МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)

А. Н. ДАНИЛЕНКО

СТРУКТУРЫ ДАННЫХ И АНАЛИЗ СЛОЖНОСТИ АЛГОРИТМОВ

Рекомендовано редакционно-издательским советом федерального государственного автономного образовательного учреждения высшего образования «Самарский национальный исследовательский университет имени академика С.П. Королева» в качестве учебного пособия для студентов, обучающихся по основной образовательной программе высшего образования по направлению подготовки 02.03.02 Фундаментальная информатика и информационные технологии

С А М А Р А
Издательство Самарского университета
2018

УДК 004.42(075)
ББК 32.973-018.2я7
Д181

Рецензенты: д-р техн. наук, проф. С. А. П р о х о р о в,
канд. техн. наук, доц. О. В. А р и п о в а

Даниленко, Александра Николаевна

Д181 **Структуры данных и анализ сложности алгоритмов:** учеб. пособие / А.Н. Даниленко. – Самара: Изд-во Самарского университета, 2018. – 76 с.: ил.

ISBN 978-5-7883-1272-9

Краткая теоретическая часть, предшествующая постановке задачи, позволяет подготовиться к выполнению лабораторных работ в условиях ограниченного количества методической литературы по данной дисциплине. Выполнение лабораторных работ направлено на знакомство студентов с основными структурами данных и критериями оценки сложности алгоритмов. Студентам предлагается реализовать алгоритмы ряда задач, широко применяемых в практике программирования. В пособии затрагиваются вопросы амортизационного анализа алгоритмов в терминах O, **Ω** и θ-символики и определения порядка их сложности. Пособие состоит из введения и 6 основных разделов и содержит в себе задания на работу с массивами, строками, деревьями, графами и другими линейными и нелинейными структурами.

Учебное пособие предназначено для студентов дневного отделения, обучающихся по направлению подготовки 02.03.02 Фундаментальная информатика и информационные технологии, изучающих курс «Алгоритмы и анализ сложности».

Подготовлено на кафедре программных систем Самарского университета.

УДК 004.42(075)
ББК 32.973-018.2я7

ISBN 978-5-7883-1272-9

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ В ТЕОРИЮ АЛГОРИТМОВ

В последние годы большое распространение получила концепция структурного программирования. Понятие *структурного программирования* включает определенные принципы проектирования, кодирования, тестирования и документирования программ в соответствии с заранее определенной жесткой дисциплины.

Полное построение алгоритма предусматривает последовательное выполнение следующих этапов:

- постановка задачи;
- построение модели;
- разработка алгоритма;
- проверка правильности алгоритма;
- реализация, т.е. программирование алгоритма;
- анализ алгоритма и его сложности;
- проверка (отладка) программы;
- составление документации.

Не все эти этапы четко различимы между собой, особенно эта различимость делается мало заметной при программировании простых задач. При программировании простых задач некоторые этапы могут вообще не выполняться – настолько очевидны их результаты.

При программировании сложных, объемных задач некоторые из вышеперечисленных этапов приходится выполнять не в том порядке, как выше указано, или выполнять их не один раз [1].

Рассмотрим более подробно каждый из этапов построения алгоритма.

Постановка задачи. Прежде чем понять задачу, ее нужно точно сформулировать. Это условие само по себе не является достаточным для понимания задачи, но оно абсолютно необходимо.

Обычно процесс точной формулировки задачи сводится к постановке правильных вопросов. Перечислим некоторые полезные вопросы для плохо сформулированных задач:

— Понятна ли терминология, используемая в предварительной формулировке?

— Что дано? Что нужно найти?

— Как определить решение?

— Каких данных не хватает или, наоборот, все ли перечисленные в формулировке задачи данные используются?

— Какие сделаны допущения?

Возможны и другие вопросы, возникающие в зависимости от конкретной задачи.

Построение модели. Задача четко поставлена, теперь нужно сформулировать для нее математическую модель. Выбор модели существенно влияет на остальные этапы в процессе решения.

Выбор модели в большей степени искусство, чем наука. Хотя при решении типовых задач целесообразно воспользоваться ранее наработанными правилами. Поэтому изучение удачных моделей – это наилучший способ приобрести опыт в моделировании.

Приступая к разработке модели, следует задать, по крайней мере, два основных вопроса:

— Какие математические структуры больше всего подходят для задачи?

— Существуют ли решенные аналогичные задачи?

Второй вопрос, возможно, самый полезный во всей математике [1].

Разработка алгоритма. Выбор метода разработки алгоритма зачастую сильно зависит от выбора модели и может в значительной степени повлиять на эффективность алгоритма решения. Два различных алгоритма могут быть правильными, но очень сильно отличаться по эффективности.

Правильность алгоритма. Доказательство правильности алгоритма – это один из наиболее трудных, а иногда и особенно утомительных этапов создания алгоритма. Вероятно, наиболее распространенный прием доказательства правильности программы – это прогон ее на разных тестах. Если выданные программой ответы могут быть подтверждены известными или вычисленными вручную данными, возникает искушение сделать вывод, что программа «работает» правильно. Однако этот метод редко исключает все сомнения; может существовать случай, в котором программа не работает.

Рассмотрим следующую общую методику доказательства правильности алгоритма. Предположим, что алгоритм описан в виде последовательности шагов, допустим, от шага 0 до шага m. Постараемся предложить некое обоснование правомерности для каждого шага. В частности, может потребоваться формулировка утверждения об условиях, действующих до и после пройденного шага. Затем постараемся предложить доказательство конечности алгоритма, при этом будут проверены все подходящие входные данные и получены все подходящие выходные данные.

Другой метод доказательства правильности алгоритма, который не имеет специального названия, состоит в следующем. Для каждого цикла, который имеется в программе (алгоритме), вручную (например, на калькуляторе) подсчитываются две контрольные точки. Если контрольные точки совпадают со значениями, выданными программой, можно быть уверенным, что все циклы в программе работают правильно. Почему речь идет о двух контрольных точках? Дело в том, что первое контрольное значение в программе может быть вычислено правильно, а затем в этом цикле будут произведены некоторые некорректные действия, которые приведут к искажению всех последующих результатов. Совпадение второго контрольного значения как раз и подтверждает, что в данном цикле некорректности нет. Таким обра-

зом, два контрольных вычисления должны быть сделаны для каждого цикла программы.

Следует подчеркнуть и тот факт, что правильность алгоритма еще ничего не говорит о его эффективности. В этом смысле исчерпывающие алгоритмы, или, как их еще называют, алгоритмы полного перебора, редко бывают хорошими во всех отношениях [1].

Реализация алгоритма. Как только алгоритм выражен, допустим, в виде последовательности шагов и мы убедились в его правильности, настает черед реализации алгоритма, т.е. написания программы для компьютера.

При этом возникают следующие проблемы.

Очень часто отдельно взятый шаг алгоритма может быть выражен в форме, которую трудно перевести непосредственно в конструкции языка программирования. Например, один из шагов алгоритма может быть записан в виде, требующем целой подпрограммы для своей реализации.

Реализация может оказаться трудным процессом потому, что перед тем, как написать программу необходимо построить целую систему структур данных для представления важных аспектов используемой модели. Чтобы сделать это, необходимо ответить, например, на такие вопросы:

— Каковы основные переменные?

— Каких они типов?

— Сколько нужно массивов и какой размерности?

— Имеет ли смысл пользоваться связными списками?

— Какие нужны подпрограммы (возможно, уже записанные в памяти)?

— Каким языком программирования пользоваться?

Конкретная реализация может существенно влиять на требования к памяти и на скорость алгоритма.

Заметим, что одно дело доказать правильность конкретного алгоритма, описанного в словесной форме, другое – доказать, что данная машинная программа, предположительно являющаяся реализацией этого алгоритма, также правильна. Поэтому необходимо очень тщательно следить, чтобы процесс преобразования правильного алгоритма (в словесной форме или форме схемы алгоритма) в программу, написанную на алгоритмическом языке, заслуживал доверия [1].

Каждый, кто занимается разработкой алгоритмов, должен овладеть некоторыми основными методами и понятиями. Перед тем, кто когда-то столкнулся с трудной задачей, вставал вопрос: «С чего начать?». Один из возможных путей – просмотреть свой запас общих алгоритмических методов для того, чтобы проверить, нельзя ли с помощью одного из них сформулировать решение новой задачи. Ну, а если такого запаса нет, то как все-таки разработать хороший алгоритм?

Рассмотрим три общих метода решения задач, полезных для разработки алгоритмов.

Первый метод связан со сведением трудной задачи к последовательности более простых задач. Конечно, мы надеемся на то, что более простые задачи легче поддаются обработке, чем первоначальная задача, а также на то, что решение первоначальной задачи может быть получено из решений этих более простых задач. Такая процедура называется методом частных целей [1].

Этот метод выглядит очень разумно. Но, как и большинство общих методов решения задач или разработки алгоритмов, его не всегда легко перенести на конкретную задачу. Осмысленный выбор более простых задач скорее, искусство или интуиция, чем наука. Не существует общего набора правил для определения класса задач, которые можно решать с помощью такого подхода. Размышление над любой конкретной задачей начинается с постановки вопросов. Частные цели

могут быть установлены, когда получены ответы на следующие вопросы:

— Можем ли мы решить часть задачи? Можно ли, игнорируя некоторые условия, решить оставшуюся часть задачи?

— Можем ли мы решить задачу для частных случаев? Можно ли разработать алгоритм, который дает решение, удовлетворяющее всем условиям задачи, но входные данные которого ограничены некоторым подмножеством всех входных данных?

— Есть ли что-то, относящееся к задаче, что мы недостаточно хорошо поняли? Если попытаться глубже вникнуть в некоторые особенности задачи, сможем ли мы что-то узнать, что поможет нам подойти к решению?

— Встречались ли мы с похожей задачей, решение которой известно? Можно ли видоизменить ее решение для решения нашей задачи? Возможно ли, что эта задача эквивалентна известной нерешенной задаче?

Второй метод разработки алгоритмов известен как метод подъема. Алгоритм подъема начинается с принятия начального предположения или вычисления начального решения задачи. Затем начинается насколько возможно быстрое движение «вверх» от начального решения по направлению к лучшим решениям. Когда алгоритм достигнет такой точки, из которой больше невозможно двигаться наверх, алгоритм останавливается. К сожалению, мы не можем всегда гарантировать, что окончательное решение, полученное с помощью алгоритма подъема, будет оптимальным. Эта ситуация часто ограничивает применение метода подъема.

Вообще методы подъема являются «грубыми». Они запоминают некоторую цель и стараются сделать все, что могут и где могут, чтобы подойти ближе к цели. Это делает их несколько недальновидными.

Третий метод известен как отрабатывание назад, т.е. работа этого алгоритма начинается с цели или решения задачи и затем осуществляется движение к начальной постановке задачи. Затем, если эти действия обратимы, производится движение обратно от постановки задачи к решению [1].

Принципы разработки эффективных алгоритмов

Для большинства проблем существует много различных алгоритмов. Какой из них выбрать для решения конкретной задачи? Этот вопрос тщательно прорабатывается в программировании.

Эффективность программы (кода) является очень важной ее характеристикой. Пользователь всегда предпочитает более эффективное решение даже в тех случаях, когда эффективность не является решающим фактором [2].

Алгоритмом называется система формальных правил, четко и однозначно определяющая процесс решения поставленной задачи в виде конечной последовательности действий или операций.

Свойства, которыми должен обладать алгоритм:

1. *Конечность (финитность) алгоритма*. Алгоритм должен приводить к решению задачи обязательно за конечное время. Последовательность правил, приведшая к бесконечному циклу, алгоритмом не является.

2. *Определенность, или детерминированность, алгоритма.* Это свойство означает, что неоднозначность толкования записи алгоритма недопустима.

3. *Результативность алгоритма.* Под результативностью понимается доступность результата решения задачи для пользователя, иными словами, алгоритм должен обеспечить выдачу результата решения задачи на печать, на экран монитора или в файл.

4. *Массовость алгоритма.* Это означает, если правильный результат по алгоритму получен для одних исходных данных, то пра-

вильный результат по этому же алгоритму должен быть получен и для других исходных данных, допустимых в данной задаче.

5. *Эффективность алгоритма.* Под эффективностью алгоритма будем·понимать такое его свойство (качество), которое позволяет решить задачу за приемлемое для разработчика время [1].

Эффективность программы имеет 3 основные составляющие:

1. пространственная эффективность (память);
2. временная эффективность;
3. интеллектуальная сложность.

Пространственная эффективность измеряется количеством памяти, требуемой для выполнения программы.

