Лабораторная работа №1
ИЗУЧЕНИЕ МЕТОДОВ ОЦЕНКИ АЛГОРИТМОВ
Краткая теория

Цель работы. Изучение методов оценки алгоритмов и программ и определение временной и емкостной сложности типовых алгоритмов и программ.

Алгоритм – это точное предписание о выполнении в определенном порядке некоторых операций, приводящих к решению всех задач данного класса.

Важнейшей характеристикой алгоритма и соответствующей ему программы является их сложность, которая может оцениваться:

a) временем решения задачи (трудоемкостью алгоритма);

b) требуемой емкостью памяти.

В общем случае сложность алгоритма можно оценить по порядку величины. Этот метод применим как к временной, так и к емкостной сложности.

Оценка порядка (временная оценка)

Эта оценка считается наиболее существенной. Для определения трудоемкости алгоритма обычно используют следующий набор «элементарных» операций, характерных для языков высокого уровня.

1. Простое присваивание: *a*<-- *b*, будем считать ее сложность, равной 1;
2. Одномерная индексация *a[i*] : (адрес (*a*)+*i**длина элемента),будем считать ее сложность, равной 2;
3. Арифметические операции: (*, /, -, +), будем считать ее сложность, равной 1;
4. Операции сравнения: *a* < *b*, будем считать ее сложность, равной 1;

5. Логические операции ($l1$) {or, and, not} ($l2$), будем считать ее сложность, равной 1;

С их помощью трудоемкость основных алгоритмических конструкций можно представить так.

1. *Линейная конструкция* из k последовательных блоков

Трудоемкость конструкции есть сумма трудоемкостей блоков, следующих друг за другом.

$$\Theta = f_1 + f_2 + \ldots + f_k,$$

где f_k – трудоемкость k-го блока.

2. Конструкция «Ветвление»

```
if(Условие) {
    Операторы
} else {
    Операторы
}
```

Общая трудоемкость конструкции «Ветвление» требует анализа вероятности выполнения операторов, стоящих после «Then» (p) и «Else» ($1-p$) и определяется как:

$$\Theta = f_{then}p + f_{else}(1-p),$$

где f_{then} и f_{else} – трудоемкости соответствующих ветвей.

3. Конструкция «Цикл»

```
for(i=1;i<=n;i++){
    Тело_цикла
}
```

После сведения этой конструкции к элементарным операциям ее трудоемкость определяется как:

$$\Theta = 1 + 3*n + n*f_{\text{цикла}},$$

где $f_{\text{цикла}}$ – сложность тела цикла,

n– число его повторений,

$3*n$ – трудоемкость изменения параметра и проверки условия выполнения цикла.

Сложность линейного участка и ветвления, как правило, не зависит от объема данных и считается равной некоторой константе. При этом алгоритмы, содержащие циклы и рекурсии, являются более сложными. Величина Θ возрастает с увеличением вложенности циклов, а также при многократном вызове методов обработки данных (процедур).

Рассмотрим примеры анализа простых алгоритмов.

Пример 1.Задача суммирования элементов квадратной матрицы размерностью *n*n.*

```
SumM (A, n; Sum)
Sum<-- 0
For i<-- 1 to n
For j<-- 1 to n
Sum <-- Sum + A[i,j]
endfor
Return (Sum)
End
```

Алгоритм выполняет одинаковое количество операций при фиксированном значении *n* и, следовательно, является количественно-зависимым. Применение методики анализа конструкции «Цикл» дает:

$$\Theta(n)=1+f_{\text{цикла по i}}= 1 + 1+ 3*n+ n *f_{\text{цикла по j}} =$$

$$1 + 1+ 3*n + n *(1+ 3*n + n *(2 + 2 + 1 + 1))$$

$$= 2 + 3n + n *(1 + 3n + 6\ n) = 9n^2+4\ n +2. \qquad (1.1)$$

Пример 2.Задачапоискамаксимумаводномерном массиве.

```
MaxS (S,n; Max)
Max <-- S[1]
For i<-- 2 ton
if Max <S[i]
Max <-- S[i]
end for
```

```
return Max
End
```

Приведенный алгоритм является количественно-параметрическим, т.е. зависит от значений исходных данных. Поэтому для фиксированной размерности исходных данных необходимо проводить анализ для худшего, лучшего и среднего случая.

1) Худший случай

Максимальное количество переприсваиваний максимума (на каждом проходе цикла) будет в том случае, если элементы массива отсортированы по возрастанию. При этом трудоемкость алгоритма будет равна:

$$\Theta_x(n)=1+ 2 +f_{\text{цикла по i}}= 1+2+1+ 3(n-1) +(n-1)\ (3+2+1+1) =$$
$$3 +1 + 7(n–1) = 7n–3. \qquad (1.2)$$

Здесь 1+2 – задание начального значения максимума (`Max<--S[1])`,

3(n-1) – на выполнение цикла,

(2 +1)(n-1) – на сравнение,

(n-1) – на запись S[i] на место Max.

2) Лучший случай

Минимальное количество переприсваиваний максимума (ни одного на каждом проходе цикла) будет в том случае, если максимальный элемент расположен на первом месте в массиве. При этом трудоемкость алгоритма будет равна:

$$\Theta_{л}(n)=1+ 2 +1+ 3(n-1) + (n-1)\ (2 +1) = 6n-2. \qquad (1.3)$$

3) Средний случай

Алгоритм поиска максимума последовательно перебирает элементы массива, сравнивая текущий элемент с текущим значением максимума. На очередном шаге, когда просматривается k-тый элемент массива, переприсваивание максимума произойдет, если в подмассиве из первых k элементов максимальным является последний. Очевидно, что в случае равномерного распределения

исходных данных, вероятность того, что максимальный из k элементов расположен в определенной (последней) позиции равна $1/k$. Тогда в массиве из n элементов общее количество операций переприсваивания максимума определяется как:

$$\sum_{k=1}^{n} 1/k = H_n \approx \ln(n) + \gamma, \gamma \approx 0.57 . \qquad (1.4)$$

Величина H_n называется n-ым гармоническим числом. Таким образом, при бесконечном количестве испытаний точное значение среднего числа операций присваивания в алгоритме поиска максимума в массиве из n элементов определяется величиной H_n, а средняя трудоемкость:

$$\Theta_{ср}(n)=1 + 2 + (n-1)(3+2) + 2(Ln(n) + \gamma)=$$
$$5n +2\ Ln(n) - 1 +2\ \gamma . \qquad (1.5)$$

Общий случай определения временной сложности

Выражения (1.1) – (1.5) представляют собой ***асимптотические*** оценки сложности соответствующих алгоритмов. В теории алгоритмов они являются семейством функций, дающих множество значений, в том числе верхнее и нижнее. Основным свойством $\Theta(n)$является то, что при увеличении размерности входных данных n время выполнения алгоритма возрастает с той же скоростью, что и функция $f(n)$.

Верхняя оценка сложности алгоритма$f(n)$обозначается $O(n)$ – *греческая буква «омикрон».* Считается, что она пропорциональна максимальному элементу в формуле $\Theta(n)$(см. (1.1) – (1.5)). Такая оценка легче вычисляется и дает предельное значение. Именно она получила наибольшее распространение.

Так, в примере 1 (нахождение суммы элементов квадратной матрицы) она определяется величиной $O(n^2)$, а в примере 2 (нахождение максимума в одномерном массиве) для всех случаев– $O(n)$. Вообще для циклов вида:

```
for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        for(k=0;k<n;k++){
            Телоцикла
        }
```

сложность равна O(n^3), т.е. пропорциональна числу повторений самого внутреннего цикла.

Сложность рекурсивных алгоритмов зависит не только от сложности внутренних циклов, но и от количества итераций рекурсии. Такая процедура может выглядеть достаточно простой, но она может серьёзно усложнить программу, многократно вызывая себя.

Рассмотрим рекурсивную реализацию вычисления факториалав среде java.

```
intfactorial(intn){
    if(n > 1)
        returnn*factorial(n-1);
    else
        return1;
};
```

Она выполняется *n* раз. Таким образом, вычислительная сложность этого алгоритма равна O*(n).*

O-функции выражают относительную скорость алгоритма в зависимости от некоторой переменной или переменных. При этом используются три ***правила оценки сложности***.

1. O*(k*f)* = O*(f),*
2. O*(f*g)* = O*(f)**O*(g),*
3. O*(f+g)* = *Max (*O*(f),* O*(g)*).

Здесь *k* обозначает константу, а *f* и *g* - функции.

Первое правило означает, что постоянные множители не имеют значения для определения порядка сложности, например, O*(2*n)* = O*(n).* В соответствии со вторым порядок сложности произведения двух функций равен произведению их сложностей. Например,

$O((10*n)*n) = O(n)*O(n) = O(n^2)$. На основании третьего правила порядок сложности суммы двух функций равен максимальному (доминирующему) значению из их сложностей. Например, $O(n^4 + n^2) = O(n^4)$.

На практике применяются следующие виды О-функций:

a) O(1) – константная сложность, для алгоритмов, сложность которых не зависит от размера данных.

b) $O(n)$ – линейная сложность, для одномерных массивов и циклов, которые выполняются n раз,

c) $O(n^2)$, $O(n^3)$, … – полиномиальная сложность, для многомерных массивов и вложенных циклов, каждый из которых выполняется n раз,

d) $O(log(n))$ – логарифмическая сложность, для программ, в которых большая задача делится на мелкие, которые решаются по-отдельности,

e) $O(2^n)$ – экспоненциальная сложность, для очень сложных алгоритмов, использующих полный перебор или так называемый «метод грубой силы».

Функции перечислены в порядке возрастания сложности. Чем выше в этом списке находится функция, тем быстрее будет выполняться алгоритм с такой оценкой.

Обычно определение сложности алгоритма, в основном, сводится к анализу циклов и рекурсивных вызовов.

Экспериментальный метод оценки трудоемкости (сложности) алгоритма

Метод основан на измерении времени выполнения алгоритма на некотором наборе данных (тесте). При этом используются стандартные средства языка программирования, позволяющие определить системное время компьютера. Например, для среды java это могут быть методы System.CurrentTimeMillis (),позволяющий фиксировать время с точность до миллисекунд и System.nanoTime() –

до наносекунд. Последний имеет ограничения и работает не на всех платформах.

Для оценки трудоемкости некоторого алгоритма достаточно запомнить время перед его выполнением и зафиксировать в конце, например, с помощью следующего фрагмента:

```
longstart = System.nanoTime();
// Фрагмент программы, реализующий
// исследуемыйалгоритм
long end = System.nanoTime();
longtraceTime = end-start;
```

Искомое время в наносекундах будет находиться в переменной *traceTime*.

