфа на основе алгоритма Флойда. Определить количество вершин, имеющих только входящие ребра
3	Определить кратчайшие пути от вершины-источника до всех вершин орграфа на основе алгоритма Дейкстры. Определить количество вершин, не имеющих исходящих ребер
4	Определить кратчайшие расстояния между каждой парой вершин орграфа на основе алгоритма Флойда. Определить количество вершин, не имеющих входящих ребер
5	Определить кратчайшие пути от вершины-источника до всех вершин орграфа на основе алгоритма Дейкстры. Найти вершины, соединенные двусторонними ребрами
6	Определить кратчайшие расстояния между каждой парой вершин орграфа на основе алгоритма Флойда. Найти вершину, из которой можно достичь наибольшего количества других вершин
7	Определить кратчайшие пути от вершины-источника до всех вершин орграфа на основе алгоритма Дейкстры. Найти вершину, достижимую из наибольшего количества других вершин

Вариант	Задание
8	Определить кратчайшие расстояния между каждой парой вершин орграфа на основе алгоритма Флойда. Определить, сколько различных путей существует между двумя заданными вершинами
9	Определить кратчайшие пути от вершины-источника до всех вершин орграфа на основе алгоритма Дейкстры. Найти вершину, через которую проходит наибольшее количество кратчайших путей
10	Определить кратчайшие расстояния между каждой парой вершин орграфа на основе алгоритма Флойда. Найти все вершины, имеющие равное количество входящих и исходящих ребер
11	Определить кратчайшие пути от вершины-источника до всех вершин орграфа на основе алгоритма Дейкстры. Найти самую длинную цепь в графе
12	Определить кратчайшие расстояния между каждой парой вершин орграфа на основе алгоритма Флойда. Найти все вершины, достижимые из заданной вершины не более чем за три шага
13	Определить кратчайшие пути от вершины-источника до всех вершин орграфа на основе алгоритма Дейкстры. Вывести все вершины, имеющие четное число входящих ребер
14	Определить кратчайшие расстояния между каждой парой вершин орграфа на основе алгоритма Флойда. Найти вершину, имеющую наибольшее расстояние до всех остальных вершин
15	Определить кратчайшие пути от вершины-источника до всех вершин орграфа на основе алгоритма Дейкстры. Найти количество путей заданной длины между двумя вершинами
16	Определить кратчайшие расстояния между каждой парой вершин орграфа на основе алгоритма Флойда. Определить число вершин, имеющих только исходящие ребра
17	Определить кратчайшие пути от вершины-источника до всех вершин орграфа на основе алгоритма Дейкстры. Определить количество вершин, имеющих только входящие ребра
18	Определить кратчайшие расстояния между каждой парой вершин орграфа на основе алгоритма Флойда. Определить количество вершин, не имеющих исходящих ребер
19	Определить кратчайшие пути от вершины-источника до всех вершин орграфа на основе алгоритма Дейкстры. Определить количество вершин, не имеющих входящих ребер

Вариант	Задание
20	Определить кратчайшие расстояния между каждой парой вершин орграфа на основе алгоритма Флойда. Найти вершины, соединенные двусторонними ребрами
21	Определить кратчайшие пути от вершины-источника до всех вершин орграфа на основе алгоритма Дейкстры. Найти вершину, из которой можно достичь наибольшее количество других вершин
22	Определить кратчайшие расстояния между каждой парой вершин орграфа на основе алгоритма Флойда. Найти вершину, которая достижима из наибольшего количества других вершин
23	Определить кратчайшие пути от вершины-источника до всех вершин орграфа на основе алгоритма Дейкстры. Определить, сколько различных путей существует между двумя заданными вершинами
24	Определить кратчайшие расстояния между каждой парой вершин орграфа на основе алгоритма Флойда. Найти вершину, через которую проходит наибольшее количество кратчайших путей
25	Определить кратчайшие пути от вершины-источника до всех вершин орграфа на основе алгоритма Дейкстры. Найти все вершины, имеющие равное количество входящих и исходящих ребер
26	Определить кратчайшие расстояния между каждой парой вершин орграфа на основе алгоритма Флойда. Найти самую длинную цепь в графе
27	Определить кратчайшие пути от вершины-источника до всех вершин орграфа на основе алгоритма Дейкстры. Найти все вершины, достижимые из заданной вершины не более чем за три шага
28	Определить кратчайшие расстояния между каждой парой вершин орграфа на основе алгоритма Флойда. Вывести все вершины, имеющие четное количество входящих ребер
29	Определить кратчайшие пути от вершины-источника до всех вершин орграфа на основе алгоритма Дейкстры. Найти вершину, имеющую наибольшее расстояние до всех остальных вершин
30	Определить кратчайшие расстояния между каждой парой вершин орграфа на основе алгоритма Флойда. Найти количество путей заданной длины между двумя вершинами

Вариант	Задание
31	Определить кратчайшие пути от вершины-источника до всех вершин орграфа на основе алгоритма Дейкстры. Определить количество вершин, имеющих только исходящие ребра

2.3 Пример выполнения задания

Для выполнения задания (вариант 0) сначала определим структуру графа. Граф хранится в виде матрицы смежности, где каждая ячейка отражает наличие ребра между вершинами. Функция `createGraph` создает граф с заданным числом вершин, инициализируя матрицу смежности нулями.

Функция `printGraph` выводит матрицу смежности, демонстрируя структуру графа.

Функция `findAdjacentVertices` возвращает список вершин, непосредственно смежных с заданной (таких, в которые ведет ребро из этой вершины), а функция `findVerticesToReach` определяет список вершин, из которых можно попасть в указанную вершину.

Алгоритм Дейкстры, реализованный в функции `dijkstra`, находит кратчайшие пути от исходной вершины до всех остальных, считая вес ребра равным единице. Для оптимизации поиска используется приоритетная очередь.

В основной функции (`main`) происходит ввод количества вершин, заполнение матрицы смежности, вывод структуры графа, а затем выполнение анализа: определяется список смежных вершин, список вершин, ведущих в заданную, и вычисляются кратчайшие пути от указанной вершины с последующим выводом результата.

Программная реализация алгоритма решения задания для варианта 0 представлена в листинге 2.2.

Листинг 2.2 – Программная реализация индивидуального задания по лабораторной работе № 2 на языке программирования C++

```
#include <iostream>
#include <vector>
#include <limits>
#include <queue>
using namespace std;
const int INF = numeric_limits<int>::max();
struct Graph {
    int V;
    vector<vector<int>> adjMatrix;
};
Graph createGraph(int vertices) {
 Graph g;
 g.V=vertices;
 g.adjMatrix=vector<vector<int>>(vertices,vector<int>(vertices,0));
 return g;
}
```

```
17  void printGraph(Graph& g) {
18      for (int i = 0; i < g.V; i++) {
19          for (int j = 0; j < g.V; j++)
20              cout << g.adjMatrix[i][j] << " ";
21          cout << endl;
22      }
23  }
24
25  vector<int> findAdjacentVertices(Graph& g, int v) {
26      vector<int> adjacentVertices;
27      for (int i = 0; i < g.V; i++) {
28          if (g.adjMatrix[v][i] == 1)
29              adjacentVertices.push_back(i);
30      }
31      return adjacentVertices;
32  }
33
34  vector<int> findVerticesToReach(Graph& g, int v) {
35      vector<int> verticesToReach;
36      for (int i = 0; i < g.V; i++) {
37          if (g.adjMatrix[i][v] == 1) {
38              verticesToReach.push_back(i);
39          }
40      }
41      return verticesToReach;
42  }
43
44  vector<int> dijkstra(Graph& g, int source) {
45      vector<int> dist(g.V, INF);
46      dist[source] = 0;
47      priority_queue<pair<int, int>, vector<pair<int, int>>,
48      greater<pair<int,int>>> pq;
49      pq.push({ 0, source });
50      while (!pq.empty()) {
51          int u = pq.top().second;
52          int d = pq.top().first;
53          pq.pop();
54          if (d > dist[u]) {
55              continue;
56          }
57       vector<int> adjacentVertices=findAdjacentVertices(g, u);
58
59       for (int v : adjacentVertices) {
60              if (d + 1 < dist[v]) {
61                  dist[v] = d + 1;
62                  pq.push({ dist[v], v });
63              }
64          }
65      }
66      return dist;
67  }
```

```
68  int main() {
69      setlocale(LC_ALL, "ru");
70      int vertices;
71      cout << "Введите количество вершин графа: ";
72      cin >> vertices;
73      Graph g = createGraph(vertices);
74
75      cout << "Введите матрицу смежности графа (" << vertices
76           << "x" << vertices << "):" << endl;
77      for (int i = 0; i < vertices; i++) {
78          for (int j = 0; j < vertices; j++) {
79              cin >> g.adjMatrix[i][j];
80          }
81      }
82      cout << "Матрица смежности:" << endl;
83      printGraph(g);
84      int sourceVertex;
85      cout << "Введите номер вершины: ";
86      cin >> sourceVertex;
87      cout << "Список смежных вершин для вершины "
88           << sourceVertex << ": ";
89      vector<int> adjacentVertices=findAdjacentVertices(g,sourceVertex);
90      if (adjacentVertices.empty()) {
91          cout << "Нет смежных вершин." << endl;
92      }
93      else {
94          for (int v : adjacentVertices) {
95              cout << v << " ";
96          }
97          cout << endl;
98      }
99      cout << "Список вершин, из которых можно попасть в вершину "
100          << sourceVertex << ": ";
101     vector<int> verticesToReach=findVerticesToReach(g,sourceVertex);
102     if (verticesToReach.empty()) {
103       cout << "Нет вершин, из которых можно попасть в вершину "
104            << sourceVertex << "." << endl;
105     }
106     else {
107         for (int v : verticesToReach) {
108             cout << v << " ";
109         }
110         cout << endl;
111     }
112     cout << "Кратчайшие пути от указанной вершины до всех:" << endl;
113     vector<int> shortestPaths = dijkstra(g, sourceVertex);
114     for (int i = 0; i < vertices; i++) {
115       if (shortestPaths[i] != INF) {
116         cout << "От вершины " << sourceVertex << " до вершины "
117              << i << ": " << shortestPaths[i] << endl;
118       }
```

```
119         else {
120           cout << "От вершины " << sourceVertex << " до вершины "
121                << i << " отсутствует" << endl;
122         }
123     }
124     return 0;
125 }
```

Результаты контрольных прогонов программы, демонстрирующие работу алгоритмов и структур на различных входных данных, представлены на рисунках 2.4–2.6.

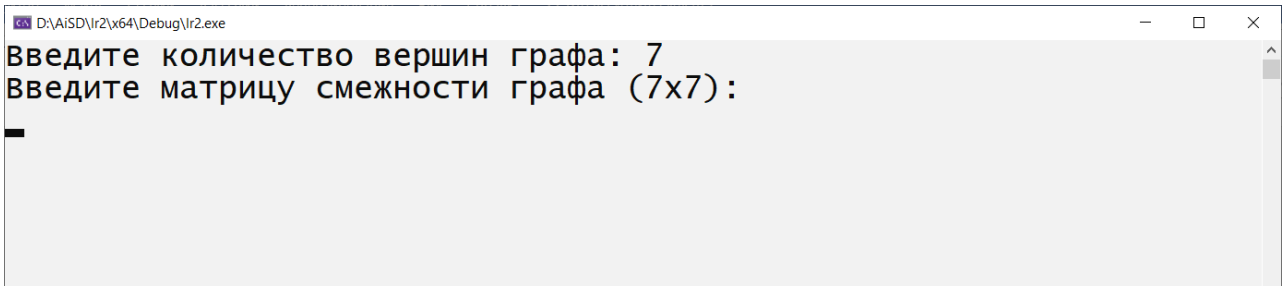


Рисунок 2.4 – Демонстрация процедуры ввода количества вершин

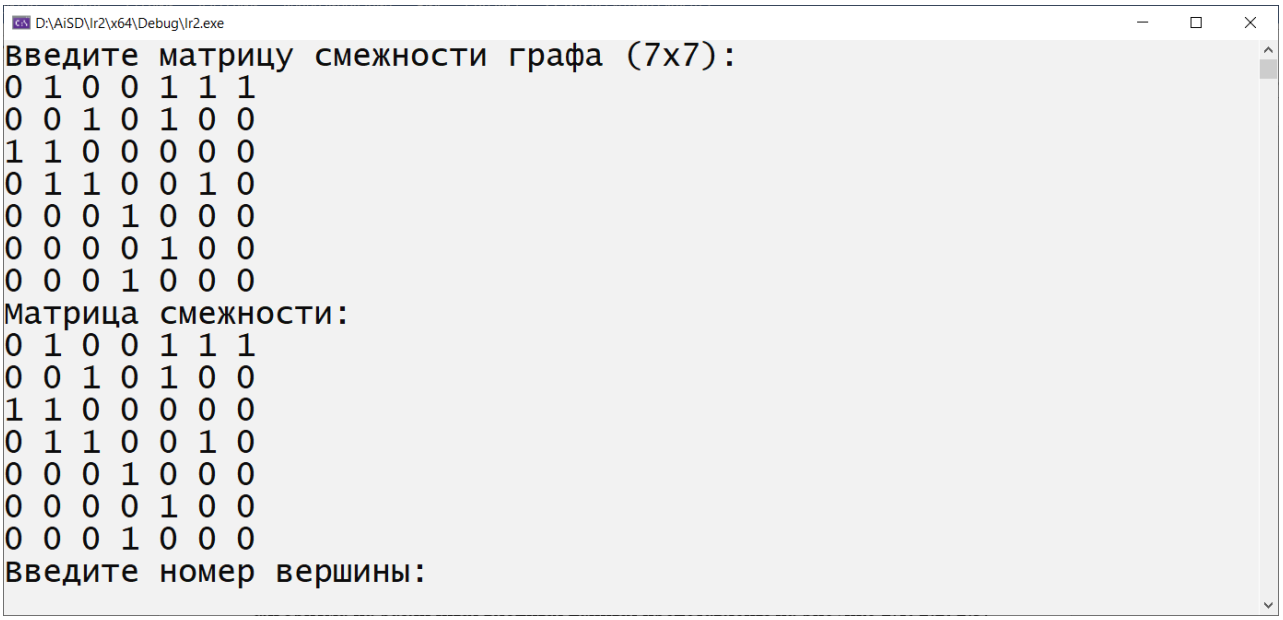


Рисунок 2.5 – Демонстрация ввода и вывода матрицы смежности

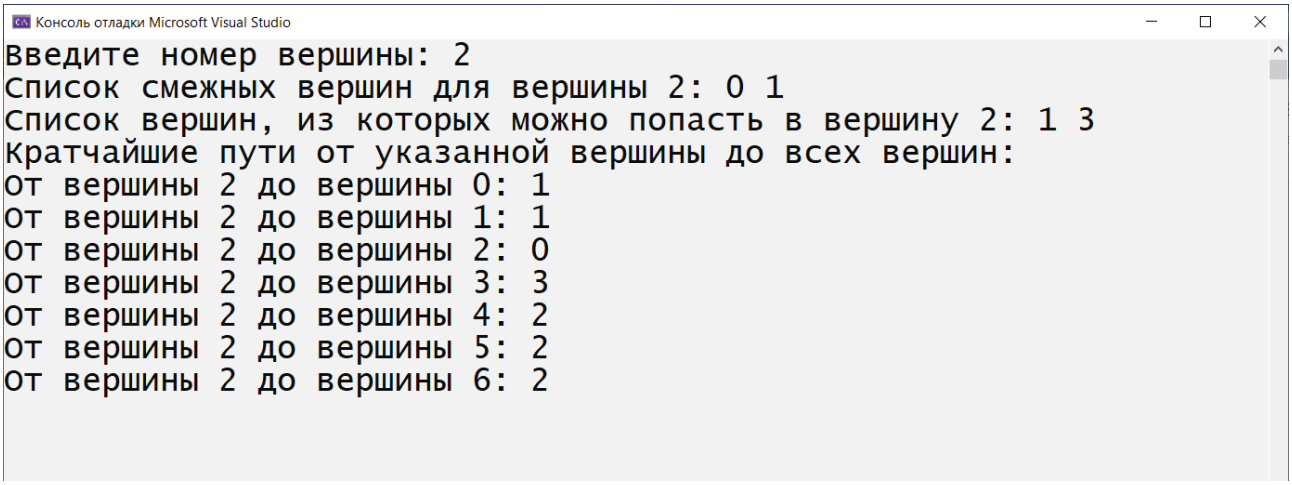


Рисунок 2.6 – Результат выполнения алгоритма Дейкстры

2.4 Контрольные вопросы

1 Что такое ориентированный граф? В чем его отличие от неориентированного графа?

2 Из каких элементов состоит ориентированный граф?

3 Что такое путь в ориентированном графе?

4 Какие есть основные способы представления ориентированных графов?

5 Как работает алгоритм поиска в глубину (DFS) на ориентированном графе? Каковы его преимущества и недостатки?

6 Как работает алгоритм поиска в ширину (BFS) на ориентированном графе? В чем его преимущества и недостатки?

7 Как работает алгоритм Дейкстры для поиска кратчайшего пути на взвешенном ориентированном графе? Каковы его ограничения?

8 Как работает алгоритм Беллмана – Форда? В чем его отличия от алгоритма Дейкстры?