Компьютеры обладают ограниченным объемом памяти. Если две программы реализуют идентичные функции, то та, которая использует меньший объем памяти, характеризуется большей пространственной эффективностью. Иногда память становится доминирующим фактором в оценке эффективности программ. Однако в последние годы в связи с быстрым ее удешевлением эта составляющая эффективности постепенно теряет свое значение.

Временная эффективность программы определяется временем, необходимым для ее выполнения.

Лучший способ сравнения эффективностей алгоритмов состоит в сопоставлении их порядков сложности. Этот метод применим как к временной, так и пространственной сложности. Порядок сложности алгоритма выражает его эффективность обычно через количество обрабатываемых данных.

Например, некоторый алгоритм может существенно зависеть от размера обрабатываемого массива. Если, скажем, время обработки удваивается с удвоением размера массива, то порядок временной сложности алгоритма определяется как размер массива.

Порядок алгоритма – это функция, доминирующая над точным выражением временной сложности [2].

О-сложность алгоритмов

Понятие *О-сложности алгоритмов* введено для того, чтобы измерять скорость роста функции в зависимости от входных данных.

Пусть даны две функции из натуральных положительных чисел f, g, которые показывают время работы двух разных алгоритмов на различных длинах входов.

Оценка роста функции сверху

$f = O(g)$, если $\exists C>0\ \forall n \in N: f(n) \le Cg(n)$

при этом говорят, что *f растёт не быстрее, чем g*, или с некоторыми оговорками $f \le g$ с точностью до константы.

Оценка роста функции снизу

$f = \Omega(g)$, если $g = O(f)$

f растёт не медленнее, чем g, можно рассматривать как аналог $f \ge g$.

Одинаковый порядок роста функции

$f = \theta(g)$, если $f = O(g)$, $g = O(f)$,

тогда говорят, что *f и g имеют один порядок роста* и *f и g растут с одинаковой скоростью.*

Таблица 1. О-сложности алгоритма

Обозначение	***Определение***
O(1)	Константная сложность. Большинство операций выполняется только раз или несколько раз
O(N)	Линейная сложность алгоритма. Время работы программы линейно. Обычно, когда элемент входных данных требуется обработать лишь линейное число раз
$O(N^2)$, $O(N^3)$, $O(N^a)$	Полиномиальная сложность алгоритмов

Окончание табл. 1

O(log(N))	Логарифмическая сложность алгоритма. Когда время работы программы логарифмировано, программа начинает работать намного медленнее с увеличением N. Такое время работы встречается обычно в программах, которые делят большую проблему на малые и решают их по отдельности
O(N*log(N))	Такое время работы имеют программы, которые делят большую проблему на маленькие, а затем, решив их отдельно, соединяют вместе
$O(2^N)$	Экспоненциальная сложность Такие алгоритмы возникают чаще всего в результате подхода «метод грубой силы»

Программист должен уметь проводить анализ алгоритмов и определять их сложность. Временная сложность алгоритма может быть посчитана исходя из анализа его управляющих структур.

Алгоритмы без циклов и рекурсивных вызовов имеют константную сложность. Если нет рекурсии и циклов, все управляющие структуры могут быть сведены к структурам константной сложности. Следовательно, и весь алгоритм также характеризуется константной сложностью.

Таблица 2. Сложность управляющих структур

Вид управляющей структуры	***Сложность***
Присваивание	O (1)

Окончание табл. 2

Простое выражение	O (1)
Вычисление S1, S2	Доминанта для O (вычисление 1) и O (вычисление 2)

IF условие THEN
S1
ELSE
S2
	Доминанта O (вычисление 1),
	O (вычисление 2) и O (вычисление условия)

FOR i:=1 to N do S1 End	O (N * вычисление 1)

Определение сложности алгоритма в основном сводится к анализу циклов и рекурсивных вызовов.

Например, рассмотрим алгоритм обработки элементов массива.

```
For i:=1 to N do
Begin
  ...
End;
```

Сложность этого алгоритма O(N), т.к. тело цикла выполняется N раз, и сложность тела цикла равна O(1).

Если один цикл вложен в другой и оба цикла зависят от размера одной и той же переменной, то вся конструкция характеризуется квадратичной сложностью.

```
For i:=1 to N do
For j:=1 to N do
Begin
...
End;
```

Сложность этой программы $O(N^2)$.

Как правило, около 90% времени работы программы требует выполнение повторений и только 10% составляют непосредственно вычисления.

Анализ сложности программ показывает, на какие фрагменты выпадают эти 90% – это циклы наибольшей глубины вложенности. Повторения могут быть организованы в виде вложенных циклов или вложенной рекурсии.

Эта информация может использоваться программистом для построения более эффективной программы следующим образом.

Прежде всего можно попытаться сократить глубину вложенности повторений.

Затем следует рассмотреть возможность сокращения количества операторов в циклах с наибольшей глубиной вложенности. Если 90% времени выполнения составляет выполнение внутренних циклов, то 30%-ное сокращение этих небольших секций приводит к 90%*30%=27%-му снижению времени выполнения всей программы.

Это наиболее простой пример.

Анализом эффективности алгоритмов занимается отдельный раздел математики и найти наиболее оптимальную функцию бывает не так-то и просто [2].

Некоторые алгоритмы хорошо изучены в том смысле, что известны точные математические формулы для среднего и худшего случаев. Такие формулы разрабатываются посредством тщательного изучения программ с целью нахождения времени работы в терминах математических характеристик, и затем производя их математический анализ.

Несколько важных причин такого рода анализа:

1. Программы, написанные на языке высокого уровня, транслируются в машинные коды, и понять сколько времени потребуется для выполнения того или иного оператора может быть трудно.

2. Многие программы очень чувствительны к входным данным, и их эффективность может очень сильно от них зависеть. Средний случай может оказаться математической фикцией не связанной с теми данными, на которых программа используется, а худший случай маловероятен.

О-анализ сложности получил широкое распространение во многих практических приложениях. Тем не менее необходимо четко понимать его ограниченность.

К основным недостаткам подхода можно отнести следующие:

— для сложных алгоритмов получение О-оценок, как правило, либо очень трудоемко, либо практически невозможно;

— часто трудно определить сложность «в среднем»;

— О-оценки слишком грубые для отображения более тонких отличий алгоритмов;

— О-анализ дает слишком мало информации (или вовсе ее не дает) для анализа поведения алгоритмов при обработке небольших объемов данных.

Определение сложности в О-обозначениях – далеко нетривиальная задача. В частности, эффективность двоичного поиска определяется не глубиной вложенности циклов, а способом выбора каждой очередной попытки [2].

Еще одна сложность – определение «среднего случая». Обычно сделать это достаточно трудно из-за невозможности предсказания условий работы алгоритма. Иногда алгоритм используется как фрагмент большой, сложной программы. Иногда эффективность работы аппаратуры и/или операционной системы, или некоторой компоненты компилятора существенно влияет на сложность алгоритма. Часто один и тот же алгоритм может использоваться в множестве различных приложений.

Из-за трудностей, связанных с проведением анализа временной сложности алгоритма «в среднем», часто приходится довольствовать-

ся оценками для худшего и лучшего случаев. Эти оценки по сути определяют нижнюю и верхнюю границы сложности «в среднем».

Основным недостатком О-функций является их чрезмерная грубость. Если алгоритм А выполняет некоторую задачу за 0,001*N с, в то время как для ее же решения с помощью алгоритма В требуется 1000*N с, то В в миллион раз медленнее, чем А. Тем не менее А и В имеют одну и ту же временную сложность O(N).

Кроме временной и пространственной сложности алгоритмов не следует забывать и интеллектуальную сложность.

При анализе *интеллектуальной сложности алгоритма* исследуется понятность алгоритмов и сложность их разработки.

Все три формы сложности обычно взаимосвязаны. Как правило, при разработке алгоритма с хорошей временной оценкой сложности приходится жертвовать его пространственной и/или интеллектуальной сложностью [2].

Например, алгоритм быстрой сортировки существенно быстрее, чем алгоритм сортировки выборками. Плата за увеличение скорости сортировки выражена в большем объеме необходимой для сортировки памяти. Необходимость дополнительной памяти для быстрой сортировки связана с многократными рекурсивными вызовами.

Алгоритм быстрой сортировки характеризуется также и большей интеллектуальной сложностью по сравнению с алгоритмом сортировки вставками.

Если предложить сотне людей отсортировать последовательность объектов, то вероятнее всего, большинство из них используют алгоритм сортировки выборками. Маловероятно также, что кто-то из них воспользуется быстрой сортировкой.

Причины большей интеллектуальной и пространственной сложности быстрой сортировки очевидны: алгоритм рекурсивный, его достаточно трудно описать, алгоритм длиннее (имеется в виду текст программы), чем более простые алгоритмы сортировки.

Рассмотрим четыре алгоритма решения одной и той же задачи, имеющие сложности log n, n, n^2 и 2^n соответственно. Предположим, что второй из этих алгоритмов требует для своего выполнения на некотором компьютере при значении параметра $n=10^3$ ровно одну минуту времени. Тогда времена выполнения всех этих четырех алгоритмов на том же компьютере при различных значениях параметра будут примерно такими, как в таблице 3.

Таблица 3. Сравнительная таблица времен выполнения алгоритмов

	n = 10	**n = 10^3**	**n = 10^6**
log n	0,2 с	0,6 с	1,2 с
N	0,6 с	1 мин.	16,6 ч.
n^2	6 с	16,6 ч.	1902 года
2^n	1 мин.	10^{235} лет	10^{300000}лет

Когда начинающие программисты тестируют свои программы, то значения параметров, от которых они зависят, обычно невелики. Поэтому даже если при написании программы был применен неэффективный алгоритм, это может остаться незамеченным. Однако, если подобную программу попытаться применить в реальных условиях, то ее практическая непригодность проявится незамедлительно.

С увеличением быстродействия компьютеров возрастают и значения параметров, для которых работа того или иного алгоритма завершается за приемлемое время. Таким образом, увеличивается среднее значение величины N, и, следовательно, возрастает величина отношения времен выполнения быстрого и медленного алгоритмов. Чем медленее компьютер, тем больше относительный проигрыш при использовании плохого алгоритма [2].

Правила для определения сложности

1. Мультипликативные константы можно опускать
Например, $5n^3 = O(n^3)$
2. n^a растёт быстрее, чем n^b, при $a > b$
Например, $n^2 = O(n^{2,1})$
3. Любая экспонента растёт быстрее любого полинома
Например, $n^5 = O(2^n)$ *или* $n^{1000} = O(1{,}01^n)$
4. Любой полином растёт быстрее полилогарифма.
Например, $(\log_2 n)^{20} = O(\sqrt{n})$

ЛАБОРАТОРНАЯ РАБОТА № 1.
АЛГОРИТМЫ СОРТИРОВКИ

Цель: Изучить несколько алгоритмов сортировки и сравнить их свойства.

Задача: Написать программу, реализующую два заданных алгоритма сортировки. Сравнить скорость работы алгоритмов на массивах различной длины.

Краткая теория

Если над некоторым множеством элементов определена операция сравнения, то такое множество может быть упорядочено. Для упорядочивания наборов данных, представленных в виде массивов, существуют различные алгоритмы. Алгоритмы классифицируются по различным характеристикам. В частности, по зависимости количества необходимых операций от размера входных данных, называемой вычислительной эффективностью (или просто быстродействием). В данной лабораторной работе рассматриваются примеры алгоритмов, обладающих квадратичной и логарифмической вычислительной эффективностью (такие алгоритмы принято называть быстрыми).

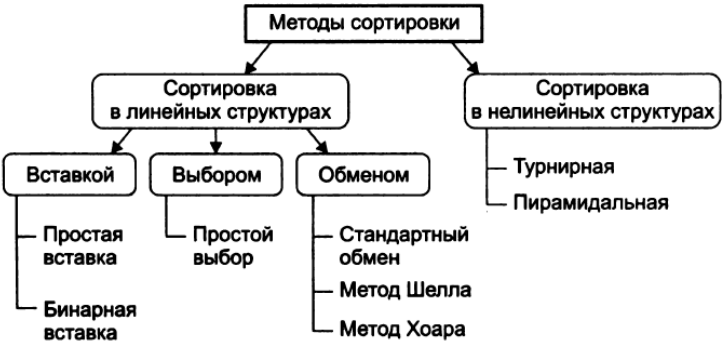


Рис. 1.1. Классификация методов сортировки

Алгоритм сортировки кучей

Сортировка кучей, пирамидальная сортировка (англ. Heapsort) – алгоритм сортировки, использующий структуру данных двоичная куча. Это неустойчивый алгоритм сортировки с временем работы O(n*logn), где n – количество элементов для сортировки, и использующий O(1) дополнительной памяти [3].

