Вид нагрузки: ЛР

Для дисциплин:

Год набора	Шифр	Форма обучения (о или о/з)	Дисциплина
2016	09.03.04	о	Алгоритмы и структуры данных
2016	09.03.01	о	Алгоритмы и структуры данных
2017	09.03.04	о	Алгоритмы и структуры данных
2017	09.03.01	о	Алгоритмы и структуры данных
2018	09.03.04	о	Алгоритмы и структуры данных
2018	09.03.01	о	Алгоритмы и структуры данных
2018	09.03.02	о	Алгоритмы и структуры данных
2018	09.03.03	о	Алгоритмы и структуры данных

Предлагаемый вариант названия:

Алгоритмы и структуры данных: методические указания по выполнению лабораторных работ

Содержание

Введение

Лабораторный практикум содержит теоретические сведения и описание лабораторных работ в области использования и реализации наиболее распространенных и показательных с точки зрения эффективного решения алгоритмических задач структур данных и алгоритмов.

Лабораторный практикум предназначен для изучения дисциплин образовательных программ подготовки бакалавров, связанных с вопросами разработки, анализа, модификации и реализации алгоритмов и структур данных.

Описание каждой лабораторной работы содержит несколько подразделов, в которых изложены основные теоретические сведения, приведены примеры и практические рекомендации.

Для выполнения лабораторных работ может использоваться любой императивный язык высокого уровня и любая реализующая его среда программирования (по выбору студента).

Способ реализации пользовательского интерфейса (в случае его необходимости), процедур ввода и вывода данных также определяется студентом. Если другое не установлено заданием на лабораторную работу, то единственным требованием является достаточность интерфейса для решения поставленной задачи.

1. Лабораторная работа «Абстрактный тип данных на примере реализации очереди в последовательной и связной памяти»

1.1. Абстрактный тип данных. Основные определения и понятия

Тип данных – множество значений, которые может принимать переменная, и множество операций с ней.

Абстрактный тип данных (АТД) – это математическая модель данных с совокупностью операторов, определенных в рамках этой модели. Например, множество целых чисел с операторами объединения, пересечения и разности множеств. Таким образом, под типом данных обычно понимается элементарная «атомарная» структура данных, встроенная в язык программирования. При решение различных алгоритмических задач возникает потребность в более сложных информационных объектах. Поэтому абстрактный тип данных рассматривается как сложная структура, которая может включать в себя другие типы данных, в том числе и абстрактные. Операторы АТД также инкапсулируют в себя как элементарные операторы языка программирования, так и сложные процедуры.

Использование концепции АТД при нисходящем проектировании позволяет на этапе разработки общих алгоритмов не уделять внимания реализации будущей структуры данных, реализующей используемый АТД. В тоже время разработчик структуры данных может не уделять внимания тому, где и как используется эта структура. В обоих случаях разработчики руководствуются лишь описанием, подобным следующему (на примере очереди):

```
АТД: Queue [1 .. MaxEl] of ElType {FIFO}
Операторы:    InitQueue (Q: Queue)
              EnQueue (Q: Queue, x: ElType): bool
              DeQueue (Q: Queue; x: ElType): bool
              Empty (Q: Queue): bool
              Full (Q: Queue): bool
```

Примечание: параметры, перечисленные в операторе после точки с запятой, обозначают возвращаемые значения (указатели или параметры-переменные).

Данная запись означает, что используется очередь, элементами которой являются переменные абстрактного типа данных ElType (обычно определяется в основной части, так как реализация очереди с точки зрения многократного использования текста модуля должна быть независима от ее наполнения). Максимальная длина очереди задается переменной MaxEl. Очередь функционирует по принципу FIFO – «первым пришел, первым обслужен». Для очереди реализуются операторы инициализации, вставки и удаления элемента, проверки на пустоту и переполнение. Необходимо отметить, что при вызове каждого оператора в него должен передаваться идентификатор конкретной очереди, для которой он вызван. Это обусловлено тем, что разработчики не могут быть уверены, что модуль очереди будет использован только ими и только один раз.

Также необходимо отметить, что данный подход является прообразом объектно-ориентированного подхода, где объект в самом простом случае можно рассматривать как АТД, реализованный специальными средствами языка программирования.

Реализация АТД заключается в выборе принципа организации хранения данных (или их структурирования) и реализации объявлений переменных и операторов АТД на языке программирования. Результатом реализации является структура данных.

Структура данных (СД) – общее свойство информационного объекта, с которым взаимодействует та или иная программа, характеризующееся:

- множеством допустимых значений данной структуры;
- набором допустимых операций;
- характером организованности.

Основным отличием СД от АТД является определенность характера организованности. Под ним понимаются принципы организации физических связей между элементами структуры данных. То есть особенности ее представле-

ния на физическом уровне, которые обеспечивают реализацию логики функционирования, заложенной в модели АТД. Например, АТД очередь может быть реализован на основе последовательности и на основе дерева (актуально для приоритетной очереди), которые в свою очередь могут быть представлены как в связной, так и в последовательной памяти.

1.2. Структура данных типа очередь

Очередь – последовательность, в которую элементы включаются с одной стороны, а исключаются – с другой.

Типовой набор допустимых операций СД типа очередь:

1. Операция инициализации.
2. Операция включения элемента в очередь.
3. Операция исключения элемента из очереди.
4. Операция проверки на пустоту.
5. Операция проверки на переполнение.
6. Операция неразрушающего доступа к элементу (может быть реализована как производная на основе комбинаций операций исключения и включения).

Необходимо отметить, что при грамотной реализации сложность всех операций не зависит от размера структуры данных – O(1).

1.3. Пример реализации очереди в последовательной памяти

В последовательной памяти (при реализации очереди как отображения на массив) очередь может быть представлена как показано на рисунке 1.1.

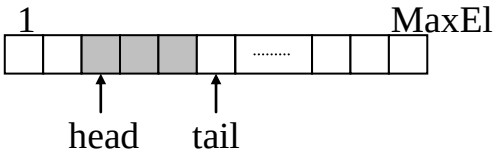


Рисунок 1.1 – Представление очереди в последовательной памяти

Здесь *head* – индекс «головы» очереди (указывает на первый элемент), а *tail* – индекс «хвоста» (указывает на свободную ячейку массива, следующую за последним элементом очереди). При использовании данного представления очереди возникают проблемы с эффективной реализацией операции включения элемента. Например, если индекс хвоста очереди вышел за правую границу массива, то вставка напрямую невозможна. Если при этом в левой части массива имеются свободные ячейки, то можно выполнить сдвиг всех элементов очереди. Однако, сложность такой операции линейна по количеству элементов в очереди – O(*N*).

Данный недостаток вытекает из свойства физической смежности элементов в последовательной памяти и нехарактерен для реализации очереди в связной памяти. Для его преодоления используют кольцевую очередь – очередь, реализованную на массиве, логически замкнутом в кольцо, т.е. за индексом *MaxEl* следует 1 (рисунок 1.2). В этом случае при достижении хвостом конца массива и при наличии незанятых ячеек в его начале осуществляется переход указателя хвоста на первую позицию. Таким образом, кольцевая очередь состоит из элементов массива, стоящих на местах с номерами *head*, *head* + 1, …, [возможно *MaxEl*, 1, ...,] *tail* – 1.

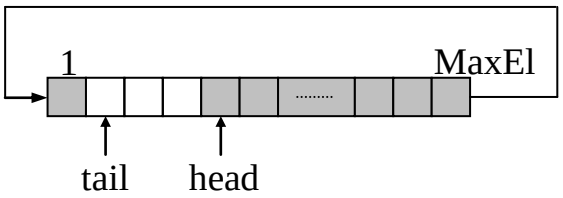


Рисунок 1.2 – Представление кольцевой очереди в последовательной памяти

Отсюда можно получить описание очереди на языке программирования или, как в данном случае, на достаточно общем псевдокоде:

```
Queue = record
        element: array [1..MaxEl] of ElType
        head, tail: integer
end
```

Необходимо отметить, что очередь представляется в виде записи, в которую помимо самого массива элементов включены и индексные переменные. Это обусловлено тем, что при создании переменных типа очередь было бы некорректно и неэффективно для каждой из них создавать дополнительные индексные переменные, которые логически никак не связаны с очередью.

Для реализации операторов АТД очередь необходимо определить условия пустоты и переполнения. Для унификации разрабатываемых процедур принимается, что условием пустоты очереди является равенство индексов *head* и *tail*, а условием заполнения – равенство $head = tail + 1$ или с учетом перехода по кольцу – равенство $head = (tail \bmod MaxEl) + 1$.

Применение таких условий позволяет отказаться от использования вспомогательной переменной N – количество элементов в очереди и операций ее модификации при включениях и исключениях. Однако, при такой реализации имеется возможность использования только $MaxEl - 1$ ячеек массива. Например, включение элемента в очередь, приведенную на рисунке 1.3 приведет к ситуации ее пустоты согласно принятым выше условиям. Данный недостаток имеет существенный вес только в случае реализации большого количества незначительных по длине очередей, в остальных же случаях такой подход обеспечивает некоторое снижение трудоемкости.

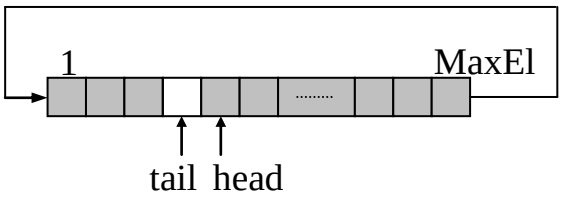


Рисунок 1.3 – Ситуация заполнения кольцевой очереди

На основании изложенного выше можно привести примеры алгоритмов, реализующих операторы АТД очередь.

1. Алгоритм инициализации (исходя из условия пустоты очереди достаточно присвоить ее индексным переменным одно и то же значение, причем, не обязательно 1):

```
Init (Q: Queue)
    Q.head ← 1
    Q.tail ← 1
End Init
```

2. Алгоритм проверки на пустоту

```
Empty (Q: Queue): Bool
    if Q.head = Q.tail then
        return true
    else
        return false
    end if
End Empty
```

3. Алгоритм исключения из очереди

```
Dequeue(Q: Queue; x: ElType): Bool
    if Empty(Q) then
        return false
    else
        x ← Q.Element[Q.head]
        Q.head ← (Q.head mod MaxEl) + 1
        return true
    end if
End Dequeue
```

Алгоритмы других операторов СД типа очередь в последовательной памяти, необходимые для выполнения задания, и СД типа очередь в связной памяти необходимо разработать самостоятельно.

1.4. Задание на лабораторную работу

Дана очередь, моделирующая работу технологической установки, обрабатывающей детали различных типов. В структуре данных хранится код детали (строка длиной 4 символа) и время, требуемое для обработки данной детали (целое число). В системе отсчитывается модельное время.

Интерфейс пользователя должен обеспечивать реализацию следующих функций приложения:

- постановка детали на обработку;
- переход к следующему моменту модельного времени (в случае, если после перехода обработка очередной детали завершена, она исключается из очереди, а на экран выводится информация об этом);
- снятие детали с обработки до ее завершения (моделируется отказ установки) с выводом на экран информации об этом;
- вывод списка обрабатываемых деталей на экран в порядке очереди;
- сброс процесса моделирования (инициализация).

Ход работы:

1. Описать очередь в терминах АТД. Кроме стандартных операторов необходимо предусмотреть оператор получения содержимого очереди от первого до последнего элемента (например, в виде массива элементов).

2. Разработать основной модуль приложения, реализующий логику диалога пользователя и соответствующие вызовы операторов. В нем же необходимо определить тип элемента очереди *ElType*.

3. Разработать алгоритмы, обеспечивающие реализацию операторов очереди. Реализовать СД типа очередь в двух различных модулях:

- 1 модуль – очередь в последовательной памяти (использовать кольцевую очередь, максимальный размер очереди – 5 деталей);
- 2 модуль – очередь в связной памяти на основе линейного односвязного списка (ограничение на количество деталей допускается не вводить).

4. Поочередно подключить модули к основной программе и провести тестирование.

Примечание: при правильной реализации концепции АТД текст основной программы (за исключением определения типа ElType) не должен зависеть от подключаемого модуля.

1.5. Контрольные вопросы

1. Дайте определение типа данных и абстрактного типа данных, охарактеризуйте их отличия и взаимосвязь.

2. Дайте определение абстрактного типа данных и структуры данных, охарактеризуйте их отличия и взаимосвязь.

3. Назовите преимущества использования концепции АТД при разработке сложных программных систем.

4. Чем обусловливается передача идентификатора очереди в качестве параметра при вызове каждого оператора?

5. Каким образом СД типа очередь представляется в последовательной и связной памяти?

6. Чем обусловливается применение кольцевых очередей?

7. Опишите принцип реализации очереди на основе линейного односвязного списка.

8. Оцените, насколько реализованные модули будут универсальны для решения других стандартных задач, требующих использования очередей.

2. Лабораторная работа «Многосвязный список»

2.1. Линейные связные списки

При реализации различных структур данных в последовательной памяти существует ряд ограничений, накладываемых физической смежностью элементов в памяти, что приводит к неэффективности реализации некоторых операций. Для снятия этих ограничений используются указатели, которые определяют логическую последовательность элементов данных независимо от физического расположения в памяти. Простейшими структурами данного класса являются линейные связные списки.

Линейный связный список – это последовательность, элементы которой связаны между собой указателями, хранящимися в самих элементах.

Обычно выделяют следующие виды линейных списков:

- односвязный линейный список;
- двусвязный линейный список;
- многосвязные линейные списки.

Представление линейного списка в связной памяти (на примере линейного двусвязного списка) приведено на рисунке 2.1.

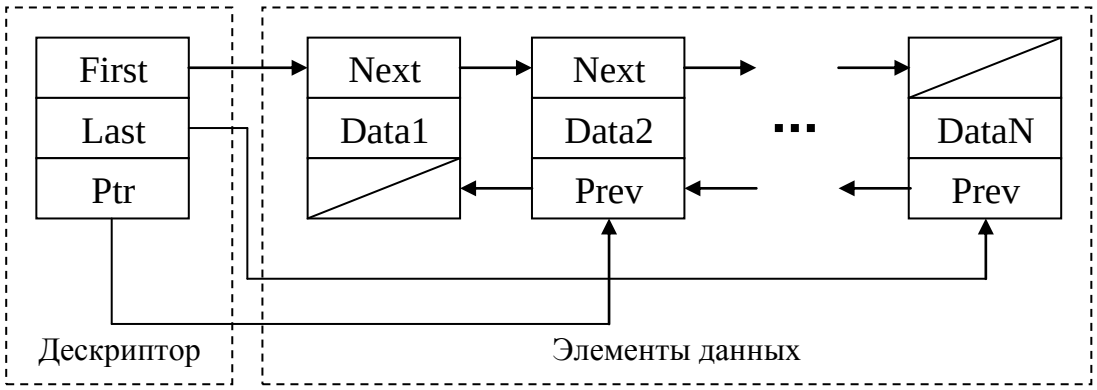


Рисунок 2.1 – Представление линейного двусвязного списка в памяти ЭВМ

Для доступа к элементам данных используется дескриптор, включающий в себя некоторые переменные, общие для данной структуры (необязатель-

но указательного типа, например, поле «количество элементов в списке»). Однако, минимальным требованием к дескриптору является наличие по крайней мере указателя *First*, обеспечивающего доступ к элементам списка. В этом случае тип «список» на языке программирования будет совпадать с типом указателя на элемент списка. В рассматриваемом случае в состав дескриптора введены:

– указатель на последний элемент списка *Last*, например, для быстрого доступа с целью включения элемента в конец списка;

– рабочий указатель *Ptr*, используемый при необходимости сохранять и передавать указатель на некоторый элемент списка в процессе работы программы (использование логически не связанной со списком переменной часто нежелательно).