Быстродействие современных процессоров с тактовой частотой в несколько Гигагерц имеет порядок нескольких десятков миллиардов операций в секунду. При этом время выполнения простых алгоритмов может быть очень малым, а измерение его предлагаемым методом будет иметь большую погрешность. Для ее уменьшения осуществляют многократное выполнение исследуемого фрагмента внутри цикла(например, повторяют его несколько десятков или сотен тысяч раз), а величину *traceTime* делят на число повторений цикла.

Оценка пространственной (емкостной) сложности

Такая оценка выполняется аналогично временной. При этом если для реализации алгоритма требуется дополнительная память для одной, двух или трех переменных, то емкостная сложность алгоритма будет константной, т.е. O(1). Такую сложность имеет, например, рассмотренный выше алгоритм поиска максимума. Если дополнительная память пропорциональна *n*, то пространственная сложность равна O(*n*) и т.д.

Для всех рекурсивных алгоритмов также важно понятие емкостной сложности. При каждом вызове метод запрашивает небольшой объём памяти, но этот объем может значительно

увеличиваться в процессе рекурсивных вызовов. Так, при вычислении факториала на первом этапе (разворачивания рекурсии) значения сомножителей помещаются в стек. Их количество равно *n*. При этом пространственная сложность алгоритма будет равна O(*n*). В других случаях эта характеристика может быть еще больше. Таким образом,всегда необходимо проводить анализ емкостной сложности рекурсивных процедур.

Временная и емкостная сложность взаимосвязаны. Это обусловлено тем, что память современных ЭВМ имеет иерархическую структуру. При выполнении программы процессор, прежде всего, работает с данными, находящимися в быстрой кэш-памяти. Если массив данных имеет большую размерность, то они могут не поместиться в кэше. При их обработке будут происходить кэш-промахи, которые сильно замедляют выполнение программы.

Оценка временной сложности в таком случае должна осуществляться экспериментально.

Порядок выполнения лабораторной работы

Работа предполагает выполнение следующих этапов.

1. Знакомство со всеми разделами руководства.
2. Получение у преподавателя задания на асимптотическую и верхнюю оценку сложности алгоритма (см. прил. 1) и выполнение этой оценки.
3. Оценка экспериментальным способом времени выполнения того же алгоритма. Значения исходных данных необходимо задавать в начале работы программы с помощью генератора случайных чисел, причем делать это до начала измерения времени работы алгоритма. Сам алгоритм в ходе измерений должен выполняться в цикле несколько миллионов раз, чтобы он не заканчивал работу слишком быстро, а выполнялся хотя бы несколько секунд.

4. Измерения необходимо повторить пять раз для различного объема исходных данных. Количество повторений алгоритма в каждом измерении должно быть одинаковым.
5. Построить график зависимости времени выполнения от объема входных данных.

Содержание отчета о выполненной работе

Отчет о выполненной работе должен содержать.

1. Название и цель работы.
2. Формулы асимптотической и верхней оценки сложности заданного алгоритма.
3. Исходный код программы экспериментальной оценки временной сложности заданного алгоритма для массива большой размерности.
4. Значения временной сложности алгоритма, полученные экспериментальным способом, а также количество повторений алгоритма и объем исходных данных, при котором были получены эти значения.
5. График зависимости времени выполнения алгоритма от объема исходных данных.

Контрольные вопросы

1. Чем характеризуется сложность алгоритма?
2. Как оценивается асимптотическая сложность алгоритма?
3. Как получается верхняя оценка сложности алгоритма?
4. Отличаются ли и на сколько асимптотическая и верхняя оценка сложности алгоритма?
5. Какие функции используются для представления верхней оценки сложности алгоритма?
6. У каких известных вам алгоритмов сложность является константной, а у каких - линейной?
7. Как оценивается сложность экспериментальным методом?
8. Совпадают ли результаты экспериментальной и верхней оценок и, если нет, то на сколько они отличаются?

9. Как влияет размер массива на временную сложность алгоритма?

10. Как влияет количество циклов повторения исследуемого алгоритма на погрешность определения времени его выполнения?

11. Как определяется емкостная сложность алгоритма?

12. Как связаны временная и емкостная сложность алгоритма?

Лабораторная работа №2
ИССЛЕДОВАНИЕ И ОЦЕНКА АЛГОРИТМОВ ПОИСКА
Краткая теория

Цель работы. Разработка программ, реализующих различные алгоритмы поиска, и оценка их временной и пространственной сложности.

Поиск. Основные определения и классы алгоритмов

Процедуры поиска широко используются в информационно-справочных системах (базах и банках данных), при разработке синтаксических анализаторов и компиляторов (поиск служебных слов, команд и т.п. в таблицах) и др. В простейшем случае считается, что исходными данными для этой процедуры являются массив (чисел, слов и т.д.) и некоторое значение, которое принято называть *аргументом поиска*.

Эталоны в таблице (массиве) могут иметь сложную структуру, но характеризуются неповторяющимися значениями некоторых *ключей*. Искомый *аргумент* сравнивается с каждым из этих ключей. Если они совпадают, то *аргумент* найден. Результат поиска может быть булевский (аргумент есть в таблице или нет) или числовой (номер ключа в таблице).

Наиболее сложным является случай, когда аргумента поиска нет в таблице. Суждение об этом можно сделать только по окончании просмотра всего массива.

Поскольку в процессе поиска участвуют только ключи, а информационная часть элементов не важна, в дальнейшем будем рассматривать только *алгоритмы поиска ключей*, значения которых обычно являются *целыми числами*.

В общем случае для выполнения этой операции могут использоваться различные ***условия***. В зависимости от способа их задания и организации поиска различают большое количество видов таких операций. Наибольшее распространение получили следующие ***виды поиска*** по простому условию:

a) совпадению,
b) близости снизу
c) близости сверху.

Первый вид предполагает нахождение элемента с заданным значением ключа *k*. Если такого ключа нет, то операция закончилась безуспешно.

При поиске ***по близости*** операция всегда заканчивается успешно. Если он выполняется снизу, то результатом будет элемент, ключ которого является ближайшим меньшим искомого. При поиске по близости сверху результат представляет собой элемент с ключом, ближайшим большим искомого.

Мы рассмотрим только алгоритмы поиска по совпадению, а по близости наиболее просто реализуются с помощью двоичных деревьев, которые будут изучаться позднее.

Общий алгоритм поиска по совпадению ключа можно представить так.

1. Ввести исходный массив (ключей).
2. Повторять
   ввести аргумент поиска и вывести результат
   пока не надоест.
3. Закончить.

Предположим в дальнейшем, что для окончания работы с

программой (когда искать надоело) необходимо ввести отрицательное число (несуществующий ключ).

Наиболее распространенными являются три алгоритма поиска:

1) Линейный;

2) Дихотомический:

3) Интерполирующий.

1. Алгоритм линейного поиска

В соответствии с этим алгоритмом эталонный массив просматривается последовательно от первого до последнего элемента. Наиболее сложным, как уже отмечалось, является случай, когда аргумента (ключа) нет в таблице (не найден).

Уточненный алгоритм будет таким.

1. Ввести исходный массив (ключей).
2. Выполнять
   2. 1. Ввести *аргумент поиска* (целое число);
   2. 2. Если *аргумент поиска* больше или равен нулю,
      2.2.1. Результат (номер в массиве, *num*) = - 1 (не найден, такого номера нет);
      2.2.2. Для индекса массива (*i)* от 0 до Длина.массива
         Если *аргумент поиска* = ключ[*i*],
            номер в массиве (*num*) = *i*;
      2.2.3. Если номер *num* = - 1,
         вывести: «Такого ключа в массиве нет»,
         Иначе
         вывести: «Ключ найден под номером *num*».

пока будет *аргумент поиска* больше или равен нулю.

3. Закончить.

Основной недостаток алгоритма линейного поиска – большое время. Он предполагает использование оператора **For** в пункте 2.2.2. При этом ВСЕГДА выполняется ровно ***n*** операций сравнения, не зависимо от того, найден ключ или нет. Программа, обнаружив

аргумент в начале массива, продолжает его просмотр до конца, т.е. выполняет бесполезную работу. Время поиска может быть существенно сокращено, если обеспечить его прекращение, когда ключ найден. При равномерном распределении элементов в таблице эталонов (ключей) среднее время поиска может стать пропорциональным величине **n / 2.**

Этого можно достичь, изменив пункт 2.2 в рассмотренном выше алгоритме следующим образом.

Ускорение линейного поиска

2. Выполнять

2. 1. Ввести *аргумент поиска* (целое число);

2. 2. Если *аргумент поиска* больше или равен нулю,

2.2.1. Результат (*num*) = - 1 (не найден);

2.2.2. Начальный индекс, *i*=0;

2.2.3. Пока (num=-1) и (i<=n – Длины.массива)

a) Если *аргумент поиска* = ключ[*i*],
номер в массиве (*num*) = *i*;

b) *i*=*i* + 1

2.2.4. Если номер *num* = - 1,
вывести: «Такого ключа в массиве нет»,
Иначе
вывести: «Ключ *num* найден».

пока будет *аргумент поиска* больше или равен нулю.

Трудоемкость (временная сложность) алгоритма линейного поиска определяется числом операций сравнения, выполняемых при просмотре таблицы эталонов. В лучшем случае количество таких операций равно 1, в худшем – *n*, а в среднем, если возможные значения ключей равновероятны, - *n* / 2. Таким образом, асимптотическая оценка O(*n*) = *n*.

2. Алгоритм дихотомического поиска

Этот алгоритм является более быстрым, чем линейный, но ***применяется только к упорядоченным массивам***. Среднее время дихотомического поиска пропорционально величине $\mathbf{log_2 n}$, где n — количество элементов таблицы эталонов. Таким образом, ускорение достигается за счет дополнительной информации о расположении элементов, а асимптотическая оценка алгоритма $O(n) = log_2 n$.

Метод основан на последовательном делении на 2 диапазона поиска. При этом на каждом шаге либо находится элемент, либо происходит переход в одну из половин диапазона. В процессе поиска выполняется не только сравнение на равенство, но и на больше - меньше. Последняя операция позволяет выбрать очередную половину диапазона таблицы. Если массив эталонов не упорядочен, то выбор будет сделан неверно, и результат можно не получить никогда (говорят, что алгоритм расходится).

Общий алгоритм поиска будет таким же, как в п. 1.

1. Ввести исходный массив (ключей).
2. Повторять
   ввести аргумент поиска и вывести результат
   пока не надоест.
3. Закончить.

Для корректности работы алгоритма необходимо упорядочить ключи (массив эталонов) по возрастанию. Для простоты их значения можно задать с помощью датчика случайных чисел. После этого необходимо выполнить операцию сортировки полученных величин.

Уточненный ***алгоритм*** можно представить в следующем виде.