Будем говорить, что массиву соответствует бинарное дерево и массив является кучей, если в каждой вершине дерева выполняется следующее свойство:

$\forall i \quad A[i] \geq A[leftchild(i)]$

$\forall i \quad A[i] \geq A[rightchild(i)]$

Например, задан массив

10	3	5	4	3	2	1	10	2

Алгоритм сортировки кучей состоит из следующих шагов:

1) строим из массива бинарное дерево;

2) находим максимальный элемент и меняем его с последним элементом. После этого забываем про этот элемент и рассматриваем кучу на n – 1 элементах.

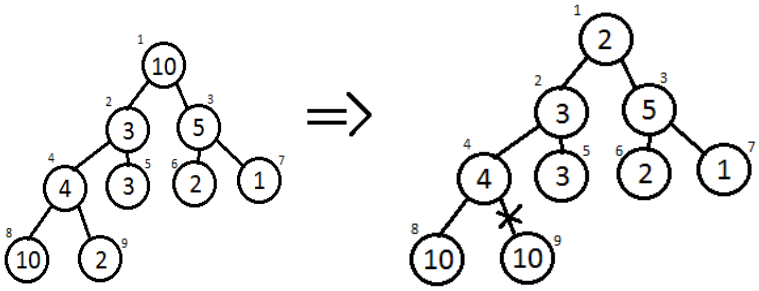


Рис. 1.2. Начало алгоритма сортировки кучей

3) проверяем, нарушилось ли в первой вершине свойство кучи. Если да, то выполняем процедуру «утапливания» элемента.

Ключевой момент – делать эту процедуру придется не больше log n раз, т.к. глубина дерева log n.

Heapify (A,i) – процедура утапливания элементов. Эта процедура работает в предположениях, что поддеревья left(i) и right(i) являются кучами.

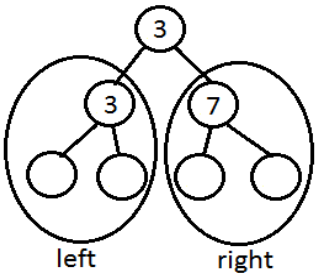


Рис. 1.3. Проверка выполнения свойства кучи в вершине

```
Heapify(A,i)
max index = i
l = left(i)
r = right(i)
if (l ≤ size & A[max index] < A[l])
   max index = l
if (r ≤ size & A[max index] < A[r])
   max index = r
if (max index ≠ i)
   swap (A, i, max index)
Heapify(A, max index)
```

Heapify будет работать время, пропорциональное высоте дерева. Процедура построения кучи будет иметь следующий вид:

```
BuildHeapify(A)
for I = n/2 down to 1
   Heapify(i)
HeapSort(A)
BuildHeap(A)
size < n
while size > 1
   swap(A, 1, size)
size = size – 1
Heapify(A,1)
```

Сортировка слиянием

Сортировка слиянием построена на глобальном принципе программирования «разделяй и властвуй», который заключается в рекурсивном разбиение решаемой задачи на две или более подзадач того же типа, но меньшего размера, и комбинировании их решений для получения ответа к исходной задаче. Разбиения выполняются до тех пор, пока все подзадачи не окажутся элементарными [3].

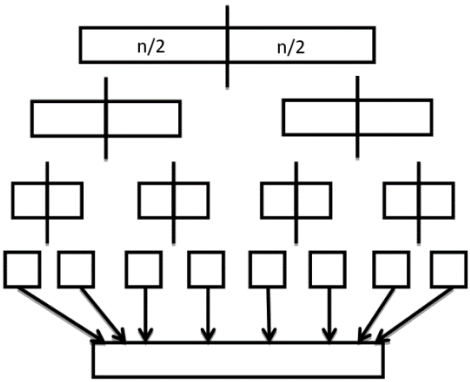


Рис. 1.4. Схема работы алгоритма сортировки слиянием

```
// a – сортируемый массив, lb – левая граница массива, rb – правая
template<class T>
void mergeSort(T a[], long lb, long rb) {
```

```
long split;                    // индекс, по которому делится массив
if (lb < rb)                          // если есть более 1 элемента
  split = (lb + rb)/2;
  mergeSort(a, lb, split);     // сортировать левую половину
  mergeSort(a, split+1, last); // сортировать правую половину
  merge(a, lb, split, rb);     // слить результаты в общий массив
```

Функция merge на месте двух упорядоченных массивов a[lb]...a[split] и a[split+1]...a[rb] создает единый упорядоченный массив a[lb]...a[rb].

Рекурсивный алгоритм обходит получившееся дерево слияния в прямом порядке. Каждый уровень представляет собой проход сортировки слияния – операцию, полностью переписывающую массив.

Деление происходит до массива из единственного элемента. Такой массив можно считать упорядоченным, а значит, задача сводится к написанию функции слияния merge.

Оценим сложность алгоритма. Время работы определяется рекурсивной формулой:

$T(n) = 2T(n/2) + O(n)$.

$T(n) = O(n \log n)$

Плюс сортировки слиянием заключается в отсутствие «худшего случая», что обеспечивает устойчивость метода. Однако, несмотря на хорошее общее быстродействие, у сортировки слиянием есть и серьезный минус: она требует дополнительной памяти.

Сортировка слиянием является одним из наиболее эффективных методов для односвязных списков и файлов, когда есть лишь последовательный доступ к элементам [3].

Быстрая сортировка

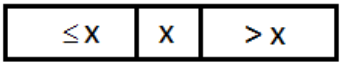
Быстрая сортировка является наиболее широко применяемым и одним из самых эффективных алгоритмов. Хотя время его работы в худшем случае составляет $O(n^2)$, на практике он является самым

быстрым. Алгоритм быстрой сортировки относится к вероятностным алгоритмам. Поэтому время работы алгоритма в среднем определяется его математическим ожиданием и составляет O (n*logn). Причем множитель n*logn довольно мал. Кроме того, быстрая сортировка относится к классу сортировок на месте (in-place sort), то есть не требует дополнительной памяти [4].

Быстрая сортировка также, как и сортировка слиянием, основана на подходе «разделяй и властвуй». Ее алгоритм состоит из следующих шагов:

1. Из массива случайным образом выбирается некоторый опорный элемент x.

2. Запускается процедура разделения массива, которая перемещает все ключи, меньшие, либо равные x, влево от него, а все ключи, большие, либо равные x – вправо.



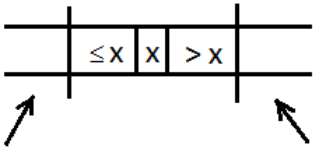
3. Теперь массив состоит из двух подмножеств, причем левое меньше, либо равно правого.

4. Для обоих подмассивов: если в подмассиве более двух элементов, рекурсивно запускаем для него ту же процедуру.

5. В конце получится полностью отсортированная последовательность.

Процедура Partition

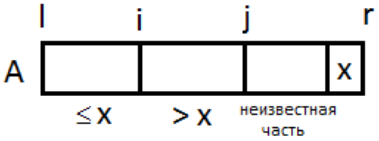
```
QuickSort(A, l, r)
if (l ≥ r)
  return
m = Partition(A, l, r)   //процедура присвоения индекса
QuickSort(A, l, m-1)
QuickSort(A, m+1, r)
```



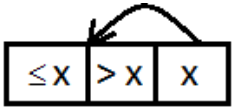
Выбираем опорный элемент x и начинаем последовательно «наращивать» два региона, пока будут выполняться следующие условия:

$\forall\ l \le k \le i : A[k] \le x$

$\forall\ i + 1 \le k \le j : A[k] > x$



После этого перемещаем опорный элемент на место.




```
Partition(A, l, r)
   i = l – 1
   j = l – 1
for j = l to r
   if A[j] ≤ A[r]
   swap(A, j, i + 1)
   i = i + 1
return i
```

Отличие алгоритма быстрой сортировки от сортировки слиянием в том, что в слиянии куски массива делятся на размер n/2, а размер кусков быстрой сортировки мы не знаем.

$T(n) = O(n) + T(n-1) \Rightarrow O(n^2)$ – худший случай быстрой сортировки.

$T(n) = O(n) + T(n/2) + T(n/2) => O(n*\log n)$

В действительности вне зависимости от выбора опорного элемента на каждом шаге время будет логарифмично. Для этого нам необходимо разбивать массив под константное соотношение.

swap(A, r, rand(l,r)) – выбор случайного элемента

Так как QuickSort вероятностный алгоритм, на одном и том же массиве он будет работать разное время. Время его работы так же случайно, поэтому мы будем оценивать его среднюю величину или математическое ожидание.

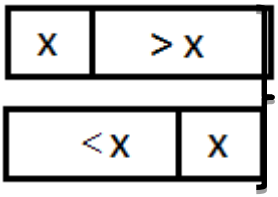
Математическое ожидание время работы алгоритма быстрой сортировки $O(n*\log n)$.

Говоря о среднем времени, мы имеем ввиду не усреднение по входам, а математическое ожидание времени работы, которое для любого входа не превосходит $O(n*\log n)$.

То есть время работы быстрой сортировки будет зависеть не от входного массива, а от выбора опорного элемента.

Перейдем к анализу сложности быстрой сортировки. Для упрощения в доказательстве мы будем рассматривать массив с неповторяющимися числами.

«Неприятной» ситуацией будет называть ситуацию, когда в качестве опорного элемента выпадает минимальный или максимальный элемент массива.



В этом случае время работы: $T(n) = T(n-1) + O(n) = O(n^2)$

«Приятная» ситуация, когда опорный элемент является медианой массива. Тогда время работы алгоритма:

$T(n) = 2T(n/2) + O(n) = O(n*\log n)$

Время работы алгоритма будет зависеть от количества сравнений элементов с опорным. То есть количество сравнений это и есть какая-то случайная величина, которую мы будем оценивать.

QuickSort вызывает процедуру Partition, после чего два раза рекурсивно вызывает себя. Partition зависит от количества сравнений, а количество сравнений пропорционально длине отрезка r-l. Далее мы забудем, что сравнения производятся в Partition, т.к. в одном случае их мало, а в другом много, в зависимости от выбора опорного элемента. Мы будем оценивать количество сравнений в среднем, то есть их математическое ожидание.

Если рассмотреть один запуск быстрой сортировки, то время его работы будет случайной величиной.

$E(\text{время работы}) = O(E(\#\text{сравнений})) = O(\sum_{(1 \le i \le j \le n)} P\{A[i] \text{ сравнился с } A[j]\})$

При сравнении пары элементов во время работы в зависимости от того, как выбран опорный элемент, некоторые из них могут не сравниваться. *Никакие два элемента в течение работы алгоритма больше одного раза не сравнятся*.

Так как Partition сравнивает все элементы с опорным, после чего опорный элемент оказывается на месте и в процедуре больше не участвует.

Математическое ожидание количества сравнений равно сумме вероятностей сравнений каждой конкретной пары элементов, т.е. какое-то число от 0 до 1.

$O(\sum_{(1 \le i \le j \le n)} P\{A[i] \text{ сравнился с } A[j]\}) = O(\sum_{(1 \le i \le j \le n)} P\{A'[i] \text{ сравнился с } A'[j]\})$,

где A’[] –отсортированная копия массива A.

Мы вправе поставить знак равенства между этими массивами, т.к. в обоих случаях используются одни и те же пары элементов, расставленные в другом порядке.

Нас интересует опорный элемент, который попадает в отрезок от i до j. Как только он попадет в этот отрезок, отработает процедура Partition. При этом элементы i и j разойдутся в разные стороны и больше никогда не встретятся. Элементы i и j сравнятся только в том случае, если один из них будет выбран в качестве опорного.

P(A'[i] сравнился с A'[j]) = P{первый опорный элемент отрезка A'[i..j] выбран A'[i] или A'[j]} = 2/(j-i+1)

Нас интересует, какой из двух элементов был выбран первым – i или j. Этих элементов будет j-i+1, так как за секунду до того как выбран опорный элемент, опорным может быть каждый.

Вероятность того, что опорным стал элемент, равна 1/(j-i+1), у нас два элемента и они независимы, поэтому вероятность равна 2/(j-i+1).