В данном случае тип список будет определяться как запись-дескриптор:

```
List = record
        First: ^CellType
        Last: ^CellType
        Ptr: ^CellType
end
```

Где *CellType* – тип элемента списка, представляющий собой запись:

```
CellType = record
        Next: ^CellType
        Data: ElType
        Prev: ^CellType
end
```

Где *ElType* – также обычно запись, содержащая непосредственно данные.

Для линейного двусвязного списка обычно реализуется следующий типовой набор операторов:

– инициализация;

– сделать список пустым (обязательное условие освобождения динамически выделенной памяти);

– проверка на пустоту;

– включить элемент в список до рабочего указателя / после рабочего указателя;

– исключить элемент из списка до рабочего указателя / после рабочего указателя.

Для унификации процедур при необходимости включения или исключения из начала или конца списка можно использовать два приведенных выше оператора с предварительной установкой рабочего указателя на нужный элемент. Однако возможна и целенаправленная реализация указанных операторов, если включение и исключение всегда осуществляется из крайних позиций, например, список используется для реализации очереди.

2.2. Принцип организации данных в линейном многосвязном списке

На практике часто возникают задачи выборки подмножества элементов данных по некоторому критерию, или поиска элемента в таком подмножестве. Полный перебор элементов данных не всегда является целесообразным с точки зрения эффективности по времени. Если незначительное увеличение памяти, затрачиваемой на указатели, не является критичным, то имеется возможность содержать один и тот же элемент в нескольких односвязных списках. Причем принадлежность элемента к каждому из поддерживаемых списков определяется на основе соответствия какого-либо его поля или полей критерию данного списка. Тогда выборка данных по критерию потребует просмотра лишь того односвязного списка, элементы которого априорно удовлетворяют заданным условиям.

Линейные списки, в элементах которых количество указателей более двух называются *многосвязными*.

Таким образом, линейный многосвязный список – это набор элементов данных, «прошитый» несколькими односвязными списками. В целях поддержания целостности структуры данных обязательно поддерживается по крайней

мере один односвязный список, содержащий в себе все элементы данных без исключения. Остальные списки выделяются по критериям, определенным конкретной задачей. Для обеспечения доступа к спискам также используются дескрипторы, которые целесообразно объединить в массив. На рисунке 2.2 приведен многосвязный список, элементы которого содержат указатели трех односвязных списков: первый включает все элементы, два других – элементы, информационные поля которых соответствуют некоторым критериям.

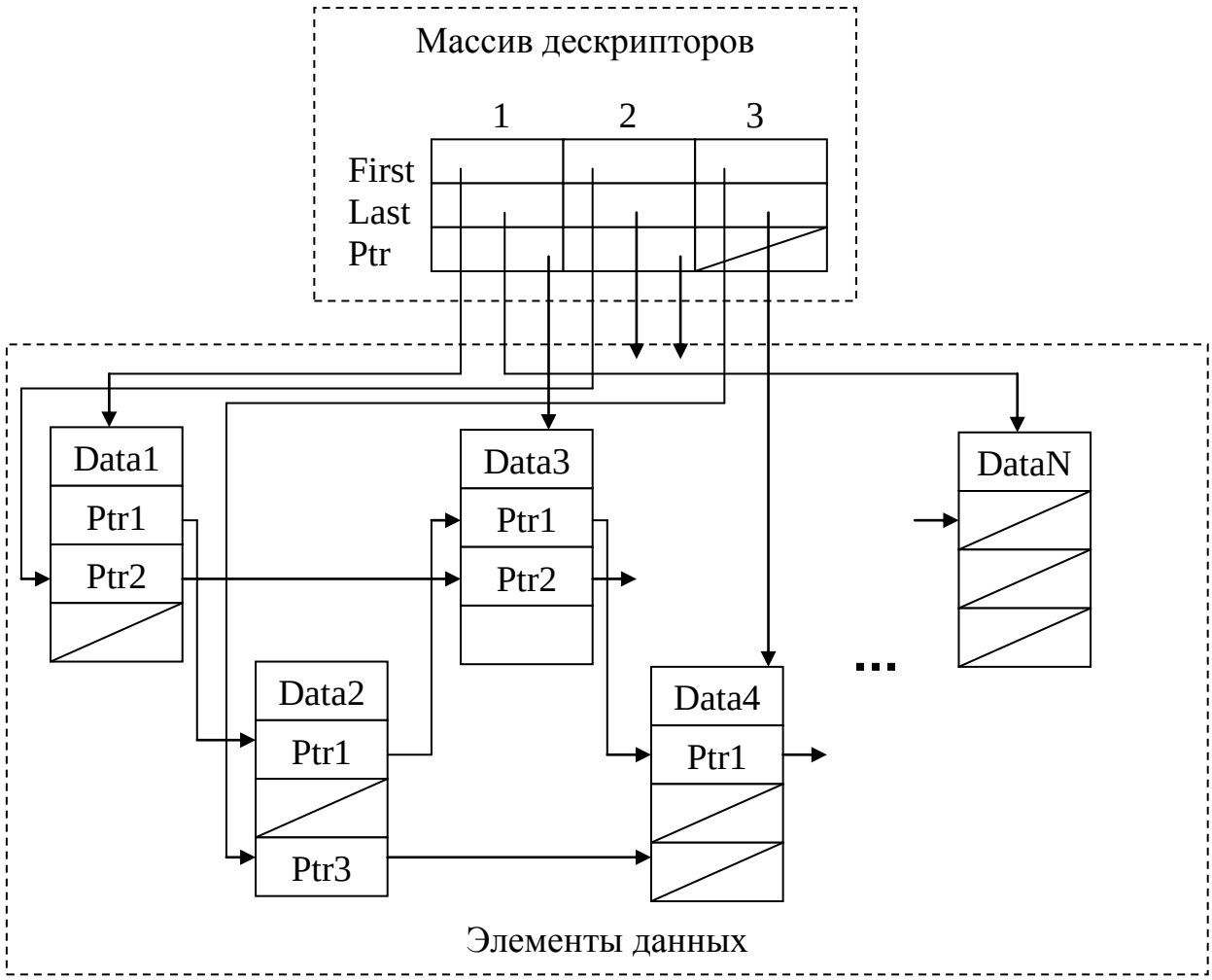


Рисунок 2.2 – Представление многосвязного списка в памяти ЭВМ

В данном примере:

1. Первый список выключает все элементы от *Data1* до *DataN*.

2. Второй список включает элементы *Data1*, *Data3*, ..., некоторые элементы из диапазона от *Data5* до *DataN* – 1. Указатель на последний элемент списка и рабочий указатель также ссылаются на элементы из не отображенного на рисунке диапазона.

3. Третий список включает только элементы *Data2* и *Data4*, причем рабочий указатель не ссылается ни на один из элементов списка.

2.3. Задание на лабораторную работу

Дан список абитуриентов университета. Запись об абитуриенте содержит следующую информацию: фамилия; оценки по трем вступительным экзаменам; сведения о наличии аттестата с отличием; наименование населенного пункта, в котором проживает абитуриент; сведения о необходимости предоставления общежития.

Программное приложение должно реализовывать следующие функции:

- ввод записи об абитуриенте;
- формирование и вывод списков абитуриентов по следующим критериям: все экзамены сданы на «отлично»; имеется аттестат с отличием; проживает за пределами населенного пункта, в котором расположен университет; нуждается в общежитии;
- формирование и вывод полного списка абитуриентов;
- удаление записи об абитуриенте;
- удаление всех записей многосвязного списка.

Ход работы

1. Определить и описать необходимые структуры данных и список операторов.

2. Разработать основной модуль приложения, реализующий логику диалога пользователя и соответствующие вызовы операторов.

3. Разработать алгоритмы, обеспечивающие реализацию операторов. Разработать модуль, реализующий структуру данных многосвязный список.

4. Провести тестирование программы, осуществить наполнение структуры данных.

2.4. Контрольные вопросы

1. Каковы преимущества и недостатки использования связного распределения памяти и динамических структур данных?

2. Приведите основные определения связных списков, набор допустимых операций.

3. Сравните с точки зрения эффективности по времени операции включения, исключения, поиска и доступа к элементу в связном списке и массиве. Чем обусловливаются равенство или неравенство в оценках сложности данных операций?

4. Охарактеризуйте класс задач, при решении которых эффективно применение многосвязных списков.

5. В чем заключаются преимущества и недостатки использования многосвязных списков?

6. Поясните принципы реализации операций включения и исключения элемента в многосвязном списке.

7. Дайте приблизительную оценку сложности по времени операций включения и исключения элемента.

3. Лабораторная работа «Представление графов в памяти ЭВМ»

3.1. Основные понятия и определения

Структура данных типа *граф* – это нелинейная многосвязная структура, обладающая следующими свойствами:

- на каждый элемент (узел, вершину) может быть произвольное количество ссылок;
- каждый элемент может иметь связь с любым количеством других элементов;
- каждая связка (ребро, дуга) может иметь направление и вес.

Граф, все связи которого ориентированные, называется *ориентированным графом* или *орграфом*. Граф, все связи которого неориентированные, называется *неориентированным графом*. Граф со связями обоих типов – *смешанным графом*.

Если ребрам графа соответствуют некоторые значения, то граф и ребра называются *взвешенными*. *Мультиграфом* называется граф, имеющий параллельные (соединяющие одни и те же вершины) ребра, в противном случае граф называется *простым*.

Путь в графе – это последовательность узлов, связанных ребрами. Путь от узла к самому себе называется *циклом*, а граф, содержащий такие пути – *циклическим*.

Два узла графа *смежны*, если между ними существует путь длины один. Узел называется *инцидентным* к ребру, если он является его вершиной.

3.2. Представление графа в последовательной памяти

В последовательной памяти граф обычно представляется матрицей смежности.

Матрицей смежности для n узлов называется квадратная матрица a порядка n. Элемент матрицы $a(i, j)$ равен 1, если узел j смежен с узлом i, и 0 – в противном случае.

Если граф неориентирован, то $a(i, j)=a(j, i)$, т.е. матрица симметрична относительно главной диагонали.

Матрица смежности мультиграфа состоит из нулей и целых чисел, взвешенного графа – из нулей и вещественных чисел, задающих вес каждого ребра.

В реальных задачах как с узлом, так и с ребром графа может быть связана информация в виде некоторого АТД. Это следует учитывать при рассмотрении вопроса сложности по памяти в целях выбора рационального способа представления графа в памяти ЭВМ.

3.3. Представление графа в связной памяти

В связной памяти граф обычно представляется в виде списка смежности. Такая структура данных состоит из:

– односвязного линейного списка, содержащего все элементы графа и обеспечивающего его целостность даже в случае отсутствия ребер;

– односвязных линейных списков по одному на каждый узел, содержащих указатели на узлы, связанные с данным.

Для реализации списка смежности потребуются два вида элементов: элемент данных и связующий элемент. Элемент данных представляет собой запись, включающую информационное поле, поле указателя на следующий элемент данных и поле указателя на список связей данного элемента. Связующий элемент содержит указатель на элемент данных и указатель на следующий элемент списка связей. Пример представления ориентированного графа списком смежности представлен не рисунке 3.1.

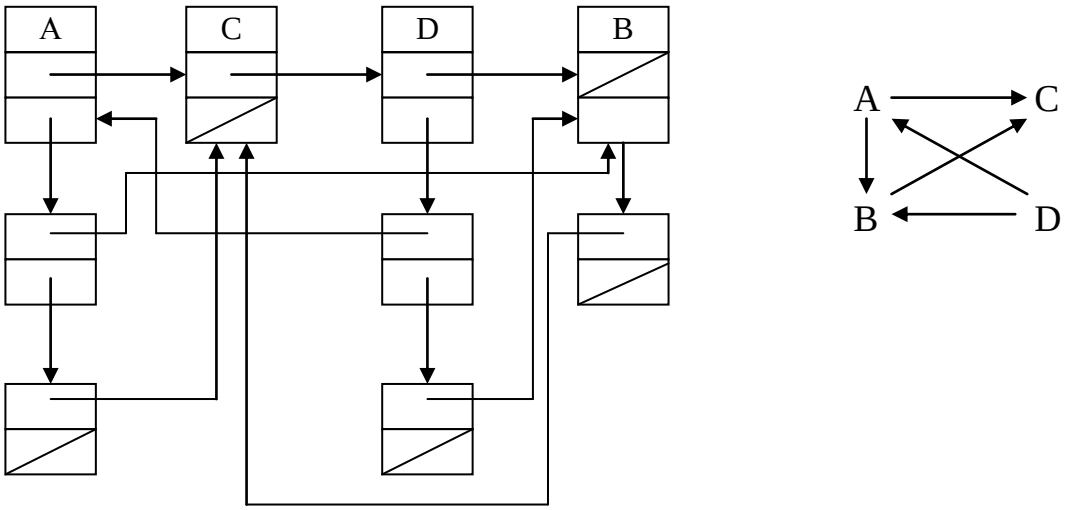


Рисунок 3.1 – Представление ориентированного графа в связной памяти

Доступ к самим элементам может обеспечиваться так же как и к односвязному списку – с помощью дескриптора, который может иметь различный набор указателей.

Применение связного представления графа в памяти снижает эффективность операций, связанных с получением информации о ребрах, так как в этом случае реализуется последовательный просмотр связующих элементов в поисках нужного. Поэтому использование списка смежности рекомендуется в двух случаях:

1. Часто осуществляются операции включения и исключения ребер, причем размер графа заранее непредсказуем.

2. Большой граф является слабо связным, то есть количество ребер невелико. В этом случае сложность доступа к ребрам в связной памяти незначительна. А реализация на основе матрицы смежности приводит к нецелесообразному расходованию памяти, так как большая часть ячеек не используется.

3.4. Задание на лабораторную работу

Дан ориентированный граф. Элементы графа содержат буквы латинского алфавита. Информация о связях представлена в файле в виде матрицы смежности.

Приложение должно выполнять следующие функции:

– построение графа в связной памяти на основе данных файла;

– включение в граф заданной пользователем вершины со ссылками на другие вершины, также задаваемые пользователем;

– исключение из графа вершины, заданной пользователем;

– поиск ребра связывающего заданные пользователем вершины (результат поиска – логическое значение).

Ход работы

1. Определить и описать необходимые структуры данных и список операторов.
2. Сформировать файл входных данных.
3. Разработать основной модуль приложения, реализующий логику диалога пользователя и соответствующие вызовы операторов.
4. Разработать алгоритмы, обеспечивающие реализацию операторов. Реализовать их в отдельном модуле.
5. Провести тестирование приложения.

3.5. Контрольные вопросы

1. Дайте основные определения структуры данных типа граф.
2. Приведите примеры объектов и процессов, которые эффективно описываются графами.
3. Каким образом граф может быть представлен в последовательной памяти?
4. Каким образом граф может быть представлен в связной памяти?

5. Назовите достоинства и недостатки последовательного и связного представления?

6. Какие рекомендации можно дать при выборе способа представления графа в памяти ЭВМ? Чем они обусловливаются?

7. Поясните принцип реализации операций включения и исключения вершин графа в связной памяти.

4. Лабораторная работа «Бинарное поисковое дерево. Алгоритмы обхода деревьев»

4.1. Основные понятия и определения

В силу рекурсивной природы дерева и удобства использования рекурсивных алгоритмов на нем дадим следующее определение.

Дерево – конечное непустое множество T, такое, что выполняются следующие условия:

1. Имеется один специально обозначенный узел, называемый корнем данного дерева.