9 Как работает алгоритм Флойда – Уоршелла?

10 Как определить наличие циклов в ориентированном графе? Какие алгоритмы используются для этого?

11 Как найти эйлеров путь или цикл в ориентированном графе? Каковы условия их существования?

12 Что такое топологическая сортировка неориентированного графа?

13 Что такое сильно связные компоненты (SCC) в ориентированном графе? Как их найти?

14 Для решения какой задачи на ориентированном графе удобно использовать алгоритм Дейкстры?

15 Для решения какой задачи на ориентированном графе удобно использовать алгоритм Флойда – Уоршелла?

3 ЛАБОРАТОРНАЯ РАБОТА № 3. ХЕШИРОВАНИЕ ДАННЫХ

3.1 Краткие теоретические сведения

Хеширование – это преобразование массива информации любого типа данных произвольного размера в уникальную комбинацию (набор) символов фиксированного размера, которая соответствует этому массиву входящей информации. Хеширование осуществляет хеш-функция.

Хеш-функция – это какая-либо функция, преобразующая массив исходных данных произвольного размера в битовую строку установленного размера по определенному алгоритму.

Исходные данные для хеширования называются *сообщением* или *ключом*, а результат – *хешем* или *сверткой*.

Хеш по своей сути представляет «отпечаток пальца» исходного сообщения, который позволяет идентифицировать его, но не позволяет восстановить его данные.

Принцип хеширования проиллюстрирован на рисунке 3.1.

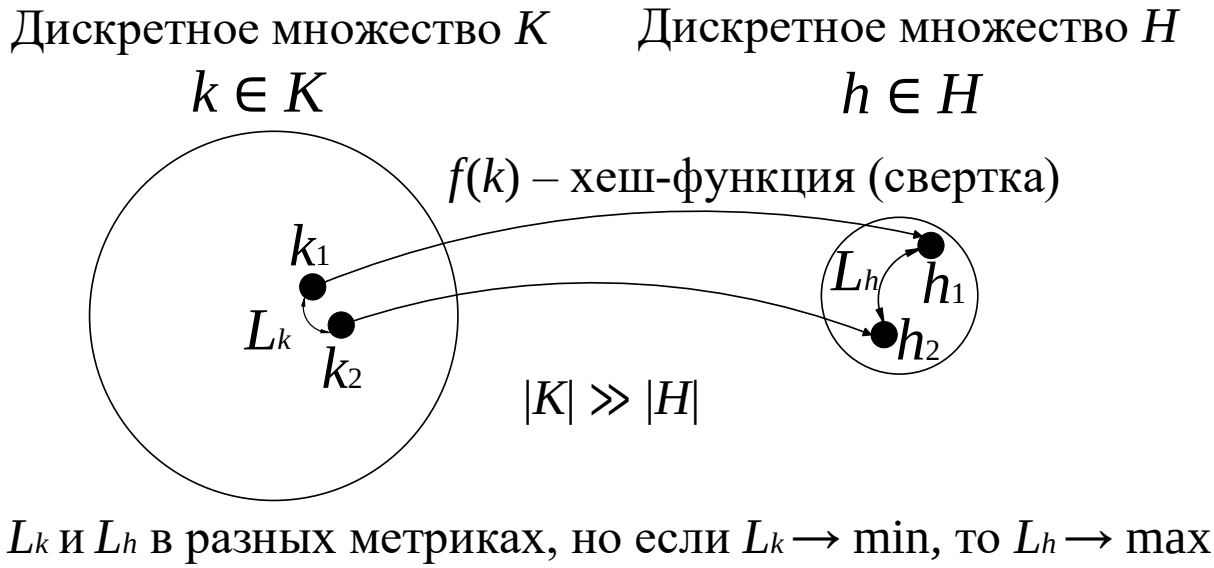


Рисунок 3.1 – Принцип хеширования данных

Как видно из рисунка 3.1, мощность множества исходных данных K много больше мощности множества хешей H, именно поэтому хеш-функция также называется сверткой. При этом хеш-функция $f(k)$ должна обеспечивать «перемешивание» таким образом, чтобы хеши исходных данных, максимально близких друг к другу (минимально отличающихся), отличались максимально значительно.

Однако, поскольку $|K| \gg |H|$, то разные по значению исходные данные могут иметь одинаковый хеш. Такая ситуация называется *коллизией*.

Коллизия – это случай, когда хеш-функция преобразует (свертывает) более одного ключа в один и тот же хеш.

Хеширование в том числе используется для обеспечения быстрого доступа к информации, хранящейся во вторичной памяти.

Структурой данных, основанной на хешировании, является хеш-таблица.

Хеш-таблица – это структура данных, реализующая интерфейс ассоциативного массива («ключ – значение») и позволяющая выполнять операции вставки и удаления пары «ключ – значение», поиска по ключу.

Ключи, выдающие одинаковые адреса в хеш-таблице, называются *ключами-синонимами*.

Коэффициент заполнения хеш-таблицы – это количество хранимых элементов массива, деленное на число возможных значений хеш-функции.

При оценке качества хеш-функции учитываются такие критерии, как:

– вычислительная сложность хеш-функции;

– равномерность распределения ключей в хеш-таблице;

– отсутствие отображения связи между значениями ключей на связь между значениями адресов хеш-таблицы;

– вероятность возникновения коллизий.

Практическое применение получили хеш-функции прямого доступа, остатков от деления, середины квадрата, свертки.

Пример хеш-таблицы, реализующей телефонный справочник, представлен на рисунке 3.2.

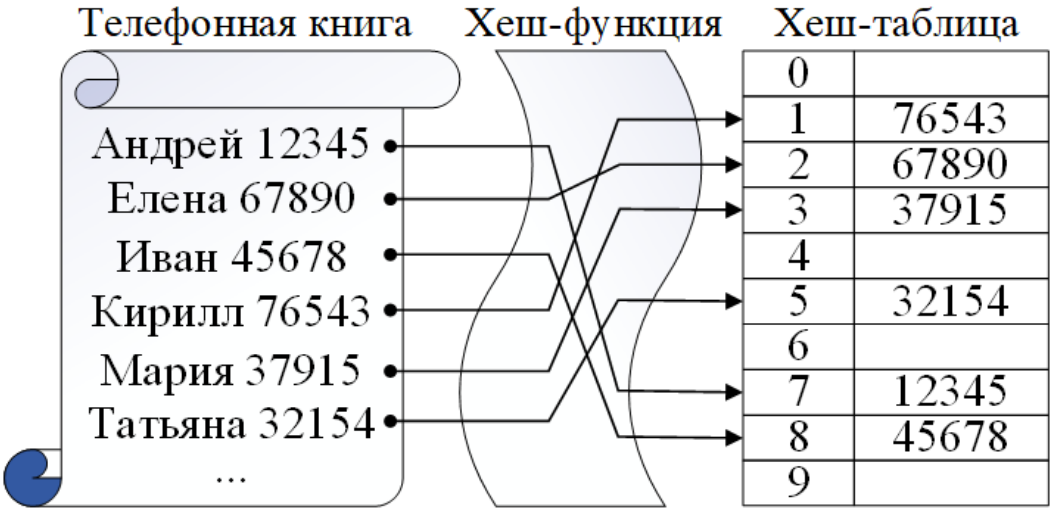


Рисунок 3.2 – Пример хеш-таблицы

По способу разрешения коллизий различают следующие методы:

1 Открытое хеширование (метод цепочек) – в хеш-таблице хранятся не сами элементы, а заголовки списков элементов. Поэтому в каждой записи (сегменте) может храниться несколько элементов в виде связанного списка.

При открытом хешировании (метод цепочек) хеш-таблица представляет собой массив связанных списков. Каждый такой список называется блоком и содержит записи, отображаемые хеш-функцией в один и тот же табличный адрес. Для разрешения коллизий во все записи вводятся указатели, используемые для организации списков – цепочек переполнения.

В случае возникновения коллизии при заполнении хеш-таблицы в список для требуемого адреса хеш-таблицы добавляется еще один элемент (рисунок 3.3).

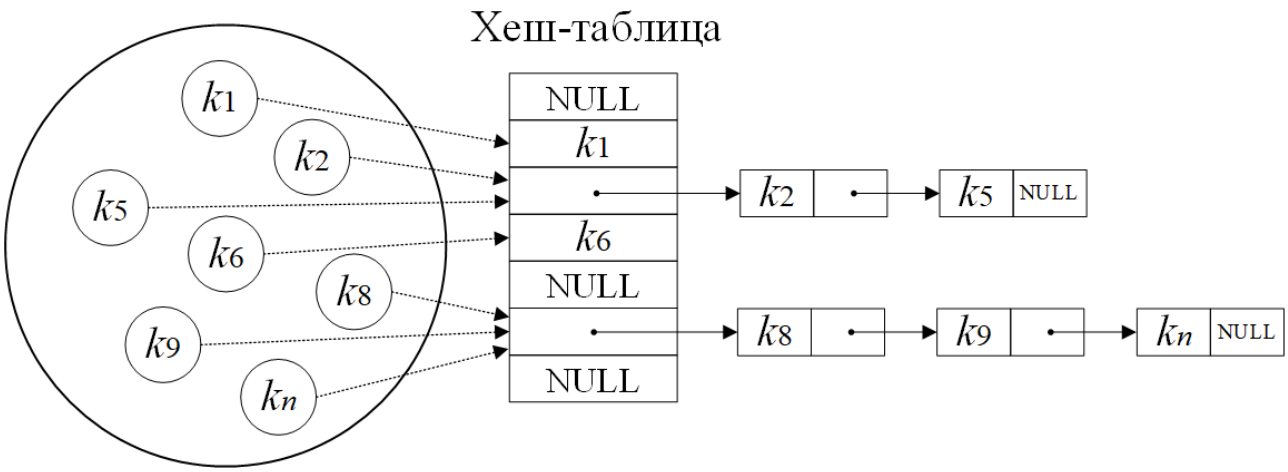


Рисунок 3.3 – Пример разрешения коллизии при открытом хешировании (метод цепочек)

2 Закрытое хеширование (метод открытой адресации) – в хеш-таблице хранятся непосредственно сами элементы, а не заголовки списков элементов, а в каждой записи (сегменте) может храниться только один элемент.

Разрешение коллизий при закрытом хешировании (методом открытой адресации) состоит в том, чтобы, пользуясь каким-либо алгоритмом, обеспечивающим перебор элементов таблицы, просматривать их в поисках свободного места для новой записи (повторное хеширование).

Повторное хеширование – это поиск местоположения для очередного элемента таблицы с учетом шага перемещения.

Если осуществляется попытка поместить элемент X в сегмент с номером $h(x)$, который уже занят другим элементом, то выбирается последовательность других номеров сегментов $h_1(x)$, $h_2(x)$, ..., куда можно поместить элемент X. Каждое из этих местоположений последовательно проверяется, пока не будет найдено свободное (рисунок 3.4). Если свободных сегментов нет, то элемент X добавить нельзя, т. к. таблица заполнена и необходимо выполнить перехеширование.

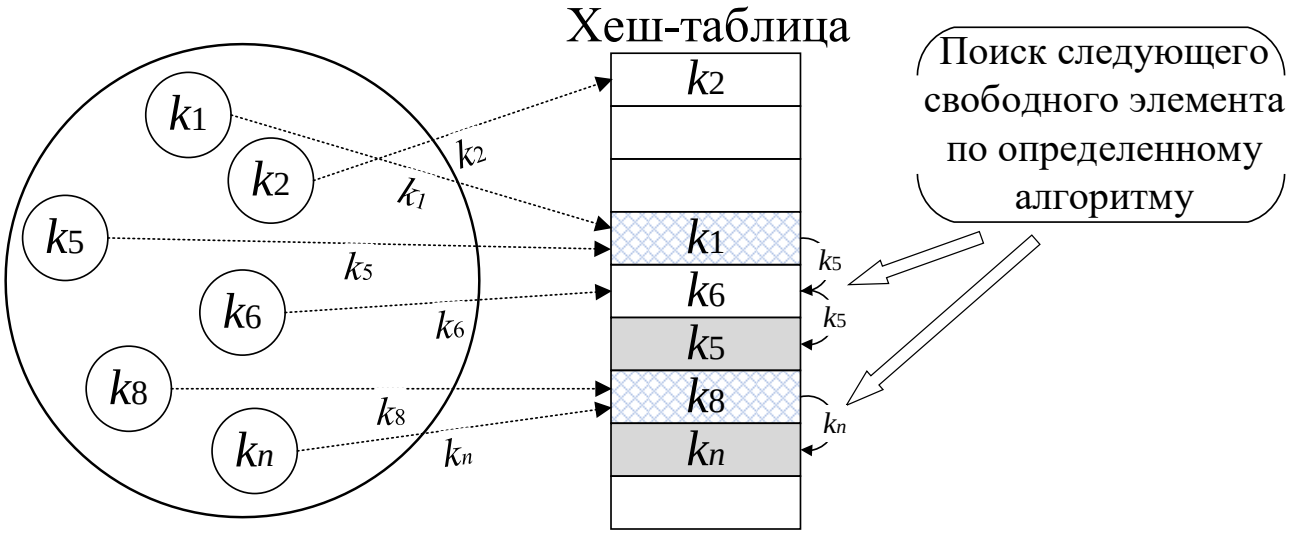


Рисунок 3.4 – Пример разрешения коллизии при закрытом хешировании

На рисунке 3.4 штриховой заливкой обозначены элементы хеш-таблицы, в которых возникают коллизии, а сплошной заливкой – найденные свободные элементы, в которые будут записаны ключи-синонимы.

Используют следующие алгоритмы повторного хеширования (перебора элементов):

а) линейное опробование;

б) квадратичное опробование;

в) двойное хеширование.

Линейное опробование сводится к последовательному перебору элементов таблицы с некоторым фиксированным шагом c:

$$\text{Искомый адрес в хеш-таблице} = h(key) + c \cdot i,$$

где c – константа, определяющая шаг перебора;

i – номер попытки разрешить коллизию.

При шаге, равном единице, происходит последовательный перебор всех элементов после текущего.

Очевидным недостатком линейного опробования является образование кластеров (несколько занятых подряд элементов), что увеличивает время доступа к данным в хеш-таблице.

Квадратичное опробование характеризуется нелинейной зависимостью шага перебора элементов от номера попытки найти свободный элемент:

$$\text{Искомый адрес в хеш-таблице} = h(key) + c \cdot i + d \cdot i^2,$$

где c, d – константы, определяющие шаг перебора;

i – номер попытки разрешить коллизию.

Достоинством квадратичного опробования является сокращение числа попыток (проб) найти свободный элемент при большом числе ключей-синонимов.

Недостатком же является достаточно быстрый выход за адресное пространство небольших хеш-таблиц даже при относительно небольшом числе проб из-за квадратичной зависимости адреса от номера попытки.

Двойное хеширование реализует нелинейную адресацию посредствам использования (суммирования) основной и дополнительной хеш-функций:

$$\text{Искомый адрес в хеш-таблице} = h_1(key) + i \cdot h_2(key),$$

где i – номер попытки разрешить коллизию.

Аналогично другим алгоритмам повторного хеширования при разрешении коллизии очередной адрес может выйти за пределы адресного пространства хеш-таблицы. Для решения этой проблемы можно увеличить размер хеш-таблицы, сделав его больше по сравнению с диапазоном адресов, выдаваемым хеш-функцией.

Достоинством такого решения является сокращение числа коллизий и ускорение доступа к данным хеш-таблицы, а недостатком – нерациональное расходование памяти.

В общем случае открытое хеширование работает быстрее закрытого, поскольку необходимо просматривать только те ключи, которые попадают в один и тот же табличный адрес. Кроме того, закрытое хеширование предполагает наличие таблицы фиксированного размера, а в открытом хешировании элементы хеш-таблицы создаются динамически, при этом длина списка ограничена лишь объемом памяти. Однако открытое хеширование требует дополнительных затрат памяти на поля указателей.

3.2 Задание

1 Согласовать индивидуальный вариант задания с преподавателем (таблица 3.1), внимательно изучить формулировку задачи и пример.

2 Разработать алгоритм и написать программный код для решения следующей задачи.

Ввести массив из *n* целых чисел из заданного диапазона. Создать хеш-таблицу из *M* элементов. Осуществить поиск элемента в хеш-таблице. Вывести на экран исходный массив, хеш-таблицу, количество коллизий и результат поиска.

3 Организовать текстовый пользовательский интерфейс в программе. При выполнении задания реализовать ввод исходных данных пользователем с клавиатуры, а вывод результатов выполнения программы в консоль (на экран). Предусмотреть возможность автоматического заполнения случайными данными в заданном диапазоне.

4 Проверить правильность вычислений на тестовых примерах, выполнив серию контрольных прогонов программы.

5 ***Дополнительно*** реализовать удаление элементов из хеш-таблицы.