1.1. Ввести количество элементов массива (n).
1.2. Инициировать датчик случайных чисел.
1.3. Для номера элемента (i) от 0 до n
   1.3.1. Вычислить ключ[i].

2. Упорядочить ключи по возрастанию:

2.1. Для номера просмотра (*k*) от 1 до *n* - 1

2.1.1. Для номера слова (*i*) от 0 до *n* - *k*

Если ключ [*i*] > ключ [*i*+1],

поменять местами ключ[*i*] и ключ[*i*+1].

3. Выполнять

3. 1. Ввести *аргумент поиска* (целое число);

3. 2. Если *аргумент поиска* больше или равен нулю,

3.2.1 Граница_левая (диапазона поиска) = 0;

3.2.2. Граница_правая (диапазона поиска) = *n* - 1 (Длина.массива – 1);

3.2.3. Если *аргумент поиска* = ключ [*n* - 1],

a) Признак = «Найдено»;

b) *k* = *n* – 1.

Иначе

3.2.4. Признак = «Не найдено».

3.2.5. Выполнять

a) Номер *аргумент поиска* (*k*) = Целое ((Граница_левая + Граница_правая)/2);

b) Если *аргумент поиска* = ключ [*k*],

Признак:= «Найдено»

Иначе

Если *аргумент поиска* > ключ [*k*],

Граница_левая = *k*

Иначе

Граница_правая = *k*.

Пока не «Найдено» Или (Граница_левая = Граница_правая - 1).

3.3. Если «Найдено»,

вывести: «Ключ найден под номером *k*»,

Иначе

вывести: «Такого ключа в массиве нет».

Пока будет *аргумент поиска* больше или равен нулю.
4. Закончить.

В алгоритме пункт 3.2.3 применен для обеспечения нахождения последнего элемента массива. Дело в том, что при целочисленном делении на 2 дробная часть частного отбрасывается, и результат всегда будет на 1 меньше, чем длина таблицы.

3. Алгоритм интерполирующего поиска

Интерполирующий поиск основан на принципе поиска в телефонной книге или, например, в словаре. Вместо сравнения каждого элемента с искомым как при линейном поиске, производится предсказание местонахождения элемента. Процедура похожа на двоичную, но вместо деления области поиска на две части, производится оценка новой области по расстоянию между ключом и текущим значением аргумента. Другими словами, бинарный поиск учитывает лишь знак разности между ключом и текущим значением, а интерполирующий - ещё и модуль этой разности и по этому значению производит предсказание позиции следующего элемента для проверки.

В среднем, интерполирующий поиск производит $\log(\log(n))$ операций, где n - число элементов в массиве. Количество необходимых операций зависит от равномерности распределения значений среди элементов. В плохом случае (например, когда значения элементов экспоненциально возрастают) интерполяционный поиск может потребовать до $O(n)$ операций.

На практике, интерполяционный поиск часто быстрее бинарного, так как их отличают, прежде всего, применяемые арифметические операции:

a) интерполирование — в интерполирующем поиске и

b) деление на два — в двоичном.

При этом скорость вычисления отличается незначительно. С другой стороны интерполирующий поиск использует такое

принципиальное свойство данных, как однородность распределения значений. Ключом может быть не только номер, число, но и, например, текстовая строка, тогда становится понятна аналогия с телефонной книгой: если мы ищем имя в телефонной книге, начинающееся на «А», то нужно искать его в начале, но никак не в середине. В принципе, ключом может быть всё что угодно, так как те же строки, например, кодируются посимвольно, в простейшем случае символ можно закодировать значением от 1 до 33 (только русские символы) или, например, от 1 до 26 (только латинский алфавит) и т. д.

Интерполяция может производиться на основе функции, аппроксимирующей распределение значений, либо набора кривых, выполняющих аппроксимацию на отдельных участках. В этом случае поиск может завершиться за несколько проверок. Преимущества метода состоят в уменьшении запросов на чтение медленной памяти (такой, например, как жесткий диск), если запросы происходят часто.

Часто анализ и построение аппроксимирующих кривых не требуется, например, когда все элементы отсортированы по возрастанию. В таком списке минимальное значение будет по индексу 0, а максимальное по индексу n - 1. В этом случае аппроксимирующую кривую можно принять за прямую и применять линейную интерполяцию.

Общий ***алгоритм интерполирующего поиска*** без задания исходного массива может быть таким. На каждой стадии рассчитывается позиция mid (средняя) для следующей проверки, по формуле:

mid = low + (аргумент - ключ [low])*(high - low) /
(ключ [high] - ключ[low])

где low – нижняя граница диапазона,
high – его верхняя граница.

Затем в зависимости от результата сравнения переносится верхняя или нижняя граница, определяя новую область поиска. Процесс

заканчивается, если элемент с индексом mid равен искомому ключу или нижняя и верхняя границы поиска совпали.

Пошаговый алгоритм интерполирующего поиска можно записать следующим образом.

1. Выполнять

1.1. Ввести *аргумент поиска* (целое число);

1.2. Если *аргумент поиска* больше или равен нулю,

1.2.1. Искомый номер (num) = - 1.

1.2.1. Нижняя_граница (low) = 0;

1.2.2. Верхняя_граница (high) = *n* - 1;

1.2.3. Пока (ключ[low] < *аргумент* И ключ [high] > *аргумент*)

a) Средний(mid) = low + (*аргумент* - ключ [low])*(high - low) / (ключ [high] - ключ[low]);

b) Если ключ [mid]< *аргумента,*

low = mid + 1;

Иначе

Если ключ [mid]< *аргумента,*

high = mid - 1;

Иначе

mid – искомый номер.

1.2.4. Если ключ[low] = *аргумент,*

low – искомый номер num

Иначе

Если ключ[high] = *аргумент,*

high – искомый номер num

Иначе

Номер num не найден

В заключение можно отметить, что рассмотренные алгоритмы могут применяться не только к числам, но и к строкам. Примером являются задачи поиска слова в словаре, справочнике, таблице и т.д. Применение дихотомического поиска требует упорядочения слов по алфавиту.

Порядок выполнения лабораторной работы

Работа предполагает выполнение следующих этапов.

1. Знакомство со всеми разделами руководства.

2. Получение у преподавателя задания на разработку программы для алгоритмов поиска (см. прил. 2).

3. Разработка и отладка заданной программы.

4. Получение верхней и экспериментальной оценки времени выполнения заданного алгоритма и программы.

5. Нахождение предельной оценки емкости памяти, необходимой для выполнения разработанной программы.

Содержание отчета о выполненной работе

Отчет о выполненной работе должен содержать.

1. Название и цель работы.
2. Словесное описание заданного алгоритма поиска.
3. Текст программы.
4. Формулы верхней оценки временной и емкостной сложности заданного алгоритма.
5. Результаты экспериментальной оценки временной и емкостной сложности заданного алгоритма.

Контрольные вопросы

1. Что такое поиск и для чего он нужен?
2. Что является исходными данными для поиска?
3. Какие алгоритмы поиска Вы знаете?
4. Приведите словесное описание простейшего алгоритма линейного поиска (с циклами For).
5. Приведите словесное описание ускоренного алгоритма линейного поиска.
6. Приведите словесное описание алгоритма дихотомического поиска.
7. Приведите словесное описание алгоритма интерполирующего поиска.
8. Какова верхняя оценка трудоемкости алгоритма линейного поиска?

9. Какова верхняя оценка трудоемкости алгоритма дихотомического поиска?

10. Какова верхняя оценка трудоемкости алгоритма интерполирующего поиска?

11. Какова верхняя оценка емкостной сложности алгоритма линейного поиска?

12. Какова верхняя оценка емкостной сложности алгоритма дихотомического поиска?

13. Какова верхняя оценка емкостной сложности алгоритма интерполирующего поиска?

14. На сколько отличаются результаты оценки трудоемкости предложенного Вам алгоритма, полученные аналитическими (по формулам) и экспериментальными методами?

15. На сколько отличаются результаты оценки емкостной сложности предложенного Вам алгоритма, полученные аналитическими (по формулам) и экспериментальными методами?

Лабораторная работа № 3
РАЗРАБОТКА РЕКУРСИВНЫХ АЛГОРИТМОВ

Краткая теория

Цель работы. Разработка программ, реализующих различные рекурсивные алгоритмы, и оценка их временной и пространственной сложности.

Понятие рекурсии и примеры ее использования

В математике под рекурсией понимается способ организации вычислений, при котором функция вызывает сама себя с другим аргументом. Большинство современных языков высокого уровня поддерживают механизм рекурсивного вызова. Ввиду отсутствия в языке Java понятия функции рассматриваются рекурсивно вызываемые методы.

Таким образом, в языке Java **рекурсия** – это вызов метода из самого себя непосредственно (**простая рекурсия**) или через другие методы (**сложная** или **косвенная рекурсия**). Примером сложной рекурсии является случай, когда метод А вызывает метод В, а метод В – метод А.

Количество вложенных вызовов методов называется **глубиной рекурсии.**

Реализация рекурсивных вызовов опирается на механизм стека вызовов. Адрес возврата и локальные переменные метода записываются в стек, благодаря чему каждый следующий рекурсивный вызов этого метода пользуется своим набором локальных переменных. Обычно рекурсия реализуется в два прохода:

1) формирование в стеке последовательности аргументов;
2) собственно вычисления (реализация метода), выполняемые над данными, помещенными в стек.

На каждый рекурсивный вызов требуется некоторое количество оперативной памяти компьютера, т.е. для рекурсии необходимо оценивать емкостную сложность. При чрезмерно большой глубине рекурсии может наступить переполнение стека, и возникнуть исключительная ситуация **StackOverfowError** (переполнение стека).

Графическое представление цепочки рекурсивных вызовов, порождаемой данным алгоритмом, называется деревом рекурсивных вызовов.

Можно заметить, что некоторая ветвь дерева рекурсивных вызовов обрывается при достижении такого значения передаваемого параметра, при котором функция может быть вычислена непосредственно. Таким образом, рекурсия эквивалентна конструкции цикла, в котором каждый проход есть выполнение рекурсивной функции с заданным параметром.