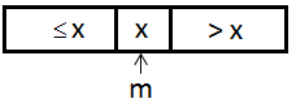
$O(\sum_{(1 \le i \le j \le n)} 2/j-i+1) = O(\sum_{(1 \le i \le n)}\sum_{(1 \le j \le n)} 2/(j-i+1)) = O(n*\log n)$

Для каждого i сумма будет не больше чем logn – это гармоническая сумма. Следовательно мы доказали, что время работы процедуры Partition $T(n) = 2T(n/2) + O(n) = O(n*\log n)$.

```
QuickSort(A, l, r)
while(l < r)
  m = Partition(A, l, r)
QuickSort(A, l, m-1)
QuickSort(A, m+1, r)
l = m+1
```

Далее нам необходимо определить, сколько вызовов Partition будет сделано за время работы QuickSort.

Пусть есть массив длиной n, дано число k и нужно выдать число, которое будет стоять на k-ом месте в отсортированном массиве.



Вызываем процедуру Partition.

1. Если k = m, то сразу выдаем элемент, потому что Partition ставит x на свое место.
2. Если k < m, вызываем рекурсию для левой части.
3. Если k > m, вызываем рекурсию для правой части.

Процедура Partition отработает только на одной части массива. В этом и заключается ее разница с QuickSort. Мы хотим оценить математическое ожидание времени работы алгоритма, и оно будет зависеть от того, насколько хорошо будет разбиваться массив.

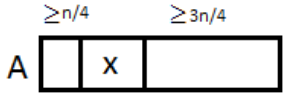
$T(n) = O(n) + T(n-1) => n^2$

$T(n) = O(n) + T(/2) => n$

$C(n) + C(n/2) + C(n/4)$

Рассмотрим отсортированный массив A’[].

Все элементы, попавшие в промежуток от ¼ до ¾ размера массива, будем называть «хорошими», т.к. они делят массив в «хорошем» отношении.



Если x попадет в выбранный промежуток = > 3n/4 – количество рекурсивных вызовов, а количество «хороших» элементов n/2. Тогда

$E(n) \le E(3n/4) + O(n)$

Однако мы не можем гарантировать, что с первого раза опорный элемент таким образом разобьет массив. Тогда время работы QuickSort будет включать в себя время работы Partition и математическое

ожидание времени, которое пройдет до того, пока массив не станет размера ¾.

Вероятность попадания опорного элемента в «хорошие» элементы равна ½. Это можно сравнить с ситуацией подбрасывания монетки и выпадение на ней «орла».

$E = \sum_{(от\ 1\ до\ \infty)} iP\{выпал\ орел\ на\ i\text{-}ом\ подбрасывании\} =$

$\sum_{(от\ 1\ до\ \infty)} i*1/2^i$

При разложении данной суммы мы получим, что уже через 2 выбора опорного элемента мы попадем в «хорошие» элементы [4] и соответственно массив уменьшится до размера 3n /4.

Следовательно, время работы быстрой сортировки:

$T(n) = 2T(n/2) + O(n) = O(n*logn)$.

Задание на лабораторную работу:

1. Написать две функции сортировки массива целых чисел, реализующих заданные алгоритмы сортировки – один из класса квадратичных алгоритмов, другой из класса быстрых алгоритмов.

2. Исследовать вычислительную эффективность этих алгоритмов. Для этого необходимо использовать функции стандартной библиотеки, которые позволяют измерять время работы участка программы. Программа для измерения вычислительной эффективности должна создавать массив для сортировки, заполняя его случайными числами, потом вызывать функцию сортировки, замеряя время её работы. Если время одной сортировки слишком мало, надо проводить генерацию случайного массива и его сортировку в цикле большое число раз и измерять суммарное (либо среднее) время сортировки. Результатом данного пункта должны быть времена выполнения сортировки обоими методами для размеров массивов примерно от 1000 до 100000 (6-7 значений размера).

3. Для измерения времени работы программы можно воспользоваться функцией clock( ) (заголовочный файл time.h), которая при вы-

зове возвращает значение, равное количеству «тиков» системных часов, прошедших от начала работы программы до момента вызова. В time.h также определена константа для перевода «тиков» в секунды.

Варианты заданий:

1. Шейкер-сортировка и пирамидальная сортировка (heapsort).
2. Шейкер-сортировка и сортировка слиянием.
3. Шейкер-сортировка и быстрая сортировка (quicksort).
4. Пузырьковая сортировка и сортировка слиянием.
5. Пузырьковая сортировка и быстрая сортировка (quicksort).

ЛАБОРАТОРНАЯ РАБОТА № 2.
РЕКУРСИЯ

Цель работы: познакомиться с одним из эффективных способов решения сложных задач – рекурсией.

Краткая теория

Очень часто, разрабатывая программу, удается свести исходную задачу к более простым. Среди этих задач может оказаться и первоначальная, но в упрощенной форме. Например, вычисление функции F(n) может потребовать вычисления F(n-1) и еще каких-то операций. Иными словами, частью алгоритма вычисления функции будет вычисление этой же функции.

Алгоритм называется *рекурсивным*, если он прямо или косвенно обращается к самому себе. Часто в основе такого алгоритма лежит рекурсивное определение какого-то понятия. Например, о факториале числа N можно сказать, что N! = N*(N – 1)!, если N > 0 и N! = 1 если N = 0. Это – рекурсивное определение.

Еще одно определение может быть таким: 3 коровы – это стадо коров. Стадо из n коров – это стадо из n – 1 коровы и еще одна корова.

Попробуем применить это определение для проверки, является ли стадом группа из пяти коров (обозначим ее К5). Объект К5 не удовлетворяет первому пункту определения, поскольку пять коров – это не три коровы. Согласно второму пункту К5 – стадо, если там есть одна корова, а остальная часть К5, назовем ее К4, – тоже стадо коров. Решение относительно объекта К5 откладывается, пока не будет принято решение относительно К4. Объект К4 снова не подходит под первый пункт, а второй пункт гласит, что К4 – стадо, если объект К3, полученный из К4 путем отделения одной коровы, тоже стадо. Решение о К4 тоже откладывается. Наконец, объект К3 удовлетворяет пер-

вому пункту определения, и мы можем смело утверждать, что К3– стадо коров. Теперь и о К4 можно утверждать, что это стадо, а значит, и К5 является стадом коров.

Любое рекурсивное определение состоит из двух частей. Эти части принято называть базовой и рекурсивной частями. Базовая часть является нерекурсивной и задает определение для некоторой фиксированной части объектов. Рекурсивная часть определяет понятие через него же и записывается так, чтобы при цепочке повторных применений она редуцировалась бы к базе [5].

Пример 1. Написать рекурсивную программу поиска минимального элемента массива.

Решение. Опишем функцию P_{min}, которая определяет минимум среди первых n элементов массива a. Параметрами этой функции является количество элементов в рассматриваемой части массива – n и значение последнего элемента этой части – a[n]. При этом если n>2, то результатом является минимальное из двух чисел – a[n] и минимального числа из первых (n-1) элементов массива. В этом заключается рекурсивный вызов. Если же n=2, то результатом является минимальное из первых двух элементов массива. Чтобы найти минимум всех элементов массива, нужно обратиться к функции P_{min}, указав в качестве параметров значение размерности массива и значение последнего его элемента. Минимальное из двух чисел определяется с помощью функции Min, параметрами которой являются эти числа [5].

```
Program Example _1;
Const n=10;
Type MyArray=Array[1..n] of Integer;
Const a : MyArray = (4,2, -1,5,2,9,4,8,5,3);
Function Min (a, b : Integer) : Integer;
Begin
if a>b then Min := b else Min:=a;
End;
```

```
Function Pmin(n, b : Integer) : Integer;
Begin
if n = 2 then Pmin := Min(n,a[1]) else Pmin := Min(a[n], Pmin(n-1,a[n]));
End;
BEGIN
Writeln('Минимальный элемент массива  - ', Pmin(n,a[n]));
END.
```

Пример 2. Ханойские башни. Имеется три стержня A, B, C. На стержень A нанизано n дисков радиуса 1, 2,..., n таким образом, что диск радиуса i является i-м сверху. Требуется переместить все диски на стержень B, сохраняя их порядок расположения (диск с большим радиусом находится ниже). За один раз можно перемещать только один диск с любого стержня на любой другой стержень. При этом должно выполняться следующее условие: на каждом стержне ни в какой момент времени никакой диск не может находиться выше диска с меньшим радиусом.

Решение. Предположим, что мы умеем перекладывать пирамиду из (n-1) диска. Рассмотрим пирамиду из n дисков. Переместим первые (n-1) дисков на стержень C (это мы умеем). Затем перенесем последний n-й диск со стержня A на стержень B. Далее перенесем пирамиду из (n-1) диска со стержня C на стержень B. Так как n-й диск самый большой, то условие задачи не будет нарушено. Таким образом, вся пирамида будет на стержне B. Аналогичным образом можно перенести n – 2, n – 3 и т. д. дисков. Когда n=1, осуществить перенос очень просто: непосредственно с первого стержня на второй. При этом для решения задачи будет достаточно 2n - 1 перекладываний [5].

```
Program Example_2;
Const k = 3;
Var a,b,c : Char;
Procedure Disk(n : Integer; a, b, c: Char);
Begin
```

```
if n>0 then
    begin
      Disk(n-1,a,c,b);
Writeln('Диск ',n, ' с ', a,'->', b);
      Disk(n-1,c,b,a);
    end;
End;
BEGIN
a := 'A'; b := 'B'; c := 'C';
  Disk(k,a,b,c);
ReadLn;
END.
```

Варианты заданий:

1. Найдите первые N чисел Фибоначчи двумя способами: с помощью рекурсии и с помощью итерации. Сравните эффективность алгоритмов.

2. Напишите традиционную функцию умножения двух чисел и функцию, использующую только операцию сложения. Сравните эффективность алгоритмов.

3. Напишите метод, находящий максимальное из двух чисел, не используя операторы if-else или любые другие операторы сравнения. Оцените сложность алгоритма.

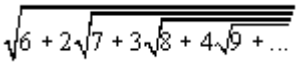
4. Напишите функцию суммирования двух целых чисел без использования «+» и других арифметических операторов. Оцените сложность алгоритма.

5. Вычислите несколько значений функции Аккермана для неотрицательных чисел m и n. Оцените сложность.

$$A(n,m) = \begin{cases} m+1, & n = 0 \\ A(n-1,1), & n \neq 0, m = 0 \\ A(n-1, A(n,m-1)), & n > 0, m \geq 0 \end{cases}$$

6. Вычислите произведение элементов одномерного массива двумя способами: с помощью рекурсии и с помощью итерации. Оцените сложность алгоритма.

7. Вычислите, используя рекурсию, выражение:



ЛАБОРАТОРНАЯ РАБОТА № 3
ПРОСТЫЕ СТРУКТУРЫ ДАННЫХ

Цель: Изучить понятие об абстрактных типах данных и структурах для их реализации в памяти прямого доступа.

Задача: Написать программу, реализующую решение задач, разрешимых за полиномиальное время.

Краткая теория

Очередь

Очередь – абстрактный тип данных с дисциплиной доступа к элементам «первый пришёл – первый вышел» (FIFO, англ. first in, first out). Добавление элемента (принято обозначать словом enqueue – поставить в очередь) возможно лишь в конец очереди, выборка – только из начала очереди (что принято называть словом dequeue – убрать из очереди), при этом выбранный элемент из очереди удаляется [6].

Существует несколько способов реализации очереди в языках программирования.

Первый способ представляет очередь *в виде массива* и двух целочисленных переменных head и tail.

Обычно head указывает на голову очереди, tail – на элемент, который заполнится, когда в очередь войдёт новый элемент. При добавлении элемента в очередь в q[tail] записывается новый элемент очереди, а tail уменьшается на единицу. Если значение tail становится меньше 1, то мы как бы циклически обходим массив, и значение переменной становится равным n. Извлечение элемента из очереди производится аналогично: после извлечения элемента q[head] из очереди переменная head уменьшается на 1. С такими алгоритмами одна ячейка из n всегда будет незанятой (так как очередь с n элементами невозможно отличить от пустой), что компенсируется простотой алгоритмов.

Преимущества данного метода: возможна незначительная экономия памяти по сравнению со вторым способом; проще в разработке.

Недостатки: максимальное количество элементов в очереди ограничено размером массива. При его переполнении требуется перевыделение памяти и копирование всех элементов в новый массив.

Второй способ основан на работе с динамической памятью. Очередь представляется в качестве *линейного списка*, в котором добавление/удаление элементов идет строго с соответствующих его концов.

Преимущества данного метода: размер очереди ограничен лишь объёмом памяти.

Недостатки: сложнее в разработке; требуется больше памяти; при работе с такой очередью память сильнее фрагментируется; работа с очередью несколько медленнее.