Таблица 3.1 − Варианты индивидуальных заданий для лабораторной работы № 3

Вариант	**Задание**
0	*n* = 7. *M* = 10. Диапазон значений: 47000–89000. Схема хеширования: метод цепочек
1	*n* = 12. *M* = 15. Диапазон значений: 23000–45000. Схема хеширования: с открытой адресацией
2	*n* = 8. *M* = 10. Диапазон значений: 53000−78000. Схема хеширования: метод цепочек

Вариант	Задание
3	n = 15. M = 20. Диапазон значений: 12000−34000. Схема хеширования: с открытой адресацией
4	n = 9. M = 10. Диапазон значений: 11000−53000. Схема хеширования: метод цепочек
5	n = 16. M = 20. Диапазон значений: 45000−76000. Схема хеширования: с открытой адресацией
6	n = 12. M = 10. Диапазон значений: 24000−54000. Схема хеширования: метод цепочек
7	n = 8. M = 10. Диапазон значений: 32000−68000. Схема хеширования: с открытой адресацией
8	n = 14. M = 10. Диапазон значений: 26000−77000. Схема хеширования: метод цепочек
9	n = 9. M = 15. Диапазон значений: 38000−58000. Схема хеширования: с открытой адресацией
10	n = 11. M = 10. Диапазон значений: 24000−79000. Схема хеширования: метод цепочек
11	n = 12. M = 15. Диапазон значений: 27000−58000. Схема хеширования: с открытой адресацией
12	n = 7. M = 10. Диапазон значений: 47000−89000. Схема хеширования: метод цепочек
13	n = 11. M = 15. Диапазон значений: 44000−73000. Схема хеширования: с открытой адресацией
14	n = 9. M = 10. Диапазон значений: 39000−76000. Схема хеширования: метод цепочек
15	n = 12. M = 15. Диапазон значений: 23000−58000. Схема хеширования: с открытой адресацией
16	n = 14. M = 10. Диапазон значений: 51000−68000. Схема хеширования: метод цепочек

Вариант	Задание
17	n = 10. M = 14. Диапазон значений: 28000–55000. Схема хеширования: с открытой адресацией
18	n = 12. M = 10. Диапазон значений: 63000−98000. Схема хеширования: метод цепочек
19	n = 15. M = 20. Диапазон значений: 27000−34000. Схема хеширования: с открытой адресацией
20	n = 8. M = 10. Диапазон значений: 12000−54000. Схема хеширования: метод цепочек
21	n = 14. M = 20. Диапазон значений: 49000−66000. Схема хеширования: с открытой адресацией
22	n = 12. M = 10. Диапазон значений: 24000−54000. Схема хеширования: метод цепочек
23	n = 8. M = 10. Диапазон значений: 36000−58000. Схема хеширования: с открытой адресацией
24	n = 12. M = 10. Диапазон значений: 36000−77000. Схема хеширования: метод цепочек
25	n = 12. M = 15. Диапазон значений: 33000−58000. Схема хеширования: с открытой адресацией
26	n = 11. M = 10. Диапазон значений: 28000−79000. Схема хеширования: метод цепочек
27	n = 9. M = 15. Диапазон значений: 26000−58000. Схема хеширования: с открытой адресацией
28	n = 8. M = 10. Диапазон значений: 27000−89000. Схема хеширования: метод цепочек
29	n = 11. M = 15. Диапазон значений: 44000−73000. Схема хеширования: с открытой адресацией
30	n = 9. M = 10. Диапазон значений: 39000−76000. Схема хеширования: метод цепочек

Вариант	Задание
31	n = 10. M = 15. Диапазон значений: 25000−68000. Схема хеширования: с открытой адресацией

3.3 Пример выполнения задания

Для выполнения задания (вариант 0) сначала определим структуру хеш-таблицы. В данной реализации используется метод цепочек, при котором каждый индекс таблицы содержит список значений, соответствующих одной хеш-ячейке. Это позволяет эффективно разрешать коллизии.

Функция `hashFunction` выполняет хеширование ключа по модулю размера таблицы, что позволяет равномерно распределять элементы по ячейкам.

Класс `HashTable` представляет собой хеш-таблицу с методами:

– `insert` – вставляет элемент, вычисляя его хеш и добавляя в соответствующую цепочку;

– `search` – выполняет поиск элемента в таблице, перебирая элементы в цепочке по соответствующему индексу;

– `display` – выводит текущее состояние таблицы, отображая содержимое каждой ячейки.

Функция `validateInt` реализует валидацию пользовательского ввода, проверяя, является ли введенное значение целым числом.

Функция `randInt` используется для генерации случайного числа в заданном диапазоне.

Программа начинает работу с инициализации хеш-таблицы размером M = 10 и массива случайных значений размером n = 7. Далее массив заполняется случайными числами в диапазоне [47000, 89000], которые затем добавляются в хеш-таблицу.

После заполнения таблицы программа отображает ее содержимое и запрашивает у пользователя число для поиска. Выполняется поиск введенного значения в таблице, и программа сообщает, найден ли элемент. В завершение программа ожидает нажатия клавиши перед выходом.

Программная реализация алгоритма решения задания для варианта 0 представлена в листинге 3.1.

Листинг 3.1 – Программная реализация индивидуального задания по лабораторной работе № 3 на языке программирования C++

```
#include <iostream>
#include <vector>
#include <list>
#include <string>
using namespace std;
int hashFunction(int key, int tableSize) {
    return key % tableSize;
}
```

```
9   class HashTable {
10      vector<list<int>> table;
11      int tableSize;
12  public:
13      HashTable(int size) : tableSize(size) {
14          table.resize(size);
15      }
16
17      void insert(int key) {
18          int index = hashFunction(key, tableSize);
19          table[index].push_back(key);
20      }
21      bool search(int key) {
22          int index = hashFunction(key, tableSize);
23          for (int element : table[index]) {
24              if (element == key) {
25                  return true;
26              }
27          }
28          return false;
29      }
30      void display() {
31          for (int i = 0; i < tableSize; i++) {
32              cout << " " << i << ": ";
33              for (int element : table[i]) {
34                  cout << element << " ";
35              }
36              cout << endl;
37          }
38      }
39  };
40
41  void validateInt(int& a, string varName) {
42      string sA = "";
43      do {
44          try {
45              if (varName != "")
46                  cout << "Введите " << varName << ": ";
47              cin >> sA;
48              a = stoi(sA);
49              break;
50          }
51          catch (invalid_argument ex) {
52              cout << "Введенные данные не число" << endl;
53          }
54          catch (out_of_range ex) {
55              cout << "Введено слишком большое число" << endl;
56          }
57      } while (true);
58  }
59  int randInt(int min, int max){return rand()%(max-min+1)+min;}
```

```
int main() {
    system("chcp 1251");
    system("cls");

    int n = 7, M = 10;

    vector<int> arr(n);
    for (int i = 0; i < n; i++) {
        arr[i] = randInt(47000, 89000);
    }
    cout << endl << "Исходный массив: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << endl << endl;

    HashTable hashTable(M);

    for (int i = 0; i < n; i++) {
        hashTable.insert(arr[i]);
    }
    cout << "Хеш-таблица" << endl;
    hashTable.display();
    cout << endl;
    int key;
    cout << "Введите элемент для поиска: ";
    cin >> key;
    if (hashTable.search(key)) {
    cout << "Элемент " << key << " найден в хеш-таблице." << endl;
    }
    else {
    cout << "Элемент " << key << " не найден в хеш-таблице." << endl;
    }
    system("pause");
    return 0;
}
```

Результаты, демонстрирующие работу алгоритмов и структур на различных входных данных, представлены на рисунках 3.5 и 3.6.

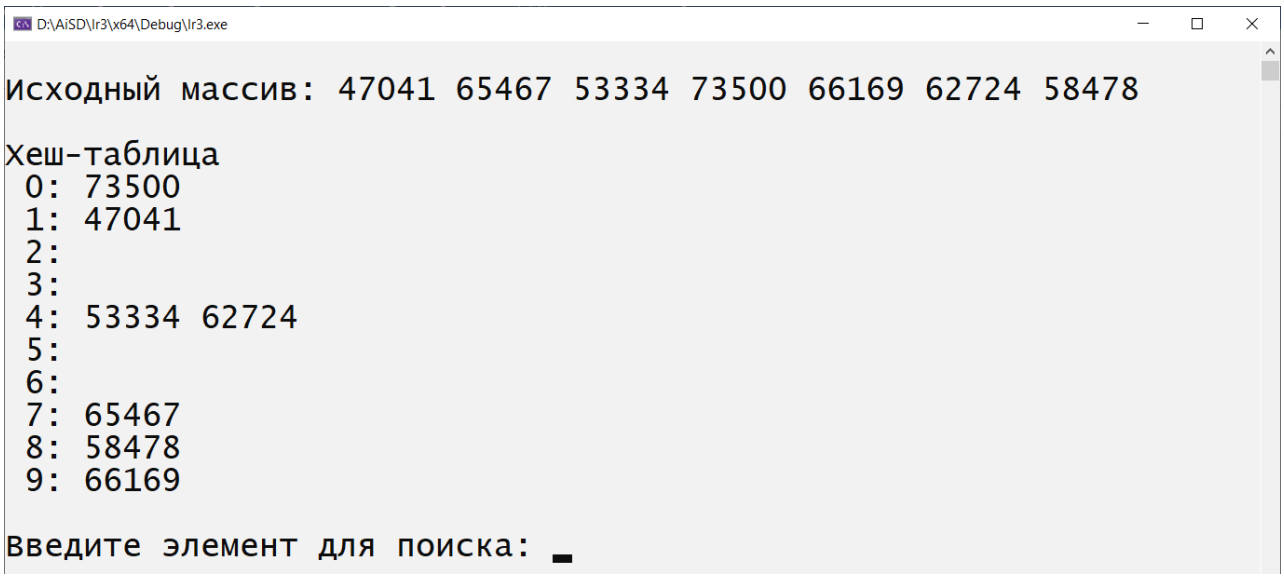


Рисунок 3.5 – Автоматическое заполнение и построение хеш-таблицы

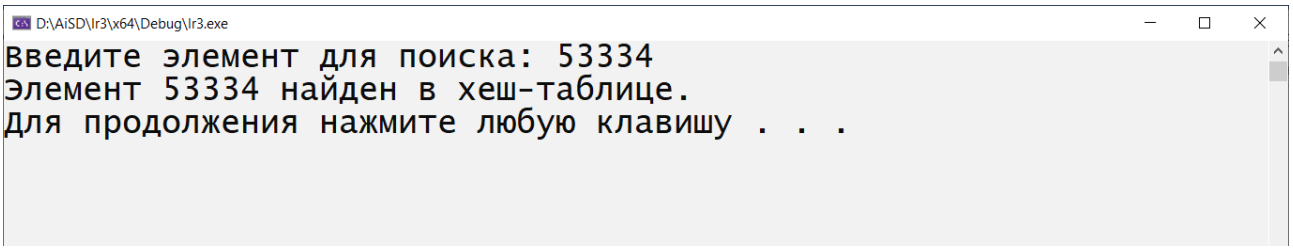


Рисунок 3.6 – Поиск элемента в хеш-таблице

3.4 Контрольные вопросы

1 Что такое хеширование данных?

2 Какие свойства должна иметь хорошая хеш-функция?

3 Что такое коллизия в контексте хеширования данных?

4 Что такое закрытое хеширование (хеширование с открытой адресацией)?

5 В чем отличия между открытым и закрытым хешированием данных?

6 Что такое хеш-таблица?

7 Какие проблемы могут возникнуть при использовании закрытого хеширования? Объясните суть коллизий.

8 Какие методы разрешения коллизий используются в закрытом хешировании?

9 Как выполняется вставка элемента в хеш-таблицу при использовании закрытого хеширования?

10 Как выполняется поиск элемента в хеш-таблице при использовании закрытого хеширования?

11 Что такое коэффициент заполнения хеш-таблицы? Как он влияет на производительность?

4 ЛАБОРАТОРНАЯ РАБОТА № 4. ВНЕШНЯЯ ПАМЯТЬ. B-ДЕРЕВЬЯ

4.1 Краткие теоретические сведения

Классификация памяти вычислительной машины и ее некоторые особенности представлены на рисунке 4.1.

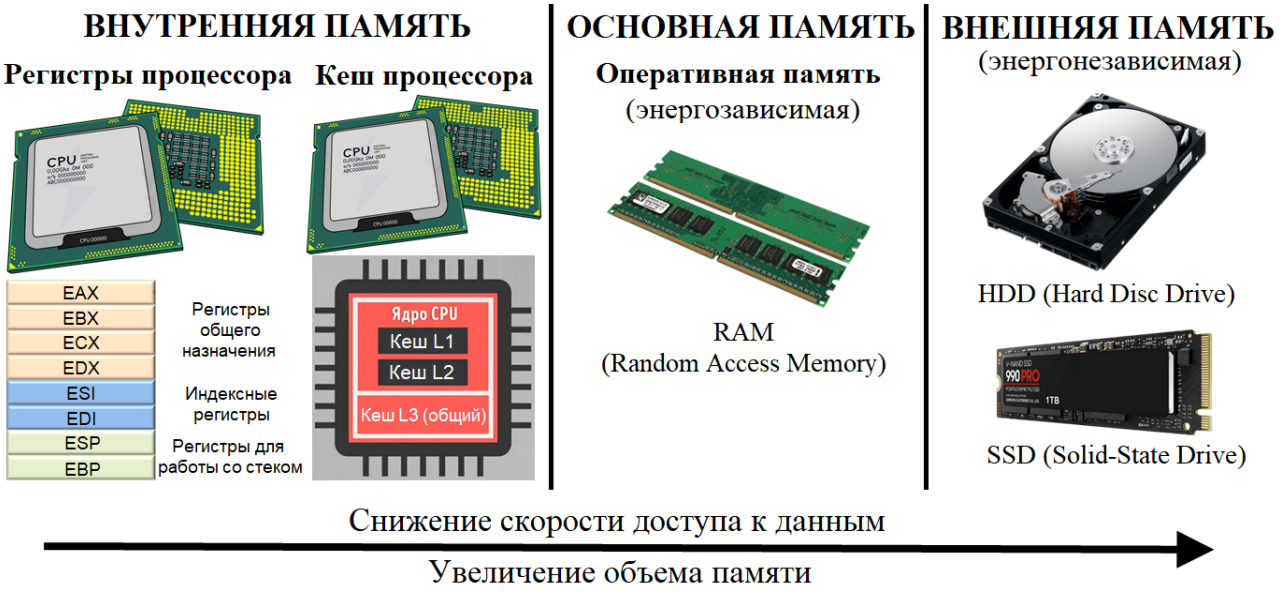


Рисунок 4.1 – Классификация памяти вычислительной машины

Характерные особенности внешней памяти:

– предназначена для длительного хранения файлов (основное назначение), однако может использоваться процессором при вычислениях больших объемов данных, превосходящих по размеру объем основной памяти;

– размер внешней памяти фактически не ограничен и много больше размера основной памяти;

– самый медленный вид памяти по скорости доступа к данным;

Характеристики доступа к устройствам внешней памяти существенно отличаются от характеристик доступа к устройствам основной памяти. Для повышения эффективности использования внешних устройств используется ряд специальных алгоритмов и структур данных.

Структурой данных для работы с внешней памятью является ***файл***.

В языках программирования высокого уровня предусмотрен файловый тип данных, предназначенный для представления данных, хранящихся во внешней памяти.

Работа с файлами определяется файловой системой (частью операционной системы), накладывающей определенные ограничения, касающиеся способов доступа к файлам. Операционная система делит пространство внешней памяти на блоки одинакового размера (кластеры), зависящего от типа и настроек операционной системы (обычно от 512 до 4096 байт).

Блок – минимальная единица данных, расположенных во внешней памяти, равная размеру кластера. Каждый файл состоит из ряда блоков.

Базовой операцией, выполняемой по отношению к файлам во внешней памяти (как структуре данных), является перенос одного блока файла в буфер, находящийся в основной памяти.

Буфер представляет собой зарезервированную область в основной памяти, размер которой соответствует размеру блока.

Операционная система обеспечивает чтение блоков в том порядке, в котором они появляются в списке блоков, содержащем соответствующий файл, т. е. сначала в буфер читается первый блок файла, затем он заменяется на второй блок, который записывается в тот же буфер, и т. д., пока в буфер не будет считан последний блок файла из внешней памяти.

Каждый файл хранится в виде определенной последовательности блоков, а каждый блок содержит целое количество записей.

Указатель считывания всегда указывает на одну из записей в блоке, которая в данный момент находится в буфере. Когда этот указатель должен переместиться на запись, отсутствующую в буфере, необходимо прочитать следующий блок файла во внешней памяти.

Принцип работы с данными из внешней памяти показан на рисунке 4.2.