2. Остальные узлы (исключая корень) содержатся в $m \geq 0$ попарно не пересекающихся множествах T_1, T_2, …, T_m, каждое из которых в свою очередь является деревом.

Деревья T_1, T_2, …, T_m называются поддеревьями данного корня. Если подмножества T_1, T_2, …, T_m упорядочены, то дерево называют *упорядоченным*. Конечное множество непересекающихся деревьев называется *лесом*. Между узлами дерева действует отношение «родитель-порожденный». Количество подмножеств (непосредственных потомков) для данного узла называется *степенью узла*. Если такое количество равно нулю, то узел является *листом*. Максимальная степень узла в дереве – *степень дерева*. *Уровень узла* – длина пути от корня до рассматриваемого узла (количество посещений узлов, включая рассматриваемый). Максимальный уровень дерева – *высота дерева*.

Бинарным называется дерево со степенью два.

Поисковым называется бинарное дерево, в котором для каждого узла выполняется условие: значения ключей всех узлов левого поддерева меньше, а значения ключей всех узлов правого поддерева больше, чем значение ключа данного узла.

Существует несколько способов представления дерева в связной памяти (также возможно представление и в последовательной памяти). Наиболее распространенным является «стандартный», когда каждый узел содержит в себе

указатели, ссылающиеся на непосредственных потомков. Пример представления дерева стандартным способом приведен на рисунке 4.1.

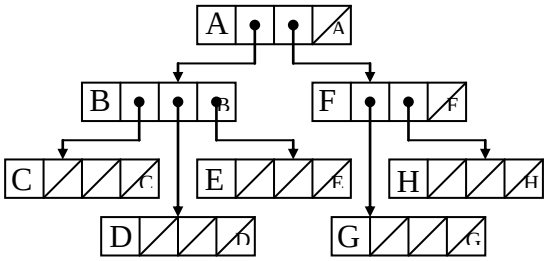


Рисунок 4.1 – Представление дерева в связной памяти стандартным способом

Очевидным недостатком такого представления является значительное количество неиспользованной памяти, выделенной под указатели. Однако, стандартный способ обеспечивает наивысшую по времени эффективность реализации подавляющей части алгоритмов.

В случае бинарного дерева и стандартного способа представления структура данных узла представляет собой запись, содержащую информационное поле и два поля указателей на левое и правое поддеревья. Тип данных «дерево» обычно совпадает с типом указателя на узел.

Для деревьев обычно выделяют следующие типовые операции:

– поиск узла (в случае непоискового дерева осуществляется последовательным перебором узлов в определенном порядке);

– включение узла (для поисковых деревьев – поиск включением);

– исключение узла (для поисковых деревьев – поиск исключением);

– обход дерева в определенном порядке (нисходящий, восходящий и смешанный).

4.2. Алгоритмы поиска, включения и исключения в бинарном поисковом дереве

Основное достоинство бинарного поискового дерева заключается в высокой эффективности поиска по нему (а следовательно включения и исключения узлов). В отличие от последовательного поиска, где один шаг алгоритма приводит к уменьшению размера списка рассматриваемых элементов на единицу, в случае бинарного поискового дерева в лучшем случае возможно уменьшение списка вдвое. Это становится возможным благодаря свойству упорядоченности узлов дерева по ключам.

Алгоритмы поиска, исключения и включения могут быть реализованы как в виде рекурсивных, так и виде итеративных процедур. В данных методических указаниях приведены рекурсивные алгоритмы, однако, при выполнении лабораторной работы допускается использовать и нерекурсивные реализации.

Поиск

Приведем словесное описание алгоритма поиска. На вход алгоритма подается указатель на дерево и аргумент поиска, на выходе – указатель на искомый элемент. Если переданный указатель нулевой, то алгоритм заканчивается неудачно. Если дерево не пустое, значение аргумента поиска сравнивается с ключом корня дерева. Если аргумент поиска меньше, тот же алгоритм рекурсивно выполняется для левого поддерева, иначе, если аргумент поиска больше, тот же алгоритм рекурсивно выполняется для правого поддерева, иначе текущий узел является искомым, алгоритм заканчивается удачно.

Включение

Алгоритм включения в бинарное поисковое дерево базируется на алгоритме поиска, так как вследствие упорядоченности по ключам узел должен быть вставлен в соответствующую ему позицию. Вставка осуществляется в том случае, если элемента с таким же ключом нет в дереве, следовательно, случай неудачного поиска является основанием для вставки.

На вход алгоритма подается указатель на дерево и включаемое информационное поле (в простейшем случае – ключ), на выход может передаваться результат вставки (логическое значение). Если переданный указатель нулевой, то алгоритм заканчивается удачно – производится вставка. Если дерево не пустое, значение вставляемого ключа сравнивается с ключом корня дерева. Если вставляемый ключ меньше, тот же алгоритм рекурсивно выполняется для левого поддерева, иначе, если вставляемый ключ больше, тот же алгоритм рекурсивно выполняется для правого поддерева, иначе такой ключ уже имеется в дереве, алгоритм заканчивается неудачно.

Действие «производится вставка» заключается обычно в выделении памяти под новый элемент, инициализации его полей и прикреплении к дереву путем модификации указателя, переданного в данный вызов процедуры, т.е. указателя родительского узла.

Очевидно, что операция поиска вставкой может использоваться при построении дерева на основе набора случайных данных. Поисковые деревья, построенные таким образом называются *случайными*. Также существуют способы построения деревьев, которые придают структуре дерева определенные свойства, обеспечивающие снижение его дерева и, как следствие, повышение эффективности операций поиска, включения и исключения. К таким деревьям относятся оптимальные и сбалансированные.

Исключение

Алгоритм исключения также строится на основе алгоритма поиска, так как для удаления элемента необходимо найти указатель на него.

На вход алгоритма подается указатель на дерево и ключ исключаемого узла (или все информационное поле), на выход может передаваться результат исключения (логическое значение). Если переданный указатель нулевой, то алгоритм заканчивается неудачно – такого узла нет. Если дерево не пустое, значение удаляемого ключа сравнивается с ключом корня дерева. Если удаляемый ключ меньше, тот же алгоритм рекурсивно выполняется для левого поддерева, иначе, если удаляемый ключ больше, тот же алгоритм рекурсивно выполняется

для правого поддерева, иначе удаляемый узел найден, производится исключение, алгоритм заканчивается удачно.

Действие «производится исключение» является тривиальным в трех случаях:

1. Исключаемый узел имеет только левого потомка.
2. Исключаемый узел имеет только правого потомка.
3. Исключаемый узел является листом.

Рассмотрим принцип исключения в первом случае. Во-первых, модифицируется указатель родителя исключаемого узла (параметр, переданный в процедуру): он ссылается на единственного левого потомка исключаемого узла. То есть исключаемый узел логически удаляется из дерева. Во-вторых, производится физическое удаление, то есть освобождение памяти, занимаемой исключаемым узлом.

Алгоритм реализации исключения во втором случае получается путем зеркального отражения указателей: левые на правые и наоборот. При этом можно объединить обработку третьего случая с любым из первых двух, так как при модификации родительский указатель в любом случае примет нулевое значение, что и требуется при удалении листа.

Более сложной является обработка ситуации, когда у исключаемого узла имеются оба потомка. В этом случае необходимо заменить данные исключаемого узла данными самого левого узла из правого поддерева или данными самого правого узла из левого поддерева. Эти узлы обладают следующим свойством (на примере правого поддерева): ключ самого левого узла меньше всех ключей в этом поддереве, но больше всех ключей левого поддерева удаляемого узла. Следовательно, такой крайний ключ (информационное поле) может занять место исключаемого узла, и при этом свойство поискового дерева не нарушится.

Таким образом, во-первых, необходимо перейти, например, к корню правого поддерева исключаемого узла, спуститься по левой ветви до узла, не имеющего левого потомка и скопировать его данные в исключаемый узел. Са-

мый левый узел в правом поддереве по определению либо является листом, либо имеет только правого потомка, что является тривиальным случаем, описанным выше. Поэтому для его исключения достаточно рекурсивно вызвать процедуру удаления, передав в нее указатель на данный узел и его же ключ.

4.3. Алгоритмы обхода деревьев

При решении практических задач часто возникает необходимость перебора всех элементов дерева. Причем в случае непоисковых деревьев такой перебор необходим в том числе для организации поиска. Вследствие нелинейной структуры дерева перебор его элементов не является тривиальной задачей, как, например, в случае линейного списка. Поэтому существует несколько порядков обхода, каждый из которых часто используется для решения различных алгоритмических задач.

Обход заключается в посещении всех узлов дерева в определенном порядке строго по одному разу. Под посещением понимаются некоторые действия, которые необходимо выполнить с узлом, в соответствии с требованиями задачи.

Алгоритмы обхода могут быть реализованы как в виде рекурсивных, так и виде итеративных процедур. В целях краткости в данных методических указаниях приведены все рекурсивные алгоритмы, и один нерекурсивный, реализация которого требуется при построении одной из процедур, описанных в задании на лабораторную работу. Необходимо отметить, что итеративные алгоритмы обходов основаны на явном использовании стека (в рекурсивных неявно используется системный стек).

Ниже приведены алгоритмы обходов бинарных деревьев. В случае сильно ветвящихся деревьев используется тот же подход, изменяется лишь количество рекурсивных вызовов в соответствии со степенью дерева.

Нисходящий обход

Схематично нисходящий обход можно представить следующим образом: Корень – Левое поддерево – Правое поддерево. То есть первым посещается корень, затем таким же образом посещаются узлы левого поддерева, затем таким же образом – узлы правого поддерева.

Общий вид рекурсивной процедуры нисходящего обхода (на вход передается указатель на корень дерева):

1. Если указатель не нулевой, то перейти к следующему пункту, иначе – выход.
2. Обработать текущий узел (корень).
3. Вызвать процедуру нисходящего обхода для левого поддерева.
4. Вызвать процедуру нисходящего обхода для правого поддерева.

Общий вид итеративной процедуры нисходящего обхода:

1. В качестве очередного узла выбрать корень дерева.
2. Произвести обработку очередного узла в соответствии с требованиями задачи.
3. Выбор следующего узла:

3.а) Если очередной узел имеет обоих потомков, то в качестве нового очередного узла выбрать левого потомка, а правого (указатель на него) занести в стек; перейти к пункту 2.

3.б) Если очередной узел является листом, то выбрать в качестве нового очередного узел из стека, если он не пуст, и перейти к пункту 2; если же стек пуст, то это означает, что обход всего дерева окончен, перейти к пункту 4.

3.в) Если очередной узел имеет только одного потомка, то выбрать его в качестве очередного узла; перейти к пункту 2.

4. Конец алгоритма.

Необходимо отметить, что, несмотря на то, что в словесном описании явно используются метки, реализация алгоритма возможна и должна осуществляться без них путем грамотного использования циклической и условных конструкций.

Восходящий обход

Схематично восходящий обход можно представить следующим образом: Левое поддерево – Правое поддерево – Корень.

Общий вид рекурсивной процедуры восходящего обхода:

1. Если указатель не нулевой, то перейти к следующему пункту, иначе – выход.
2. Вызвать процедуру восходящего обхода для левого поддерева.
3. Вызвать процедуру восходящего обхода для правого поддерева.
4. Обработать текущий узел (корень).

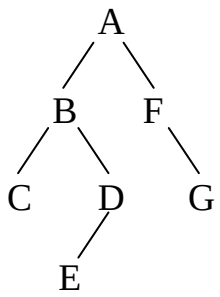
Смешанный обход

Схематично смешанный обход можно представить следующим образом: Левое поддерево – Корень – Правое поддерево.

Общий вид рекурсивной процедуры смешанного обхода:

1. Если указатель не нулевой, то перейти к следующему пункту, иначе – выход.
2. Вызвать процедуру смешанного обхода для левого поддерева.
3. Обработать текущий узел (корень).
4. Вызвать процедуру смешанного обхода для правого поддерева.

Примеры реализации порядков обхода приведены на рисунке 4.2.



Нисходящий обход: A – B – C – D – E – F – G

Восходящий обход: C – E – D – B – G – F – A

Смешанный обход: C – B – E – D – A – F – G

Рисунок 4.2 – Иллюстрация порядков обхода бинарного дерева

4.4. Задание на лабораторную работу

Дан текстовый файл с последовательностью неповторяющихся целых чисел.

Приложение должно выполнять следующие функции:

− построение в связной памяти бинарного поискового дерева по данным из файла на основе операции поиска вставкой;

− поиск вершин, у которых количество потомков в левом поддереве отличается от количества потомков в правом поддереве на единицу, на основе рекурсивного алгоритма восходящего обхода;

− нахождение высоты дерева на основе итеративного алгоритма нисходящего обхода;

− поиск *k*-го в порядке слева направо листа дерева (*k* задается пользователем) на основе рекурсивного алгоритма смешанного обхода;

− удаление вершины по ключу, заданному пользователем, на основе операции поиска исключением.

Ход работы

1. Описать структуру данных и операторы, необходимые для реализации указанных функций.

2. Подготовить файл входных данных.

3. Разработать интерфейсную часть приложения, реализующую вызовы соответствующих операторов.

4. Разработать детальные алгоритмы операторов, реализовать соответствующие процедуры.

5. Провести тестирование приложения.

4.5. Контрольные вопросы

1. Дайте основные определения структуры данных типа дерево.

2. Какими способами дерево может представляться в связной памяти? Назовите их достоинства и недостатки.

3. Что понимается под бинарным поисковым деревом? Каким образом его свойства обеспечивают повышение эффективности операций поиска, включения и исключения?

4. Опишите принципиальную схему реализации операций поиска, вставки и исключения в бинарном поисковом дереве.

5. Каковы отличия организации процедуры поиска в поисковых и непоисковых деревьях?

6. В чем заключаются и чем обусловливаются недостатки случайных поисковых деревьев по сравнению с оптимальными и сбалансированными?

7. Опишите принципы реализации рекурсивных алгоритмов обхода.

8. Опишите принцип реализации итеративного алгоритма нисходящего обхода.

9. Опишите принцип реализации итеративного алгоритма восходящего обхода.

10. Опишите принцип реализации итеративного алгоритма смешанного обхода.

5. Лабораторная работа «Полностью сбалансированное и оптимальное деревья поиска»

5.1. Полностью сбалансированное поисковое дерево

Бинарные поисковые деревья являются эффективным по времени инструментом организации доступа к данным. Очевидно, что сложность поиска будет иметь порядок высоты дерева. Поэтому случайные деревья, полученные путем многократного повторения операции поиска включением на имеющемся наборе данных, не всегда обеспечивают ожидаемую эффективность. Худшим случаем является построение такого дерева на изначально упорядоченном наборе данных, когда поисковое дерево вырождается в линейный односвязный список со сложностью операций O(N).

Одним из способов повышения эффективности операций на деревьях является введение ограничений на их структуру с целью уменьшения высоты. Наиболее эффективным по времени при равной вероятности появления ключей является полностью сбалансированное дерево.

Полностью сбалансированным называется бинарное поисковое дерево, для всех узлов которого выполняется условие сбалансированности: количество узлов в левом поддереве отличается от количества узлов в правом поддереве не более чем на единицу.

При выполнении условия сбалансированности высота дерева не превышает $\log_2(N+1)$, где N – количество узлов дерева. Следовательно, сложность поиска $O(\log_2 N)$.