Реализация очереди на двух стеках

Методы очереди могут быть реализованы на основе двух стеков S1 и S2, как показано ниже:

```
Процедура enqueue(x):
  S1.push(x)
Функция dequeue():
  если S2 пуст:
    если S1 пуст:
      сообщить об ошибке: очередь пуста
    пока S1 не пуст:
      S2.push(S1.pop())
  вернуть S2.pop()
```

Такой способ реализации наиболее удобен в качестве основы для построения персистентной очереди [6].

Стек

Стек (англ. Stack – стопка) – абстрактный тип данных, представляющий собой список элементов, организованных по принципу LIFO (англ. last in – first out, «последним пришёл – первым вышел»).

Чаще всего принцип работы стека сравнивают со стопкой книжек: чтобы взять вторую сверху, нужно снять верхнюю.

Операции со стеком

Возможны три операции со стеком: добавление элемента (иначе проталкивание, push), удаление элемента (pop) и чтение головного элемента (peek).

При проталкивании (push) добавляется новый элемент, указывающий на элемент, бывший до этого головой. Новый элемент теперь становится головным.

При удалении элемента (pop) убирается первый, а головным становится тот, на который был указатель у этого объекта (следующий элемент). При этом значение убранного элемента возвращается [7].

Списки

Односвязный список (однонаправленный связный список)

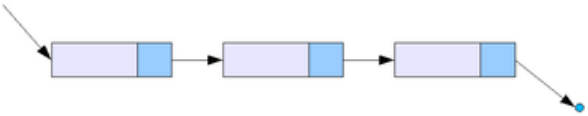


Рис. 3.1. Однонаправленный связный список

Линейный однонаправленный список – это структура данных, состоящая из элементов одного типа, связанных между собой последовательно посредством указателей. Каждый элемент списка имеет указатель на следующий элемент. Последний элемент списка указывает на NULL. Элемент, на который нет указателя, является первым (головным) элементом списка. Здесь ссылка в каждом узле указывает на следующий узел в списке. В односвязном списке можно передвигать-

ся только в сторону конца списка. Узнать адрес предыдущего элемента, опираясь на содержимое текущего узла, невозможно [4].

Двусвязный список (двунаправленный связный список)

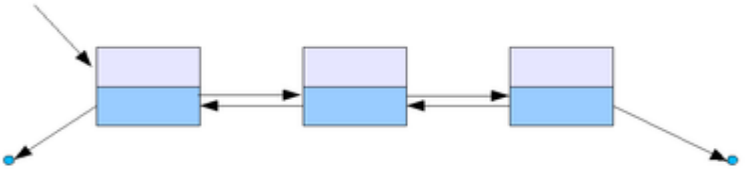


Рис. 3.2. Двунаправленный связный список

Здесь ссылки в каждом узле указывают на предыдущий и на последующий узел в списке. Как и односвязный список, двусвязный допускает только последовательный доступ к элементам, но при этом дает возможность перемещения в обе стороны. В этом списке проще производить удаление и перестановку элементов, так как легко доступны адреса тех элементов списка, указатели которых направлены на изменяемый элемент [4].

Кольцевой связный список

Разновидностью связных списков является кольцевой (циклический, замкнутый) список. Он тоже может быть односвязным или двусвязным. Последний элемент кольцевого списка содержит указатель на первый, а первый (в случае двусвязного списка) — на последний [4].

Рис. 3.3. Кольцевой связный список

Как правило, такая структура реализуется на базе линейного списка. С каждым кольцевым списком дополнительно хранится указатель на первый элемент. В этом списке ссылки на NULL не встречается.

Также существуют циклические списки с выделенным головным элементом, облегчающие полный проход через список [4].

Варианты заданий:

1. Реализуйте «вручную» стек со стандартными функциями push/pop и дополнительной функцией min, возвращающей минимальный элемент стека. Все эти функции должны работать за O(1). Память должна быть оптимальна.

2. Дана строка со скобками. Проверьте правильность расстановки скобок за время O(n).

а) в строке содержатся только круглые скобки;

б) скобки могут быть любые.

3. Дан однонаправленный список с петлёй. Его «последний» элемент содержит указатель на один из элементов этого же списка, причём не обязательно на первый. Найдите начальный узел петли. Элементы списка менять нельзя, память должна быть константна.

4. Дан список с двумя указателями у каждого элемента. Зацикленность списка не допускается. Скопируйте данный список за время O(n) без использования дополнительной памяти. Выделение памяти под все данные одним блоком (как под массив) не допускается, список должен быть разбросанным по частям.

5. Есть однонаправленный список из структур. В нём random указывает на какой-то элемент этого же списка. Напишите функцию, которая копирует этот список с сохранением структуры (т.е. если в старом списке random первой ноды указывал на 4-ю, в новом списке должно быть то же самое – рандом первой ноды указывает на 4-ю ноду нового списка). Сложность алгоритма должна быть O(n), константная дополнительная память плюс память под элементы нового списка.

Выделение памяти под все данные одним блоком (как под массив) не допускается, список должен быть разбросанным по частям.

6. Удалите дубликаты из несортированного связного списка. Память должна быть константна.

7. Разбейте связный список вокруг некоторого значения так, чтобы все меньшие узлы оказались перед узлами, большими или равными этому значению.

8. Найдите в односвязном списке k-ый с конца элемент. Список реализован «вручную», есть только операция получения следующего элемента и указатель на первый элемент. Алгоритм должен быть оптимален по времени и памяти.

9. Задача «Поддержания max в окне». Дан массив размером n и счетчик k, определяющий размер окна в массиве. Окно двигается от начала до конца массива. Необходимо найти максимум в окне и напечатать все их значения. Время работы алгоритма должно быть O(n) и не зависеть от k.

10. Дан массив размера 1 млн. ячеек. Достаньте 100 минимальных элементов, пробегая по массиву менее 100 раз.

11. Дан массив размера n+1. Элементы массива числа из множества {1, 2, 3 … n}. Найдите повторяющееся число за время O(n), не используя дополнительной памяти. Повторяющихся элементов может быть несколько.

12. Дан массив с целыми числами, в том числе и отрицательными. Найдите самое большое произведение трех чисел из этого массива. Например: дан массив, содержащий числа -10, -10, 1, 3, 2. Функция, которая обрабатывает этот массив, должна вернуть 300, так как -10 * -10 * 3 = 300. Время и память должны быть оптимальны.

13. Обнулите столбец N и строку M матрицы, если элемент в ячейке (N, M) нулевой. Затраты памяти и времени работы должны быть минимизированы.

ЛАБОРАТОРНАЯ РАБОТА № 4.
ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Цель: Изучить метод динамического программирования, как способ решения сложных задач путём разбиения их на более простые подзадачи для сокращения времени работы алгоритма.

Задача: Написать программу, реализующую задачу с оптимальной подструктурой, выглядящей как набор перекрывающихся подзадач, сложность которых чуть меньше исходной.

Краткая теория

В оптимизационных задачах может существовать множество различных решений; их «качество» определяется значением какого-то параметра, и требуется выбрать оптимальное решение, при котором значение параметра будет минимальным или максимальным (в зависимости от постановки задачи).

Жадные алгоритмы

Многие задачи оптимизации сравнительно быстро и просто решаются с помощью ***жадных алгоритмов***. Такой алгоритм делает на каждом шаге локально оптимальный выбор, допуская, что итоговое решение также окажется оптимальным. Однако это не всегда так, но для большого числа алгоритмических задач жадные алгоритмы действительно дают оптимальное решение. Нужно отметить, что вопрос о применимости жадного алгоритма в каждом классе задач решается отдельно, однако для обоснования общего решения существует теория матроидов [8].

Рассмотрим схему работы жадных алгоритмов на конкретных примерах.

Задача о выборе заявок

Даны n заявок на проведение занятий в некоторой аудитории. В каждой заявке указаны начало и конец занятия (si и fi для i-й заявки). Заявки с номерами i и j совместны, если интервалы $[si, fi)$ и $[sj, fj)$ не пересекаются (т.е. $fi \le sj$ или $fj \le si$). Задача о выборе заявок состоит в том, чтобы набрать максимальное количество совместных друг с другом заявок.

Приведем жадный алгоритм, решающий данную задачу. При этом полагаем, что заявки упорядочены в порядке возрастания времени окончания.

```
Activity-Selector(s,f)
1  n ← length[s]
2  A ← {1}
3  j ← 1
4  for i ← 2 to n
5  do if si ≥ fj
6  then A ← A ∪ {i}
7  j ← i
8  return A
```

На вход данному алгоритму подаются массивы начала и окончания занятий. Множество A состоит из номеров выбранных заявок, а j — номер последней заявки. Жадный алгоритм ищет заявку, начинающуюся не ранее окончания j-той, затем найденную заявку включает в A, а j присваивает ее номер.

Алгоритм работает за O(n*logn+n), т. е. сортировка плюс выборка. На каждом шаге выбирается наилучшее решение. Покажем, что в итоге получится оптимум.

Доказательство.

Заметим, что все заявки отсортированы по неубыванию времени окончания. Заявка номер 1, очевидно, входит в оптимум (если нет, то заменим самую раннюю заявку в оптимуме на нее, от этого хуже не

станет). Выкинув все заявки, противоречащие первой, получим исходную задачу с меньшим количеством заявок. Рассуждая по индукции, аналогичным образом приходим к оптимальному решению.

Применимость жадного алгоритма

В общем случае нельзя сказать, можно ли получить оптимальное решение с помощью жадного алгоритма применительно к конкретной задаче. Но есть две особенности, характерные для задач, которые решаются с помощью жадных алгоритмов: принцип жадного выбора и свойство оптимальности для подзадач.

Принцип жадного выбора

Говорят, что к задаче оптимизации применим *принцип жадного выбора*, если последовательность локально оптимальных выборов дает глобально оптимальное решение. В этом состоит главное отличие жадных алгоритмов от динамического программирования: во втором просчитываются сразу последствия всех вариантов.

Чтобы доказать, что жадный алгоритм дает оптимум, нужно попытаться провести доказательство, аналогичное доказательству алгоритма задачи о выборе заявок. Сначала мы показываем, что жадный выбор на первом шаге не закрывает путь к оптимальному решению: для любого решения есть другое, согласованное с жадным выбором и не хуже первого. Потом мы показываем, что подзадача, возникшая после жадного выбора на первом шаге, аналогична исходной. По индукции будет следовать, что такая последовательность жадных выборов дает оптимальное решение.

Оптимальность для подзадач

Это свойство говорит о том, что оптимальное решение всей задачи содержит в себе оптимальные решения подзадач. Например, в задаче о выборе заявок можно заметить, что если A — оптимальный набор заявок, содержащий заявку номер 1, то $A' = A \setminus \{1\}$ — оптимальный набор заявок для меньшего множества заявок S', состоящего из тех заявок, для которых $si \geq f1$ [8].

Динамическое программирование

Динамическое программирование применимо в случаях, когда при решении задач:

— необходимо сохранить оптимальность задач;

— есть перекрытие подзадач.

Алгоритм динамического программирования состоит из трех этапов:

1. *Формулировка подзадач*. Если дана последовательность большой размерности, ее необходимо сначала разбить на подзадачи, а затем последовательно увеличивать размерность решаемых подзадач.

2. *Определение способа пересчета*. Если решена 1, 2, … i подзадача, нужно определить, как решать i+1-ую подзадачу.

3. *Определение порядка решения подзадач*. В большинстве случаев порядок решения подзадач простой, но бывают такие случаи, где он не очевиден.

Рассмотрим DAG (directed acyclic graph). Это ориентированный граф без циклов с множеством вершин и ребер.

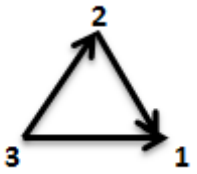


Рис. 4.1. Ориентированный ациклический граф

Свойства этого графа заключаются в том, что все его вершины можно расположить так, что все они будут идти слева направо.

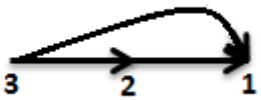


Рис. 4.2. Топологическая сортировка вершин графа, представленного на рис. 4.1

Такая нумерация вершин называется *топологической сортировкой.*

Ориентированный граф является *ациклическим* тогда и только тогда, когда существует топологическая сортировка вершин.

Доказательство:

1) если есть цикл, то топологической сортировки нет;

2) если есть топологическая сортировка, то цикла нет.

Лемма 1. В любом ациклическом графе есть исток и есть сток.

Доказательство:

Допустим, что стока нет, следовательно, из каждой вершины можно куда-то идти. Получили противоречие.

Цикла нет, следовательно, обязательно есть исток. Получается, что все вершины будут справа от истока [4].