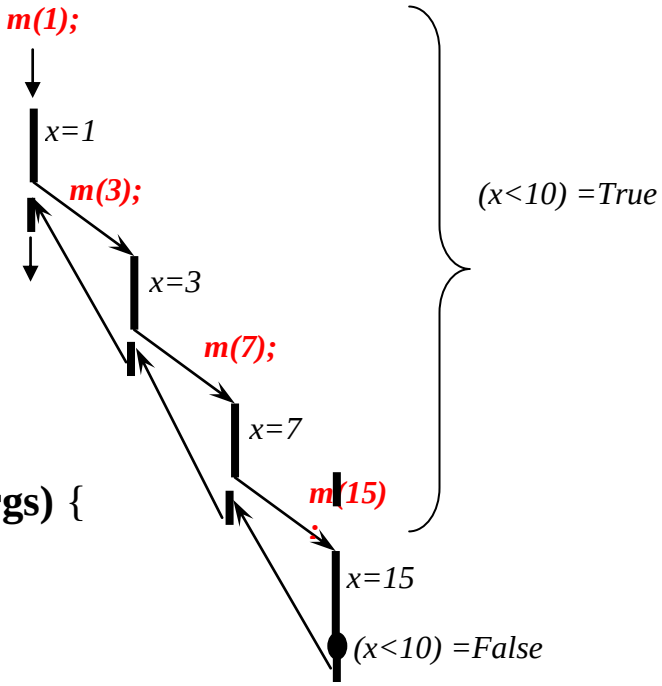
Рассмотрим несколько примеров простой рекурсии, когда метод вызывает сам себя. Проиллюстрируем их рисунком, представляющим дерево вызовов.

Пример 1. Для заданного параметра **x** вывести последовательность числового ряда в соответствии со следующими требованиями:

– очередной элемент x= 2*x+1 (новое значение вычисляется с использованием старого);
– x<10.

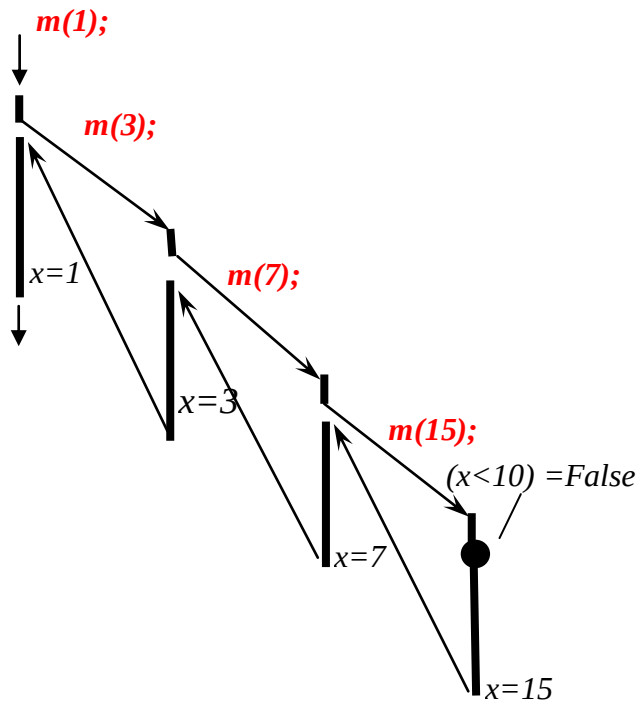
Ниже приведен метод *m*, вычисляющий элемент ряда, и фрагмент основной программы, которая его вызывает. Рядом с текстом метода представлено дерево вызовов.

```
public class Rec1 {
   public static void m(int x) {
       System.out.println("x="+x);
      if (x<10) {
        m(2*x+1);
      }
   }

public static void main(String[] args) {
      m(1);
   }
  }
```



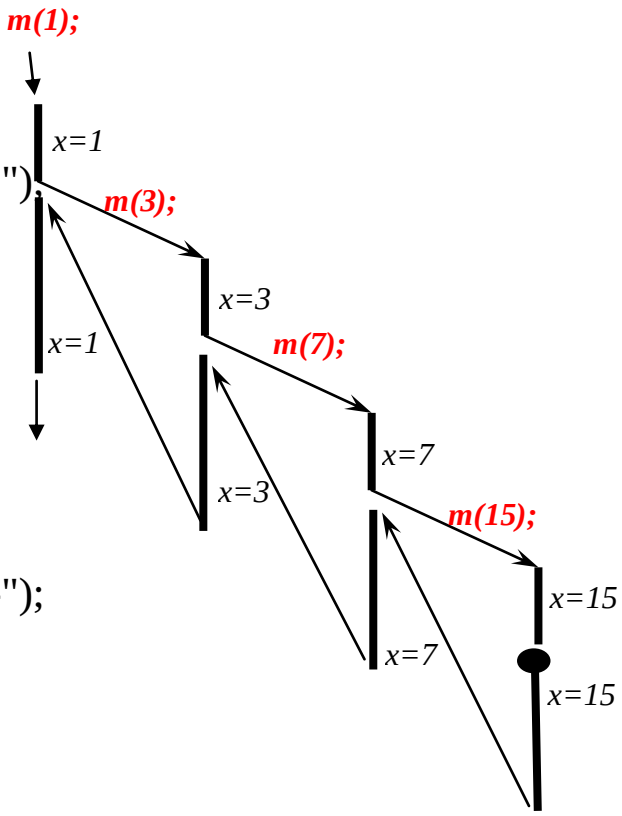
Пример 2. Вывести последовательность из примера 1 в обратном порядке.

```
public class Rec2 {
   public static void m(int x) {
      if (x<10) {
        m(2*x+1);
      }
      System.out.println("x="+x);
    }

   public static void main(String[]
args) {
      m(1);
      }
      }
```



Пример 3. Для примера 1 вывести параметр перед вхождением в рекурсию и после него.

```
public class Rec3 {
   private static int step=0;
   public static void m(int x) {
      space();
      System.out.println(""+x+"-> ");
      step++;
      if (x<10) {
        m(2*x+1);
      }
      step--;
      space();
      System.out.println(""+x+ " <-");
   }
```



```
    public static void space() {
        for (int i = 0; i < step; i++) {
            System.out.print("  ");
        }
    }

    public static void main(String[] args) {
      m(1);
    }
}
```

Следует отметить, что рекурсию всегда можно заменить циклом.

Классическим примером рекурсивных алгоритмов считаются вычисление факториала и чисел Фибоначчи. Рассмотрим примеры записи соответствующих методов.

Пример 3. Вычислить факториал числа n с использованием рекурсии.

Факториал числа *n* (обозначается *n*!) — произведение всех натуральных чисел от 1 до n включительно, т.е. n!=1*2*…*n. Например, 5!=1*2*3*4*5 =4!*5 . В общем случае формулу для нахождения факториала можно записать так

n!=(*n*-1)!**n*, если *n* > 1 и

1, если *n* = 1.

```
public static  int fact(int n){
    int result;
    if (n==1)
       return 1;
    else{
       result=fact(n-1)*n;
       return result;
        }
       }
```

Пример 4. Подсчитать ряд чисел Фибоначчи до заданного значения *x*.

Последовательность Фибоначчи формируется так: первый и второй члены последовательности равны единицам, а каждый следующий — сумме двух предыдущих:

$$f(k) = f(k\text{-}1) + f(k\text{-}2),$$

$$f(0) = 0,\ f(1) = 1.$$

№ числа	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*...*	*20*
число	*0*	*1*	*1*	*2*	*3*	*5*	*8*	*13*	*21*	*...*	*6765*

```
public static int f(int x){
if (x==0){
      return 0;
}else
    if (x==1){
        return 1;
    } else {
          return f(x-2)+f(x-1);
       }
}
```

Таким образом, трудоемкость рекурсивных алгоритмов зависит как от количества операций, выполняемых при одном вызове функции, так и от количества таких вызовов. Так как рекурсию можно заменить циклом, то порядок временной сложности соответствующего алгоритма можно считать равным числу повторений цикла.

Порядок выполнения лабораторной работы

Работа предполагает выполнение следующих этапов.

1. Знакомство со всеми разделами руководства.

2. Получение у преподавателя задания на разработку программы для рекурсивных алгоритмов (см. прил. 3).

3. Разработка и отладка заданной программы.

4. Получение верхней и экспериментальной оценки времени выполнения заданного алгоритма и программы.

5. Нахождение предельной оценки емкости памяти, необходимой для выполнения разработанной программы.

Содержание отчета о выполненной работе

Отчет о выполненной работе должен содержать.

1. Название и цель работы.
2. Словесное описание заданных рекурсивных алгоритмов.
3. Тексты программ.
4. Формулы верхней оценки временной и емкостной сложности заданных алгоритмов.
5. Результаты экспериментальной оценки временной и емкостной сложности заданных алгоритмов.

Контрольные вопросы

1. Что такое рекурсия и для чего она нужна?
2. Чем отличается простая рекурсия от сложной?
3. Какие классические рекурсивные алгоритмы Вы знаете?
4. Чем можно заменить рекурсию?
5. Какими характеристиками сложности описывается рекурсия?
6. Что такое глубина рекурсии?
7. Что представляет собой дерево рекурсии?
8. Как можно вычислить факториал без использования рекурсивного алгоритма?
9. Как можно вычислить числа Фибоначчи без использования рекурсивного алгоритма?
10. Как можно оценить емкостную сложность рекурсивного алгоритма?

Лабораторная работа №4
ИССЛЕДОВАНИЕ И ОЦЕНКА АЛГОРИТМОВ СОРТИРОВКИ
Краткая теория

Цель работы. Разработка программ, реализующих различные алгоритмы сортировки, и оценка их временной и пространственной сложности.

Сортировка. Основные определения и классы алгоритмов

Сортировка — это упорядочение элементов в списке. В случае, когда элемент имеет несколько полей, поле, служащее критерием порядка, называется ключом сортировки. На практике в качестве ключа часто выступает число, а в остальных полях хранятся любые данные, не влияющие на работу алгоритма.

Мы рассмотрим наиболее распространенные алгоритмы сортировки для чисел, хотя они могут быть распространены на строки и списки элементов, о которых говорилось в общем определении.

Классификация алгоритмов сортировки

При классификации учитывают следующие основные характеристики алгоритмов.

1) *Устойчивость* (stability) — устойчивая сортировка не меняет взаимного расположения равных элементов.

1) *Естественность поведения* — эффективность метода при обработке уже упорядоченных или частично упорядоченных данных. Алгоритм ведёт себя естественно, если учитывает эту особенность входной последовательности и работает лучше.

2) *Использование операции сравнения*. Алгоритмы, применяющие для сортировки сравнение элементов между собой, называются основанными на сравнениях. Минимальная трудоемкость

худшего случая для этих алгоритмов составляет $O(n^2)$, но они отличаются гибкостью применения. Для специальных случаев (типов данных) существуют более эффективные алгоритмы.

Ещё одним важным свойством алгоритма является сфера его применения. При этом различают два основных типа упорядочения.

1. *Внутренняя сортировка,* которая оперирует с массивами, целиком помещающимися в оперативной памяти с произвольным доступом к любой ячейке. Данные обычно упорядочиваются на том же месте, без дополнительных затрат. В современных персональных компьютерах, как уже отмечалось, широко применяется подкачка и кэширование памяти. Алгоритм сортировки должен хорошо сочетаться с этими операциями.