Для построения полностью сбалансированного дерева необходима отсортированная в порядке возрастания ключей последовательность элементов данных с индексами от 1 до N. Алгоритм представляет собой рекурсивную процедуру, на вход которой подаются:

– индексы $i=l-1$ и $j=r$, где l и r – соответственно левая и правая границы участка массива, на котором должен работать данный вызов (уменьшение

левой границы на единицу обусловлено спецификой алгоритма, обеспечивающей одновременное получение цены поиска по дереву);

− указатель родительского (по отношению к добавляемому на данном вызове) узла.

Алгоритм построения полностью сбалансированного дерева:

1. Если размер входного набора данных менее единицы ($i \geq j$), то текущий узел пустой, завершить алгоритм, иначе перейти к следующему пункту.

2. В качестве корня выбрать средний элемент с индексом $k=((i+j)$ div $2)+1$, создать узел по переданному в процедуру указателю, проинициализировать поля созданного узла.

3. Выполнить тот же алгоритм для левой части списка, передав индексы i и k-1 и левый указатель созданного корня.

4. Выполнить тот же алгоритм для правой части списка передав индексы k и j и правый указатель созданного корня.

Полностью сбалансированным деревьям поиска свойственны следующие недостатки:

1. Необходима начальная упорядоченность элементов. Затраты на сортировку могут свести к нулю эффект получаемый за счет снижения высоты дерева, если поиск осуществляется не так часто.

2. При вставке или исключении элемента может нарушаться условие сбалансированности. Восстановление баланса возможно только за счет полной перестройки дерева, что делает реализацию операций включения и исключения неэффективной по времени – O(N).

5.2. Оптимальное поисковое дерево: определение и цена поиска

Оптимальным называется поисковое дерево, обеспечивающее минимальную цену поиска.

С этой точки зрения полностью сбалансированное дерево является оптимальным только при равной вероятности появления ключей, что на практике не всегда возможно.

Если имеются сведения о вероятности поиска каждого ключа дерева, то имеется возможность изменить его структуру так, чтобы при сохранении свойств поискового дерева расположение ключей обеспечивало минимальную цену поиска. Это означает, что ключи должны быть удалены от вершины дерева по критерию убывания вероятности их поиска.

Также имеет смысл рассматривать случаи неудачного поиска, так как он заканчивается всегда в листьях и может вносить существенный вклад в общую цену поиска.

Пусть имеется дерево (рисунок 5.1), построенное некоторым алгоритмом на основе упорядоченного массива с индексами от 1 до *N*. Всем узлам присвоены номера в соответствии с их индексами в исходном массиве. Также показаны абстрактные узлы (обозначены квадратами), представляющие собой возможные случаи окончания неудачного поиска.

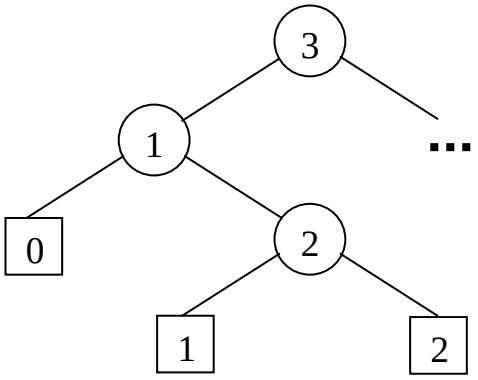


Рисунок 5.1 – Фрагмент дерева

Для формализации понятия цены поиска введем следующие обозначения:

– p_i – частота (вероятность), с которой аргумент поиска примет значение, равное ключу i-го узла, где i=1...N;

– q_0 – частота (вероятность), с которой аргумент поиска окажется меньше значения 1-го узла (частота неудачного поиска, завершающегося в нулевом абстрактном узле);

– q_i – частота (вероятность), с которой аргумент поиска окажется больше значения i-го и меньше значения (i+1)-го внутреннего узла (i-й абстрактный узел);

– q_N – частота (вероятность), с которой аргумент поиска окажется больше значения N-го узла (N-й абстрактный узел).

Если заданы p_i (i=1..N) и q_j (j=0..N), то можно найти математическое ожидание числа сравнений ключей (в соответствии с допущениями асимптотического анализа принимаем, что при посещении узла выполняется одна операция проверки условия) или взвешенную среднюю длину пути при поиске по данному дереву:

$$C(1,N) = \sum_{i=1}^{N} p_i l_i + \sum_{j=0}^{N} q_j (l'_j - 1), \qquad (5.1)$$

где l_i и l'_i – уровни i-го узла и j-го абстрактного узла соответственно.

Вычитание единицы в правой сумме означает, что хотя неудачный поиск всегда заканчивается в одном из абстрактных узлов, аргумент поиска сравнивается только со значениями ключей узлов дерева.

Таким образом, задача построения оптимального дерева состоит в определении, какая из всех возможных структур бинарного поискового дерева для заданных вероятностей минимизирует математическое ожидание числа сравнений ключей. Это значение будем называть *ценой поиска*.

При таком определении оптимального дерева при подсчете цены произвольного дерева нет необходимости требовать, чтобы общая сумма вероятностей обязательно равнялась единице (т.е. не обязательно для частоты использовать нормализованное значение). Благодаря этому появляется возможность говорить о дереве с минимальной ценой для заданной последовательности «весов» узлов.

Оптимальному дереву свойственны те же недостатки, что и полностью сбалансированному. Причем приведенный ниже алгоритм его построения имеет сложность по времени O(N^3).

5.3. Алгоритм построения оптимального поискового дерева

Метод поиска оптимальных деревьев на основе полного перебора структур и прямого вычисления цены неэффективен – по мере увеличения числа внутренних узлов экспоненциально увеличивается число соответствующих им бинарных деревьев.

Алгоритм построения оптимального дерева строится на утверждении, что если дерево является поддеревом оптимального дерева, затраты на поиск по нему будут минимальными.

Обозначим через $t(i, j)$ поддерево дерева $t(1, N)$, имеющего веса $(p_1,\ldots, p_{i+1},\ldots, p_j,\ldots, p_N; q_0,\ldots, q_i,\ldots, q_j,\ldots, q_N)$. $t(i, j)$ содержит внутренние узлы $i+1\ldots j$ и внешние узлы $i\ldots j$.

Из формулы 5.1 можно получить вклад поддерева $t(i, j)$ в общую цену дерева (вывод в целях краткости не приводится):

$$C'(i,j) = (l_m - 1)\left\{\sum_{k=i+1}^{j} p_k + \sum_{k=i}^{j} q_k\right\} + \left\{\sum_{k=i+1}^{j} p_k L_k + \sum_{k=i}^{j} q_k (L'_k - 1)\right\}, \qquad (5.2)$$

где L_k и L'_k – уровни k-го узла и k-го абстрактного узла в поддереве $t(i, j)$ соответственно;

l_m – уровень корня поддерева $t(i, j)$ в дереве.

Для узлов, образующих это поддерево, первое слагаемое правой части формулы является константой. Оно выражает статический вклад суммы весов узлов поддерева $t(i, j)$, взвешенный уровнем дерева $t(1, N)$, на котором находится его корень. Обозначим сумму весов узлов:

$$W(i,j) = \sum_{k=i+1}^{j} p_k + \sum_{k=i}^{j} q_k \ . \qquad (5.3)$$

Второе слагаемое выражает затраты на поиск по поддереву, которые будут зависеть от его структуры. Оно эквивалентно правой части 5.1 и является ценой поиска по поддереву $t(i, j)$, как самостоятельному дереву:

$$C(i, j) = \sum_{k=i+1}^{j} p_k L_k + \sum_{k=i}^{j} q_k (L'_k - 1). \qquad (5.4)$$

Действительно, если рассматривать поддерево $t(i, j)$ как самостоятельное ($l_m = 1$), то левое слагаемое обращается в ноль.

Теперь рассмотрим дерево $t(i, j)$ как состоящее из корня с номером k и двух поддеревьев: левого $t(i, k\text{-}1)$, состоящего из узлов $i{+}1 \ldots k\text{-}1$, и правого левого $t(k, j)$, состоящего из узлов $k{+}1 \ldots j$. Тогда его цену можно представить в виде

$$C(i, j) = p_k + C'(i, k-1) + C'(k, j), \qquad (5.5)$$

где p_k – вклад пути поиска длины 1, заканчивающегося в корне k.

Представив вклады поддеревьев $C'(i, k-1)$ и $C'(k, j)$ в соответствии с формулой 5.2 ($l_m = 2$, так как их корни находятся на втором уровне дерева $t(i, j)$) и обозначениями, введенными в 5.3 и 5.4, получим:

$$C(i, j) = p_k + \left(\frac{2}{2} - 1\right) \cdot W(i, k-1) + C(i, k-1) + \left(\frac{2}{2} - 1\right) \cdot W(k, j) + C(k, j).$$

Сгруппируем постоянные и переменные члены выражения:

$$C(i, j) = \left(p_k + W(i, k-1) + W(k, j)\right) + C(i, k-1) + C(k, j).$$

Отметим, что выражение в первой фигурной скобке есть ни что иное, как сумма весов узлов дерева $i{+}1 \ldots j$. Отсюда получим окончательное рекуррентное выражение для цены произвольного дерева $t(i, j)$:

$$\begin{cases} C(i, j) = W(i, j) + C(i, k-1) + C(k, j) & \text{при } i < j, \\ C(i, j) = 0 & \text{при } i = j; \end{cases} \qquad (5.6)$$

Базовый случай представляет собой пустое поддерево $t(i, j)$, где $i{=}j$.

Теперь потребуем, чтобы дерево $t(i, j)$ было оптимальным. Если оно является таковым, то и оба его поддерева также должны быть оптимальными деревьями. Из этого следует, что минимально возможная цена дерева $t(i, j)$ с корнем k равна

$$\begin{cases} C(i,j) = W(i,j) + \min\limits_{i<k\le j} \{C(i,k-1) + C(k,j)\} \text{ при } i < j, \\ C(i,j) = 0 \quad \text{при } i = j; \end{cases} \tag{5.7}$$

На основе полученных зависимостей можно сформулировать общий вид алгоритма построения полностью сбалансированного дерева. На вход алгоритма передается массив элементов от 1 до N, отсортированный в порядке возрастания ключей, и массивы весов $p[0..N]$, где p_0=0, и $q[0..N]$. Построение осуществляется в два этапа.

1. Формирование логической структуры дерева.

На данном этапе находятся корни оптимальных поддеревьев, начиная с поддеревьев, состоящих из одного узла, затем из двух узлов и т.д. Ниже дана итеративная реализация подготовительного этапа, но возможна и рекурсивная реализация пунктов 1.2 и 1.3.

1.1. Осуществляется начальная инициализация матриц $W(i, j)$, $C(i, j)$ и $R(i, j)$ нулевыми значениями (i=0...N и j=0...N).

1.2. По формуле 5.6 строится матрица весов $W(i, j)$.

1.3. Последовательно, сначала для всех i–j=1, затем для всех i–j=2 и т.д., для всех i–j=N по формуле 5.5 вычисляются значения матрицы $C(i, j)$. При этом значения номеров корней, на которых был достигнут минимум, фиксируются в матрице $R(i, j)$.

2. Физическое построение дерева.

На основе логической структуры, сформированной в матрице $R(i, j)$, строится дерево $t(i, j)$. В процедуру при первом вызове передается указатель на будущий корень дерева и индексы участка массива элементов данных, на котором строится дерево i и j (при первом вызове i=0, j=N).

1. Если R(i, j)=0, то поддерево пустое, конец алгоритма, иначе перейти к следующему пункту.

2. В качестве корня выбрать элемент m=$R(i, j)$. Создать узел по переданному указателю, проинициализировать поля.

3. Вызовом того же алгоритма построить левое поддерево $t(i, m$-1).

4. Вызовом того же алгоритма построить правое поддерево $t(m, j)$.

Имеется возможность улучшить приведенный алгоритм построения оптимально дерева. Для этого можно использовать в процессе нахождения значения k, минимизирующего второе слагаемое формулы 5.5, информацию о структуре оптимальных поддеревьев, полученную на предыдущих шагах. То есть нет необходимости проверять все k из диапазона $i \le k \le j$, достаточно проверить только область $R(i, j-1) \le k \le R(i+1, j)$. Модифицированный таким образом алгоритм будет иметь сложность $O(N^2)$.

5.4. Задание на лабораторную работу

Деревья строятся по данным, приведенным в таблице 5.1.

Таблица 5.1

i	0	1	2	3	4	5	6	7	8	9	10
$key[i]$	-	1	3	5	7	9	11	13	15	17	19
$p[i]$	0	100	350	700	280	50	300	200	150	30	400
$q[i]$	900	500	600	120	40	50	140	230	60	300	110

Приложение должно выполнять следующие функции:

– построение оптимального и полностью сбалансированного деревьев в связной памяти;

– отображение построенных деревьев на экране, допускается отображение в текстовом режиме с помощью смешанного обхода, направленного справа налево;

– вычисление цены поиска по полностью сбалансированному и оптимальному деревьям с учетом весов узлов.

Примечание: Цену полностью сбалансированного дерева можно определить в процессе построения, если в реализующей его процедуре «поднимать» вверх по рекурсии значения в соответствии с рекуррентной зависимостью 5.6.

Ход работы

1. Описать структуры данных и операторы, необходимые для реализации указанных функций.

2. Разработать интерфейсную часть приложения, реализующую вызовы соответствующих операторов.

3. Разработать детальные алгоритмы операторов, реализовать соответствующие процедуры.

4. Провести тестирование приложения.

5.5. Контрольные вопросы

1. Чем может быть обусловлена необходимость применения полностью сбалансированного и оптимального деревьев поиска вместо случайных деревьев?

2. В чем заключается условие полной сбалансированности? Чего позволяет достичь его соблюдение?

3. Сформулируйте достоинства и недостатки полностью сбалансированного дерева.

4. Дайте определение оптимальному дереву, цене поиска.

5. Обоснуйте необходимость учета вероятностей неудачного поиска.

6. На каком свойстве оптимального дерева базируется алгоритм его построения? Как формально описывается процесс минимизации?

7. Поясните принцип реализации процедур формирования матриц весов, цен и корней.

8. Поясните принцип реализации процедуры построения дерева по матрице корней.

6. Лабораторная работа «Сортировка во внешней памяти многопутевым сбалансированным слиянием»

6.1. Особенности задач сортировки во внешней памяти

В случае невозможности или нежелательности размещения всего объема сортируемых данных в оперативной памяти необходимо осуществлять сортировку на внешних носителях информации, таких как накопители на жестких магнитных дисках. В таком случае мы говорим, что данные представляют собой (последовательный) файл. А эффективность алгоритмов во внешней памяти определяется количеством операций доступа к ее блокам.

Наиболее эффективные алгоритмы сортировки, применяемые во внутренней памяти, базируются на возможности прямого доступа к элементу массива по его индексу. Если попытаться применить их для сортировки последовательностей, то это приведет не только к логическому усложнению алгоритмов, но и к значительной неэффективности вследствие нерационально большого числа обращений к блокам из последовательных процедур доступа.

Поэтому во внешней памяти применяются методы сортировки, обеспечивающие последовательную обработку считываемых записей в процесе сортировки. Таким методов является сортировка с помощью слияния.

6.2. Отрезки и слияние отрезков

Выберем как можно более длинные участки, представляющие собой отсортированные последовательности в конечной числовой последовательности $a_1, a_2, \ldots, a_n$. Эти участки будем называть *отрезками* или *сериями*. Они должны удовлетворять условию $(a_{i-1} > a_i) \,\&\, (\forall k : i \le k < j : a_k > a_{k+1}) \,\&\, (a_j > a_{j+1})$.

Отсортированная последовательность состоит из одного отрезка длиной n, а последовательность, отсортированная в обратном порядке, состоит из n отрезков длиной 1.