Задача о максимальной возрастающей последовательности

Пусть дана последовательность A [1..n]

Например, 6 9 12 4 1 21 30

Шаг 1. Заведем массив D[1..n], где D[j] – длина максимальной возрастающей подпоследовательности, заканчивающаяся на элементе D[j].

A: ***6 9 12*** 4 1 ***21 30***

D: 1 2 3 1 1 4 5

Определив D[j], мы определили подзадачи.

Шаг 2.

```
D[1] = 1
for I = 2 to n            //порядок пересчета задачи
D[i] = 1 + max{D[j]: 1 ≤ j ≤ i-1, A[j] < A[i]}
Return max{D[j]: 1 ≤ j ≤ n}
```

Рассмотрим предыдущий оптимальный элемент в i возрастающей подпоследовательности. Мы не знаем, на каком месте он стоит, но мы точно знаем, что он есть. Это называется оптимальностью на

подзадачах. Отбросив i-ый элемент, оставшаяся подпоследовательность будет так же оптимальной.

Перебираем все возможные кандидаты с 1 до i-1 на то, чтобы быть оптимальным перед i в возрастающей последовательности. Длину оптимальной подпоследовательности мы уже знаем, она сохранена в D, и к ней мы прибавляем 1.

Время работы $O(n^2)$, так как поиск максимума это цикл, получается арифметическая прогрессия.

Замечание. Для D[i] нужно знать несколько предыдущих D[j], и каких именно, зависит от того, что содержится в i-ой ячейке. Это связано с тем, что задачи могут часто перекрываться.

Задача о стоимости редактирования

Стоимость редактирования – это такое число, которое определяется для двух слов и показывает, насколько они похожи. Чем меньше стоимость, тем больше слова похожи.

Есть *три возможные ситуации*:

— напечатан лишний символ (удаление);

— символ пропущен (вставка);

— напечатан другой символ (замена).

Например,

ПОЛИН_ _ ОМЫ

ПАЛИНДРОМ_

Пропуски внутри слова – вставка, пропуск в конце слова – удаление, первое вхождение буквы «О» в первом слове и буквы «А» во втором – замена.

Такое положение называется *выравниванием*, его можно рассматривать, как таблицу. Стоимость данного выражения – 4. Стои-

мость редактирования двух слов – это стоимость их оптимального выравнивания.

```
A[1…p]  B[1…q]
D[0…p, 0…q]   D[i, j]  //стоимость редактирования префиксов
                          A[1…i], B[1…j]
for i = 0 to p
  D[i,0] = i
   for j = 0 to g
```

Рассмотрим пример:

		К	**О**	**Т**
	0	1	2	3
С	1	1	2	3
К	2	1	2	3
О	3	2	1	2
Т	4	3	2	1

Идея такая же, как в предыдущей задаче. На i-ом шаге мы не знаем, какая максимальная стоимость редактирования, но, если убрать i-ый элемент – стоимость редактирования оптимальная.

— при замене D [i-1, j-1]

— при вставке D [i, j-1]

— при удаление D [i-1, j]

Другими словами, нам нужно посмотреть, что находится в клетке с трёх сторон.

```
for i = 1 to p
 for j = 1 to q
    D [i, j] = min { D [i-1, j] + 1;
D [i, j-1] + 1, D [i-1, j-1];
A [i] == B [j]?0*1}
return D[p,q]
```

Для решения данной задачи мы могли завести матрицу, которая бы определила максимальную стоимость распределения для подстрок, но матрица получилось бы четырёхмерной. Этот пример показывает, насколько важно определить подзадачи.

Самое простое применение данной задачи можно найти в редактирование текста в Microsoft Word или словаре T9.

Задача об оптимальном заполнении рюкзака

Есть рюкзак размера W и есть n предметов, у каждого предмета своя стоимость и объём.

Задача заключается в том, чтобы собрать рюкзак так, чтобы предметы в нём имели максимальную стоимость.

Задача с повторениями (предметы могут повторяться)

Пусть есть рюкзак, который заполнен не полностью. Если мы вытащим i-ый предмет, то останется оптимальное заполнение, и рюкзак будет размером $W- w_i$.

```
A [0] = 0
  for w = 1 to w
     A[w] = max A [w-wi]
      w ≥wi+ci
      while A [w]
```

Стратегия выбора предметов, которые имеют наименьший объём и наибольшую стоимость.

Однако эта стратегия не всегда приводит к оптимальному результату. Эта задача относится к классу NP-трудных задач. У неё не может быть оптимального решения.

Сложность алгоритма O (n*W), но это не квадратичный алгоритм. Время работы алгоритма будет зависеть от длины входа экспо-

ненциально. Чтобы записать числа для W, например, числа от 1 до 1000 достаточно 10 бит. Длина входа будет n + logW.

Таким образом, алгоритм, который работает пропорционально числу, которое дали на входе, называется алгоритмом с экспоненциальным временем работы.

Задача без повторений (предметы не повторяются)

Пусть n = 100 предметов, объем рюкзака w = 1000.

A [w, i] – оптимальная стоимость первых i предметов для рюкзака w.

Раньше мы ограничивались только объёмом, а теперь будем постоянно наращивать количество предметов, которые будем использовать.

```
f [a, i] = max {A [w-wi, i-1] + ci, A [w, i-1]}
for i = 1 to n
   for  w = 1 to w
```

Например, w = 10, c_i – стоимость i-го предмета

i	**1**	**2**	**3**	**4**
c	3	2	10	1
w	5	7	4	2

A [5, 1] = max {A [0,0] + 3, A[5, 0]} = 3
A [7, 2] = max {A [0, 1] + 2, A [7, 1]} = 3
A [9, 3] = max {A [9-4, 2] + 10, A [9, 2]}

w\i	**0**	**1**	**2**	**3**	**4**
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0

3	0	0	0	0	0
4	0	0	0	10	10
5	0	3	3	10	10
6	0	3	3	10	10
7	0	3	3	10	10
8	0	3	3	10	10
9	0	3	3	13	13
10	0	3	3	13	13

Варианты заданий:

1. Реализуйте алгоритм поиска максимальной длины убывающей подпоследовательности.

2. Реализуйте алгоритм поиска стоимости редактирования двух слов.

3. Проверьте, являются ли два слова или фразы анаграммами. Два слова являются анаграммами, если путем перестановки букв местами можно получить одно слово из другого.

4. Реализуйте алгоритм, определяющий, является ли одна строка перестановкой другой. Под перестановкой понимаем любое изменение порядка символов. Регистр учитывается, пробелы являются существенными.

5. Реализуйте алгоритм решения задачи об оптимальном заполнении рюкзака с повторениями.

6. Реализуйте алгоритм решения задачи об оптимальном заполнении рюкзака без повторений.

ЛАБОРАТОРНАЯ РАБОТА № 5.
ДЕРЕВЬЯ

Цель: Изучить несколько алгоритмов работы с деревьями и способами их балансировки. Оценить трудоемкость операций.

Краткая теория

Дерево — это иерархическая структура данных, состоящая из элементов (вершин, или узлов), которые связаны между собой отношениями типа «родительская вершина – дочерняя вершина» [9].

Чаще всего дерево изображается в виде графа, вершинами которого являются вершины дерева, а ребрами — его ветви. Начальная вершина дерева, называемая *корнем*, изображается в верхней части графа, и считается, что она находится на нулевом уровне. Вершина Y, расположенная ниже вершины X и соединенная с ней ветвью, называется непосредственным *потомком вершины* X, или ее *дочерней вершиной* (а вершина X, соответственно, — непосредственным *предком вершины* Y, или ее *родительской вершиной*). Если вершина X находится на уровне K, то считается, что все ее непосредственные потомки находятся на уровне K + 1. На графе, изображающем дерево, все вершины одного уровня располагаются на одной горизонтали. Номер максимального уровня дерева называется *глубиной (или высотой) дерева*. Если вершина не имеет потомков, то она называется *терминальной вершиной*, или *листом*. Нетерминальная вершина называется *внутренней*. Число непосредственных потомков внутренней вершины дерева называется *степенью этой вершины*. Максимальная степень всех вершин дерева называется *степенью дерева*. Дерево называется *упорядоченным*, если все непосредственные потомки любой вершины упорядочены [10].

Особым видом деревьев являются *бинарные деревья*. Бинарное дерево с вершинами типа T можно определить следующим образом:

— это либо пустое дерево, не содержащее ни одной вершины;

— либо некоторая вершина типа T, соединенная ветвями с двумя бинарными деревьями с вершинами типа T (эти деревья называются левым и правым поддеревом) [11].

Дерево поиска

Пусть есть объекты произвольной природы, обращение к которым происходит по ключам (key; data). Все ключи различны.

Бинарное дерево поиска

Бинарное дерево поиска – это такое двоичное дерево, в котором, если в вершине хранится x, то в левом поддереве ключи < x, в правом – > x.

Дерево является бинарным деревом поиска, если это свойство выполняется для любой вершины.

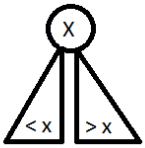


Рис. 5.1. Бинарное дерево поиска

Напишем процедуру обхода дерева, которая будет печатать ключи друг за другом:

```
print(x)
if (x → left ≠ 00)
   print (x → left)
   print (x → key)
if (x → right ≠ 00)
   print (x → right)
```

Она работает за линейное время и печатает ключи в возрастающем порядке.

Операции для работы с деревьями

Find(K) – поиск элемента по ключу.

Сложность – O(n).

Insert(K) – вставка элемента.

Сложность – O(n).

Delete(K) – удаление элемента.

У каждого элемента есть три указателя: на левого ребенка (leftchild), правого ребенка (rightchild) и родителя. При реализации алгоритма на практике меняются не сами вершины, а перенаправляются ссылки. Иначе, если удалить элемент, на который ссылается другой элемент, появится «висячая» (пустая) ссылка.

Алгоритм удаления элемента:

1) листья удаляются за константное время;

2) если у элемента есть 1 ребенок, то элемент удаляется, а ребенок перевешивается на уровень выше;

3) если у элемента 2 ребенка, то удаление происходит в 3 шага:

— y ← min (x → right)

— Delete(x)

— вместо x → y.

Удаление будет работать за высоту, на которой расположен данный элемент, то есть пропорционально логарифму.

Поиск следующего элемента

Если есть правое поддерево, то поиск следующего элемента дерева очевиден.

Если у нас есть ключ k, у которого есть левое поддерево, но нет правого, то k максимальный элемент в этом поддереве. Значит, необходимо подниматься налево на уровень выше до тех пор, пока не попадем в правую ветку. Если ветка не найдена, значит, этот элемент максимальный во всем дереве.

Для поиска предыдущего элемента используется похожий алгоритм. Пусть дан массив. Из него необходимо построить двоичное де-

рево поиска. За линейное время это сделать нельзя, т.к. для этого необходимо отсортировать массив. В лучшем случае бинарное дерево можно построить за n*logn, если дерево будет сбалансированным, а в худшем – за n^2.

Рассмотрим два способа поддержания сбалансированности дерева: ABL – дерево и Splay – дерево.

АВЛ-дерево

АВЛ-дерево – сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на единицу [12].

АВЛ – аббревиатура, образованная первыми буквами фамилий создателей (советских учёных) Адельсон-Вельского Георгия Максимовича и Ландиса Евгения Михайловича [13].

$h(x \to left) - h(x \to right) \leq 1$

Высота такого дерева логарифмическая. Если T – высота АВЛ – дерева с n вершинами, то $h(T) = \theta(\log n)$.

Нам необходимо сделать верхнюю и нижнюю оценку. Нижняя оценка очевидна: $h(T) \geq \log_2 n$

Верхняя оценка: $h(T) \leq c*\log_2 n$

Другими словами, мы хотим доказать, что у дерева большой высоты не может быть мало элементов.

m(h) – минимальное количество элементов в АВЛ – дереве высотой h.

$m(1) = 1$

Т.к. высота дерева h, то одно из поддеревьев будет h-1, а второе как минимум h-2.

Можно написать рекуррентное соотношение:

$m(h) = 1 + m(h-1) + m(h-2)$ или $h = 1 + \max\{h-1, h-2\}$

Если мы посмотрим на вершины, то можно сказать, что количество элементов растет не меньше чем числа последовательности

Фибоначчи, т.е. $m(h) \approx 1.7^h$, а соответственно $m(h) \geq 1.6^h$, т.е. кол-во элементов экспоненциально относительно высоты дерева.

При вставке и удалении баланс дерева будет изменяться на единицу.