2. *Внешняя сортировка,* которая применяется для запоминающих устройств большого объёма, с последовательным доступом (упорядочение файлов). При этом в каждый момент доступен только один элемент, а затраты на перемотку по сравнению с памятью неоправданно велики. Это накладывает дополнительные ограничения на алгоритм и приводит к методам упорядочения, обычно использующим дополнительное дисковое пространство. Кроме того, доступ к данным на носителе производится намного медленнее, чем операции с оперативной памятью. Их объём настолько велик, что не позволяет разместить информацию в ОЗУ.

К алгоритмам ***устойчивой сортировки*** относят следующие.

1. Сортировка *пузырьком* (англ. Bubble sort) — сложность алгоритма: $O(n^2)$; для каждой пары индексов производится обмен, если элементы расположены не по порядку.

2. Сортировка *перемешиванием* (Шейкерная, Cocktail sort, bidirectional bubble sort) — сложность алгоритма: $O(n^2)$.

3. Сортировка *вставками* (Insertion sort) — сложность алгоритма: $O(n^2)$; определяют место элемента в упорядоченном списке и вставляют его туда.

4. *Гномья* сортировка — сложность алгоритма: O(n^2); сочетает методы пузырьковой сортировки и вставками.

5. *Блочная* сортировка (Корзинная, Bucket sort) — сложность алгоритма: O(n); требуется O(k) дополнительной памяти и знание о природе сортируемых данных.

6. Сортировка *подсчётом* (Counting sort) — сложность алгоритма: O($n+k$); требуется O($n+k$) дополнительной памяти (существует три варианта).

7. Сортировка *слиянием* (Merge sort) — сложность алгоритма: O($n \log n$); требуется O(n) дополнительной памяти; упорядочивают две половины списка отдельно (первую и вторую), а затем — сливают их воедино.

8. Сортировка *с помощью двоичного дерева* (англ. Tree sort) — сложность алгоритма: O($n \log n$); требуется O(n) дополнительной памяти.

Алгоритмами ***неустойчивой сортировки*** являются следующие методы.

1. Сортировка *выбором* (Selection sort) — сложность алгоритма: O(n^2); выполняется поиск наименьшего или наибольшего элемента и помещение его в начало или конец упорядоченного списка.

2. Сортировка *Шелла* (Shell sort) — сложность алгоритма: O($n \log_2 n$); попытка улучшить сортировку вставками.

3. Сортировка *расчёской* (Comb sort) — сложность алгоритма: O($n \log n$)

4. *Пирамидальная* сортировка (Сортировка кучи, Heapsort) — сложность алгоритма: O($n \log n$); список превращается в кучу, берется наибольший элемент и добавляется в конец списка.

5. *Плавная* сортировка (Smoothsort) — сложность алгоритма: O($n \log n$).

6. *Быстрая* сортировка (Quicksort), в варианте с минимальными затратами памяти сложность алгоритма: O($n \log n$) — среднее время, O(n^2) — худший случай; широко известен как быстрейший из

известных для упорядочения больших случайных списков; исходный набор данных разбивается на две половины так, что любой элемент первой половины упорядочен относительно любого элемента второй; затем алгоритм применяется рекурсивно к каждой половине. При использовании O(n) дополнительной памяти сортировка становится устойчивой.

7. *Поразрядная* сортировка — сложность алгоритма: O($n \cdot k$); требуется O(k) дополнительной памяти.

8. *Сортировка перестановкой* — O($n \cdot n!$) — худшее время. Для каждой пары осуществляется проверка верного порядка и генерируются всевозможные перестановки исходного массива.

Описание алгоритмов сортировки

Рассмотрим наиболее распространенные алгоритмы.

Алгоритм 1. Пузырьковая сортировка

Этот метод - наиболее распространенный и простой. Он требует минимального объема памяти для данных, но затраты времени на его реализацию велики. При упорядочении выполняются следующие операции:

1) элементы массива сравниваются *попарно*: первое со вторым; второе с третьим; i-тое – с (i+1) - вым;

2) если они стоят неправильно (при *упорядочении* по возрастанию первый должен быть меньше второго или равен ему), то элементы меняются местами.

За один такой просмотр массива при сортировке по возрастанию минимальный элемент «вытолкнется», по крайней мере, на одно место вверх (вперед), а максимальный – переместится в самый конец (вниз). Таким образом, минимальный элемент как легкий пузырек воздуха в жидкости постепенно

«всплывает» вверх (в начало последовательности). Отсюда – название метода.

Поскольку последний элемент после первого просмотра массива окажется на своем месте, следующий просмотр можно закончить раньше, т.е. в конце сравнивать (n-2)-рой и (n-1)-вый элементы. Вообще, при k-том просмотре достаточно сравнить элементы от первого до (n-k) –того.

За n-1 просмотр произойдет полное упорядочение массива при любом исходном расположении элементов в нем.

Алгоритм сортировки пузырьком

1. Задать массив из n чисел.
2. Для номера_просмотра (k) от 1 до n-1 выполнить
2.1. Для номера_элемента (i) от 1 до n - k выполнить
Если элементы i-тый и (i+1)-вый стоят неправильно, то
Поменять их местами;
3. Вывести отсортированный массив.
4. Закончить.

Сокращение количества просмотров улучшает временные характеристики метода. Из алгоритма можно понять, что если на одном из просмотров не было перестановок, то их не будет и потом – данные уже отсортированы, а процесс следует закончить. Такой подход дает значительную экономию времени при работе с большими «почти отсортированными» массивами.

В ускоренном алгоритме будем использовать булевскую переменную «Перестановка», которой сначала присваивается значение «ложь», так как перестановок элементов не было. Затем, если в процессе просмотра массива некоторые элементы менялись метами, то признак получает значение «истина». Это позволит прервать выполнение внешнего цикла при условии, что на очередном просмотре перестановки не выполнялись.

Алгоритм ускоренной сортировки пузырьком

1. Задать массив из n чисел.

2.1 Номер_просмотра (k) = 1.

2.2. Повторять

2.2.1. Перестановка = *ложь*.

2.2.2. Для i от 1 до n-k выполнить

Если элементы i-тый и (i+1)-вый стоят неправильно, то

a) Поменять местами i-тый и (i+1)-вый элементы;

b) Перестановка = *истина*.

2.2.3. $k = k + 1$

Пока Перестановка.

3. Вывести отсортированный массив.

4. Закончить.

Алгоритм 2. Сортировка вставками

Это достаточно простой, но эффективный в некоторых случаях метод. В начале считается, что первый элемент находится на своем месте. Далее, начиная со второго, каждый элемент сравнивается со всеми, стоящими перед ним, и если они стоят неправильно, то меняются местами. Таким образом, новый элемент сравнивается и меняется местами не со всем массивом, а только до тех пор, пока в начале не найдется элемент, меньший его. Поэтому рассматриваемый алгоритм примерно в два раза быстрее сортировки пузырьком. Для уже частично отсортированных массивов такой метод является наилучшим.

Словесное описание алгоритма можно представить так.

Алгоритм сортировки вставками

1. Задать исходный массив A из n элементов, например, с помощью генератора случайных чисел.
2. Для i от 0 до n

*2.1.*Индекс элемента в конце массива $j = i$

2.2.Пока ($j > 0$) И ($A_{j-1} > A_i$)

2.2.1. *Сдвиг большего элемента, чтобы было место для вставки*

$A_j = A_{j-1}$

2.2.2. $j = j - 1$

2.3.Вставка j - го элемента на его место

$A_j = A_i$

3. Вывести массив *A*.

Время сортировки можно сократить, если учесть, что начальная часть массива (до индекса *j*) упорядочена. При этом точку вставки можно найти методом деления пополам диапазона индексов этой части (от 0 до *j*).

Алгоритм сортировки двоичными вставками

1. Задать исходный массив *A* из *n* элементов.
2. Для *i* от 0 до *n*

2.1. *Вставляемый элемент* $x = A_i$

2.2. Левая граница диапазона $L = 0$.

2.3. Правая граница диапазона $R = i$.

2.4. Пока ($L<R$)

2.4.1. Искомый индекс $m = (L + R)/2$

2.4.2. Если $A_m \leq x$

$L = m + 1$

Иначе

$R = m$.

2.5. Перепись элементов на одну позицию в конец массива (освобождение места для вставки)

2.5.1. Для *j* от *i* до $R + 1$ с шагом -1

$A_j = A_{j-1}$

2.6. Вставка последнего элемента

$A_R = x$

3. Вывести массив *A*.

Число сравнений и пересылок рассмотренным методом в худшем случае пропорционально *n*log(n)*, т.е. сложность алгоритма равна O(*n*log(n)*).

Алгоритм 3. Сортировка выбором

Этот метод – один из наиболее простых. Он требует выполнения *n*-1 просмотра массива. Во время *k*-го просмотра выявляется наименьший элемент, который затем меняется местами с *k*-м.

Алгоритм сортировки выбором

1. Задать массив *A* из *n* чисел.
2. Для *k* от 0 до *n*-1
    2.1. $j = k$
    2.2. Для i от $k + 1$ до n-1
        2.2.1. Если $A_i < A_j$,
            $j = i$.
    2.3. Если $k \neq j$, то
        Поменять местами A_k и A_j.
3. Вывести массив *A*.
4. Закончить.

Алгоритм 4. Сортировка Шейкером

Если данные сортируются не в оперативной памяти, а на жестком диске, то количество перемещений элементов существенно влияет на время работы. Этот алгоритм уменьшает количество таких перемещений. За один проход из всех элементов выбираются минимальный и максимальный. Потом минимальный элемент помещается в начало массива, а максимальный - в конец. Таким образом, за каждый проход два элемента оказываются на своих местах, поэтому понадобится $n/2$ проходов, где n — количество элементов.

На каждом проходе массив просматривается слева направо до середины, а потом – наоборот (справа налево). В результате, как при сортировке «пузырьком», при просмотре слева направо максимальный элемент попадает на свое место, а при обратном – минимальный.

Алгоритм сортировки Шейкером

1. Задать массив из *n* чисел.

2.1 Левая_граница = 0.

2.2. Правая_граница = *n* - 1.

2.3. Повторять

2.3.1. *Движение слева направо:*

1) Для *i* = Левая_граница; *i* < Правая_граница; *i*++

Если элементы *i*-тый и (*i*+1)-вый стоят неправильно, то

a) Поменять их местами;

b) Номер = *i*.

2.3.2. *Движение справа налево:*

1) Правая_граница = Номер.

2) Для *i* = Правая_граница; *i* > Левая_граница; *i*--

Если элементы *i*-тый и (*i*+1)-вый стоят неправильно, то

a) Поменять их местами;

b) Номер = *i*.