Слияние – объединение двух или более упорядоченных отрезков в один упорядоченный более длинный отрезок. Рассмотрим простой случай, когда сливаются два отрезка, входящие в различные файлы (рисунок 6.1). При этом производят сравнение начальных элементов каждого из отрезков: меньший элемент выводят, а на его место становится следующий элемент из этой последовательности. Эту операцию повторяют до тех пор, пока один из отрезков не закончится. Оставшиеся элементы другого отрезка выводят, не изменяя порядка.

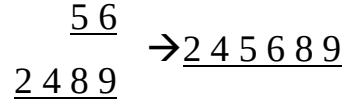


Рисунок 6.1 – Пример слияния двух отрезков

В случае наличия нескольких отрезков в последовательности необходимо обнаружение конца отрезка. Для этого могут применяться несколько способов:

1. Ключ выведенной записи сравнивают с ключом следующей записи. Если значение ключа новой записи меньше то отрезок закончен.

2. В качестве метки, указывающей на конец отрезка, вставляют специальный ключ, значение которого не входит в диапазон значений реальных ключей.

3. Длина отрезков устанавливается фиксированной.

Последний подход применяется в алгоритме прямого слияния, где начальная последовательность рассматривается как набор отрезков единичной длины. На первом проходе отрезки попарно сливаются в отрезки, состоящие из двух элементов. На втором проходе полученные отрезки сливаются в отрезки, состоящие из четырех элементов, и т.д. до получения единственного отрезка, то есть отсортированной последовательности.

В случае прямого слияния размер сливаемых на k-м проходе подпоследовательностей меньше или равен 2^k и не зависит от существования более длин-

ных, уже упорядоченных подпоследовательностей. Слияние более длинных отрезков позволяет повысить эффективность алгоритма.

6.3. Естественное слияние

Сортировка, при которой всегда сливаются упорядоченные подпоследовательности максимальной длины, называется *естественным слиянием*. Действия по однократной обработке всего множества данных называются *фазой*. Наименьший подпроцесс, повторение которого составляет процесс сортировки, называется *проходом* или *этапом*.

В случае естественного слияния обычно говорят о несбалансированной, двухфазной сортировке слиянием с тремя лентами (двумя путями слияния). Схема сортировки приведена на рисунке 6.2.

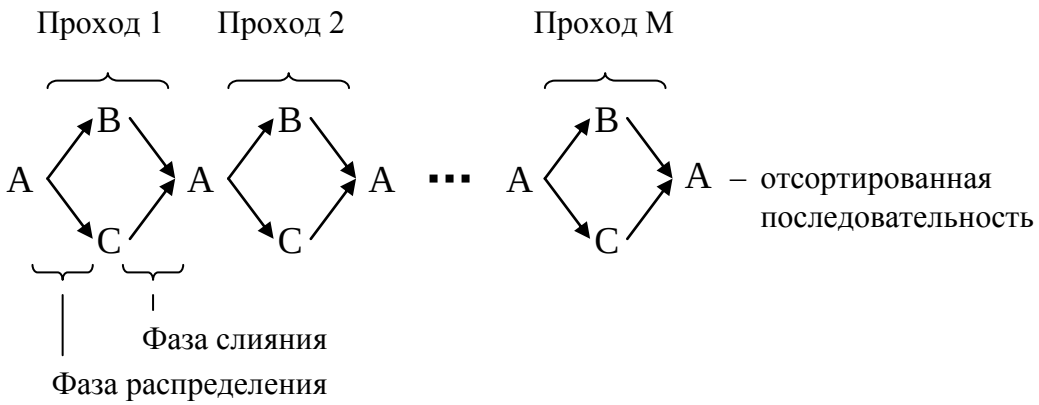


Рисунок 6.2 – Схема двухфазной сортировки слиянием

Переменная *A* представляет собой исходную последовательность элементов. *B* и *C* – вспомогательяые переменные-последовательности. Каждый проход состоит из фазы распределения и фазы слияния.

В первой фазе осуществляется распределение отрезков из последовательности *A* поочередно в последовательности *B* и *C* до завершения *A*. На фазе слияния последовательности *B* и *C* являться входными. На каждой итерации из *B* и *C* сливается по одному отрезку и результирующий отрезок записывается в

A. Итерации повторяются до завершения одной из входных последовательностей. Оставшиеся отрезки второй последовательности копируются в *A* без изменения порядка. Пример сортировки естественным слиянием приведен на рисунке 6.3.

A= 17 31\05 59\13 41 43 67\11 23 29 47\03 07 71

Проход 1 – распределение

B= 17 31\13 41 43 67\03 07 71 C= 05 59\11 23 29 47

Проход 1 – слияние

A= 05 17 31 59\11 13 23 29 41 43 47 67\03 07 71

Проход 2 – распределение

B= 05 17 31 59\03 07 71 C= 11 13 23 29 41 43 47 67

Проход 2 – слияние

A= 05 11 13 17 23 29 31 41 43 47 59 67\03 07 71

Проход 3 – распределение

B= 05 11 13 17 23 29 31 41 43 47 59 67 C= 03 07 71

Проход 3 – слияние

A= 03 05 07 11 13 17 23 29 31 41 43 47 59 67 71

Рисунок 6.3 – Пример сортировки естественным слиянием

6.4. Многопутевое сбалансированное слияние

Затраты на любую последовательную сортировку пропорциональны числу требуемых проходов, так как по определению при каждом из проходов копируются все данные. Общее число проходов, необходимых в худшем случае (начальная последовательность упорядочена по убыванию ключей) для сортировки *n* элементов с помощью *N*-путевого слияния, равно $k = \lceil \log_N n \rceil$.

Очевидно, что для повышения эффективности сортировки слиянием необходимо повышать количество путей слияния. Однако, на практике не рекомендуется использовать более 5-6 путей. Это ограничение обусловлено повышением количества переключений между последовательности в процессе

сортировки, каждое из которых приводит к необходимости выполнения наиболее времяемких операций позиционирования головки и ожидания поворота диска до нужного сектора.

Таким образом, первым отличием многопутевого сбалансированного слияния является наличие более двух путей. Вторым отличием является ее «сбалансированность». Если рассмотреть процесс двухфазной сортировки, то можно заметить, что на фазе распределения не происходит перестановок элементов, то есть сортировки. Поэтому для сокращения количества операций копирования можно использовать однофазную схему (рисунок 6.4), где проход состоит из одной фазы, в которой одновременно со слиянием осуществляется и распределение. «Сбалансированность» заключается в равном количестве входных и выходных файлов на каждой фазе.

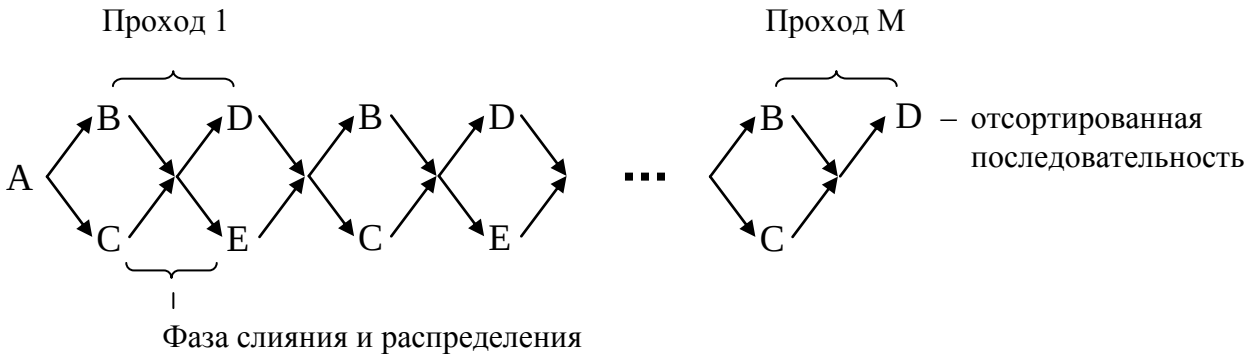


Рисунок 6.4 – Схема однофазного слияния

На подготовительном этапе осуществляется начальное распределение отрезков исходного файла *A* в *B* и *C*. В ходе 1-го прохода записи получаемого в результате слияния отрезка сразу же копируются в один из выходных файлов (осуществляется распределение): первый в *D*, второй в *E*, третий в *D* и т.д. до завершения *B* или *C*. Остаток второго входного файла без изменения копируется в очередной выходной. На втором проходе выходные файлы становятся входными и наоборот. Последним является проход, в результате которого будет получен лишь один отрезок, расположенный в первом выходном файле.

Однофазная схема позволяет почти вдвое сократить количество операций копирования. Очевидным ее недостатком является увеличение количества файлов, участвующих в слиянии, до $2N$ по сравнению с N+1 при двухфазном, где N – количество путей слияния.

На основании изложенного выше можно сформулировать основные особенности построения алгоритма сбалансированного многопутевого слияния:

1. Циклически осуществляется обращение к нескольким файлам. Целесообразно использовать массив файлов, что позволит получать к ним доступ с помощью индексной переменной.

2. В рамках прохода слияние производится одновременно из некоторого количества файлов (более 2). Факт завершения одного из файлов необходимо учитывать (путем уменьшения переменной количества активных файлов) для завершения прохода при опустошении всех входных файлов.

3. Проход реализуется в виде цикла слияний отрезков. Факт завершения одного из отрезков нужно учитывать (путем уменьшения переменной количества активных отрезков) для завершения цикла слияния отрезков.

4. Слияние должно осуществляться только из активных файлов и только из тех из них, в которых есть активные отрезки. Для этого необходимо поддерживать массив индексов активных файлов, в которых имеются активные отрезки.

5. После каждого прохода необходимо осуществлять переключение входных и выходных файлов. Чтобы избежать применения операций проверки условия при каждом обращении к файлу, необходимо использовать технику индексных карт. Индексная карта $t[1..2N]$, где N – количество путей слияния, – это массив индексов файлов. На первом проходе индексная карта инициализируется по правилу $t[i]=i$, то есть для двух путей слияния имеем (1, 2, 3, 4). Осуществляется слияние из файлов $t[1]=1$ и $t[2]=2$ в файлы $t[3]=3$ и $t[4]=4$. Перед вторым проходом индексы в карте «переключаются», и она принимает вид (3, 4, 1, 2). Слияние осуществляется из файлов $t[1]=3$ и $t[2]=4$ в файлы $t[3]=1$ и $t[4]=2$, что и требуется на втором проходе. Однако, индексы, по которым ведет-

ся обращение к индексной карте остаются неизменными, что позволяет обеспечить независимость основной логики сортировки от номера прохода.

При описании алгоритма сбалансированного многопутевого слияния используются следующие обозначения:

*f*0 – исходный файл;

f[1..2*N*] – массив файлов, на которых реализуется сортировка;

t[1..2*N*] – индексная карта доступа к файлам;

L – количество полученных в результате слияния за проход отрезков;

af – количество активных файлов;

ao – количество активных файлов с активными отрезками;

j – индекс очередного выходного файла;

m – индекс файла, из которого в процессе слияния был скопирован минимальный ключ;

ta[1..*N*] – массив индексов активных файлов, в которых имеются активные отрезки.

Алгоритм:

```
BMerge
   открыть для чтения f0
   открыть для записи f[1]..f[N]
   j ← 1
   L ← 0
   Repeat          //начальное распределение отрезков
      копировать отрезок из f0 в f[j]
      переключить j на следующий выходной файл
      L ← L +1
   Until конец f0
   выполнить начальную инициализацию индексной карты
   Repeat          // слияние из  f[t[1]].. f[t[N]] в f[t[N+1]].. f[t[2N]]
      af ← min(L, N)
      открыть для чтения f[t[1]].. f[t[af]]
      открыть для записи f[t[N+1]].. f[t[2N]]
      инициализировать ta[1].. ta[af] индексами из ta[]
```

```
L ← 0
j ← N+1
Repeat        //проход
   L ← L+1
   ao ← af
   Repeat     //слияние первых активных отрезков в f[t[j]]
      m ← индекс файла с минимальным ключом
      скопировать запись из f[ta[m]] в f[t[j]]
      if конец f[ta[m]] then
         модифицировать af, ao, ta[]
      else
         if конец отрезка в f[ta[m]] then
            модифицировать ao, ta[]
         end if
      end if
   Until ao=0
   переключить j на следующий выходной файл
Until af=0
переключить индексную карту
Until L=1
                  //отсортированная последовательность в f[t[1]]
End BMerge
```

6.5. Задание на лабораторную работу

Файл содержит случайный набор записей фиксированного размера (200 байт). Одно из полей записи содержит целочисленный ключ (ключи могут повторяться). Количество записей – 10000.

Приложение должно реализовывать следующие функции:

– сортировка файла сбалансированным многопутевым слиянием на количестве путей слияния, заданном пользователем;

– определение количества проходов и времени сортировки для заданного количества путей слияния;

− вывод в виде таблицы и графиков зависимости количества проходов и времени сортировки от количества путей слияния.

Ход работы

1. Детализировать составные части алгоритма сортировки сбалансированным многопутевым слиянием.

2. Подготовить файл входных данных.

3. Разработать приложение, реализующее сортировку.

4. Провести тестирование и отладку.

5. Реализовать функции определения количества проходов и времени сортировки, вывода отчетной информации.

6. Провести эксперименты на количестве путей слияния 2, 3, …, 10, построить соответствующие таблицы и графики.

6.6. Контрольные вопросы

1. Чем обусловлено применение особых методов для решения задач сортировки во внешней памяти? В чем заключается их специфика?

2. Дайте определения терминам отрезок, слияние. В чем заключается основная идея слияния?

3. Какой принцип положен в основу естественного слияния? Какой положительный эффект оно дает?

4. Поясните принцип двухфазного слияния.

5. Поясните принцип однофазного слияния.

6. Назовите преимущества и недостатки однофазной и двухфазной схем слияния.

7. Назовите преимущества сбалансированного многопутевого слияния по сравнению с двухфазным двухпутевым естественным слиянием. Чем они обусловливаются?

8. Поясните принцип реализации алгоритма сбалансированного многопутевого слияния.

9. Каким образом можно интерпретировать полученные в ходе выполнения лабораторной работы данные?

7. Лабораторная работа «Алгоритмы обхода графов»

7.1. Задачи обхода графов

Доступ к вершинам графа как в случае последовательного, так и в случае связного представления в памяти ЭВМ является тривиальной задачей, реализуемой последовательным просмотром. То же можно сказать и о переборе ребер. Однако при решении практических задач часто возникает необходимость посещения всех вершин графа в некотором порядке по путям, заданным его ребрами.

Обход заключается в посещении всех вершин графа в определенном порядке строго по одному разу. Под посещением понимаются некоторые действия, которые необходимо выполнить с вершиной, в соответствии с требованиями задачи.

7.2. Алгоритм обхода графа в глубину

При обходе в глубину посещается и помечается как посещенная первая вершина, а затем осуществляется проход (с посещением и пометкой вершин) вдоль ребер графа, пока не будет достигнут «тупик». Вершина неориентированного графа является тупиком, если уже посещены все примыкающие к ней вершины. В ориентированном графе тупиком также оказывается вершина, из которой нет исходящих ребер.

После попадания в тупик осуществляется возврат по пройденному пути пока не будет найдена вершина, у которой есть еще не посещенный сосед, и в этом направлении таким же образом осуществляется процесс прохождения до тупика. Обход оказывается завершенным, когда процесс вернется в отправную точку, а все примыкающие к ней вершины окажутся уже посещенными.