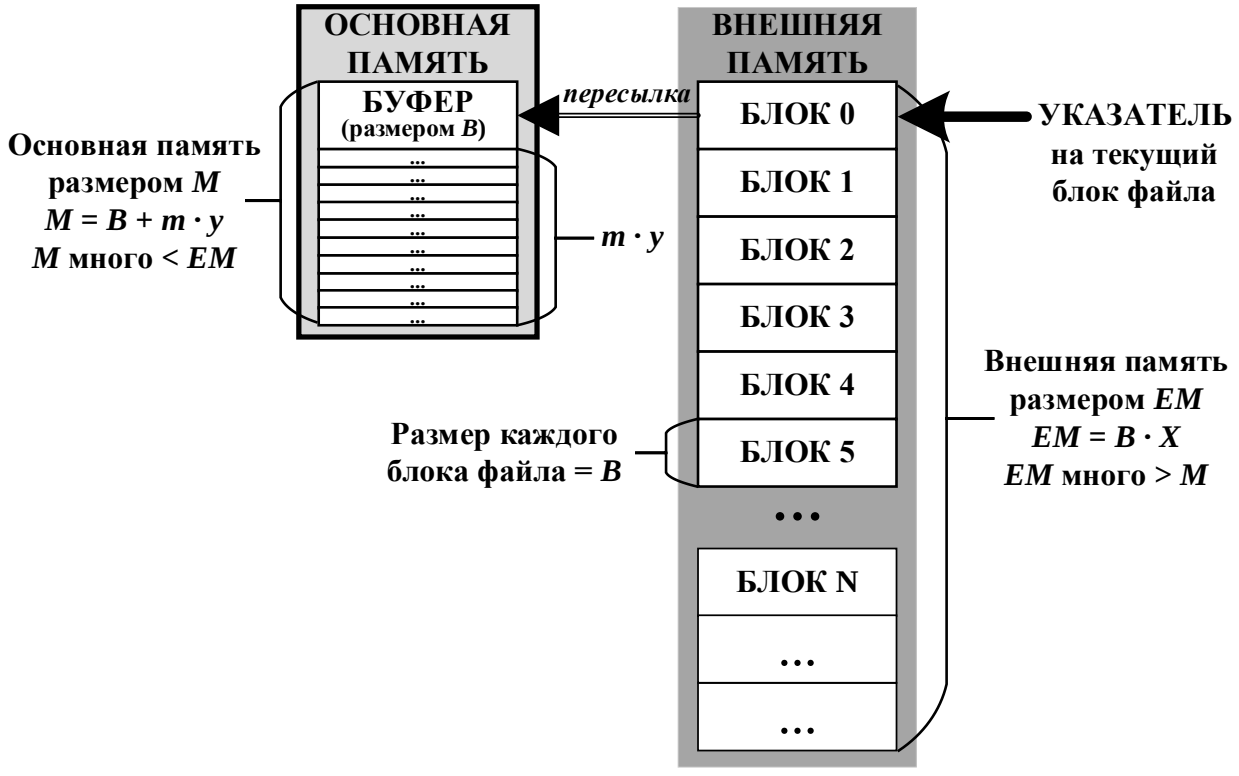


Рисунок 4.2 – Принцип работы с данными во внешней памяти

Физика работы устройств внешней памяти такова, что время, необходимое для поиска блока и чтения его в основную память, достаточно велико даже по сравнению со временем, которое требуется для относительно простой обработки данных, содержащихся в этом блоке.

Таким образом, общее время доступа к данным во внешней памяти составляет сотни миллисекунд и определяется как

$$T_{\text{доступа к данным}} = t_{\text{позиционирования головки}} + t_{\text{ожидания блока}} + t_{\text{чтения/записи}} + t_{\text{передачи данных}}.$$

Для работы с файлами применяются следующие операторы:

– `INSERT` вставляет определенную запись в определенный файл;

– `DELETE` удаляет из определенного файла все записи, содержащие указанные значения в указанных полях;

– `MODIFY` изменяет все записи в определенном файле, задав указанные значения определенным полям в тех записях, которые содержат указанные значения в других полях;

– `RETRIEVE` отыскивает все записи, содержащие указанные значения в указанных полях.

Существуют следующие способы организации файлов:

1) последовательные файлы;

2) хешированные файлы;

3) индексированные файлы.

Последовательные файлы характеризуются тем, что поиск записи с указанными значениями в определенных полях осуществляется путем полного просмотра файла и проверки каждой его записи на наличие в ней заданных значений, т. е. выполнение каждой операции требует прочтения всего файла (через буфер), а затем еще, возможно, выполнения перезаписи некоторых блоков.

Хешированные и индексированные файлы предусматривают наличие у каждой записи ключа, т. е. совокупности полей, которая уникальным образом идентифицирует каждую запись. По ним можно выполнять поиск соответствующих блоков, позволяя считывать в основную память (буфер) лишь небольшую часть файла, реализуя непосредственный доступ к блокам файла.

Файл можно рассматривать как связанный список блоков. Однако для представления внешних файлов удобнее использовать древовидные структуры данных, при которых блоки, составляющие файл, являются листьями дерева, а каждый внутренний узел содержит указатели на множество блоков файла.

B-дерево, являющееся обобщением бинарных деревьев, удачно подходит для представления внешней памяти. Поэтому оно стало стандартным способом организации индексов в системах баз данных.

Обобщением дерева двоичного поиска является *m*-арное дерево, в котором каждый узел имеет не более *m* сыновей.

B-дерево – это особый вид сбалансированного *m*-арного дерева, который позволяет выполнять операции поиска, вставки и удаления записей из внешнего файла с гарантированной производительностью для самой неблагоприятной ситуации.

Характеристики *B*-дерева:

1) корень либо является листом, либо имеет хотя бы двух сыновей;

2) каждый узел, за исключением корня и листьев, имеет от $m / 2$ до m сыновей;

3) все пути от корня до любого листа имеют одинаковую длину.

Поиск записей. Если требуется найти запись *r* со значением ключа *x*, нужно проследить путь от корня до листа, который содержит *r*, если эта запись вообще существует в файле.

Вставка записей. Если требуется вставить в *B*-дерево запись *r* со значением ключа *x*, нужно сначала воспользоваться процедурой поиска, чтобы найти лист *L*, которому должна принадлежать запись *r*. Если в *L* есть место для новой записи, то она вставляется в требуемом порядке в *L*. В этом случае не требуется внесений каких-либо изменений в предков листа *L*.

Удаление записей. Если требуется удалить запись *r* со значением ключа *x*, нужно сначала найти лист *L*, содержащий запись *r*. Затем, если такая запись существует, она удаляется из *L*. Если *r* является первой записью в *L*, после этого выполняется переход в узел *P* – родителя листа *L*, чтобы установить новое значение первого ключа для *L*. Если *L* является первым сыном узла *P*, то первый ключ *L* не зафиксирован в *P*, а появляется в одном из предков *P*. Таким образом, надо распространить изменение в наименьшем значении ключа *L* в обратном направлении вдоль пути от *L* к корню.

4.2 Задание

1 Согласовать индивидуальный вариант задания с преподавателем (таблица 4.1), внимательно изучить формулировку задачи и пример.

2 Разработать алгоритм и написать программный код для решения следующей задачи.

Организовать ввод от 10 до 15 целых чисел, из которых построить *B*-дерево порядка *n*. Реализовать процедуры поиска, вставки и удаления элементов *B*-дерева. Вывести результаты работы процедур. В соответствии с индивидуальным заданием реализовать пользовательскую функцию обработки *B*-дерева. Организовать вывод результатов работы.

3 Организовать текстовый пользовательский интерфейс в программе. При выполнении задания реализовать ввод исходных данных пользователем с клавиатуры, а вывод результатов выполнения программы в консоль (на экран). Предусмотреть возможность автоматического заполнения случайными данными в заданном диапазоне.

4 Проверить правильность вычислений на тестовых примерах, выполнив серию контрольных прогонов программы.

5 ***Дополнительно*** реализовать визуальное отображение построенного *B*-дерева в консоли.

Таблица 4.1 − Варианты индивидуальных заданий для лабораторной работы № 4

Вариант	Задание
0	Реализуйте функции вставки, удаления и поиска элементов. Выведите элементы *B*-дерева по возрастанию
1	Реализуйте функцию поиска, которая также выводит путь поиска от корня до нужного ключа
2	Реализуйте операцию удаления всех ключей, значение которых превышает заданный порог
3	Реализуйте функцию вычисления среднего арифметического значений всех листьев дерева
4	Реализуйте функцию, вычисляющую сумму всех ключей второго уровня дерева
5	Подсчитайте количество узлов, имеющих *n* детей
6	Реализуйте функцию, вычисляющую сумму всех ключей последнего уровня дерева
7	Реализуйте функцию нахождения суммы всех значений на нечетных уровнях дерева
8	Реализуйте функцию вывода узлов, находящихся только на четных уровнях
9	Реализуйте функцию вычисления среднего арифметического значений в листьях
10	Реализуйте функцию вычисления разницы между максимальным и минимальным значениями в листьях
11	Реализуйте функцию поиска ключа с последующим выводом его родительского узла
12	Реализуйте функцию нахождения суммы всех значений на четных уровнях дерева
13	Реализуйте функцию, которая выводит номер уровня, на котором сумма значений узлов нечетная
14	Реализуйте функцию подсчета узлов с нечетными значениями
15	Реализуйте функцию подсчета узлов с четными значениями
16	Реализуйте функцию поиска узлов со значениями, которые в сумме дают минимальный и максимальный результаты
17	Реализуйте функцию, которая ищет заданный узел и, если он найден, предлагает заменить его значение на новое, введенное пользователем
18	Реализуйте функцию удаления всех листовых узлов дерева
19	Реализуйте функцию нахождения ближайшего большего элемента для заданного значения
20	Реализуйте функцию определения уровня узла с заданным значением

Вариант	Задание
21	Реализуйте функцию вывода всех листовых узлов дерева
22	Реализуйте функцию нахождения суммы всех значений на четных уровнях дерева
23	Реализуйте функцию подсчета количества узлов с четными значениями
24	Реализуйте функцию вывода самого длинного пути в дереве
25	Реализуйте функцию определения уровня самого глубокого узла
26	Реализуйте функцию нахождения всех узлов на заданном уровне
27	Реализуйте функцию подсчета узлов с четными значениями
28	Реализуйте функцию вычисления среднего арифметического значений узлов второго уровня
29	Реализуйте вывод узла-предшественника для заданного ключа
30	Реализуйте функцию, вычисляющую произведение значений всех узлов дерева на втором уровне
31	Реализуйте функцию, которая выводит номер уровня, на котором сумма значений узлов четная

4.3 Пример выполнения задания

Для выполнения задания (вариант 0) сначала реализуем структуру данных, известную как *B*-дерево, которое представляет собой сбалансированную версию дерева поиска. В *B*-дереве каждый узел может содержать несколько ключей, что позволяет поддерживать сбалансированность дерева и эффективно выполнять операции вставки, поиска и удаления. Каждый узел может содержать от t до $2t - 1$ ключей, где t – порядок дерева.

Определяем два ключевых класса: `TreeNode` (представляющий узел дерева) и `BTree` (само дерево). Класс `TreeNode` описывает узел, который хранит массив ключей, массив указателей на дочерние узлы, количество текущих ключей и флаг, указывающий, является ли узел листом. В этом классе также реализованы методы для добавления и удаления ключей, для разделения узлов, обхода дерева и поиска ключей.

Метод `insertNonFull(int k)` используется для добавления нового ключа в узел, если он не полный. Если узел уже полон, вызывается метод `splitChild(int i, TreeNode* y)`, который разделяет дочерний узел и перераспределяет ключи между родительским узлом и новым дочерним. Для удаления ключей используется метод `deleteKey(int k)`, который в зависимости от того, является ли узел внутренним или листом, выполняет различные действия по удалению ключей и балансировке дерева.

Класс `BTree` реализует методы для добавления нового ключа в дерево `insert(int k)`, поиска ключа `search(int k)`, удаления ключа `deleteKey(int k)` и вывода дерева по уровням `printTreeByLevels()`.

Метод `insert(int k)` добавляет ключ в дерево, создавая новый корень, если дерево пусто, или выполняет разделение узлов, если корень заполнен. Вставка ключа происходит с использованием метода `insertNonFull()`, который находит подходящий узел и добавляет ключ в него.

Метод `printTreeByLevels()` выводит структуру дерева построчно, отображая каждый уровень дерева отдельно. Для этого используется очередь, в которой содержатся узлы на каждом уровне, и они выводятся по очереди. Также предусмотрены функции для автоматического и ручного заполнения дерева ключами: `autoFill(BTree& tree)` добавляет ключи автоматически, а `manualFill(BTree& tree)` позволяет пользователю вводить ключи вручную.

Программная реализация алгоритма решения задания для варианта 0 представлена в листинге 4.1.