2.3.3. Левая_граница = Номер.

Пока Левая_граница < Правая_граница.

3. Вывести отсортированный массив.

4. Закончить.

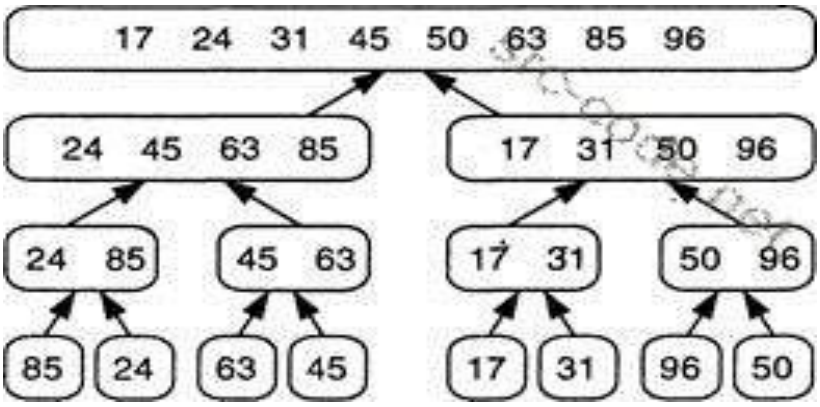
Алгоритм 5. Сортировка слиянием

Этот метод является одним из самых простых и быстрых. Особенностью его является то, что он работает с элементами массива последовательно, благодаря чему может использоваться, например,

для данных на жёстком диске. Кроме того, алгоритм, может быть эффективен для сортировки таких структур, как связанные списки. Он был предложен Джоном фон Нейманом в 1945 году.

Алгоритм использует методику «разделяй и властвуй», которая может быть представлена в виде трех стадий:

1) разделение — данные разбиваются на две, три или более составляющих, обработка которых осуществляется простыми методами;



2) рекурсия — применяется для обработки полученных составляющих;

3) слияние — результаты действий над составляющими «сливаются» в единое целое.

Основой алгоритма является *подзадача слияния* двух отсортированных массивов в один, тоже отсортированный. Пусть исходные массивы A и B имеют размерность n_a и n_b. Результат слияния – массив С размерностью $n_a + n_b$. Он формируется так. Из первых элементов массивов A и B в C записывается сначала меньший, а потом – больший. Как показано на рисунке 4.1. Считая, что индексы элементов начинаются с 0 (как это принято в системах C++ и java), алгоритм можно описать так.

Рис. 4.1. Сортировка слиянием

Алгоритм слияния двух отсортированных массивов

1. Номер A (i_a) =0.
2. Номер B (i_b) =0.
3. Пока ($i_a < n_a$) *и* ($i_b < n_b$)

    3.1. Если $A[i_a] \le B[i_b]$, то

        а) $C[i_a + i_b] = A[i_a]$;

b) $i_a = i_a + 1$.

Иначе

a) $C[i_a + i_b] = B[i_b]$;

b) $i_b = i_b + 1$.

4. Если $i_a < n_a$, то

Переписать оставшийся массив *A* в *C*.

5. Если $i_b < n_b$, то

Переписать оставшийся массив *B* в *C*.

Собственно сортировка слиянием

Процедура слияния использует два отсортированных массива. Очевидно, что массив из одного элемента по определению является отсортированным. Тогда сортировку можно осуществить в соответствии с рисунком 2.1 следующим образом.

1. Разбить элементы массива на пары и осуществить слияние элементов каждой пары, получив отсортированные цепочки длины 2 (кроме может быть, одного элемента, для которого не нашлось пары);

2. Разбить отсортированные цепочки на пары, и осуществить слияние цепочек каждой пары;

3. Если число отсортированных цепочек больше единицы, перейти к шагу 2.

Этот алгоритм можно реализовать рекурсивным способом или с помощью обычного цикла. Рекурсивный алгоритм будет таким.

Рекурсивный алгоритм сортировки слиянием

1. Если Правая_граница = 0, то

Результат = 0

Иначе

1.1. Если Правая_граница = 1, то

Результат = A_0

Иначе

1.1.1. Если Правая_граница = 2, то

Если $A_1 < A_0$

Результат = A_1, A_0

Иначе

Результат = A_0, A_1

1.1.2. Иначе

a) Упорядочить первую половину массива A:

Вызвать Сортировку(*A*; Левая_граница;
Правая_граница/2);

b) Упорядочить вторую половину массива A:

Вызвать Сортировку (*A*; Левая_гр +Правая_гр/2+1;
Правая_гр);

c) *Слияние Первой и Второй половин массива A:*

Слияние *A*[0… Левая_гр + Правая_гр/2] и
A[Левая_гр + Правая_гр/2 + 1; Правая_гр]

2. Вывести массив *A*.
3. Закончить.

Алгоритм 6. Сортировка Боуза- Нельсона

Этот метод реализует один из способов слияния. Он удобен для сортировки больших массивов, находящихся на устройствах последовательного доступа (например, винчестерах). Массив *A* разбивается на две половины: *B* и *C*. Эти половины сливаются в упорядоченные пары, объединяя первые элементы из *B* и *C* в первую пару, вторые – во вторую и т.д. Полученному массиву присваивается имя *A*, после чего операция повторяется. При этом пары сливаются в упорядоченные четверки. Предыдущие шаги повторяются: четверки сливаются в восьмерки и т.д., пока не будет упорядочен весь массив, т.к. длины частей каждый раз удваиваются. Если размер массива нечетный, или на некотором шаге получатся неполные части, то выполняют отдельно слияние начал, концов и центральной частей. Описанный алгоритм можно проиллюстрировать следующим примером.

Пример. Исходная последовательность

A = 44 55 12 42 94 18 06 67

Шаг 1

Первая половина *B* = 44 55 12 42

Вторая половина *C* = 94 18 06 67

A = 44 94 18 55 06 12 42 67 – для наглядности нечетные пары выделены.

Шаг 2

Первая половина *B* = 44 94 18 55

Вторая половина *C* =06 12 42 67

A = 06 12 44 94 18 42 55 67 – выделены упорядоченные четверки.

Шаг 3

Первая половина *B* = 06 12 44 94

Вторая половина *C* = 18 42 55 67

A = 06 12 18 42 44 55 67 94 – отсортированный массив.

Судя по описанию, алгоритм проще всего реализовать с помощью рекурсии, причем, в отличие от традиционной сортировки слиянием он не требует дополнительной памяти. Обозначим *r* - расстояние между началами сливаемых частей, *m* - их размер и *j* - наименьший номер элемента. Будем считать, что выполняется сортировка по возрастанию.

Рекурсивный алгоритм сортировки Боуза- Нельсона

1. *Слияние*

1.1.*Начало рекурсии.* Если $j+r \le n$, то

1.1.1. Если $m = 1$, то

а) Если $A[j] > A[j+r]$, то

Поменять местами $A[i]$ и $A[j+r]$.

1.2.Иначе

1.2.1. $m = m / 2$.

1.2.2. Выполнить Слияние «начал» от *j* до *m* с расстоянием *r*.

1.2.3. Если $j+r+m \le n$, то

Выполнить Слияние «концов» от $j+m$ до m с расстоянием r.

1.2.4. Выполнить Слияние «центральной части» от $j+m$ до m с расстоянием $m - r$.

2. *Основная часть рекурсии*

2.1.$m = 1$.

2.2.Повторять

2.3.*Цикл слияния списков равного размера*

2.3.1. $j = 1$.

2.3.2. Пока $j+m \leq n$

1) Выполнить Слияние (j,m,m);

2) $j = j + 2m$

2.3.3. Удвоение размера списка перед началом нового прохода

$m = 2m$.

Конец цикла 2.2, реализующего все слияния

Пока $m < n$.

3. Вывести массив A.

4. Закончить.

Число сравнений и пересылок при реализации сортировки методом Боуза - Нельсона в самом худшем случае пропорционально $n \log n$, т.е. сложность алгоритма O($n \log n$). Дополнительная память не требуется.

Алгоритм 7. Быстрая сортировка

Метод использует тот факт, что число перестановок в массиве может резко сократиться, если менять местами элементы, находящиеся на большом расстоянии. Он реализует метод «разделяй и властвуй». Для этого обычно в середине массива выбирается один элемент (опорный). Далее слева от опорного располагают все элементы, меньше его, а справа – больше. Затем тот же прием

применяют к половинкам массива. Процесс заканчивается, когда в частях массива останется по одному элементу.

В алгоритме используются два разнонаправленных процесса. Первый выполняется от начала массива и ищет элемент, больший опорного. Второй - работает с конца и ищет элемент, меньший опорного. Как только такие элементы найдены, производится их обмен местами. Далее поиск продолжается с того места, где процессы остановились.

Таким образом, когда процессы встречаются, любой элемент в первой части меньше любого во второй. Это значит, что их уже сравнивать друг с другом не придется. Остается только провести такую же операцию по отношению к полученным половинам, и так далее, пока в очередной части массива не останется один элемент. Это будет *означать*, что массив отсортирован. Очевидно, что наиболее удобный способ реализации рассмотренного метода – рекурсивный.

Общий алгоритм можно представить так.

1. Из массива выбирается некоторый опорный элемент, Опорный = $a[i]$, $i=n/2$.
2. Запускается процедура разделения массива, которая перемещает все элементы, меньшие, либо равные Опорному, влево от него, а все, большие, равные Опорному, - вправо.
3. Теперь массив состоит из двух подмассивов, причем длина левого меньше, либо равна длине правого.
4. Для обоих подмассивов, если в них больше двух элементов, рекурсивно заполняется та же процедура.

   В конце получится полностью отсортированная последовательность.

Уточненный рекурсивный алгоритм быстрой сортировки.

Исходные данные: Массив A; Левая_граница (0); Правая_граница (n-1)

1. Индекс в начале, *i* = Левая_граница.
2. Индекс в конце, *j* =Правая_граница.
3. Опорный = *A*[(Левая_граница + Правая_граница) / 2].
4. Повторять
    4.1. *Движение слева направо:*
        Пока *A[i*] < Опорный
            i = *i* + 1
    4.2. *Движение справа налево:*
        Пока *A[j*] > Опорный
            j = *j* - 1.
    4.3. Поменять местами *A[i*] и *A[j*].
    4.4. *i* = *i* + 1.
    4.5. *j* = *j* - 1.

Пока Левая_граница < Правая_граница.

5. Если Левая_граница < *j,* то отсортировать *A* от Левая_граница до *j,*
6. Если *i* > Правая_граница, то отсортировать *A* от *i* до Правая_граница.
7. Вывести массив *A*.