Иллюстрируя алгоритмы обхода, при выборе одной из двух вершин всегда будем выбирать вершину с меньшей (в числовом или лексикографическом

порядке) меткой. При реализации алгоритма выбор обычно определяться способом хранения ребер графа. Рассмотрим граф, приведенный на рисунке 7.1.

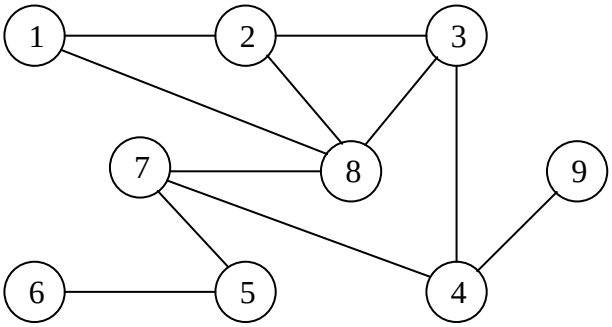


Рисунок 7.1 – Пример графа

Начав обход в глубину в вершине 1, посетим последовательно вершины 2, 3, 4, 7, 5 и 6 и достигнем тупика. Затем вернемся в вершину 7 и обнаружим, что вершина 8 осталась непосещенной. Перейдя в эту вершину, мы оказываемся в тупике и т.д. Возвращаемся назад в исходную вершину, и поскольку все соседние с ней вершины оказались посещенными, обход закончен:

1 нач. → 2 → 3 → 4 → 7 → 5 → 6 туп. → 5 → 7 → 8 туп. → 7 →4 → 9 туп. →
→ 4 → 3 → 2 → 1 нач.

В приведенной последовательности посещений вершин маркером отмечены пути возврата. Результирующая последовательность обхода:

1 → 2 → 3 → 4 → 7 → 5 → 6 → 8 → 9.

Ниже представлен общий вид алгоритма обхода в глубину. В рекурсивную процедуру передается начальная вершина обхода *v*, переменная *w* указывает на проверяемую на конкретном этапе вершину, смежную *v*:

```
DepthTraversal (v)
   Visit(v)          //посещение вершины
   Mark(v)           //пометка посещенной вершины
   for каждого ребра (v, w) do
      if вершина w непомечена then
         DepthTraversal(w)
      end if
   end for
End DepthTraversal
```

Этот рекурсивный алгоритм использует системный стек для отслеживания текущей вершины графа, что позволяет правильно осуществить возвращение, наткнувшись на тупик.

7.3. Алгоритм обхода графа по уровням (в ширину)

После посещения первого узла посещаются все соседние с ним вершины. На втором проходе посещаются все вершины «на расстоянии двух ребер» от начальной и т.д. В графе могут быть циклы, поэтому не исключено, что одну и ту же вершину можно соединить с начальной двумя различными путями. Вершина посещается впервые, когда алгоритм дойдет до нее по самому короткому пути. Чтобы предупредить повторное посещение, необходимо либо вести список посещенных вершин, либо завести в каждой вершине флаг посещения.

В основе обхода в ширину лежит структура данных типа очередь. На подготовительном этапе в очередь заносится корень дерева обхода, он посещается и помечается. На первом уровне корень удаляется из очереди, его соседние вершины просматриваются, непомеченные посещаются, помечаются и заносятся в очередь. После посещения всех соседних с корнем вершин на следующем уровне алгоритм возвращается к очереди. Так как узлы добавляются к концу очереди, ни одна из вершин, находящихся на расстоянии двух ребер от корня, не будет рассмотрена повторно, пока не будут обработаны и удалены из очереди все вершины на расстоянии одного ребра от корня.

Ниже представлен общий вид алгоритм обхода по уроням. В процедуру передается начальная вершина обхода v:

```
BreadthTraversal(v)
   InitQueue(Q)    //инициализация очереди
   Visit(v)        //посещение
   Mark(v)           //пометка
   Enqueue(Q, v)  //добавление в очередь
```

```
    while not Empty(Q) do
      Dequeue(Q, x)  //извлечение из очереди
      for каждого ребра (x, w) do
        if вершина w непомечена then
          Visit(w)
          Mark(w)
          Enqueue(Q, w)
        end if
      end for
    end while
  End BreadthTraversal
```

Например, рассмотрим процесс обхода приведенного в предыдущем пункте графа, начинающийся в вершине 7 (маркером выделены посещаемые вершины, в строке Q: … приведено состояние очереди на конец этапа):

Подготовительный этап

7

Q: 7

1 уровень ----------

7 из очереди, ее соседи: 4, 5, 8

Q: 4, 5, 8

2 уровень ----------

4 из очереди, ее соседи: 3, 9

5 из очереди, ее соседи: 6

8 из очереди, ее соседи: 1, 2

Q: 3, 9, 6, 1, 2

3 уровень ----------

3 из очереди, ее соседи: нет

9 из очереди, ее соседи: нет

6 из очереди, ее соседи: нет

1 из очереди, ее соседи: нет

2 из очереди, ее соседи: нет

Q: пуста

Результирующая последовательность обхода:

7→4→5→8→3→9→6→1→2.

7.4. Задание на лабораторную работу

Дан неориентированный граф. Элементы графа содержат буквы латинского алфавита. Информация о связях представлена в файле в виде матрицы смежности.

Приложение должно выполнять следующие функции:

− построение графа в последовательной памяти на основе данных файла;

− вывод вершин в соответствии с порядком обхода в глубину, начиная с вершины, заданной пользователем;

− вывод вершин в соответствии с порядком обхода по уровням, начиная с вершины, заданной пользователем.

Ход работы

1. Определить и описать необходимые структуры данных и список операторов. Допускается использовать модуль очереди, реализованный в ходе выполнения первой лабораторной работы.

2. Сформировать файл входных данных.

3. Разработать интерфейсную часть приложения, реализующую логику диалога пользователя и соответствующие вызовы операторов.

4. Разработать детальные алгоритмы процедур обхода, реализовать их.
5. Провести тестирование приложения.

7.5. Контрольные вопросы

1. В чем заключается специфика задачи обхода графа?
2. Опишите принцип обхода графа в глубину.
3. Поясните принцип реализации алгоритма обхода графа в глубину.
4. Опишите принцип обхода графа по уровням.
5. Поясните принцип реализации алгоритма обхода графа по уровням.
6. Дайте приблизительную оценку сложности алгоритмов обхода по времени.

8. Лабораторная работа «Анализ сложности поиска по АВЛ-сбалансированному дереву»

8.1. АВЛ-сбалансированные деревья

Бинарные поисковые деревья являются эффективным по времени инструментом организации доступа к данным. Очевидно, что сложность поиска будет иметь порядок высоты дерева. Поэтому случайные деревья, полученные путем многократного повторения операции поиска включением на имеющемся наборе данных, не всегда обеспечивают ожидаемую эффективность.

Одним из способов повышения эффективности операций на деревьях является введение ограничений на их структуру с целью уменьшения высоты. Например, полностью сбалансированные деревья при равной вероятности появления ключей являются оптимальными с точки зрения эффективности поиска: сложность $O(\log_2 N)$. Основным недостатком полностью сбалансированного дерева является необходимость его полного перестроения для восстановления баланса при включении или исключении элементов, что приводит к сложности этих операций $O(N)$.

Ослабив условия сбалансированности, можно построить «почти сбалансированное» дерево, такое, чтобы и в самой неблагоприятной реализации число сравнений при поиске (высота дерева) ограничивалось функцией $c \cdot \log_2 N$, где c – константа.

Одно из решений этой проблемы предложили два советских математика – Г.М. Адельсон-Вельский и Е.М. Ландис. Отсюда название таких деревьев – АВЛ-сбалансированные деревья или АВЛ-деревья.

Бинарное дерево называется *АВЛ-сбалансированным*, если высота левого поддерева каждого узла отличается от высоты правого поддерева не более чем на 1.

Показателем сбалансированности узла АВЛ-дерева называется разность высот его левого и правого поддеревьев. Исходя из определения, показа-

тели сбалансированности узлов АВЛ-дерева могут принимать значения –1, 0, +1.

8.2. Балансировка при включении

При вставке новых ключей в дерево могут иметь место различные ситуации. Пусть имеется корень *r*, левое и правое поддеревья *L* и *R* соответственно, и включение в поддерево *L* приводит к увеличению его высоты:

Если $h_L = h_R$, *L* и *R* станут разной высоты, но показатель сбалансированности не будет нарушен.

Если $h_L < h_R$, *L* и *R* станут равной высоты, т.е. показатель сбалансированности улучшится.

Если $h_L > h_R$, условие сбалансированности нарушается, появляются узлы с показателем сбалансированности –2 (для правого дерева +2), и для сохранения свойства сбалансированности АВЛ-дерева потребуется некоторая корректировка его структуры.

Для восстановления баланса дерева осуществляются «повороты» его узлов. При этом допускаются лишь вертикальные перемещения вершин, в то время как их относительное горизонтальное расположение должно оставаться неизменным, т.е. обеспечивается сохранение поисковых свойств дерева. На рисунках 8.1-8.4 приведены примеры различных вариантов поворотов (цветом выделены включаемые узлы).

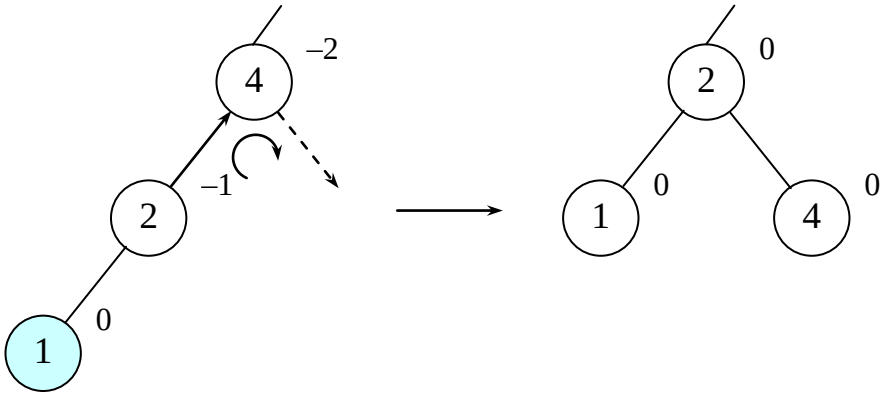


Рисунок 8.1 – Однократный поворот

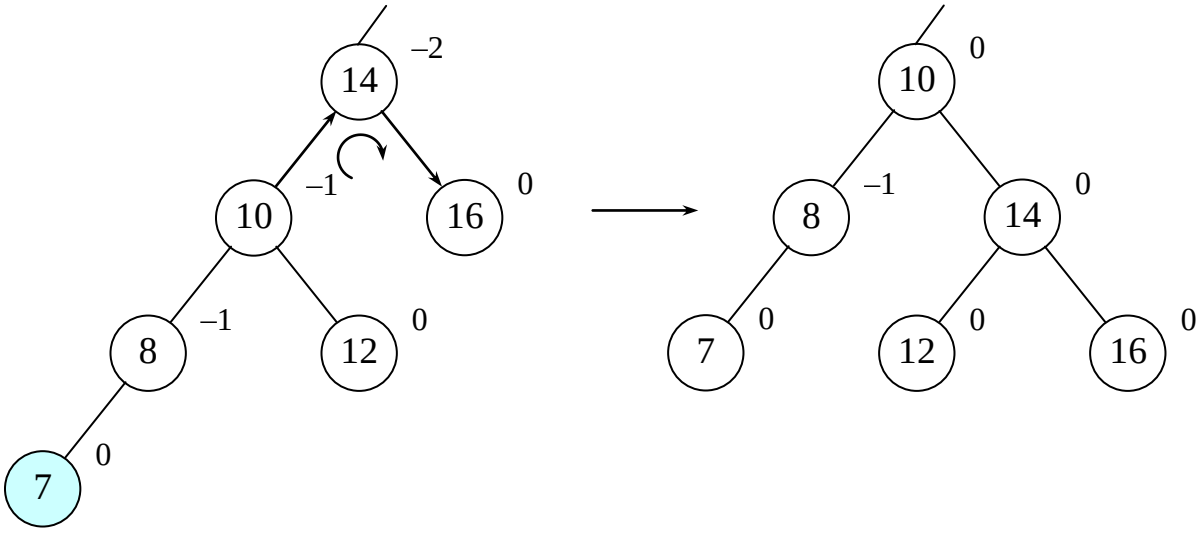


Рисунок 8.2 – Однократный поворот с переприкреплением поддерева

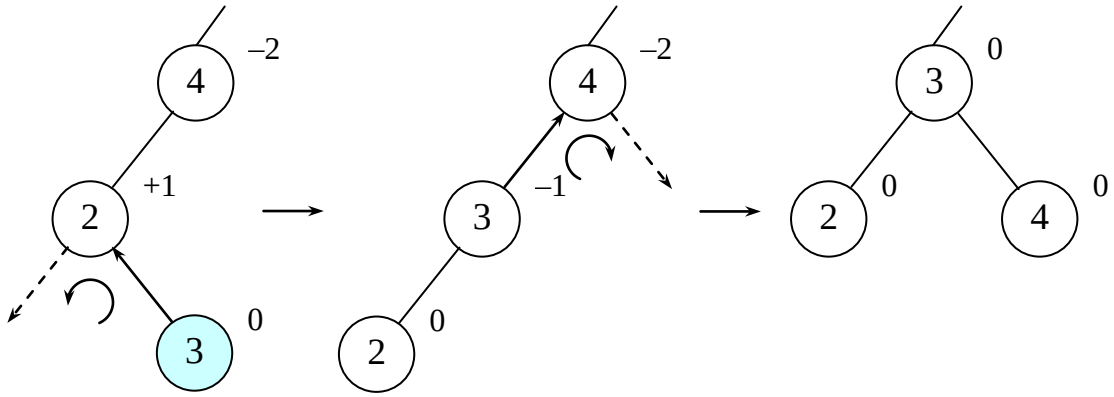


Рисунок 8.3 – Двукратный поворот

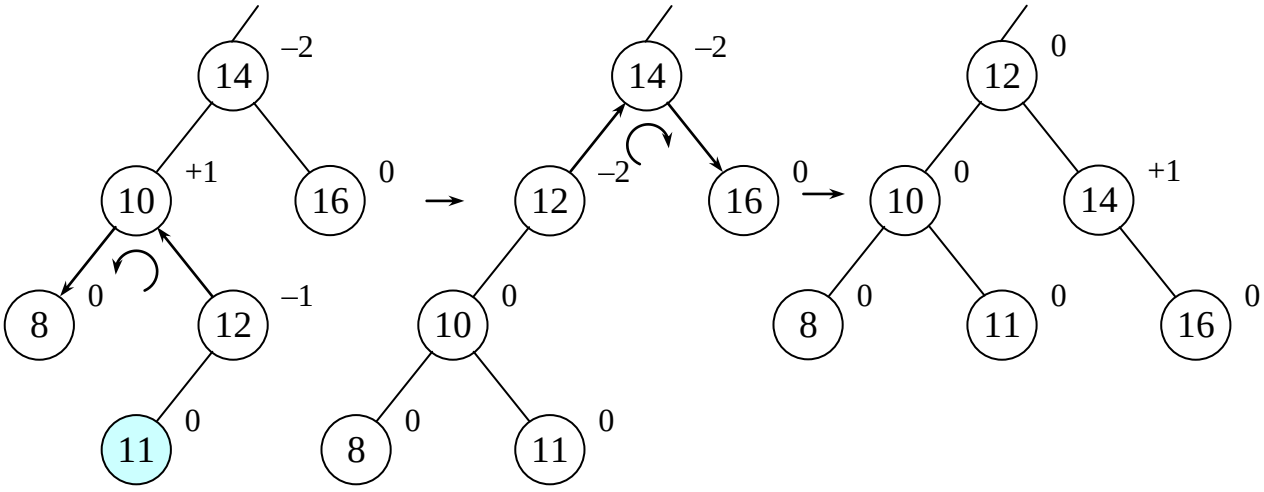


Рисунок 8.4 – Двукратный поворот с переприкреплением поддерева

Рассматривая приведенные примеры, можно заметить следующее:

1. Изменения затрагивают только поддерево, корнем которого является самый нижний узел с нарушенным показателем сбалансированности. В верхней части дерева в результате включения также могут появиться узлы с нарушенным показателем сбалансированности. Но в процессе балансировки высота изменяемого поддерева всегда уменьшается на единицу, что в любом случае приведет к нормализации нарушенных выше показателей.