Листинг 4.1 – Программная реализация индивидуального задания по лабораторной работе № 4 на языке программирования C++

```
#include <iostream>
#include <queue>
using namespace std;

class TreeNode {
    int* keys;
    int t;
    TreeNode** C;
    int n;
    bool leaf;

public:
    TreeNode(int temp, bool bool_leaf);

    void insertNonFull(int k);
    void splitChild(int i, TreeNode* y);
    void traverse();
    TreeNode* search(int k);
    void deleteKey(int k);
    void removeFromLeaf(int idx);
    void removeFromNonLeaf(int idx);
    int getPred(int idx);
    int getSucc(int idx);
    void fill(int idx);
    void merge(int idx);
    friend class BTree;
};
```

```
28  class BTree {
29      TreeNode* root;
30      int t;
31
32  public:
33      BTree(int temp) {
34          root = NULL;
35          t = temp;
36      }
37
38      void traverse() {
39          if (root != NULL)
40              root->traverse();
41      }
42
43      TreeNode* search(int k) {
44          return (root == NULL) ? NULL : root->search(k);
45      }
46
47      void insert(int k);
48      void deleteKey(int k);
49      void printTreeByLevels();
50  };
51
52  TreeNode::TreeNode(int t1, bool leaf1) {
53      t = t1;
54      leaf = leaf1;
55      keys = new int[2 * t - 1];
56      C = new TreeNode * [2 * t];
57      n = 0;
58  }
59
60  void TreeNode::traverse() {
61      int i;
62      for (i = 0; i < n; i++) {
63          if (leaf == false)
64              C[i]->traverse();
65          cout << " " << keys[i];
66      }
67      if (leaf == false)
68          C[i]->traverse();
69  }
70
71  TreeNode* TreeNode::search(int k) {
72      int i = 0;
73      while (i < n && k > keys[i]) {
74          i++;
75      }
76
77      cout << "Проверяем ключи в узле: ";
```

```
78      for (int j = 0; j < n; j++) {
79          cout << keys[j] << " ";
80      }
81      cout << endl;
82
83      if (i < n && keys[i] == k) {
84          cout << "Ключ " << k << " найден!" << endl;
85          return this;
86      }
87
88      if (leaf) {
89         cout << "Ключ " << k << " не найден в дереве." << endl;
90         return NULL;
91      }
92      return C[i]->search(k);
93  }
94
95  void BTree::insert(int k) {
96      if (root == NULL) {
97          root = new TreeNode(t, true);
98          root->keys[0] = k;
99          root->n = 1;
100     }
101     else {
102         if (root->n == 2 * t - 1) {
103             TreeNode* s = new TreeNode(t, false);
104             s->C[0] = root;
105             s->splitChild(0, root);
106
107             int i = 0;
108             if (s->keys[0] < k)
109                 i++;
110             s->C[i]->insertNonFull(k);
111
112             root = s;
113         }
114         else
115             root->insertNonFull(k);
116     }
117 }
118
119 void TreeNode::insertNonFull(int k) {
120     int i = n - 1;
121     if (leaf == true) {
122         while (i >= 0 && keys[i] > k) {
123             keys[i + 1] = keys[i];
124             i--;
125         }
126         keys[i + 1] = k;
127         n = n + 1;
128     }
```

```
129     else {
130         while (i >= 0 && keys[i] > k)
131             i--;
132         if (C[i + 1]->n == 2 * t - 1) {
133             splitChild(i + 1, C[i + 1]);
134             if (keys[i + 1] < k)
135                 i++;
136         }
137         C[i + 1]->insertNonFull(k);
138     }
139 }
140
141 void TreeNode::splitChild(int i, TreeNode* y) {
142     TreeNode* z = new TreeNode(y->t, y->leaf);
143     z->n = t - 1;
144     for (int j = 0; j < t - 1; j++)
145         z->keys[j] = y->keys[j + t];
146
147     if (y->leaf == false) {
148         for (int j = 0; j < t; j++)
149             z->C[j] = y->C[j + t];
150     }
151     y->n = t - 1;
152     for (int j = n; j >= i + 1; j--)
153         C[j + 1] = C[j];
154     C[i + 1] = z;
155     for (int j = n - 1; j >= i; j--)
156         keys[j + 1] = keys[j];
157
158     keys[i] = y->keys[t - 1];
159     n = n + 1;
160 }
161
162 void BTree::deleteKey(int k) {
163     if (root == NULL) {
164         cout << "Дерево пусто.\n";
165         return;
166     }
167     root->deleteKey(k);
168
169     if (root->n == 0) {
170         TreeNode* tmp = root;
171         if (root->leaf) {
172             root = NULL;
173         }
174         else {
175             root = root->C[0];
176         }
177         delete tmp;
178     }
179 }
```

```
180 void TreeNode::deleteKey(int k) {
181     int idx = 0;
182     while (idx < n && keys[idx] < k) {
183         idx++;
184     }
185 
186     if (idx < n && keys[idx] == k) {
187         if (leaf) {
188             removeFromLeaf(idx);
189         }
190         else {
191             removeFromNonLeaf(idx);
192         }
193     }
194     else {
195         if (leaf) {
196             cout << "Ключ не найден!\n";
197             return;
198         }
199 
200         bool flag = ((idx == n) ? true : false);
201         if (C[idx]->n < t) {
202             fill(idx);
203         }
204 
205         if (flag && idx > n) {
206             C[idx - 1]->deleteKey(k);
207         }
208         else {
209             C[idx]->deleteKey(k);
210         }
211     }
212 }
213 
214 void TreeNode::removeFromLeaf(int idx) {
215     for (int i = idx + 1; i < n; i++) {
216         keys[i - 1] = keys[i];
217     }
218     n--;
219 }
220 
221 void TreeNode::removeFromNonLeaf(int idx) {
222     int k = keys[idx];
223 
224     if (C[idx]->n >= t) {
225         int pred = getPred(idx);
226         keys[idx] = pred;
227         C[idx]->deleteKey(pred);
228     }
229     else if (C[idx + 1]->n >= t) {
230         int succ = getSucc(idx);
```

```
231         keys[idx] = succ;
232         C[idx + 1]->deleteKey(succ);
233     }
234     else {
235         merge(idx);
236         C[idx]->deleteKey(k);
237     }
238 }
239
240 int TreeNode::getPred(int idx) {
241     TreeNode* cur = C[idx];
242     while (!cur->leaf) {
243         cur = cur->C[cur->n];
244     }
245     return cur->keys[cur->n - 1];
246 }
247
248 int TreeNode::getSucc(int idx) {
249     TreeNode* cur = C[idx + 1];
250     while (!cur->leaf) {
251         cur = cur->C[0];
252     }
253     return cur->keys[0];
254 }
255
256 void TreeNode::fill(int idx) {
257     if (idx > 0 && C[idx - 1]->n >= t) {
258         merge(idx - 1);
259     }
260     else if (idx < n && C[idx + 1]->n >= t) {
261         merge(idx);
262     }
263     else {
264         if (idx < n) {
265             merge(idx);
266         }
267         else {
268             merge(idx - 1);
269         }
270     }
271 }
272
273 void TreeNode::merge(int idx) {
274     TreeNode* child = C[idx];
275     TreeNode* sibling = C[idx + 1];
276
277     child->keys[t - 1] = keys[idx];
278
279     for (int i = 0; i < sibling->n; i++) {
280         child->keys[i + t] = sibling->keys[i];
281     }
```

```
282     if (!child->leaf) {
283         for (int i = 0; i <= sibling->n; i++) {
284             child->C[i + t] = sibling->C[i];
285         }
286     }
287     for (int i = idx + 1; i < n; i++) {
288         keys[i - 1] = keys[i];
289     }
290     for (int i = idx + 2; i <= n; i++) {
291         C[i - 1] = C[i];
292     }
293     child->n += sibling->n + 1;
294     n--;
295 
296     delete sibling;
297 }
298 
299 void autoFill(BTree& tree) {
300     int keys[] = { 8, 9, 10, 11, 15, 16, 17, 18, 20, 23 };
301     for (int key : keys) {
302         tree.insert(key);
303     }
304 }
305 
306 void manualFill(BTree& tree) {
307     int n, key;
308     cout << "Сколько ключей вы хотите вставить? ";
309     cin >> n;
310     for (int i = 0; i < n; i++) {
311         cout << "Введите ключ " << i + 1 << ": ";
312         cin >> key;
313         tree.insert(key);
314     }
315 }
316 void BTree::printTreeByLevels() {
317     if (root == NULL) {
318         cout << "Дерево пусто.\n";
319         return;
320     }
321     queue<TreeNode*> nodesQueue;
322     nodesQueue.push(root);
323 
324     int level = 0;
325     while (!nodesQueue.empty()) {
326         int nodeCount = nodesQueue.size();
327         cout << "Уровень " << level << ": ";
328 
329         while (nodeCount > 0) {
330             TreeNode* node = nodesQueue.front();
331             nodesQueue.pop();
332             cout << "[";
```

```
333             for (int i = 0; i < node->n; i++) {
334                 cout << node->keys[i];
335                 if (i < node->n - 1)
336                     cout << " ";
337             }
338             cout << "] ";
339 
340             if (!node->leaf) {
341                 for (int i = 0; i <= node->n; i++) {
342                     nodesQueue.push(node->C[i]);
343                 }
344             }
345             nodeCount--;
346         }
347         cout << endl;
348         level++;
349     }
350 }
351 
352 int main() {
353     setlocale(LC_ALL, "RU");
354 
355     int choice;
356     BTree t(3);
357 
358     cout << "Выберите способ заполнения дерева:\n";
359     cout << "1. Вручную\n";
360     cout << "2. Автоматически\n";
361     cout << "Ваш выбор: ";
362     cin >> choice;
363 
364     if (choice == 1) {
365         manualFill(t);
366     }
367     else if (choice == 2) {
368         autoFill(t);
369     }
370     else {
371         cout << "Неправильный выбор!";
372         return 0;
373     }
374 
375     int option;
376     do {
377         cout << "\nМеню:\n";
378         cout << "1. Вставить новый ключ\n";
379         cout << "2. Вывести дерево по уровням\n";
380         cout << "3. Поиск ключа\n";
381         cout << "4. Удалить ключ\n";
382         cout << "5. Выйти\n";
383         cout << "Ваш выбор: ";
```

```
384         cin >> option;
385
386         switch (option) {
387         case 1: {
388             int key;
389             cout << "Введите ключ для вставки: ";
390             cin >> key;
391             t.insert(key);
392             cout << "Ключ " << key << " вставлен.\n";
393             break;
394         }
395         case 2:
396             cout << "B-дерево (по уровням):" << endl;
397             t.printTreeByLevels();
398             break;
399         case 3: {
400             int key;
401             cout << "Введите ключ для поиска: ";
402             cin >> key;
403             TreeNode* result = t.search(key);
404             if (result == NULL)
405                 cout << "Ключ " << key << " не найден.\n";
406             break;
407         }
408         case 4: {
409             int key;
410             cout << "Введите ключ для удаления: ";
411             cin >> key;
412             t.deleteKey(key);
413             break;
414         }
415         case 5:
416             cout << "Выход из программы.\n";
417             break;
418         default:
419             cout << "Неправильный выбор! Попробуйте снова.\n";
420         }
421     } while (option != 5);
422
423     return 0;
424 }
```

Результаты выполнения программы из листинга 4.1, демонстрирующие работу алгоритмов и структур на различных входных данных, представлены на рисунках 4.3–4.6.

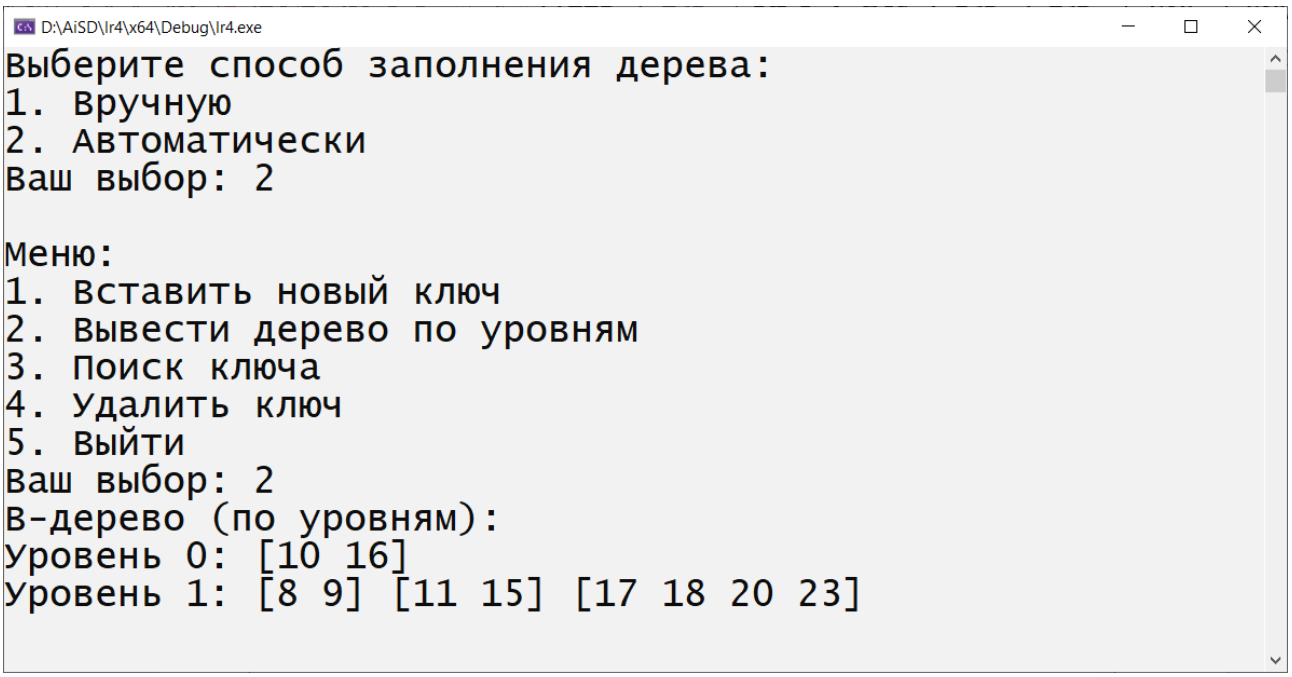


Рисунок 4.3 – Автоматическое заполнение и построение дерева по уровням

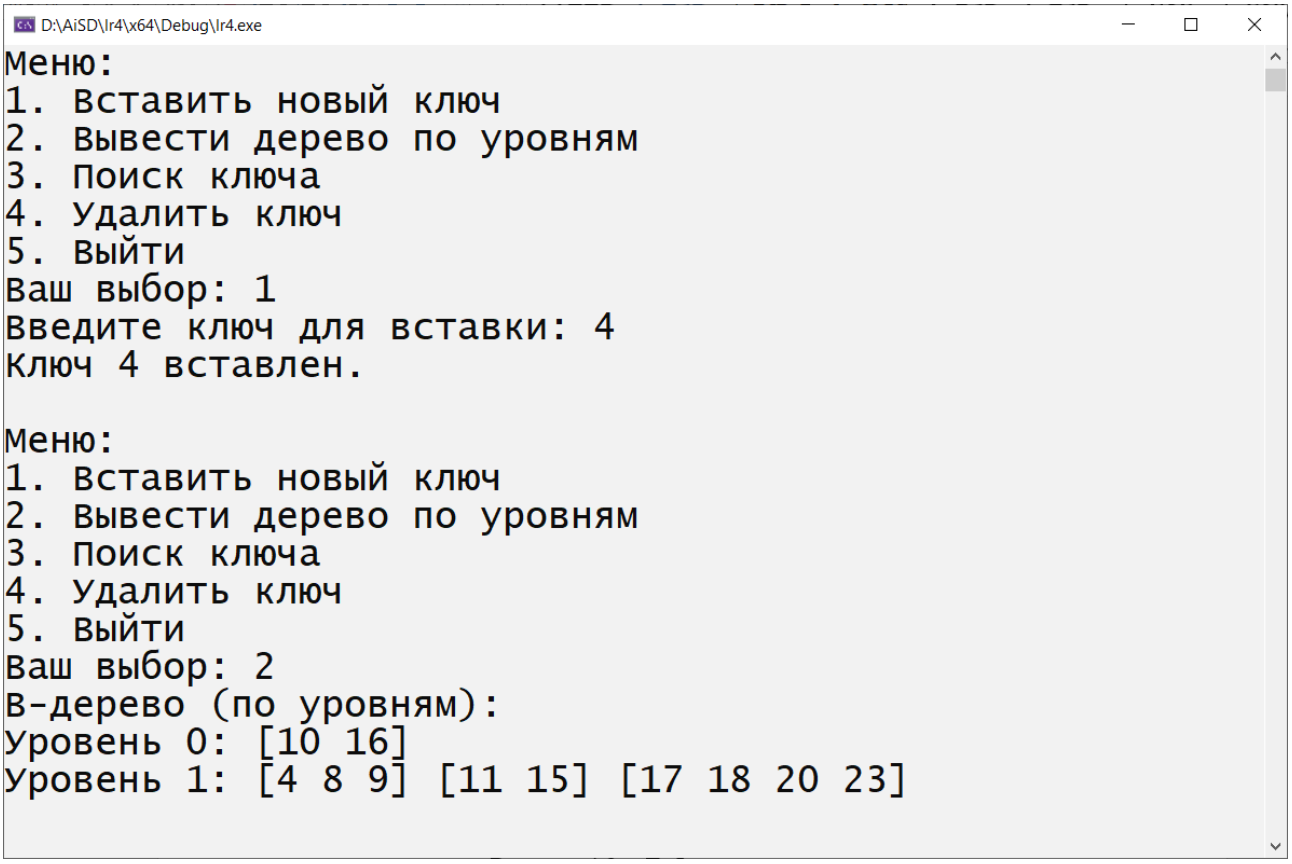


Рисунок 4.4 – Добавление элемента в дерево

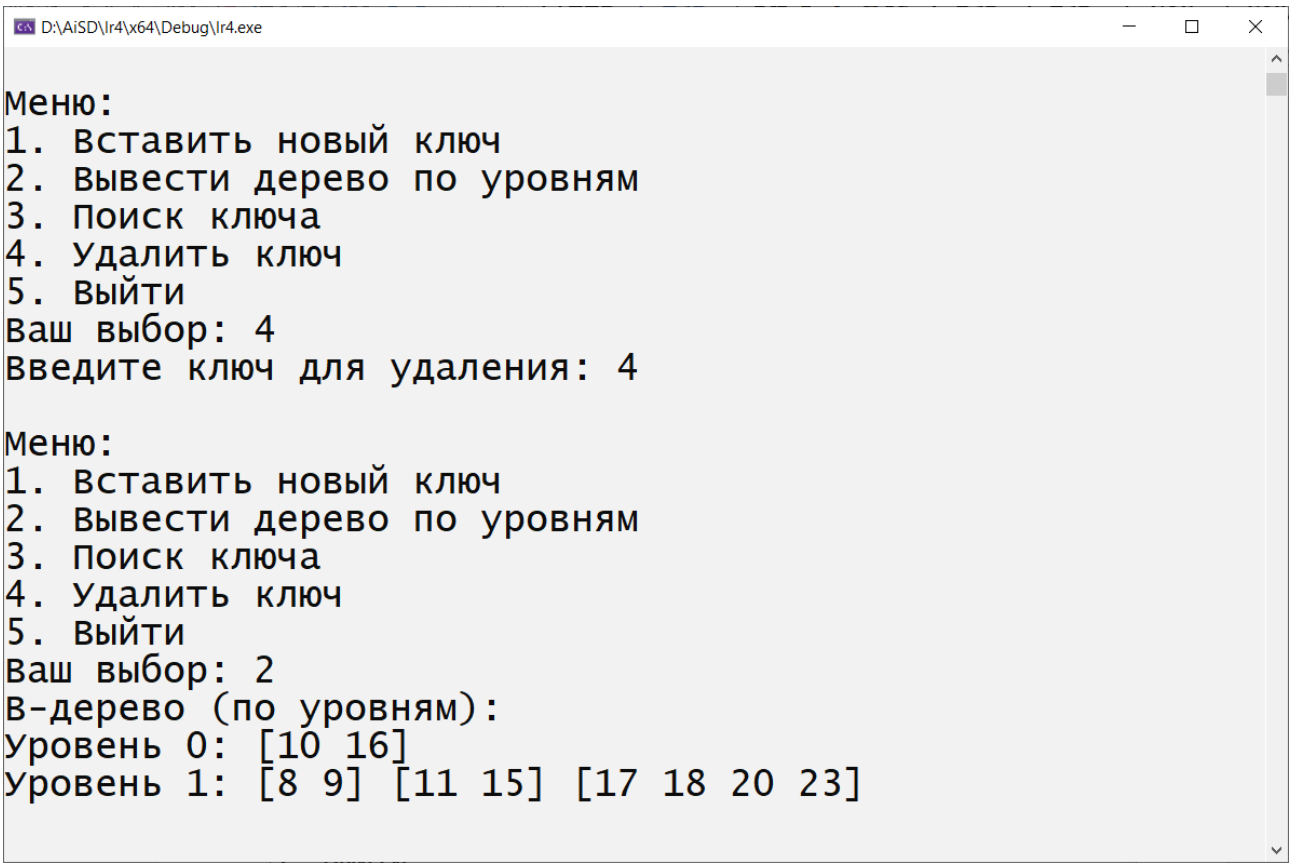


Рисунок 4.5 – Результат работы функции удаления элемента

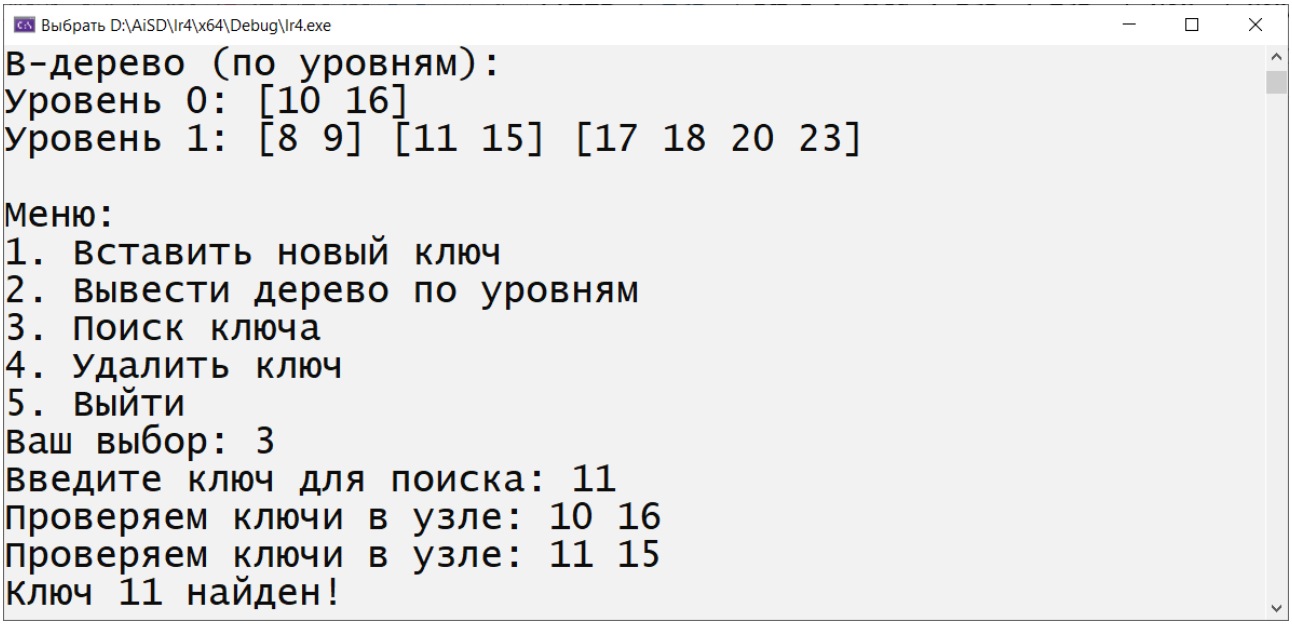


Рисунок 4.6 – Результат работы функции поиска элемента

4.4 Контрольные вопросы

1 Что такое внешняя память? Чем она отличается от оперативной памяти?