Алгоритм 8. Сортировка Шелла

Метод является ускоренным вариантом сортировки вставками. При этом ускорение достигается за счет увеличения расстояния, на которое перемещаются элементы. Исходный массив делится на *d* частей, содержащих *n* / *d* элементов каждый (последний подмассив может быть короче). Подмассивы содержат элементы с номерами [0, *d, 2 d* и т.д.], [1, *d* +1, *2 d* +1 и т.д.], [2, *d* + 2, 2 *d* +2 и т.д.] и т.д.

Вначале сравниваются и упорядочиваются с помощью алгоритма вставок элементы, отстоящие один от другого на расстоянии *d*, т.е. имеющие номера 0 и 1, *d* и *d* + 1, 2 *d* и 2 *d* + 1 и т.д. Затем процедура повторяется при меньших значениях *d*, например, *d* / 2. Завершается алгоритм упорядочением элементов при *d* = 1, то есть обычной

сортировкой вставками. Мы рассмотрим сортировку Шелла для начального значения *d*, равного *n* /2, и будем последовательно уменьшать его вдвое.

Алгоритм сортировки Шелла

1. Задать массив *A* из *n* чисел.
2. Выполнять
   2.1. $d = n / 2$.
   2.2. Для i от 0 до $n - d$ с шагом *d*
      2.2.1. $j = i$.
      2.2.2. $x = A_i$
      2.2.3. Пока ($j \geq d$) $(A[j] > A[j + d])$
         a) $A[j] = A[j + d]$
         b) $j = j - d$.
   2.3. $d = d / 2$.
   Пока $d > 0$.
3. Вывести массив *A*.
1. Закончить.

Порядок выполнения лабораторной работы

Работа предполагает выполнение следующих этапов.

1. Знакомство со всеми разделами руководства.

2. Получение у преподавателя задания на разработку программы для алгоритмов сортировки (см. Приложение 4).

3. Разработка и отладка заданных программ.

4. Получение верхней и экспериментальной оценки времени выполнения заданных алгоритмов и программ.

5. Нахождение предельной оценки емкости памяти, необходимой для выполнения разработанных программ.

Содержание отчета о выполненной работе

Отчет о выполненной работе должен содержать.

1. Название и цель работы.
2. Словесное описание заданных алгоритмов сортировки.
3. Тексты программ.
4. Формулы верхней оценки временной и емкостной сложности заданных алгоритмов.
5. Результаты экспериментальной оценки временной и емкостной сложности заданных алгоритмов.

Контрольные вопросы

1. Что такое сортировка и для чего она нужна?
2. По каким признакам выполняется классификация алгоритмов сортировки?
3. Как оценивается временная сложность алгоритмов упорядочения?
4. Как оценивается емкостная сложность алгоритмов сортировки?
5. Какой метод упорядочения самый простой?
6. Какой алгоритм сортировки самый быстрый?
7. Какие алгоритмы пригодны для упорядочения файлов?
8. Чем отличается сортировка чисел от строк?
9. Как Вы определили время выполнения Ваших алгоритмов?
10. Как Вы определили объем памяти, необходимой для выполнения Ваших алгоритмов?
11. Каковы основные отличия сортировки вставками от «пузырьковой»?
12. Каковы основные отличия упорядочения слиянием от «пузырьковой»?
13. Каковы основные отличия сортировки слиянием от метода Боуза-Нельсона?
14. Каковы основные отличия упорядочения слиянием и вставками?
15. Каковы отличительные особенности быстрой сортировки?
16. Как выполняется упорядочение Шейкером?
17. Каковы особенности сортировки Шелла и для каких данных она предпочтительна?
18. У каких известных Вам методов сортировки временная сложность зависит от объема используемой памяти?

Лабораторная работа №5
ИССЛЕДОВАНИЕ И ОЦЕНКА АЛГОРИТМОВ ПОИСКА НА ДЕРЕВЬЯХ

Краткая теория

Цель работы. Разработка программ, реализующих алгоритмы формирования и обхода двоичных и B+ деревьев, а также поиска элементов в них, и оценка их временной и пространственной сложности.

Двоичные и B+ деревья. Основные понятия и определения

Массивы относятся к линейным структурам данных. Они не позволяют реализовать эффективные алгоритмы для решения некоторых задач, например, поиска. Для таких задач более предпочтительным является древовидное представление данных.

Древовидная структура содержит множество узлов (nodes), происходящих от единственного начального, который называют корнем (root). На Рис. 5.1 корнем является узел *A*. Принято узел считать родителем (parent), указывающим на 0, 1 или более других узлов, называемых сыновьями (children). Например, узел B является родителем сыновей *E* и *F*. Дерево может представлять несколько поколений. Сыновья узла и сыновья их сыновей называются потомками (descendants), а родители и прародители – предками (ancestors) этого узла. Например, узлы *E, F, I, J* – потомки узла *B*. Каждый некорневой узел имеет только одного родителя, и каждый родитель имеет 0 или более сыновей. Узел, не имеющий детей (*E, G, H, I, J*), называется листом (leaf).

Каждый узел дерева является корнем поддерева (subtree), которое включает в себя этот узел и всех его потомков. Так, *F* есть корень поддерева, содержащего узлы *F, I* и *J*. Узел *G* является корнем

поддерева без потомков. Таким образом, узел *A* есть корень поддерева, которое само оказывается деревом.

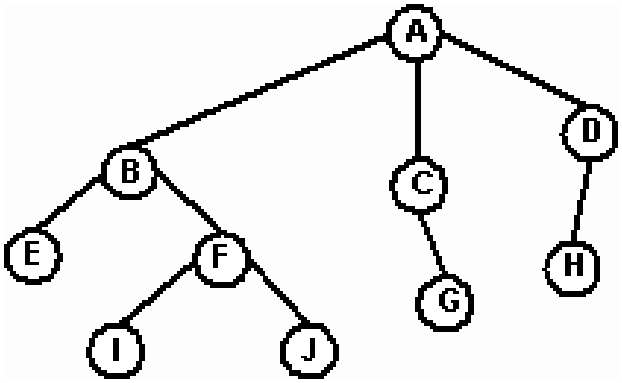


Рис. 5.1. Графическое изображение дерева

Каждый узел дерева является корнем поддерева (subtree), которое включает в себя этот узел и всех его потомков. Так, *F* есть корень поддерева, содержащего узлы *F*, *I* и *J*. Узел *G* является корнем поддерева без потомков. Таким образом, узел *A* есть корень поддерева, которое само оказывается деревом.

Переход от родительского узла к дочернему и другим потомкам осуществляется вдоль пути (path). Например, на рисунке 5.2 путь от корня *A* к узлу *F* проходит от *A* к *B* и от *B* к *F*. Тот факт, что каждый некорневой узел имеет единственного родителя, гарантирует, что существует единственный путь из любого узла к его потомкам. Длина пути от корня к этому узлу есть уровень узла. Уровень корня равен 0. Каждый сын корня является узлом 1-го уровня, следующее поколение – узлами 2-го уровня и т.д. Например, на рисунке 5.2 узел *F* является узлом 2-го уровня с длиной пути 2.

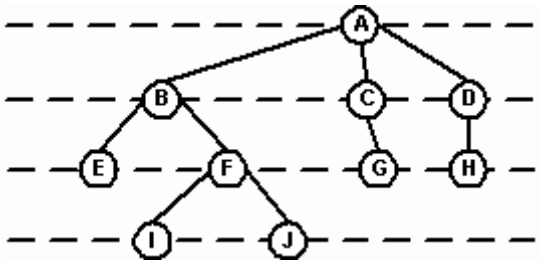


Рис. 5.2. Уровень узла и длина пути

Глубина (depth) дерева есть его максимальный уровень. Понятие глубины также может быть описано в терминах пути. Глубина дерева есть длина самого длинного пути от корня до узла. На рис. 5.2 глубина дерева равна 3.

В программировании широкое распространение получили так называемые ***двоичные*** (***бинарные*** - binary trees) деревья, которые имеют унифицированную структуру, обеспечивающую разнообразные алгоритмы эффективного доступа к элементам. Примеры таких деревьев представлены на рисунке 5.3.

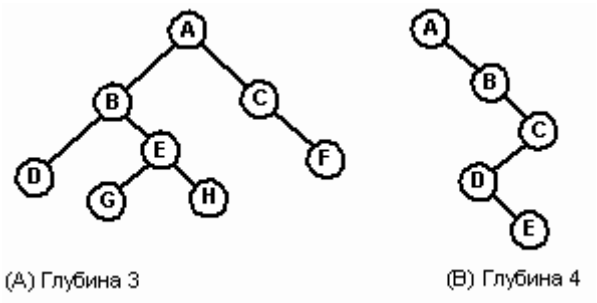


Рис. 5.3. Бинарные деревья

У каждого узла бинарного дерева может быть 0, 1 или 2 сына. Причем, узел слева называют левым сыном (left child), а справа – правым (right child). Эти наименования связаны с графическим представлением дерева. Двоичное дерево является рекурсивной структурой. Каждый узел – это корень своего собственного поддерева. У него есть сыновья, которые сами являются корнями деревьев, которые называются *левым* и *правым поддеревьями* соответственно.

Таким образом, бинарное дерево - это конечное множество элементов, которое либо пусто, либо содержит один элемент, называемый корнем. Остальные элементы делятся на два непересекающихся подмножества, каждое из которых, в свою очередь, является бинарным деревом. Эти подмножества называются правым и левым поддеревьями исходного дерева.

Рекурсивное определение двоичного дерева может быть сформулировано следующим образом.

Двоичное дерево - это такое множество узлов *B*, что

a) *B* является деревом, если множество узлов пусто (пустое дерево – тоже дерево);

b) *B* разбивается на три непересекающихся подмножества:

- $\{R\}$ корневой узел;
- $\{L_1, L_2, ..., L_m\}$ левое поддерево *R*;
- $\{R_1, R_2, ..., R_m\}$ правое поддерево *R*.

И определение, и процедуры обработки деревьев, как правило, рекурсивны.

Узлы дерева могут быть пронумерованы следующим образом.

Номер корня всегда равен 1, левый потомок получает номер 2, правый - номер 3. Левый потомок узла 2 должен получить номер 4, а правый - 5, левый потомок узла 3 получит номер 6, правый - 7 и т.д. Несуществующие узлы не нумеруются, что обычно не нарушает приведенного порядка. При такой системе нумерации в дереве каждый узел получает уникальный номер.

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

Почти полное бинарное дерево представляет собой бинарное дерево, для которого существует неотрицательное целое k такое, что:

1) каждый лист в дереве имеет уровень k или k+1;

2) если узел дерева имеет правого потомка уровня k+1, тогда все его левые потомки, являющиеся листами, также имеют уровень k+1.