2. Если после включения показатели сбалансированности корня рассматриваемого поддерева и его левого потомка имеют одинаковый знак, то восстановление баланса дерева возможно однократным поворотом. Если знаки отличаются, то потребуется двукратный поворот: первый поворот относительно левого потомка, второй – относительно корня.

3. При повороте может потребоваться переприкрепление узлов. Однако, обобщая, можно говорить, что переприкрепление осуществляется в любом случае: если указатель нулевой, то его значение также присваивают соответствующему указателю.

Основываясь на изложенном выше, можно обобщить все рассмотренные случаи к двум схемам выполнения поворотов, приведенным на рисунках 8.5 и 8.6 (перекрестиями отмечены места включения узлов).

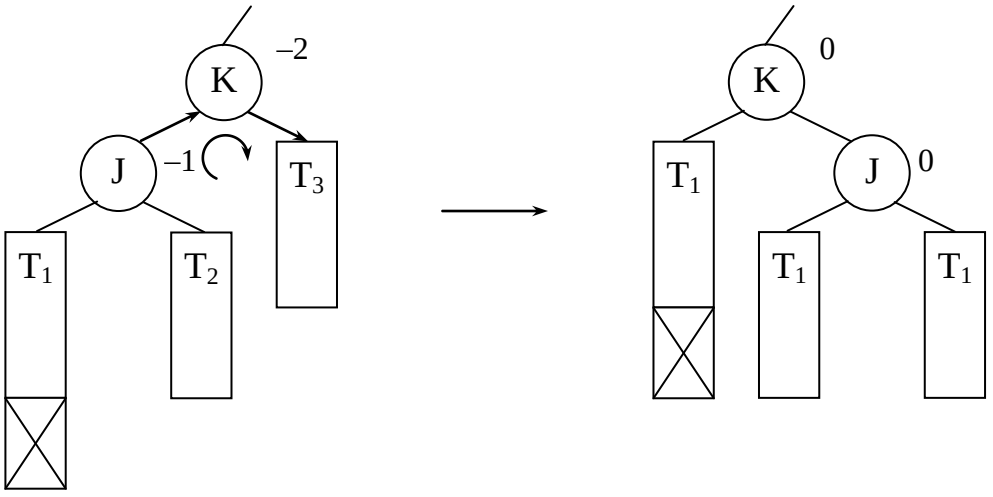


Рисунок 8.5 – Общая схема однократного поворота

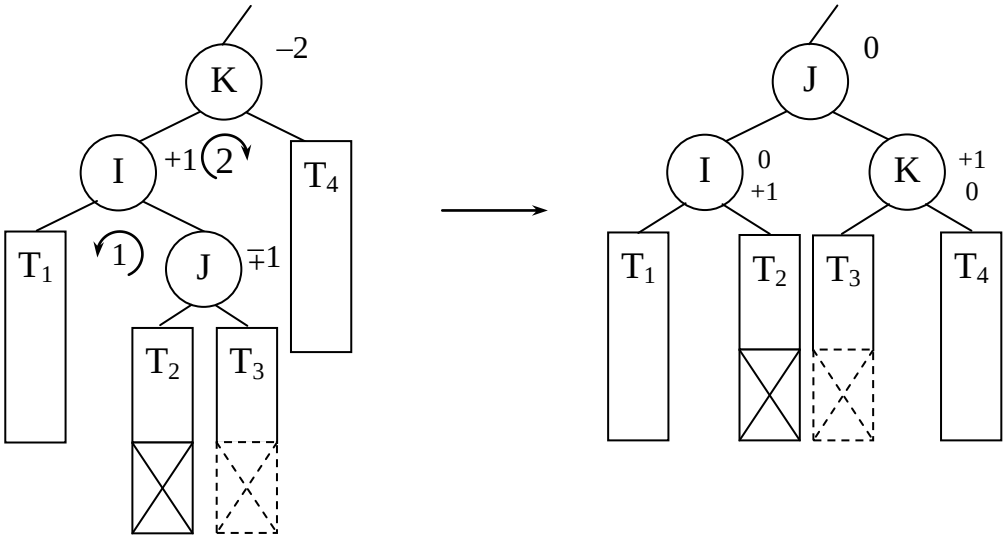


Рисунок 1.6 – Общая схема двукратного поворота

Алгоритм включения и балансировки существенно зависит от того, каким образом храниться информация о сбалансированности. Одно решение – хранить эту информацию неявно в структуре самого дерева. В этом случае возникает необходимость определять показатель сбалансированности узла всякий раз, когда включение затрагивает этот узел, что ведет к значительным затратам. Второе решение – хранить в каждой вершине показатель сбалансированности, т.е. вводить в определение узла переменную типа [-1..+1].

Процесс включения вершины фактически состоит из следующих трех последовательно выполняемых частей:

1. Проход но пути поиска до достижения нулевого указателя, определяющего место включения.

2. Включение новой вершины.

3. Если необходимо – балансировка.

Такой алгоритм рационально строить на основе рекурсивного алгоритма поиска вставкой. При этом после вставки необходимо вверх по рекурсии передавать информацию о том, увеличилась ли высота того поддерева, где произошло включение. Для этого можно использовать некую логическую параметр–

переменную h. Непосредственно из вызова, в котором произошло включение передается h = true.

Предположим, что процесс из левой ветви возвращается к вершине p, причем указывается, что высота поддерева увеличилась, то есть h = true, иначе в балансировке нет необходимости. В зависимости от начальной высоты поддеревьев перед включением необходимо различать три возможные ситуации:

1. Если $h_L < h_R$ и $p.bal = +1$, то предыдущая несбалансированность в p уравновешивается: $p.bal \leftarrow 0$, $h \leftarrow$ false.

2. Если $h_L = h_R$ и $p.bal = 0$, то левое поддерево стало «перевешивать»: $p.bal \leftarrow -1$.

3. Если $h_L > h_R$ и $p.bal = -1$, то необходима балансировка: выполнить соответствующий поворот, $p.bal \leftarrow 0$, $h \leftarrow$ false.

Рассмотренные примеры и общие схемы приведены для случаев увеличения высоты левого поддерева. При увеличении высоты правого поддерева соответствующие схемы поворотов и условия могут быть получены путем зеркального отражения.

8.3. Балансировка при исключении

Принципиальная схема алгоритма исключения из сбалансированного дерева аналогична схеме рекурсивного алгоритма поиска исключением. Как и в случае включения, вводится логический параметр-переменная h, указывающий, уменьшилась ли высота поддерева.

Балансировка осуществляется, только если h = true и имеется соответствующее соотношение высот поддеревьев. В отличие от включения истинное значение присваивается переменной h не только при обнаружении и исключении какой-либо вершины, но и при уменьшении высоты поддерева в процессе самой балансировки.

8.4. Оценки эффективности АВЛ-сбалансированных деревьев

Рассмотрим лучший и худший случаи АВЛ-деревьев. Очевидно, что лучшим случаем является полностью сбалансированное дерево, которое по определению удовлетворяет условию АВЛ-сбалансированности. Худшим случаем является дерево, которое при заданном количестве узлов имеет наибольшую высоту. Этому требованию отвечает дерево, у каждого узла которого высота, например, левого поддерева на единицу больше высоты правого. Это описание соответствует определению дерева Фибоначчи. Отсюда можно говорить, что высота h АВЛ-сбалансированных деревьев с N узлами будет удовлетворять условию:

$$1{,}4404\log_2(N+2) - 0{,}328 > h \geq \log_2(N+1).$$

Математический анализ среднего случая высоты АВЛ-дерева не произведен. Однако, экспериментальные оценки показывают, что ожидаемая высота сбалансированного дерева, построенного поиском включением на случайном множестве ключей

$$h = \log_2(N+1) + c,$$

где $c < 1$ – константа.

В случае большого количества операций включения и исключения также необходимо оценивать их сложность. Имеются следующие оценки:

4. В среднем одна балансировка приходится приблизительно на два включения. При этом одинаково вероятны и однократные, и двукратные повороты.

5. Балансировка при включении требует постоянного числа операций, то есть приводит только к одному однократному или двукратному повороту.

6. При исключении вследствие того, что уменьшение высоты поддерева при балансировке может привести к новому нарушению показателя сбалансированности в верхних узлах по пути поиска, возможны несколько поворотов. В худшем случае повороты потребуются по всему пути поиска – сложность $O(\log_2 N)$. Например, исключение самого верхнего листа из дерева Фибоначчи.

7. При исключении поворот в среднем происходит приблизительно в одном случае из пяти.

Таким образом, значительная трудоемкость операции балансировки предполагает, что сбалансированные деревья следует использовать только тогда, когда поиск информации происходит значительно чаще, чем ее включение и (или) исключение.

8.5. Задание на лабораторную работу

Входные данные представляют собой случайные наборы целых чисел. Размер наборов данных: 100, 1000, 10000, 20000, … 100000.

Необходимо произвести экспериментальную оценку сложности поиска по АВЛ-дереву.

Ход работы:

1. Реализовать приложение, обеспечивающее построение АВЛ-сбалансированного дерева поиском включением на основе входного файла целочисленных ключей.

2. Провести экспериментальную оценку средней длины пути поиска на деревьях с количеством ключей 100, 1000, 10000, 20000, … 100000.

3. По результатам экспериментов построить таблицу и график зависимости средней длины пути поиска от количества узлов дерева.

4. Сравнить полученный график с графиками функций, определяющих нижнюю и верхнюю границы сложности поиска (приведены в подразделе 8.4). Сделать выводы.

8.6. Контрольные вопросы

1. Сформулируйте определение АВЛ-дерева и условие АВЛ-сбалансированности.

2. В чем заключаются преимущества АВЛ-дерева по сравнению с оптимальным и полностью сбалансированным поисковыми деревьями?

3. За счет каких операций выполняется балансировка в АВЛ-дереве.

4. Каковы оценки трудоемкости операций балансировки при вставке и исключении?

5. Поясните процедуру балансировки при вставке.

6. Поясните процедуру балансировки при исключении.

7. Дайте оценку сложности поиска по АВЛ-дереву в худшем и лучшем случаях.

9. Лабораторная работа «Оценка качества хеш-функций»

9.1. Основные определения и сущность хеширования

Основным принципом повышения эффективности операций доступа к данным является наличие некоторой информации о них, например, об их упорядоченности, как в случае поиска делением пополам.

Однако, есть методы организации данных, которые позволяют обеспечить более высокую эффективность поиска, вставки и удаления, вплоть до оценки сложности O(1). Например, метод прямого доступа. Пусть имеется множество используемых ключей K_u, являющееся подмножеством множества всех возможных ключей $K=\{0,1, \dots , |K| - 1\}$. Для организации данных можно выделить массив T размера $[0 .. |K| - 1]$. Ключ k вставляется в ячейку с индексом k. Тогда для доступа к нужному элементу данных достаточно будет выполнить одну операцию индексации $T[k]$, то есть имеем сложность O(1). Очевидным недостатком данного подхода является нерациональное использование памяти, так как в реальных задачах мощность множества используемых ключей $|K_u| << |K|$.

Если отказаться от прямой адресации, то можно попытаться найти некоторую функцию f, преобразующую множество используемых ключей в множество адресов A мощностью M. Причем M – «разумная» величина порядка $|K_u|$. Тогда операция поиска будет заключаться в вычислении функции преобразования для аргумента поиска k: $a = f(k)$, где a – индекс (адрес) искомой записи в массиве. Такой подход, сохраняя постоянную сложность, обеспечивает рациональное использование памяти. Однако, имеются ограничения, которые не позволяют применять при решении подавляющей части практических задач:

1. Множество ключей должно быть невелико, иначе построение функции преобразования становиться невозможным за разумное время.

2. Значения ключей должны быть заранее известны, то есть при добавлении нового ключа должна быть выведена новая функция преобразования и в соответствии с ней должно быть осуществлено переразмещение всех ключей.

Для решения этой проблемы был предложен более гибкий подход, названный хешированием (от английского hash – рубить, крошить, измельчить, перемешать). Также как и в предыдущем случае данные «перемешиваются» в массиве в соответствии с определенным правилом. Отличие заключается в том, что осуществляется отказ от однозначности преобразования множества ключей в множество адресов $a = h(k)$.

Хеширование – метод организации данных, основанный на прямом доступе к ячейке массива посредством индекса, получаемого как результат хеш-функции от ключа. *Хеш-функция* – дискретная функция, рассеивающая (неоднозначно преобразующая) множество всех возможных ключей K в множество адресов $A=\{0, 1, \ldots, M-1\}$, где $|K|>M$. Массив, используемый для хранения элементов данных (обычно записей) и доступа к ним посредством хеширования, называется *хеш-таблицей*.

Отказ от требования взаимно однозначного соответствия между ключом и адресом означает, что для различных ключей $k_1 \neq k_2$ после вычисления хеш-функции может возникнуть ситуация, называемая *коллизией* $h(k_1) = h(k_2)$. В этом случае говорят, что ключи k_1, k_2 являются синонимами по функции h.

Таким образом, главными задачами при организации хеширования являются выбор хеш-функции h и нахождение способов разрешения возникающих коллизий. На данном практическом занятии рассматривается первая задача.

9.2. Показатели качества хеш-функций

При выборе хеш-функции необходимо, чтобы она минимизировала количество коллизий и не допускала «скучивания» ключей в отдельных частях таблицы.

Таким образом, основным показателем качества хеш-функции является количество коллизий, получаемых при рассеивании определенного количества ключей $k \in K$ в адреса $a \in A$. То есть, если на одних и тех же ключах при одина-

ковом размере хеш-таблицы одна хеш-функция дает меньшее количество коллизий, чем другая, то первая является более качественной.

Показателем, характеризующим степень скученности, является равномерность рассеивания ключей (или коллизий) по диапазону адресов. В лучшем случае при попытке рассеять N ключей в таблице размером M на каждый адрес будет приходится одно и то же количество ключей: $\lfloor N/M \rfloor$ или $\lceil N/M \rceil$, или коллизий: $\lfloor (N-M)/M \rfloor$ или $\lceil (N-M)/M \rceil$. Худшим является случай, когда все ключи (коллизии) приходятся на один адрес таблицы. Например, преднамеренно выберем непригодную на практике функцию $h(k)=k\cdot 0$. Тогда все ключи будут адресованы в ячейку $a = 0$. На рисунке 9.1 приведены возможные на практике желательный (слева) и нежелательный (справа) варианты рассеивания 10 ключей по 10 адресам.