2 Почему традиционные структуры данных (например, бинарные деревья) неэффективны для работы с внешней памятью?

3 Что такое *B*-дерево? Какие основные свойства и характеристики оно имеет?

4 Каковы основные преимущества *B*-деревьев перед другими структурами данных при работе с внешней памятью?

5 Чем *B*-дерево отличается от бинарного дерева?

6 На основании чего выбирается арность *B*-дерева?

7 Как происходит вставка нового элемента в *B*-дерево?

8 Что такое разделение узла в *B*-дереве? Когда и как оно происходит?

9 Как выполняется удаление элемента из *B*-дерева? Какие случаи нужно учитывать?

10 Что такое слияние узлов в *B*-дереве? Когда оно применяется?

11 Как выполняется поиск элемента в *B*-дереве? Каковы временные затраты на эту операцию?

12 Как выбор порядка *B*-дерева влияет на его производительность?

13 За счет чего удается ускорить обработку данных при использовании *B*-деревьев?

5 КОНТРОЛЬНАЯ РАБОТА № 1. ПРОШИВКА БИНАРНОГО ДЕРЕВА. ОБХОДЫ ДЕРЕВЬЕВ И ОПЕРАЦИИ НАД НИМИ

5.1 Краткие теоретические сведения

Списки и деревья – это наиболее часто используемые динамические структуры данных. В данной контрольной работе будет показано, как может повыситься эффективность решения поставленной задачи, если применять эти структуры сообща для достижения поставленной цели. Также будет рассмотрена задача нахождения *k*-го элемента списка с использованием дерева. Значение в узле дерева – это количество элементов в левом поддереве данного узла.

Элементы исходного списка представляются листьями дерева, а узлы, не являющиеся листьями, представляются как часть внутренней структуры дерева. С каждым узлом связан счетчик числа листьев в левом поддереве. Элементы списка присваиваются листьям дерева в порядке симметричного просмотра. В каждом узле `p`, не являющемся листом, алгоритм определяет по значениям `r` и `ListCount(p)`, находится ли нужный лист в левом или в правом поддереве. Если лист в левом поддереве, то поиск ведется по этому поддереву без изменения значения `r`, а если в правом, то перед поиском по этому поддереву значение `r` уменьшается на `ListCount(p)`.

Рассмотрим реализацию изложенного способа на примере.

Пусть дан список элементов (рисунок 5.1).

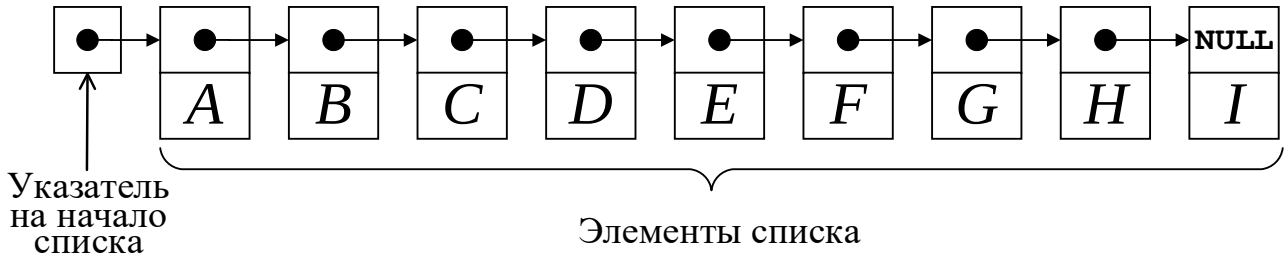


Рисунок 5.1 – Список элементов для поиска

Код реализации алгоритма нахождения *k*-го элемента списка представлен в листинге 5.1.

Листинг 5.1 – Нахождение *k*-го элемента на языке программирования C++

```
1   Node* findKthElement(Node* root, int k) {
2       if (!root) return nullptr;
3
4       int leftCount = root->count;
5       if (k == leftCount + 1) return root;
6       if (k <= leftCount) return findKthElement(root->left, k);
7       return findKthElement(root->right, k - leftCount - 1);
8   }
```

На рисунках 5.2 и 5.3 приведены два дерева, удовлетворяющие требованиям алгоритма.

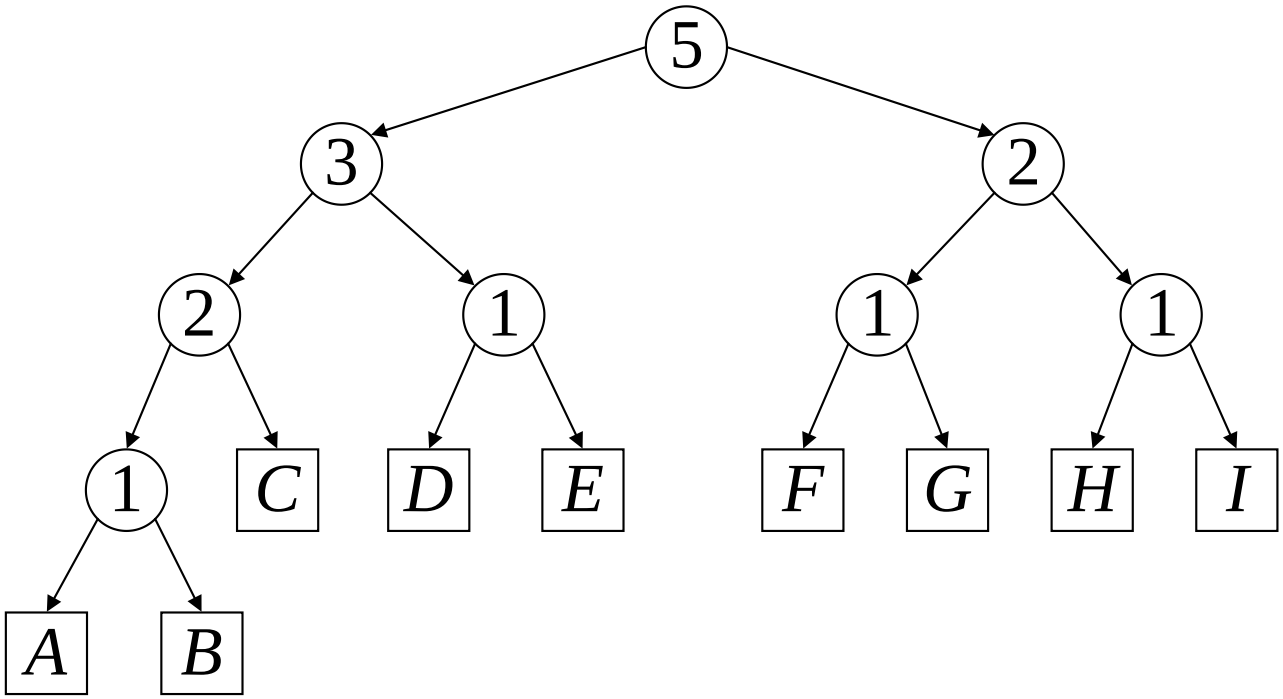


Рисунок 5.2 – Первый вариант дерева с элементами списка

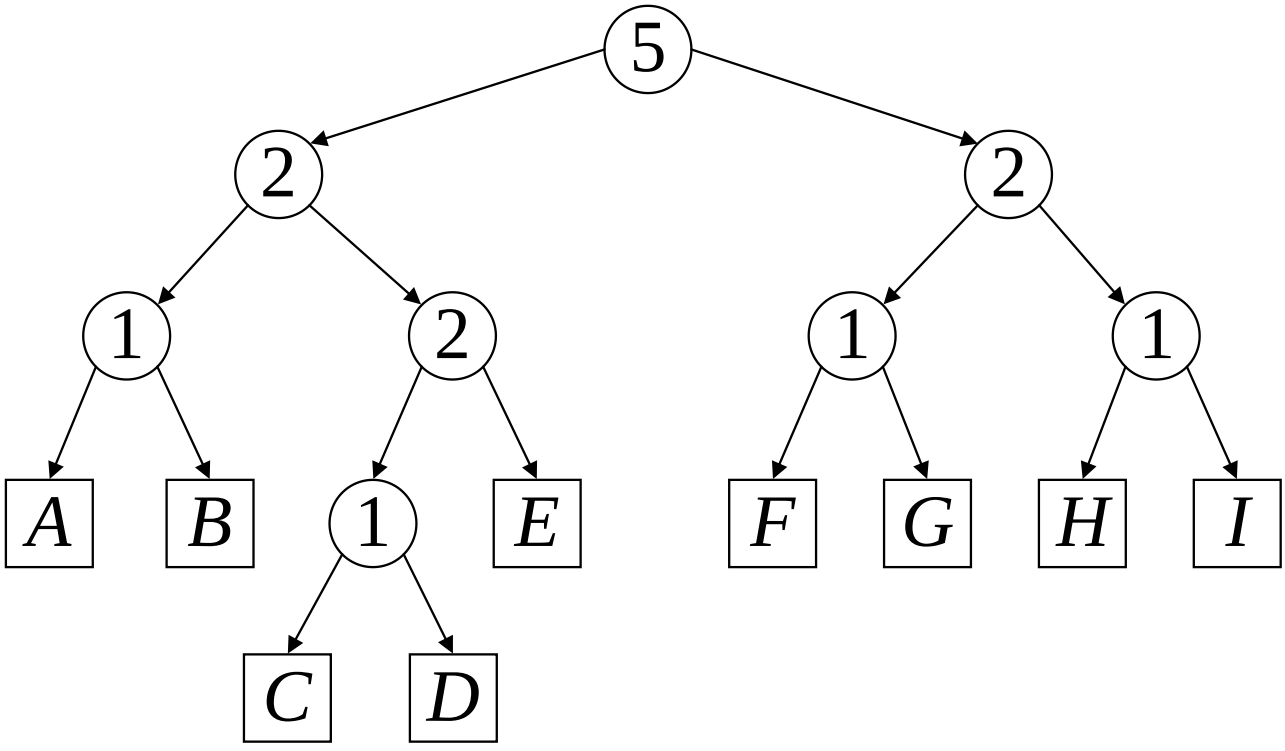


Рисунок 5.3 – Второй вариант дерева с элементами списка

Функция `findKthElement(Node* root, int k)` предназначена для поиска *k*-го наименьшего элемента в бинарном дереве поиска.

Используемые элементы:

– `root` – указатель на корень бинарного дерева;

– `k` – порядковый номер элемента, который нужно найти (1 – минимальный элемент, 2 – второй по величине и т. д.);

– `leftCount` – количество узлов в левом поддереве текущего узла. Оно хранится в поле `count`;

– `findKthElement(root, k)` – указатель на найденный узел, содержащий *k*-й элемент;

– `root->left` и `root->right` – указатели на левого и правого потомков текущего узла.

Если `root == nullptr`, значит, дерево пустое или *k* превышает количество элементов в дереве – возвращаем `nullptr`. Вычисляем `leftCount` – число узлов в левом поддереве текущего узла. Если `k == leftCount + 1`, то текущий узел содержит искомый *k*-й элемент, возвращаем его. Если `k <= leftCount`, продолжаем поиск в левом поддереве. Иначе продолжаем поиск в правом поддереве, но с обновленным значением `k = k - leftCount - 1`, т. к. мы исключаем текущий узел и все элементы левого поддерева.

Прошитые бинарные деревья

В случае представления бинарного дерева в виде узлов, содержащих информационное поле и два поля связи, количество полей связи, имеющих значения `NULL`, всегда больше числа связей, указывающих на реально существующие узлы. Поэтому часто такой способ хранения деревьев оказывается неэффективным с точки зрения использования памяти, особенно, если размер информационного поля сопоставим с размером указателя.

Эффективность прохождения дерева рекурсивными и нерекурсивными алгоритмами может быть увеличена, если использовать пустые указатели на отсутствующие поддеревья для хранения в них адресов узлов преемников, которые надо посетить при заданном порядке прохождения бинарного дерева.

Такой указатель называется *нитью*. Его следует отличать от указателей в дереве, которые используются с левым и правым поддеревьями.

Операция, заменяющая пустые указатели на нити, называется *прошивка*. Она может выполняться по-разному. Если при просмотре в симметричном порядке нити заменяют пустые указатели в узлах с пустыми правыми поддеревьями, то бинарное дерево называется *симметрично прошитым справа.* Похожим образом может быть определено бинарное дерево, *симметрично прошитое слева*: дерево, в котором каждый пустой левый указатель изменен так, что он содержит нить – связь с предшественником данного узла при просмотре в симметричном порядке.

Симметрично прошитое бинарное дерево – это то, которое симметрично прошито слева и справа. Однако левая прошивочная нить не дает тех же преимуществ, что и правая прошивочная нить.

На рисунке 5.4 показано дерево, *симметрично прошитое справа.* На нем пунктирными линиями обозначены прошивочные нити.

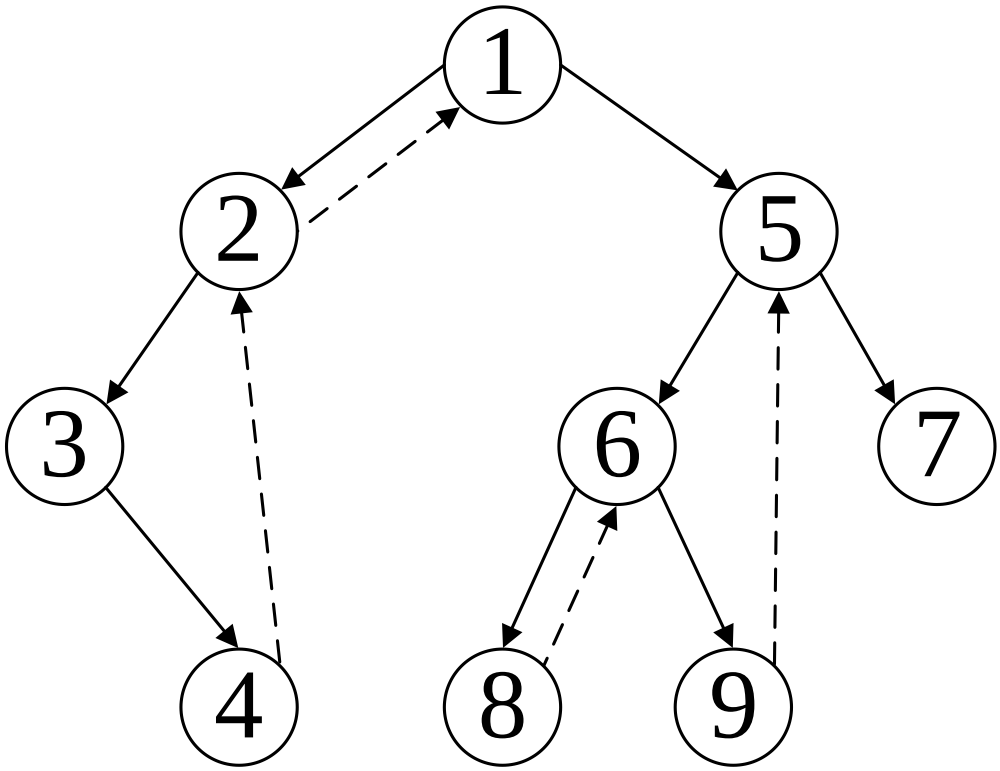


Рисунок 5.4 – Симметрично прошитое справа бинарное дерево

Также используются бинарные деревья, прямо прошитые справа и слева. В них пустые правые и левые указатели узлов заменены соответственно на их преемников и предшественников при прямом порядке просмотра. Прошитые деревья эффективно проходятся без использования стека.