Процедура восстановления баланса АВЛ-дерева похожа на процедуру Hepify. И называется малым левым вращением.

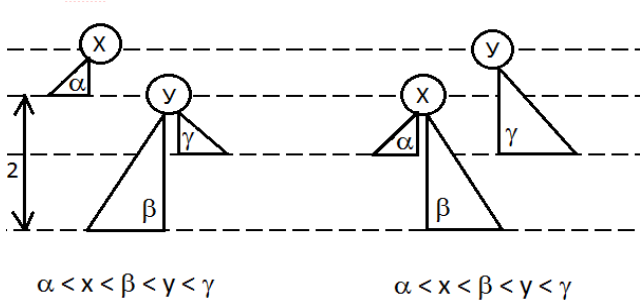


Рис. 5.2. Пример малого левого вращения АВЛ-дерева

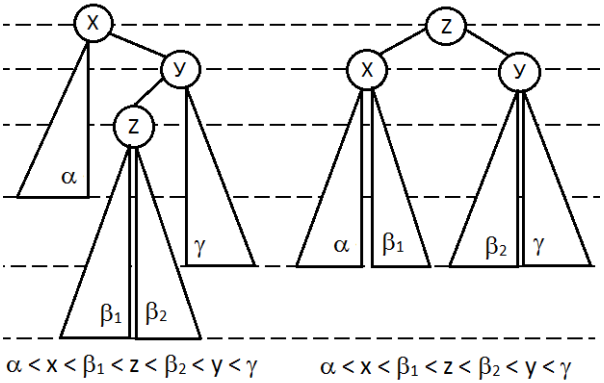


Рис. 5.3. Пример вращения АВЛ-дерева

Splay деревья

Сплей-дерево (англ. Splay-tree) – это двоичное дерево поиска. Оно позволяет находить быстрее те данные, которые использовались недавно. Относится к разряду сливаемых деревьев. Сплей-дерево было придумано Робертом Тарьяном и Даниелем Слейтером в 1983 году [14].

Splay(x) – операция вращения переводит x в корень

Утверждение 1. splay(x) = O(глубина x)

Утверждение 2. Существует такой потенциал φ, который принадлежит $R \geq 0$

1. φ(дерева с n вершинами) ≤ n*logn
2. учетная стоимость spaly(x) = O(logn)

С одной стороны, каждый раз мы отрабатываем за время пропорциональное глубине вершины, с другой – время, которое мы потратим, это суммарная стоимость учетных операций, равная m*logn плюс изменение потенциалов – n*logn

Соответственно общее время будет O(n*logn)

Это значит, что в среднем глубина нашего дерева остается пропорциональна логарифму.

C_i – истинная стоимость операции

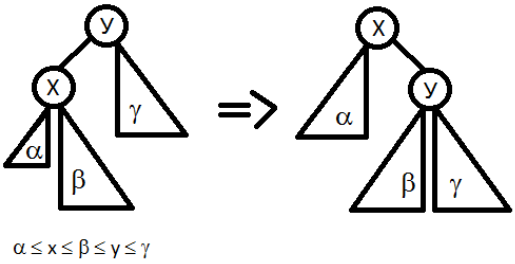
$C'_i = C_i + (\varphi_i - \varphi_{i-1})$ – учетная стоимость

$\sum C'_i = \sum C_i + (\varphi_n - \varphi_0)$

Если бы C_i было примерно одинаковым, то в результате получилось бы C_i*n. В случае, когда C_i разные, нам необходимо сбалансировать значения учетных стоимостей с помощью потенциала.

Splay будет выполняться столько раз, пока x не поднимется в корень. Будем различать три случая:

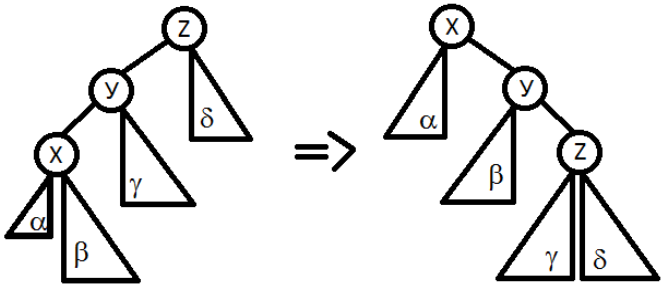
1) y является корнем дерева



$\alpha \le x \le \beta \le y \le \gamma$

Рис. 5.4. Пример Splay-дерева, когда y является корнем

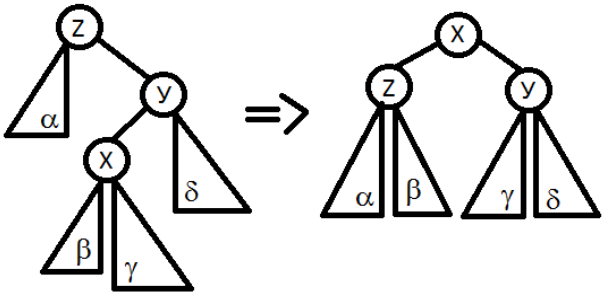
2) y находится в левом поддереве



$\alpha \le x \le \beta \le y \le \gamma \le z \le \delta$

Рис. 5.5. Пример Splay-дерева, когда y находится в левом поддереве

3) y находится в правом поддереве



$\alpha \le z \le \beta \le x \le \gamma \le y \le \delta$

Рис. 5.6. Пример Splay-дерева, когда y находится в правом поддереве

Рассмотрим пример: возьмем разбалансированное дерево и вызовем splay(1), а затем splay(2) (см. рис. 5.7-5.8).

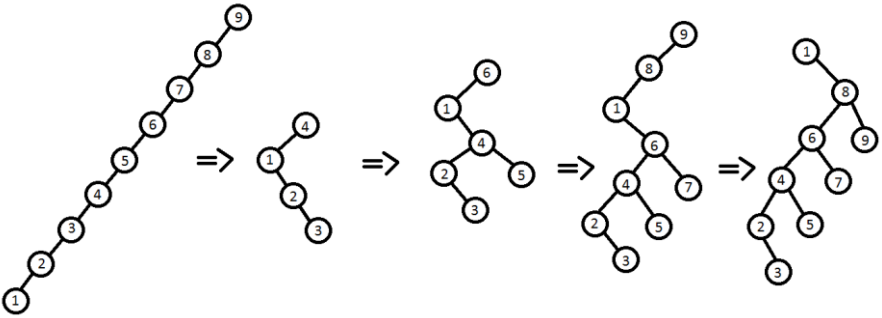


Рис. 5.7. Пример работы функции splay(1)

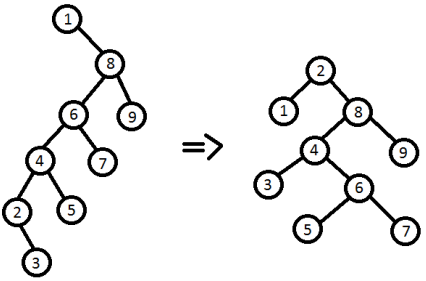


Рис. 5.8. Пример работы функции splay(2)

Нам нужно ввести такие φ, на которых учетная стоимость любой операции splay будет не больше чем какая-то константа, умноженная на логарифм.

ω(x) – вес вершины – количество вершин в поддереве с корнем x.

$a(x) = \lfloor \log_2 \omega(x) \rfloor$

$\varphi(T) = \sum a(x)$

a(x) – для любой вершины принимает значение не больше логарифма.

Лемма 1

Учетная стоимость операции splay(x) не превосходит значение $3(a'(x) - a(x)) + 1$.

Лемма 2

Процедура splay состоит из множества шагов, у каждой локальной операции есть своя учетная стоимость.

— учетная стоимость шага 1 не больше чем $3(a'(x) - a(x)) + 1$;

— учетная стоимость шага 2 и 3 не больше чем $3(a'(x) - a(x))$.

Для каждого шага мы покажем, что учетная стоимость будет не больше, чем $a'(x) - a(x)$.

Доказательство: Учетная стоимость шага равна истинной стоимости шага и разницы потенциалов.

Считаем, что истинная стоимость этого шага равна 1 и смотрим, насколько меняется потенциал. Под x в новом дереве будет то, что было под y в первоначальном дереве.

$a'(y) \le a'(x)$

$1 + (a'(x) + a'(y) - a(x) - a(y)) = 1 + (a'(y) - a(x)) \le 1 + (a'(x) - a(x))$

$\le 1 + 3(a'(x) - a(x))$

Учетная стоимость второго шага должна быть $\le 3(a'(x) - a(x))$

Здесь может измениться потенциал x, y, z.

$\Delta\varphi = (a'(x) + a'(y) + a'(z) - a(x) - a(y) - a(z)) \le 2(a'(x) - a(x))$, т.к. $a'(y), a'(z) \le a'(x)$ и $a(y) \ge a(x)$

Учетная стоимость $C' = 1 + \Delta\varphi \le 1 + 2(a'(x) - a(x)) \le 3(a'(x) - a(x))$.

Если $2(a'(x) - a(x))$ положительно, то неравенство очевидно. Если $2(a'(x) - a(x)) = 0$, тогда, например, во втором случае дерева у x потенциал не изменяется, следовательно $a'(x) = a(x)$, тогда получится $1 + (a'(x) - a(x)) \le 3(a'(x) - a(x))$, то есть $1 + 0 \le 0$.

Докажем, когда $a'(x) = a(x)$, т.е. $\Delta\varphi < 0$.

Оцениваем $\Delta\varphi$:

$a(x) \le a(y) \le a(z)$ $a(x) = a'(x)$

$a'(z) \le a'(y) \le a'(x)$ $a(z) = a'(x)$

$\Delta\varphi = a'(x) + a'(y) + a'(z) - a(x) - a(y) - a(z) =$

$= a'(y) + a'(z) - a(x) - a(y)$

В плохом случае это равно 0, но тогда $a'(y) = a'(z)$ и $a(x) = a(y)$.

Предположим $a(x) = a(y) = a(z) = a'(x) = a'(y) = a'(z) = t$.

У нас получается второй случай дерева.

$\omega(t) = 3 + \omega(\alpha) + \omega(\beta) + \omega(\gamma) + \omega(\delta) = (1 + \omega(\alpha) + \omega(\beta)) + (1 + \omega(\gamma) + \omega(\delta)) + 1 = \omega(x) + \omega'(z) + 1 = 2^t + 2^t + 1$

$a(x) = \lfloor \log_2 \omega(x) \rfloor$

$\omega(x) \ge 2^{a(x)}$

Мы получили $\omega(z) > 2^{t+1}$

$\log_2 \omega(z) \ge t + 1$, а соответственно он не может быть равен t. Получено противоречие [4].

Search

Для найденной вершины мы будем вызывать splay.

В вырожденном дереве поиск происходит за линейное время, но при этом, после того как мы вызывали splay, частые вершины поднимаются наверх. Например, если бы мы искали 1 и не нашли, мы бы подняли наверх 2. И если бы мы стали искать 1 заново, то сразу бы поняли, что ее в дереве нет.

Учетная стоимость операции search равна сумме учетных стоимостей поиска и splay.

Учетная стоимость поиска пропорциональна глубине вершины, а глубина вершины зашита в операции splay, т.к. в splay мы проработаем глубину вершины плюс изменение потенциала. Известно, что в среднем глубина пути плюс изменение потенциала это логарифм. В этом случае мы удваиваем глубину пути, но с точки зрения анализа это все равно логарифм.

Insert

В процедуре добавления элемента мы «подвешиваем» к y вершину x и вызываем splay. Учетная стоимость вставки будет тоже логарифмом. Во-первых, мы опять прошли какую-то глубину, т.е. в splay мы ее уже охватили, плюс после того как мы подвесили x, у нас изменились все веса всех вершин, которые встречаются на пути от корня к x, при этом потенциал изменяется не больше чем на logn.

Потенциал меняется, если вес становится равен степени 2. Веса вершин у нас монотонно убывают, и таких весов, у которых потенциал резко увеличился, может быть не больше чем logn.

Remove

Алгоритм удаления элемента будет следующим:

1. Вызываем splay(x)
2. Удаляем x
3. Находим максимальную вершину в левом поддереве y и делаем splay(y)
4. Соединяем получившиеся поддеревья

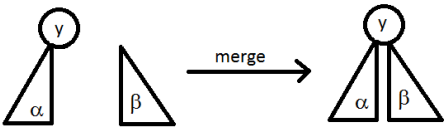


Рис. 5.9. Заключительный этап удаления элемента

При этом мы подвешиваем не больше n вершин, следовательно, потенциал изменится не больше чем на logn.