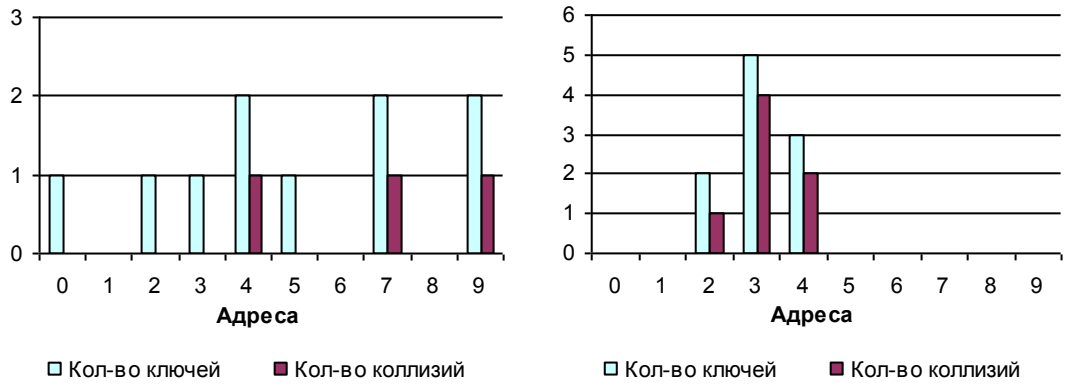


Рисунок 9.1 – Примеры рассеивания ключей

В ходе выполнения практического задания во избежание усложнения анализа экспериментальных результатов допускается визуальная оценка степени скученности коллизий.

9.3. Виды хеш-функций

В случае численных ключей для получения хеш-функций могут использоваться несколько методов. Рассмотрим наиболее простые из них.

Метод деления

В методе деления в качестве значения хеш-функции используется остаток от деления ключа на некоторое целое число M:

$$h(k) = k \bmod M.$$

Очевидно, что эффективность рассеивания ключей по таблице во многом зависит от значения M. Нежелательно, чтобы M являлось степенью основания системы счисления, так как в этом случае значением хеш-функции будут просто младшие разряды ключа.

Метод умножения

При использовании метода умножения хеш-функция имеет вид:

$$h(k) = \lfloor M \cdot \{k \cdot A\} \rfloor,$$

где $\{k \cdot A\}$ – дробная часть числа, полученного в результате умножения ключа на некоторую иррациональную константу A.

Достоинство метода умножения заключается в том, что качество хеш-функции мало зависит от выбора M. Обычно в качестве M выбирают степень двойки, так как операции с таким числом выполняются быстрее. Метод умножения работает при любой константе A, но от ее выбора зависит качество хеширования. Доказано, что всегда достаточно удачным является выбор «золотой середины»:

$$A = \frac{\sqrt{5} - 1}{2} = 0{,}6180339887...$$

Единственным ограничением является недопустимость равенства A=1/M. В этом случае метод умножения эквивалентен методу деления. В общем случае метод умножения обеспечивает большую степень рассеивания ключей по сравнению с методом деления.

В случае символьных ключей необходимо их предварительное преобразование в численные. Причем способ преобразования также может значительно повлиять на качество хеширования.

Аддитивный метод

Заключается в нахождении суммы кодов всех символов, например ASCII. К найденной сумме может быть применен любой из рассмотренных выше методов.

Недостаток метода заключается в его неспособности различать строки, одинаковые по набору символов, но отличающиеся порядком их следования, например, «АБВ» и «БАВ».

Метод исключающего ИЛИ

Данный метод позволяет устранить недостаток аддитивного. Процедура преобразования заключается в том, что к элементам строки последовательно применяется операция «исключающее или», а результаты суммируются: $sum \leftarrow sum + \text{ord}(s[i])$ xor $r[i]$. Здесь $s[i]$ – код i-го символа строки, а $r[i]$ – соответствующее i-му символу целое число (количество бинарных разрядов $s[i]$ и $r[i]$ должно совпадать). Массив r может быть задан перед началом процедуры хеширования случайным образом, но в процессе поддержания хеш-таблицы меняться не должен, так как это приведет к нарушению процедуры поиска.

9.4. Задание на лабораторную работу

Даны:

– набор ключей – случайные строки длиной 6 символов, состоящие из букв латинского алфавита (ASCII: 65-90 и 97-122 включительно) и десятичных цифр (ASCII: 48-57);

– параметры хеширования: размер таблицы и количество ключей.

Произвести экспериментальную оценку качества хеширования при использовании методов деления, умножения при преобразовании ключей аддитивным методов и методом исключающего ИЛИ.

Ход работы:

1. Разработать приложение, обеспечивающее определение количества коллизий (моделирование процесса хеширования) на каждый адрес хеш-таблицы в процессе рассеивания случайных строковых ключей.

2. По результатам моделирования работы хеш-функций построить графики (диаграммы) зависимости количества коллизий от адреса ячейки таблицы для каждой комбинации методов – четыре графика. Определить общее количество коллизий для каждой комбинации методов.

3. Оценить экспериментальные результаты, сделать выводы относительно эффективности каждого метода.

9.5. Контрольные вопросы

1. Какова основная идея хеширования, как метода хранения и организации доступа к данным в оперативной памяти?
2. В чем заключаются преимущества хеширования по сравнению с использованием поисковых деревьев?
3. Какие требования предъявляются к хеш-функции?
4. В чем преимущество метода умножения перед методом деления?

10. Лабораторная работа «Анализ методов разрешения коллизий при хешировании»

10.1. Метод цепочек

Методом цепочек называется метод, в котором для разрешения коллизий во все записи добавлены указатели, используемые для поддержания связанных списков синонимов (по одному списку на каждый возможный хеш-адрес таблицы). Сами списки могут размещаться как в памяти, принадлежащей хеш-таблице, так и в динамически выделяемой памяти. В первом случае цепочки называются *внутренними,* а во втором – *внешними.*

В случае внешних цепочек при поиске вычисляется значение хеш-функции $a = h(k)$ и осуществляется последовательный просмотр a-го списка до достижения записи с ключом k (удачный поиск). Если весь список просмотрен, и совпадения не обнаружено, то поиск завершается неудачей. При вставке также вычисляется значение хеш-функции $a = h(k)$, и запись с ключом k помещается в динамически создаваемый и включаемый в a-й список элемент. Если число синонимов станет слишком большим, и операции доступа станут неэффективными, можно вместо линейных списков использовать дерево поиска.

Хеш-таблицы с внутренними цепочками позволяют более эффективно использовать память. Для этого связанные списки синонимов по хеш-функции поддерживаются внутри таблицы. В этом случае ячейка таблицы должна содержать поле, в котором хранится индекс следующей ячейки списка синонимов. Процедуры поиска, вставки и удаления строятся на тех же принципах, что и в методе внешних цепочек, но базируются на соответствующих операциях связного списка в последовательной памяти. Например, операция вставки начинается с вычисления хеш-фукнции ключа. Если возникла коллизия, то в таблице осуществляется поиск свободной позиции, вставка в нее, и включекние этой ячейки в список синонимов путем модификации индексного поля предыдущей по списку ячейки. Для поиска сво-

бодной позиции могут использоваться различные методы, самый простой – последовательный просмотр ячеек таблицы.

10.2. Метод открытой адресации

Метод открытой адресации состоит в том, что с помощью какого-либо правила, обеспечивающего перебор всех элементов, осуществляется просмотр ячеек хеш-таблицы в поисках незанятой. Данное «правило поиска», называемое *стратегией опробования*, обеспечивает однозначное соответствие путей доступа к элементам данных при вставке, поиске и удалении.

Алгоритм вставки или поиска ключа k в хеш-таблице T в общем виде выглядит следующим образом:

1. $i \leftarrow 0$.
2. $a \leftarrow h_i(k)$.
3. Если $T[a]$ свободно, то {в случае поиска алгоритм заканчивается неудачно} {в случае вставки элемент вставляется, и алгоритм заканчивается удачно}.

Если $T[a]=k$, то {в случае поиска a – индекс искомого элемента, алгоритм заканчивается удачно} {в случае вставки ситуация наличия такого же ключа обрабатывается в соответствии с требованиями задачи, алгоритм заканчивается неудачно}.

Иначе – коллизия, $i \leftarrow i+1$.

4. Если i превысило предельное количество шагов опробования то алгоритм заканчивается неудачно, иначе перейти к пункту 2.

Сделаем некоторые пояснения:

1. В широком смысле $\{h_i\}$ ($i=0...\infty$) представляет собой последовательность хеш-функций, правило формирования которой определяется стратегией опробования.

2. В пунктах 3 и 4 фигурными скобками отмечены варианты действий в случае истинности условий в зависимости от того, производится ли поиск или вставка.

3. Условие в пункте 6 введено с целью исключения бесконечного цикла опробований. Такая ситуация может возникнуть в двух случаях: либо некорректно выбрана хеш-функция и (или) правило опробования, и алгоритм не перебирает всех ячеек (циклически проходит по одному и тому же не исчерпывающему пути), либо таблица заполнена практически полностью. В любом случае выход из цикла по данному условию нужно анализировать и принимать соответствующие меры.

10.3. Стратегии опробования в методе открытой адресации

Рассмотрим две наиболее простых стратегий опробования.

Линейное опробование

Это простейшая схема открытой адресации, при которой

$$h_i(k)=(h_0(k) + c\cdot i) \bmod M,$$

где $h_0(k)$ – исходная хеш-функция;

c – константа.

При выборе константы с необходимо учитывать следующие рекомендации:

1. Константа с должна быть достаточно большим числом, например, в 4-5 раз меньше размера таблицы M. Если выбрать c достаточно малым, то при наличии скучивания коллизий цепочки переполнения будут длиннее, так как небольшой шаг опробования не позволит быстро находить свободные позиции. Худшим случаем является c=1, когда метод открытой адресации сводится к рассмотренному выше методу внутренних цепочек с последовательным опробованием ячеек.

2. Для уменьшения степени срастания цепочек, то есть ситуации, когда одна цепочка синонимов попадает на позиции другой и длины путей поиска по

одной и другой увеличиваются, желательно, чтобы M и c были взаимно простыми числами.

3. Недопустимо, чтобы M было кратно c, так как в этом случае на шаге опробования M/c начнут посещаться ячейки, уже пройденные в данном цикле поиска. Это приведет к бесконечному циклу опробований, о котором речь шла выше.

Квадратичное опробование

Это схема открытой адресации, при которой

$$h_i(k) = (h_0(k) + c \cdot i + d \cdot i^2) \bmod M,$$

где c и d – константы.

Эта схема более эффективна по сравнению с линейным опробованием, о чем более подробно будет сказано ниже. Также она менее чувствительна к выбору констант. Единственным требованиям является то, что c должно быть достаточно большим числом, а d – достаточно малым.

10.4. Оценки эффективности методов разрешения коллизий

Если давать оценку эффективности методам разрешения коллизий по памяти, то наиболее затратными с этой точки зрения являются внешние цепочки. Остальные методы не требуют дополнительной памяти для поддержания цепочек синонимов.

При рассмотрении эффективности по времени, наиболее выгодным является использование внешних цепочек, так как здесь увеличение длины пути поиска по причине срастания цепочек невозможно по определению. Другим значительным преимуществом метода (свойственно и внутренним цепочкам) является простота процедуры удаления, которая заключается просто в исключении элемента из односвязного списка синонимов. В остальных методах операция исключения не является обратной процедуре включения.

Ранжирование остальных методов в порядке возрастания эффективности дает следующий список:

1. Внутренние цепочки.
2. Линейное опробование.
3. Квадратичное опробование.

Преимущество линейного опробования перед внутренними цепочками обуславливается в увеличении шага опробования и, следовательно, снижении вероятности попадания в ту же область высокого скучивания коллизий, в которой произошла первая.

В то же время линейному опробованию по сравнению с квадратичным более свойственно срастание цепочек. Это объясняется тем, что все цепочки синонимов имеют один и тот же шаг, и при их пересечении на любой итерации опробования обязательно возникает срастание. Квадратичная схема вследствие ее нелинейности позволяет избежать этого. Высокая степень срастания наблюдается только при высоком коэффициенте заполнения таблицы.

Результирующим показателем эффективности методов разрешения коллизий, как и процедуры хеширования в целом, является количество шагов опробования или длина пути поиска. Очевидно, что лучшим случаем будет являться порядок O(1). Оценка худшего и среднего случая является математически сложной задачей, так как эти оценки будут зависеть от множества факторов.

Однако, можно привести некоторые факты и рекомендации относительно эффективности хеширования:

1. Эффективность в общем смысле зависит от выбора хеш-функции, метода разрешения коллизий, выбранных размера таблицы, констант, набора входных данных.

2. Длина пути поиска существенно зависит от коэффициента заполнения таблицы. Обычно считается, что нормальной ситуацией, когда сложность поиска действительно можно оценивать константой – O(1), является заполнение таблицы не более, чем на 50-60%. Это необходимо учитывать при выборе *M*. Если в процессе работы коэффициент заполнения превысил этот порог, то для сохранения эффективности необходимо рехеширование, то есть

увеличение размера таблицы, формирование новой хеш-функции и хеширование всех ключей в новую таблицу.

3. Если рассматривать другие методы организации данных, широко используемые для подобного класса задач, то можно говорить, что сравнимые свойства обеспечиваются сбалансированными деревьями. Например, АВЛ-деревья, сложность основных операций для которых оценивается как $O(\log_2 N)$. Поэтому можно говорить, что использование хеширования является рациональным, если средняя длина пути поиска ограничивается сверху функцией $\log_2 N$.

10.5. Задание на лабораторную работу

Даны:

– набор ключей – случайные строки длиной 6 символов, состоящие из букв латинского алфавита (ASCII: 65-90 и 97-122 включительно) и десятичных цифр (ASCII: 48-57);

– параметры хеширования: размер таблицы и количество ключей.

Произвести экспериментальную оценку качества разрешения коллизий при использовании метода открытой адресации с линейным и квадратичным опробованием при выполнении хеширования методами деления, умножения.

Ход работы:

1. Разработать приложение, обеспечивающее интерактивный ввод всех параметров хеширования, генерацию определенного числа случайных ключей и их вставку в хеш-таблицу в соответствии с выбранной комбинацией методов. Во всех случаях для преобразования символьных ключей в численные необходимо использовать метод исключающего ИЛИ.

2. Произвести оценку качества хеширования для всех комбинаций методов путем определения средней (по нескольким экспериментам со случайными ключами и одинаковыми настройками) длины пути поиска при вставке клю-

ча для различных значений количества вставляемых ключей от 10 до размера таблицы с шагом 10.

3. Для каждой комбинации методов по результатам экспериментов построить таблицу и соответствующие графики (четыре графика) зависимости длины пути поиска от количества ключей.

4. Оценить экспериментальные результаты, сделать выводы относительно эффективности рассмотренных методов.

10.6. Контрольные вопросы

1. В чем принципиальное отличие метода внешних цепочек и метода открытой адресации при разрешении коллизий?

2. Какой недостаток линейного опробования устраняется квадратичным опробованием?

3. Какое значение имеет выбор константы *C* в стратегиях опробования?

4. Какой вывод об условии необходимости рехеширования можно сделать на основе анализа результатов эксперимента?

Список использованных источников

1. Вирт Н. Алгоритмы и структуры данных. – М.: Мир, 1989. – 360 с.

2. Сибуя М. Алгоритмы обработки данных / М. Сибуя, Т. Ямамото. – М.: Мир, 1986. – 218 с.

3. Макконенелл Дж. Анализ алгоритмов. Вводный курс. – М.: Техносфера, 2002. – 304 с.

4. Макконенелл Дж. Основы современных алгоритмов. 2-е дополненное издание. – М.: Техносфера, 2004. – 368 с.

5. Ахо А. Структуры данных и алгоритмы / А. Ахо, Д. Хопкрофт, Д. Ульман. – М.: Издательский дом «Вильямс», 2000. – 384 с.

6. Кормен Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест. – М.: МЦНМО, 2000. – 960 с.