Поскольку нужно каким-то образом отличать обычную связь от прошивочной нити, к каждому узлу добавляется два однобитовых (логических) поля тега: `ltag` и `rtag`. Если значение тега `true`, соответствующее поле связи является обычной связью. В случае значения `false` – прошивочной нитью.

В этой связи узел обычного бинарного дерева представляется структурой. Пример реализации в листинге 5.2.

Листинг 5.2 – Структура узла бинарного дерева на языке программирования C++

```
1   struct Node {
2       int info;     //Информационное поле
3       Node* left;   //Указатель на левое поддерево
4       Node* right;  //Указатель на правое поддерево
5   };
```

Узел прошитого бинарного дерева имеет иную структуру. Пример реализации в листинге 5.3.

Листинг 5.3 – Структура узла прошитого бинарного дерева на языке программирования C++

```
1 struct ThreadedNode {
2   int info; //Информационное поле
3   bool ltag, rtag; //Теги прошивочных нитей
4   ThreadedNode* left; //Указатель на левое поддерево или предшественника
5   ThreadedNode* right; //Указатель на правое поддерево или преемника
6 };
```

Логические поля в прошитом дереве могут принимать следующие значения:

1) `ltag = true`, следовательно, `left` представляет собой обычную связь;

2) `ltag = false`, следовательно, `left` указывает на узел-предшественник;

3) `rtag = true`, следовательно, `right` представляет собой обычную связь;

4) `rtag = false`, следовательно, `right` указывает на узел-преемник.

Рассмотрим вставку новой вершины слева от заданной в симметрично прошитое бинарное дерево (рисунок 5.5).

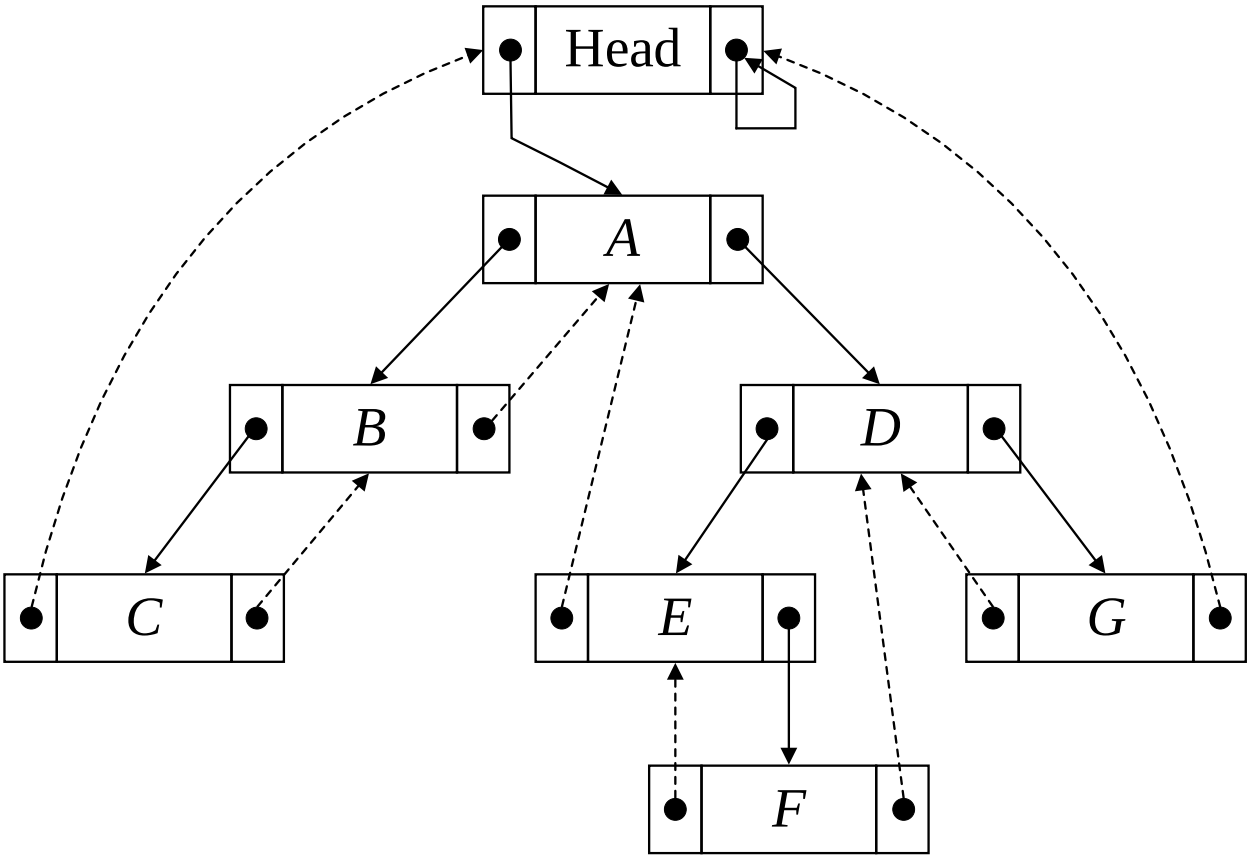


Рисунок 5.5 – Исходное симметрично прошитое дерево

На рисунке 5.6 показано результирующее дерево (после вставки нового элемента).

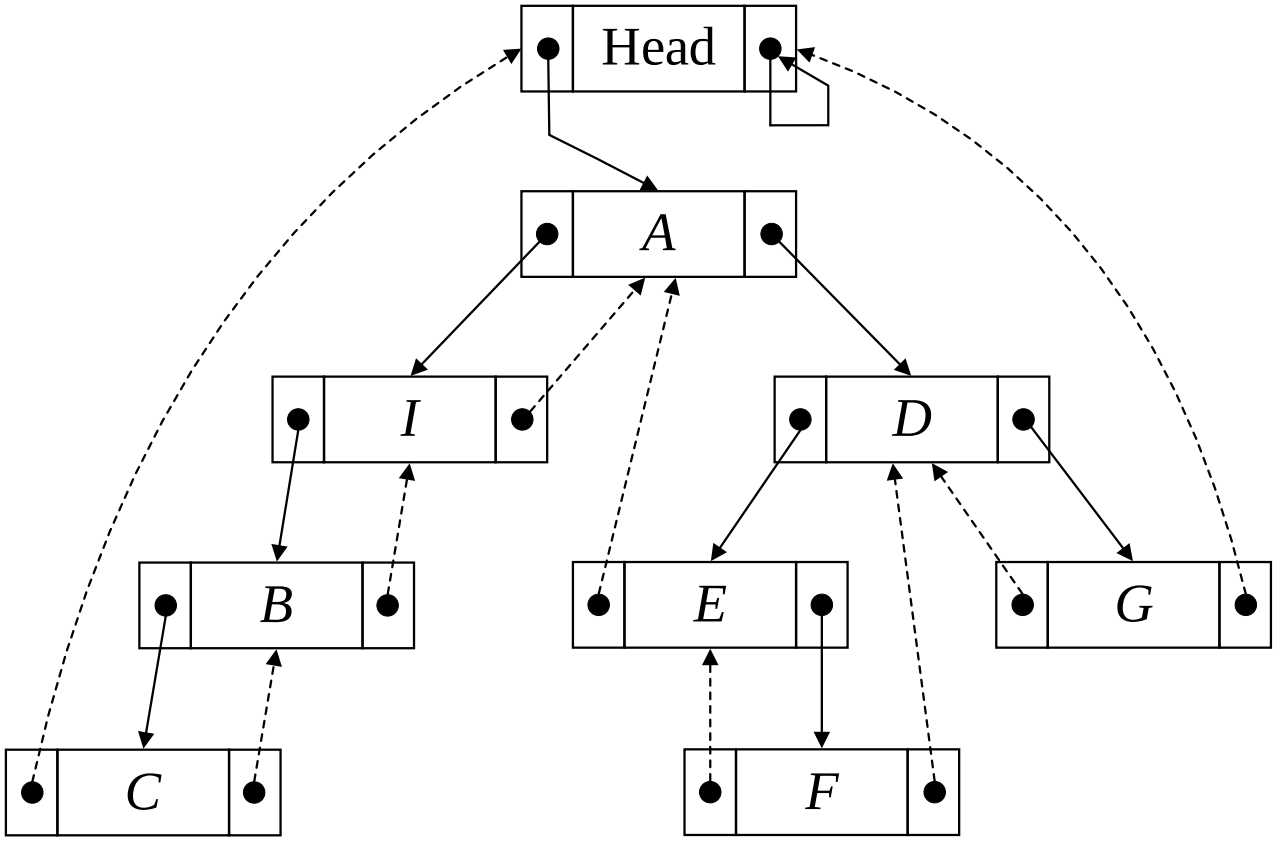


Рисунок 5.6 – Прошитое дерево после вставки в него нового элемента

Здесь требовалось вставить вершину *I* в качестве левого поддерева вершины *A*, если *A* не имеет левого поддерева. В противном случае новая вершина вставляется между *A* и ее левым сыном.

Для удобства создания и обхода дерева используется дополнительная головная вершина `Head`, которая служит при симметричном обходе предшественником его первой вершины и приемником всех его концевых вершин.

Преимущества прошитых деревьев: быстрый обход, отсутствие необходимости в стеке, возможность определить предшественника и преемника вершины.

Недостаток прошитых деревьев: включение новой вершины в дерево занимает больше времени, т. к. необходимо поддерживать два типа связей: структурные и по нитям. Поэтому прошитые деревья целесообразно использовать в тех задачах, где изменения в деревьях происходят редко, а обходы выполняются часто.

Алгоритмы прошивки бинарных деревьев

Алгоритм правосторонней симметричной прошивки бинарного дерева. Строится бинарное дерево. При этом поля `ltag` и `rtag` создаваемых узлов дерева остаются неопределенными, а `left` и `right` соответственно указывают на левое и правое поддеревья либо равны `nullptr`. На корень построенного дерева указывает `root`.

Создается головной узел, `left` которого указывает на корень дерева, а `right` на сам головной узел.

Пример реализации представлен в листинге 5.4.

Листинг 5.4 – Создание головного узла для прошитого бинарного дерева на языке программирования C++

```
1   ThreadedNode* HEAD = new ThreadedNode(0);
2   HEAD->left = root;
3   HEAD->right = HEAD;
```

Информационное поле и поля тегов головного узла можно оставить неопределенными.

Алгоритм прошивки правых связей. Вводится дополнительный глобальный указатель y (указатель на узел, предшествующий текущему узлу). Указатель на текущий узел p устанавливаем на корень дерева (`p = HEAD->left`).

Программный код процедуры прошивки правых связей `rightsew` представлен в листинге 5.5.

Листинг 5.5 – Процедура установки правой прошивки

```
1   void rightsew(ThreadedNode* p, ThreadedNode*& y) {
2       if (y != nullptr) {
3           if (y->right == nullptr) {
4               y->rtag = false;
5               y->right = p;
6           } else {
7               y->rtag = true;
8           }
9       }
10      y = p;
11  }
```

Алгоритм обхода прошитого бинарного дерева

Пусть HEAD – указатель на головной узел прошитого дерева, p – указатель на текущий узел. Тогда алгоритм симметричного обхода прошитого дерева можно сформулировать следующим образом:

– шаг 1. Перейти к корню дерева (`p = HEAD->left`);

– шаг 2. До тех пор пока `p->left != nullptr`, повторять: `p = p->left`, т. е. идти по левой ветви до самого левого узла;

– шаг 3. Обработать узел p, например напечатать `p->info`;

– шаг 4. Если `p->rtag` равен `false`, то `p = p->right` – перейти к шагу 3 (к преемнику). Иначе `p = p->right` – перейти к шагу 2.

Алгоритм заканчивает работу, когда p станет равным HEAD.

Программный код процедуры симметричного обхода `sim_print` представлен в листинге 5.6.

Листинг 5.6 – Процедура симметричного обхода

```
1   void sim_print(ThreadedNode* x, ThreadedNode*& y) {
2       if (x != nullptr) {
3           sim_print(x->left, y);
4           rightsew(x, y);
5           sim_print(x->right, y);
6       }
7   }
```

5.2 Задание

1 Согласовать индивидуальный вариант задания с преподавателем (таблица 5.1), внимательно изучить формулировку задачи и теоретический материал по теме.

2 Разработать алгоритм и написать программный код для решения следующей задачи.

Ввести 10–15 целых чисел и построить из них бинарное дерево поиска.

Выполнить симметричную прошивку бинарного дерева поиска и выполнить обход в симметричном порядке. Вывести результат обхода (порядок обхода вершин).

Реализовать поиск, вставку и удаление элементов в симметрично прошитом бинарном дереве (предусмотреть повторный ввод).

В соответствии с индивидуальным заданием реализовать пользовательскую функцию обработки дерева.

Организовать вывод результатов работы.

3 Организовать текстовый пользовательский интерфейс в программе. При выполнении задания реализовать ввод исходных данных пользователем с клавиатуры, а вывод результатов выполнения программы в консоль (на экран). Предусмотреть возможность автоматического заполнения случайными данными в заданном диапазоне.

4 Проверить правильность вычислений на тестовых примерах, выполнив серию контрольных прогонов программы.

5 ***Дополнительно*** реализовать поддержку нескольких типов прошивки.

6 ***Дополнительно*** реализовать визуальное отображение построенного дерева в консоли.

Таблица 5.1 − Варианты индивидуальных заданий для контрольной работы № 1

Вариант	**Задание**
1	Определить высоту симметрично прошитого бинарного дерева
2	Определить количество узлов в дереве
3	Определить количество листовых узлов в дереве

Вариант	Задание
4	Определить глубину заданного узла
5	Реализовать поиск максимального элемента в дереве
6	Реализовать проверку, является ли дерево строго бинарным (у каждого узла либо два потомка, либо ни одного)
7	Реализовать поиск минимального элемента в дереве
8	Реализовать подсчет количества узлов с одним потомком
9	Вывести количество узлов на третьем уровне дерева
10	Реализовать удаление всех листовых узлов дерева
11	Реализовать удаление всех узлов с четными значениями
12	Реализовать удаление всех узлов, у которых только один потомок
13	Реализовать поиск ближайшего большего элемента для заданного значения
14	Вывести родителя заданного значения
15	Реализовать поиск ближайшего меньшего элемента для заданного значения
16	Вывести всех потомков заданного значения
17	Определить, являются ли два заданных узла братьями (имеют общего родителя)
18	Найти наибольшее поддерево, содержащее только четные числа
19	Определить сумму значений всех узлов в дереве
20	Найти наибольшее поддерево, содержащее только нечетные числа
21	Определить произведение значений всех узлов в дереве
22	Реализовать функцию поиска количества узлов в заданном диапазоне значений
23	Найти расстояние между двумя узлами в дереве
24	Реализовать удаление всех узлов, не имеющих правого потомка
25	Реализовать подсчет количества узлов с одним потомком
26	Реализовать поиск самой длинной ветви дерева
27	Вывести сумму значений узлов на третьем уровне дерева
28	Реализовать функцию, которая выводит номер уровня, на котором сумма значений узлов четная
29	Реализовать функцию определения количества узлов, не имеющих левого потомка

Вариант	Задание
30	Реализовать функцию вычисления разницы между максимальным и минимальным значениями
31	Реализовать функцию подсчета узлов с четными значениями

5.3 Контрольные вопросы

1 Что такое прошивка (threading) бинарного дерева? Зачем она нужна?

2 Что такое прошитое бинарное дерево (Threaded Binary Tree) и в чем его отличие от обычного бинарного дерева поиска?

3 Какие бывают типы прошивки бинарного дерева? В чем их различия?

4 Как прошивка бинарного дерева связана с обходом дерева?

5 В чем преимущество прошитого дерева перед обычным бинарным деревом при обходе?

6 Как выполняется симметричная прошивка бинарного дерева поиска?

7 Как реализовать симметричный (inorder) обход прошитого бинарного дерева без рекурсии?

8 Как обновлять прошитые ссылки после вставки/удаления нового узла?

9 Почему балансировка прошитого дерева требует обновления прошивки?

10 Как выполнить одностороннюю прошивку бинарного дерева?

11 Как выполнить двустороннюю прошивку бинарного дерева?

12 Как прошивка влияет на поиск в бинарном дереве?

13 Чем прошитое бинарное дерево отличается от двусвязного списка?

14 Какую память экономит прошитое дерево по сравнению с обычным бинарным деревом при обходе?

15 Можно ли прошивать AVL-дерево? Как это повлияет на балансировку?

6 КОНТРОЛЬНАЯ РАБОТА № 2. ХЕШИРОВАННЫЕ И ИНДЕКСИРОВАННЫЕ ФАЙЛЫ

6.1 Краткие теоретические сведения

Хешированный файл – это структура данных во внешней памяти, в которой записи располагаются не последовательно, а в определенных позициях в файле, вычисляемых по хеш-функции.

В зависимости от реализации может быть один или несколько файлов, например:

1 Основной хешированный файл (например, *data.dat* или *dictionary.txt*). В нем хранятся записи словаря (например, пары «ключ – значение»). Данные записываются в бинарном виде (например, `key : value`), размещаются в файле по хеш-адресам, вычисленным по ключу.