Варианты заданий:

1. Ключи всех вершин бинарного дерева поиска заданы в порядке его прямого обхода. Ключ каждой вершины является натуральным значением, меньшим 10^6. Все значения заданы в разных строках (по одному числу в строке). Бинарное дерево содержит не более 10000 вершин и не содержит вершин с одинаковыми ключами. Необходимо

вывести обратный обход дерева. Каждый ключ следует выводить в отдельной строке.

2. Требуется по заданным n и h найти количество АВЛ-деревьев, состоящих из n вершин и имеющих высоту h. Так как ответ может быть очень большим, требуется найти остаток от деления искомого количества на 786433. Во входном файле даны числа n и h ($1 \le n \le 65535$, $0 \le h \le 15$). Выведите одно число — остаток от деления количества АВЛ-деревьев, состоящих из n вершин и имеющих высоту h, на 786433.

3. В первоначально пустое АВЛ-дерево были занесены (согласно стандартному алгоритму вставки) в указанном порядке следующие ключи: 20, 15, 9, 18, 40, 35, 51, 27, 37, 36. Нарисуйте АВЛ-деревья, которые получатся после добавления каждого из этих ключей.

4. Дано splay-дерево, такое, что путь от корня к узлу с ключом 90 проходит через следующие узлы в порядке: 10, 20, 30, 40, 50, 60, 70, 80, 90. Нарисуйте результат операции splay над узлом 90.

5. Пусть до выполнения операции splay все узлы дерева из задачи 3 на пути к узлу 90 имеют ранг k. Покажите, что после выполнения операции splay над узлом 90 ранги этих узлов не увеличатся, а ранги как минимум трех из них уменьшатся.

ЛАБОРАТОРНАЯ РАБОТА № 6.
ГРАФЫ

Цель: Изучить применимость использования графов с точки зрения построения эффективного алгоритма, то есть удобной структуры для хранения бинарных отношений.

Краткая теория

Наиболее известным и популярным способом представления графов является геометрический способ, основанный на изображении графа в виде образа, состоящего из точек (вершин графа) и линий (соединяющих точки) – ребер графа. При разработке алгоритмов для задач на графе, а, в последствие для программ, необходимо вводить некоторый формальный способ представления графов. Способ, основанный на использовании известных в языках программирования структурах данных. К счастью графы можно представить с помощью разнообразных алгебраических форм: матриц, ассоциируемых с графом или структур данных.

Матрицей смежности ориентированного помеченного графа с n вершинами называется матрица $A = \lfloor a_{ij} \rfloor \; i, j = 1,2,...,n$, в которой

$$a_{ij} = \begin{cases} 1, \textit{ если существует ребро } (v_i, v_j), \\ 0, \textit{ если вершины } v_i, v_j \textit{ не связаны ребром } (v_i, v_j). \end{cases}$$

Матрица смежности однозначно определяет структуру графа.

Матрицей инцидентности для неориентированного графа с n вершинами и m ребрами называется матрица $B = \lfloor b_{ij} \rfloor$ $i = 1,2,...,n;\; j = 1,2,...,m$, строки в которой соответствуют вершинам, а столбцы – ребрам. При этом:

$$b_{ij} = \begin{cases} 1, \textit{если вершина } v_i \textit{ инцидентна ребру } u_j, \\ 0, \textit{если вершина } v_i \textit{ не инцидентна ребру } u_j. \end{cases}$$

Матрицей инцидентности для ориентированного помеченного графа с n вершинами и m ребрами называется матрица $B = \lfloor b_{ij} \rfloor$ $i = 1,2,...,n;\ j = 1,2,...,m$, строки в которой соответствуют вершинам, а столбцы – ребрам. При этом:

$$b_{ij} = \begin{cases} +1, \textit{если ребро } u_j \textit{ выходит из вершины } v_i, \\ -1, \textit{если ребро } u_j \textit{ входит в вершину } v_i, \\ 0, \textit{иначе.} \end{cases}$$

Матрицей весов графа с n вершинами называется матрица $W = \lfloor w_{ij} \rfloor$ $i, j = 1,2,...,n$, где w_{ij} – вес ребра, соединяющего i и j вершины.

Веса несуществующих ребер полагают равными 0 или ∞ в зависимости от смысла решаемой задачи.

При описании графа ***списком ребер*** каждое ребро представляется парой инцидентных ему вершин. Это представление можно реализовать двумя массивами:

$Tf = (tf_1, tf_2, ..., tf_m)$,

$To = (to_1, to_2, ..., to_m)$,

Каждый элемент в массивах есть метка вершины, а *i*-е ребро выходит из вершины tf_i и входит в вершину to_i.

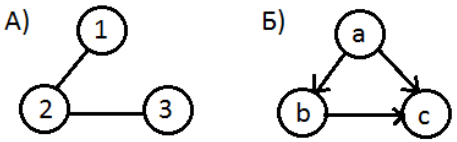
Интересно, что данное представление позволяет описывать петли и кратные ребра.

Ориентированные или неориентированные графы можно однозначно представить ***структурой смежности*** своих вершин. Такую

структуру удобно представлять массивом G линейно связанных списков A_{dj}. Каждый список содержит вершины, смежные с вершиной, из которой исходят ребра.

С точки зрения теории алгоритмов граф – это удобная структура данных, которая позволяет отражать бинарные отношения.

Рассмотрим способы хранения двух графов.



1) Хранение списка ребер

А) (1,2), (2,3) / {1,2}, {2,3}

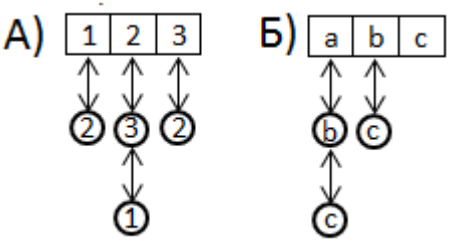
Б) (a,b), (b,c), (a,c)

2) Хранение матрицы смежности

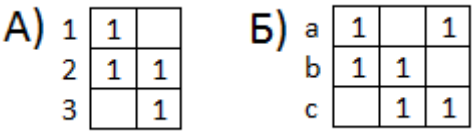
А	1	2	3
1	0	1	0
2	1	0	1
3	0	1	0

Б	a	b	c
a	0	1	1
b	0	0	1
c	0	0	0

3) Хранение списка смежности



4) Хранение матрицы инцидентности



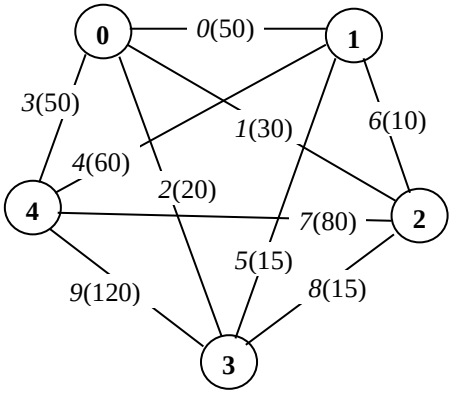
Теперь оценим плюсы и минусы способов хранения графов по трем критериям: память, проверка номера ребра и перебор соседей.

	Плюсы	***Минусы***
Список ребер	Память O(|V|+|E|)	Проверка номера ребра O(|E|) Перебор соседей O(|E|)
Матрица смежности	Проверка номера ребра O(1)	Память O($\|V^2\|$) Перебор соседей O(|V|)
Список смежности	Память O(|V|+|E|) Перебор соседей O (количество соседей)	Проверка номера ребра O(количество соседей)

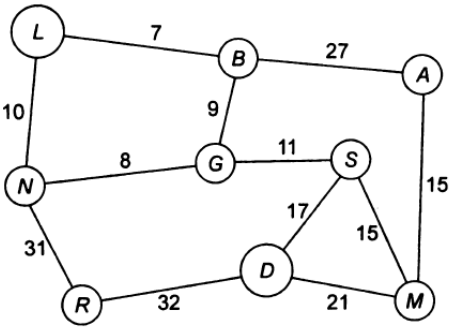
Выбор способа хранения графа зависит только от задачи. Чаще всего используется способ хранения списка смежности. Матрица инцидентности используется довольно редко, так как данная матрица зачастую сильно разряжена [15].

Варианты заданий:

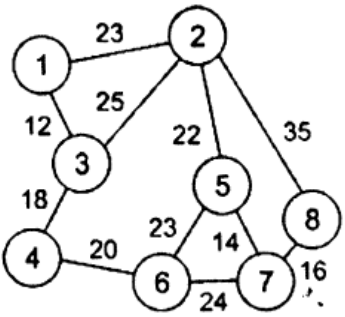
Описать граф, представленный на рисунке, с помощью структур смежности графа.



1. С помощью алгоритма Дейкстра построить минимальное остовное дерево для графа, представленного на рисунке задания 1.
2. Найти кратчайший путь на графе от вершины L до вершины D.
3.



4. Найти кратчайший путь на графе от 1-ой до 7-ой вершины.



СПИСОК ЛИТЕРАТУРЫ

1. *Гагарина, Л.Р.* Алгоритмы и структуры данных [Текст]: учеб. пособие / Л.Р. Гагарина, В.Д. Колдаев – М.: Издательство «Финансы и статистика», 2009. – 304 с.

2. *Шаров, А.А.* TURBO PASCAL [Текст] / А.А. Шаров URL: http://www.borlpasc.narod.ru/docym/stryctyr/str.htm (дата обращения 11.09.2018)

3. *Левитин, А.В.* Алгоритмы. Введение в разработку и анализ [Текст]: учебник / А.В. Левитан – М.: Издательство «Вильямс», 2006. – 576 с.

4. *Кормен, Т.* Алгоритмы: построение и анализ [Текст]: учебник / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн – М.: Издательство «Вильямс», 2016. – 1326 с.

5. *Лясин, Д.Н.* Использование рекурсивных вызовов в программах на языке Си [электронный ресурс]: методические указания/ Д.Н. Лясин, О.Ф. Абрамова //Сборник «Методические указания» Выпуск 3. – Электрон. текстовые дан. (1файл:207 Kb) – Волжский: ВПИ (филиал) ВолгГТУ,2012.

6. *Блэк, П. Dictionary of Algorithms and Data Structures* [электронный ресурс]: словарь / П. Блэк: URL: https://www.nist.gov/dads/HTML/queue.html (дата обращения 11.09.2018)

7. *Мальцев, А.И.* Алгебраические системы [Текст]: учеб. пособие / А.И. Мальцев – М.: Издательство «Наука», 1970. – 392 с.

8. *Белешко, Д.С.* Дискретная маетматика. Алгоритмы [Текст] / Д.С. Белешко URL: http://rain.ifmo.ru/cat/view.php/theory/algorithm-analysis/greedy-2004 (дата обращения 11.09.2018)

9. *Кнут, Д.* Искусство программирования [Текст]: учебник / Д. Кнут – М.: Издательство «Вильямс», 2000. – 832 с.

10. *Абрамян, М.Э.* Бинарные деревья [Текст]: учеб. пособие / М.Э. Абрамян – Ростов-на-Дону: Издательство Южного федерального университета, 2009. – 71 с.

11. *Сенюкова, О.В.* Сбалансированные деревья поиска [Текст]: учеб. пособие / М.Э. Абрамян – М.: Издательский отдел факультета ВМиК МГУ имени М.В. Ломоносова; МАКС Пресс, 2014. – 68 с.

12. *Вирт, Н.* Алгоритмы и структуры данных [Текст]: учебник / Н. Вирт – М.: Издательство «Мир», 1989. – 240 с.

13. *Адельсон-Вельский, Г. М.* Один алгоритм организации информации [Текст]: учебник / Г. М. Адельсон-Вельский, Е. М. Ландис – Доклады АН СССР. — 1962. — Т. 146, № 2. — С. 263—266.

14. *Кормен, Т.* Алгоритмы. Вводный курс [Текст]: учебник / Т. Кормен – М.: Издательство «Вильямс», 2015. – 208 с.

Учебное издание

Даниленко Александра Николаевна

СТРУКТУРЫ ДАННЫХ
И АНАЛИЗ СЛОЖНОСТИ АЛГОРИТМОВ

Учебное пособие

Редактор Т.К. Кретинина
Компьютерная вёрстка И.П. Ведмидской

Подписано в печать 07.11.2018. Формат 60×84 1/16.
Бумага офсетная. Печ. л. 4,75.
Тираж 100 экз. Заказ . Арт. – 18(Р4У)/2018.

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)
443086, САМАРА, МОСКОВСКОЕ ШОССЕ, 34.

Изд-во Самарского университета.
443086, Самара, Московское шоссе, 34.