2 Файл индексов (например, *index.dat*) используется при реализации индексации. Содержит таблицу соответствий между ключами и позициями в файле. Может ускорить поиск, если вместо полного сканирования хеш-таблицы обращаться к индексу.

3 Журнал изменений – опционально (например, *log.txt*). Например, для ведения истории действий (добавление, удаление, обновление).

При работе с хешированными файлами:

1) формируется ключ (например, ID студента);

2) вычисляется хеш по формуле (`hash(ID)` = адрес в файле);

3) запись сохраняется по найденному адресу;

4) при поиске вычисляется хеш и сразу осуществляется переход к нужному месту.

Примеры простейшей хеш-функции вычисления адреса записи в файле, реализация добавления и поиска записи представлены в листингах 6.1–6.3.

Листинг 6.1 – Программная реализация хеш-функции на языке программирования C++

```
1   int hashFunction(int key, int tableSize) {
2       return key % tableSize; // Простое деление по модулю
3   }
```

Листинг 6.2 – Программная реализация добавления записи в хешированный файл

```
1   fstream file("data.dat", ios::in | ios::out | ios::binary);
2   int key = 12345, value = 8;
3   int position = hashFunction(key, TABLE_SIZE)*sizeof(int)*2;
4   file.seekp(position);
5   file.write(reinterpret_cast<char*>(&key), sizeof(int));
6   file.write(reinterpret_cast<char*>(&value), sizeof(int));
7   file.close();
```

Листинг 6.3 – Программная реализация поиска элемента в хешированном файле

```
1   fstream file("data.dat", ios::in | ios::binary);
2   int key = 12345, value;
3   int position = hashFunction(key, TABLE_SIZE)*sizeof(int)*2;
4   file.seekg(position);
5   int foundKey;
6   file.read(reinterpret_cast<char*>(&foundKey), sizeof(int));
7   if (foundKey == key) {
8       file.read(reinterpret_cast<char*>(&value), sizeof(int));
9       cout << "Оценка студента " << key << ": " << value <<
    endl;
10  } else {
11      cout << "Студент не найден" << endl;
12  }
13  file.close();
```

Словарь студентов и оценок будет храниться в файле, где данные записаны в формате: ключ (ID студента) → значение (оценка).

Пример хеш-файла представлен на рисунке 6.1.

Хеш	Ключ (ID студента)	Значение (Оценка)
12	12345	8
23	67890	10

Рисунок 6.1 – Пример хеш-файла для словаря с номерами студентов и их оценками

Индексированный файл – это файл, в котором данные хранятся в порядке сортировки, а для быстрого доступа используется отдельный индексный файл.

Индексный файл (разряженный индекс) – это файл, содержащий пары (x, b), где x – значение ключа, а b – физический адрес блока, в котором значение ключа первой записи равняется x. Индексный файл отсортирован по значениям ключей.

При работе с индексированными файлами:

1) создается основной файл, содержащий записи;

2) создается индексный файл, где указывается позиция каждой записи;

3) сначала осуществляется поиск по индексу, а затем переход к нужной записи.

Индексированный файл содержит таблицу индексов с указанием, где лежат нужные данные.

Примеры реализации создания индексированного файла, поиска записи в нем и обновления индексированного файла после вставки новой записи представлены в листингах 6.4–6.6.

Листинг 6.4 – Пример реализации создания индексированного файла

```
fstream indexFile("index.dat", ios::out | ios::binary);
int key = 12345, position = 12; //Позиция в основном файле
indexFile.write(reinterpret_cast<char*>(&key), sizeof(int));
indexFile.write(reinterpret_cast<char*>(&position), sizeof(int));
indexFile.close();
```

Листинг 6.5 – Пример реализации поиска записи через индексированный файл

```
fstream indexFile("index.dat", ios::in | ios::binary);
fstream dataFile("data.dat", ios::in | ios::binary);

int keyToFind = 12345, key, position;
bool found = false;
while (indexFile.read(reinterpret_cast<char*>(&key), sizeof(int))) {
    indexFile.read(reinterpret_cast<char*>(&position), sizeof(int));
    if (key == keyToFind) {
        found = true;
        break;
    }
}
if (found) {
    dataFile.seekg(position);
    int value;
    dataFile.read(reinterpret_cast<char*>(&value), sizeof(int));
    cout << "Оценка студента " << keyToFind << ": " << value << endl;
} else {
    cout << "Студент не найден" << endl;
}
indexFile.close();
dataFile.close();
```

Листинг 6.6 – Программная реализация обновления индексированного файла после вставки новой записи

```
fstream indexFile("index.dat", ios::app | ios::binary);
int newKey = 67890, newPosition = 23; // Новая запись
indexFile.write(reinterpret_cast<char*>(&newKey), sizeof(int));
indexFile.write(reinterpret_cast<char*>(&newPosition), sizeof(int));
indexFile.close();
```

Пример индексного файла представлен на рисунке 6.2.

```
ID     | Адрес в data.dat
12345  | 0012
67890  | 0023
```

Рисунок 6.2 – Пример индексного файла

Данные хранятся в файле *data.dat*. При поиске информации программа сначала ищет адрес в файле *index.dat* и затем переходит по нему в файле *data.dat*.

6.2 Задание

1 Согласовать индивидуальный вариант задания с преподавателем (таблица 6.1), внимательно изучить формулировку задачи и теоретический материал по теме.

2 Разработать алгоритм и написать программный код для решения следующей задачи.

Создать словарь на основе динамических списков.
Сохранить словарь во внешней памяти в виде хешированного файла.
Реализовать основные операции: поиск, вставку и удаление.
Реализовать и применить методы разрешения коллизий.
Реализовать пользовательское меню из следующих пунктов:
– создать новый файл или загрузить существующий;
– добавить данные в файл;
– удалить запись из файла;
– найти запись в файле;
– вывести данные файла на экран;
– выполнить дополнительное задание варианта (при его наличии);
– завершить работу программы.

3 Организовать текстовый пользовательский интерфейс в программе. При выполнении задания реализовать ввод исходных данных пользователем с клавиатуры, а вывод результатов выполнения программы в консоль (на экран). Предусмотреть возможность автоматического заполнения случайными данными в заданном диапазоне.

4 Проверить правильность вычислений на тестовых примерах, выполнив серию контрольных прогонов программы.

5 ***Дополнительно*** реализовать работу с индексированными файлами для ускорения работы.

Таблица 6.1 − Варианты индивидуальных заданий для контрольной работы № 2

Вариант	**Задание**
1	*Тематика:* ID студента → средний балл. *Метод разрешения коллизий:* метод цепочек. *Дополнительно:* реализовать экспорт данных в CSV-файл
2	*Тематика:* номер заказа → сумма заказа. *Метод разрешения коллизий:* двойное хеширование. *Дополнительно:* добавить резервное копирование файла перед изменениями
3	*Тематика:* код товара → количество на складе. *Метод разрешения коллизий:* открытая адресация. *Дополнительно:* реализовать логирование всех операций

Вариант	Задание
4	*Тематика:* регистрационный номер машины → владелец. *Метод разрешения коллизий:* квадратичное пробирование. *Дополнительно:* реализовать сортированный вывод записей
5	*Тематика:* код клиента → баланс счета. *Метод разрешения коллизий:* двойное хеширование. *Дополнительно:* реализовать экспорт данных в JSON
6	*Тематика:* телефонный номер → Ф. И. О. владельца. *Метод разрешения коллизий:* метод цепочек. *Дополнительно:* реализовать поиск по имени или отчеству
7	*Тематика:* табельный номер сотрудника → должность. *Метод разрешения коллизий:* линейное пробирование. *Дополнительно:* реализовать сортировку по должности
8	*Тематика:* серийный номер устройства → год выпуска. *Метод разрешения коллизий:* двойное хеширование. *Дополнительно:* реализовать вывод устройств старше заданного года
9	*Тематика:* код авиарейса → время вылета. *Метод разрешения коллизий:* метод цепочек. *Дополнительно:* реализовать поиск по диапазону времени
10	*Тематика:* код договора → дата заключения. *Метод разрешения коллизий:* открытая адресация. *Дополнительно:* реализовать экспорт в CSV
11	*Тематика:* ID сотрудника → оклад. *Метод разрешения коллизий:* квадратичное пробирование. *Дополнительно:* реализовать поиск по диапазону оклада
12	*Тематика:* лицензия ПО → дата окончания. *Метод разрешения коллизий:* метод цепочек. *Дополнительно:* реализовать автоудаление просроченных лицензий
13	*Тематика:* код товара → цена. *Метод разрешения коллизий:* линейное пробирование. *Дополнительно:* реализовать вывод товаров в порядке убывания
14	*Тематика:* код дисциплины → название предмета. *Метод разрешения коллизий:* двойное хеширование. *Дополнительно:* реализовать поиск по подстроке названия
15	*Тематика:* серийный номер билета → дата мероприятия. *Метод разрешения коллизий:* открытая адресация. *Дополнительно:* реализовать поиск по диапазону дат
16	*Тематика:* код клиента → Ф. И. О. *Метод разрешения коллизий:* метод цепочек. *Дополнительно:* реализовать экспорт в XML

Вариант	Задание
17	*Тематика:* код заказа → статус выполнения. *Метод разрешения коллизий:* линейное пробирование. *Дополнительно:* реализовать автоудаление завершенных заказов
18	*Тематика:* номер страхового полиса → дата окончания. *Метод разрешения коллизий:* двойное хеширование. *Дополнительно:* реализовать автоматическое удаление просроченного полиса
19	*Тематика:* код книги → автор. *Метод разрешения коллизий:* метод цепочек. *Дополнительно:* реализовать сортировку по автору
20	*Тематика:* номер счета → валюта. *Метод разрешения коллизий:* квадратичное пробирование. *Дополнительно:* добавить резервное копирование файла перед изменениями
21	*Тематика:* код ученика → посещаемость. *Метод разрешения коллизий:* открытая адресация. *Дополнительно:* реализовать логирование отсутствий
22	*Тематика:* код абонента → тарифный план. *Метод разрешения коллизий:* двойное хеширование. *Дополнительно:* реализовать экспорт в CSV
23	*Тематика:* код медицинской карты → диагноз. *Метод разрешения коллизий:* метод цепочек. *Дополнительно:* реализовать поиск по частичному совпадению
24	*Тематика:* код производителя → страна. *Метод разрешения коллизий:* линейное пробирование. *Дополнительно:* реализовать поиск по стране
25	*Тематика:* артикул одежды → размер. *Метод разрешения коллизий:* открытая адресация. *Дополнительно:* реализовать сортировку по размеру
26	*Тематика:* код рейса → аэропорт прибытия. *Метод разрешения коллизий:* двойное хеширование. *Дополнительно:* реализовать автоудаление выбранного аэропорта
27	*Тематика:* код брони → дата заезда. *Метод разрешения коллизий:* метод цепочек. *Дополнительно:* реализовать поиск по диапазону дат
28	*Тематика:* код игры → разработчик. *Метод разрешения коллизий:* квадратичное пробирование. *Дополнительно:* реализовать сортировку по разработчику

Вариант	Задание
29	*Тематика:* код SIM-карты → оператор. *Метод разрешения коллизий:* двойное хеширование. *Дополнительно:* добавить резервное копирование файла перед изменениями
30	*Тематика:* код заказа → адрес доставки. *Метод разрешения коллизий:* метод цепочек. *Дополнительно:* реализовать логирование измененных адресов
31	*Тематика:* код программы → версия. *Метод разрешения коллизий:* двойное хеширование. *Дополнительно:* реализовать поиск последней версии

6.3 Контрольные вопросы

1 Что такое внешняя память? Чем она отличается от оперативной памяти?

2 Что такое хешированный файл и чем он отличается от обычного файла?

3 Какую роль играет хеш-функция в организации хешированного файла?

4 Какие основные проблемы возникают при работе с внешней памятью?

5 Какие методы разрешения коллизий используются в хешированных файлах?

6 Что такое блоки данных (data blocks) и как они используются при работе с внешней памятью?

7 Как работает хеширование для организации данных во внешней памяти? Какие задачи оно решает?

8 Как выполняется поиск в хешированном файле?

9 Как обрабатываются удаленные записи в хешированном файле?

10 Что такое индексированные файлы? Какие задачи они решают?

11 Как работает индексный файл? Какие структуры данных используются для индексации?

12 Какие типы индексов существуют?

13 В чем разница между первичными и вторичными индексами?

14 Как реализовать двухуровневую индексацию и в чем ее преимущества?

15 Какие преимущества дает использование индексов по сравнению с полным сканированием файла?

16 В чем преимущества и недостатки хешированных файлов перед индексированными файлами?

17 Какие факторы влияют на эффективность хеширования и выбор хеш-функции?

ПРИЛОЖЕНИЕ А

Сложность выполнения операций для различных структур данных

Структура данных	Временная сложность								Расход памяти
	Средняя				В худшем случае				В худшем случае
	Доступ	***Поиск***	***Вставка***	***Удаление***	***Доступ***	***Поиск***	***Вставка***	***Удаление***	
Массив	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Стек	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Очередь	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Односвязный список	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Двусвязный список	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Список с пропусками	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$
Хеш-таблица	Не-применимо	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	Не-применимо	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Бинарное дерево поиска	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Декартово дерево	Не-применимо	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	Не-применимо	$O(n)$	$O(n)$	$O(n)$	$O(n)$
B-дерево	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Красно-черное дерево	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
AVL-дерево	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Splay-дерево (расширяющееся дерево)	Не-применимо	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	Не-применимо	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
K-мерное дерево (*k-d*-дерево)	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И РЕКОМЕНДОВАННОЙ ЛИТЕРАТУРЫ

1 Парамонов, А. И. Алгоритмизация и компьютерные вычисления на языках программирования высокого уровня : учеб.-метод. пособие / А. И. Парамонов, А. Г. Савенко. – Минск : БГУИР, 2024. – 211 с.

2 Савенко, А. Г. Основы компьютерной техники : учеб.-метод. пособие / А. Г. Савенко, А. И. Парамонов. – Минск : БГУИР, 2025. – 131 с.

3 Ахо, А. В. Структуры данных и алгоритмы / А. В. Ахо, Д. Хопкрофт, Д. Д. Ульман. – М. : Вильямс, 2018. – 400 с.

4 Кормен, Т. Алгоритмы: вводный курс / Т. Кормен. – М. : Вильямс, 2014. – 208 с.

5 Алгоритмы: построение и анализ / Т. Кормен [и др.]. – 3-е изд. – М. : Вильямс, 2016. – 1328 с.

6 Вирт, Н. Алгоритмы и структуры данных. Новая версия для Оберона / Н. Вирт ; пер. с англ. под ред. Ф. В. Ткачева. – М. : ДМК Пресс, 2011. – 272 с.

7 Скиена, С. Алгоритмы. Руководство по разработке / С. Скиена. – 2-е изд. – СПб. : БХВ-Петербург, 2019. – 720 с.

8 Венгроу, Д. Прикладные структуры данных и алгоритмы. Прокачиваем навыки / Дж. Венгроу. – СПб. : Питер, 2024. – 512 с.

9 Стивенс, Р. Алгоритмы. Теория и практическое применение / Р. Стивенс. – М. : Изд-во «Э», 2016. – 544 с.

10 Зингаро, Д. Алгоритмы на практике. Решение реальных задач / Д. Зингаро ; пер. с англ. Д. И. Брайт. – СПб. : Питер, 2023. – 432 с.

11 Фридман, А. C/C++. Алгоритмы и приемы программирования / А. Фридман. – М. : Бином, 2007. – 560 с.

12 Клейнберг, Д. Алгоритмы. Разработка и применение / Д. Клейнберг, Е. Тардос ; пер. с англ. Е. Матвеева. – СПб. : Питер, 2016. – 800 с.

13 Уилсон, Р. Введение в теорию графов / Р. Уилсон ; пер. с англ. И. В. Красикова. – 5-е изд. – М. : Вильямс, 2020. – 240 с.

14 Рафгарден, Т. Совершенный алгоритм. Графовые алгоритмы и структуры данных / Т. Рафгарден. – СПб. : Питер, 2023. – 256 с.

15 Бхаргава, А. Грокаем алгоритмы: иллюстрированное пособие для программистов и любопытствующих / А. Бхаргава. – СПб. : Питер, 2022. – 288 с.

16 Макконнелл, Д. Анализ алгоритмов. Активный обучающий подход : учеб. пособие / Д. Макконнелл ; пер. с англ. С. А. Кулешова ; под ред. С. К. Ландо. – 3-е изд., доп. – М. : Техносфера, 2009. – 416 с.

17 Седжвик, Р. Computer Science: основы программирования на Java, ООП, алгоритмы и структуры данных / Р. Седжвик. – СПб. : Питер, 2018. – 1072 с.

Св. план 2025, поз. 52

Учебное издание

Парамонов Антон Иванович
Потоцкий Дмитрий Сергеевич
Савенко Андрей Геннадьевич

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

Редактор *А. Ю. Шурко*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *А. А. Лущикова*

Подписано в печать 09.02.2026. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 6,51. Уч.-изд. л. 6,7. Тираж 50 экз. Заказ 174.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
Ул. П. Бровки, 6, 220013, г. Минск