В программировании структура бинарного дерева образуется элементами, структура которых приведена на рисунке 5.4. Узел содержит поле данных и два поля с указателями: левым (left) и правым (right), поскольку они указывают на соответствующие поддеревья. Значение NULL является признаком пустого дерева.

Корневой узел определяет входную точку дерева, а поля указателей – узлы следующего уровня. Листовой узел содержит NULL в полях правого и левого указателей. В java узел

представляется объектом класса с соответствующей структурой, например.

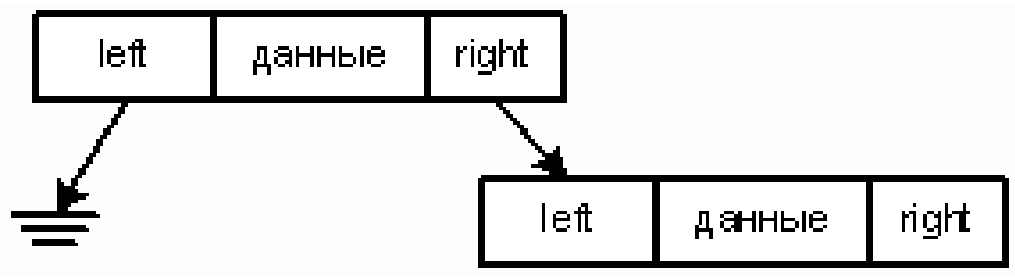


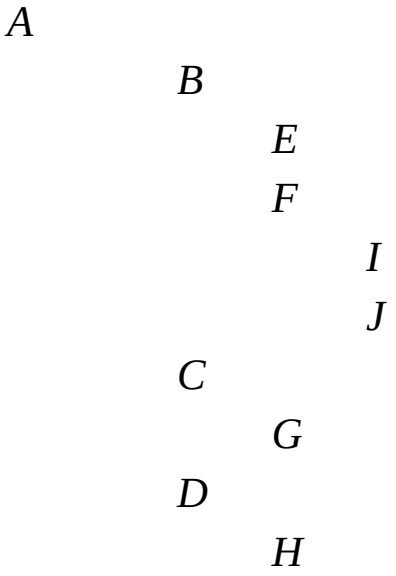
Рис. 5.4. Представление узлов дерева в программировании

```
class Tree{
        public Tree left; // левый указатель
        public Tree right;  // правый указатель
        public int key; // информационное поле целого типа
            }
```

Другой пример – дерево, узел которого хранит его обозначение в виде латинской буквы с номером и числовое поле. Корень обозначается буквой *A*, узлы 2-го уровня – *B*, 3-го – *C* и т.д. Нумерация узлов *k*-ого уровня выполняется от 1 до 2^k.

```
Class MyTree{
// Уровень узла
  private char Level;
// Номер узла
  private int Number;
// Поле узла
  private int key; }
```

При выводе из программы на экран удобно представлять дерево в виде строк текста с отступами. При этом начинают с корня, затем – его левые потомки и только потом – правые. Например, граф на рисунке 5.1 может быть представлен следующим образом.



B и B+ деревья

B-дерево является разновидностью дерева поиска. Такая структура была предложена Р. Бэйером и Е. МакКрейтом в 1970 году. Характерной для B-дерева является его малая глубина (высота), обычно равная 2 - 3. Основными свойствами таких деревьев являются: сбалансированность, ветвистость, отсортированность и логарифмическое время выполнения всех стандартных операций (поиск, вставка, удаление). Сбалансированность означает, что все листья находятся на одинаковом расстоянии от корня. В отличие от бинарных, B-деревья допускают большое число потомков для любого узла. Это свойство называется *ветвистостью*. Благодаря ему, B-деревья очень удобны для хранения крупных последовательных блоков данных, поэтому такая структура часто находит применение в базах данных и файловых системах.

Важным параметром B-дерева является его степень (порядок) *m*– максимальное число потомков для любого узла. Ключи располагаются между ссылками на потомков и, таким образом, ключей всегда на 1 меньше. В организации B-дерева можно выделить несколько правил:

1. Каждый узел содержит строго меньше *m* (порядок дерева) потомков.

2. Каждый узел содержит не менее $m/2$ потомков.

3. Корень может содержать меньше $m/2$ потомков.

4. У корневого узла есть хотя бы 2 потомка, если он не является листом.

5. Все листья находятся на одном уровне и содержат только данные (ключи).

Пример B-дерева приведен на рисунке 5.5.

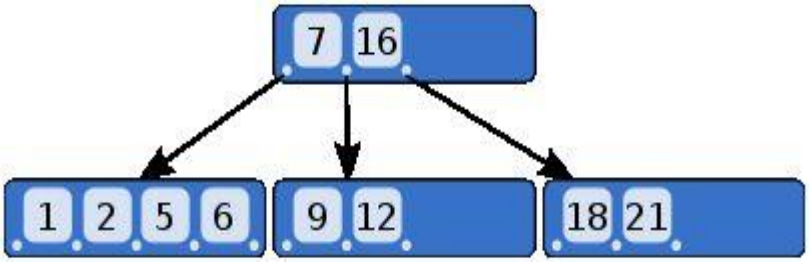


Рис. 5.5. Пример B+ дерева.
Стрелки идут к потомкам. В ключах записаны числа

Действия с бинарными и B+ деревьями

Узлы дерева обычно используются для хранения информации. Основными операциями над деревьями являются:

1) построение дерева,
2) создание узла,
3) включение узла в дерево,
4) удаление узла из дерева,
5) обход дерева,
6) поиск элементов (узлов).

Построение бинарного дерева

В программировании наибольшее распространение получили *упорядоченные двоичные деревья поиска*. Правило построения такого дерева следующее: элементы, у которых значение некоторого признака (ключа) меньше, чем у корня, всегда включаются слева от некоторого поддерева, а элементы со значениями, большими, чем у

корня - справа. Этот принцип используется как при формировании двоичного дерева, так и при поиске в нем элементов.

Таким образом, при поиске элемента с некоторым значением ключа происходит спуск по дереву, начиная от корня. Выбор направления следующего шага – направо или налево (по значению искомого ключа) - происходит в каждом очередном узле на пути. При поиске элемента результатом будет либо узел с заданным ключом, либо лист с «нулевой» ссылкой (искомый элемент отсутствует на проделанном пути). Последняя ситуация может возникнуть, например, если целью поиска было включение очередного узла в дерево. При этом справа или слева (в зависимости от значения признака) к листу будет присоединен новый узел.

Рассмотрим пример формирования двоичного дерева. Предположим, что нужно сформировать двоичное дерево, узлы (элементы) которого имеют следующие значения признака: 20, 10, 35, 15, 17, 27, 24, 8, 30. В этом же порядке они и будут поступать для включения в двоичное дерево.

Первым узлом в дереве (корнем) станет узел со значением 20. В общем случае, поиск места подключения очередного элемента всегда начинается с корня. К корню слева подключается элемент 10, а справа - 35. Далее элемент 15 подключается справа к 10, проходя путь: корень 20 - налево - элемент 10 - направо - подключение, так как дальше пути нет. Процесс продолжается до тех пор, пока не будут включены в дерево все элементы. Результат представлен на рис. 5.6.

Отметим, что при создании двоичного дерева обычно считается, что ключи элементов не повторяются.

Узлы, у которых заполнены два адреса связи считаются полными, а с одним адресом – неполными. Элементы с двумя незаполненными адресами называются концевыми (листьями).

В *упорядоченном бинарном дереве* значение ключевого атрибута каждого узла должно быть больше, чем значение ключа у любого

элемента на его левой ветви, и не меньше, чем ключ на его правой ветви.

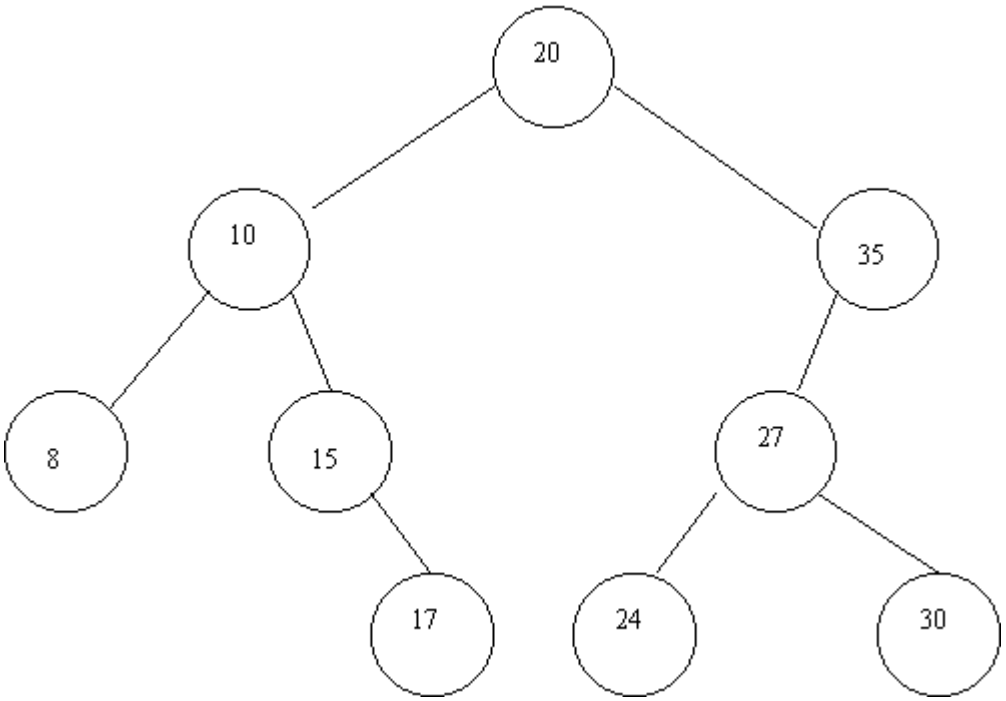


Рис. 5.6. Построение бинарного дерева.
Значения элементов дерева: 20, 10, 35, 15, 17, 27, 24, 8, 30

Алгоритм построения упорядоченного бинарного дерева может быть сформулирован так.

1.Первый элемент с ключом $p[1]$ становится корнем дерева.

2.Значение ключа второго узла $p[2]$ сравнивается с $p[1]$ (корня дерева). Если $p[2] < p[1]$, то второй элемент помещается на левой от корня ветви, в противном случае – на правой. В результате будет получено упорядоченное дерево из первых двух узлов.

3. Далее на каждом шаге создается упорядоченное дерево из первых i элементов. Выбор i-го узла производится следующим образом. Ключ $p[i]$ сравнивается с корневым значением и выполняется переход по левому адресу (если $p[1]>p[i]$), в противном случае (при $p[1] <= p[i]$) – по правому адресу. Далее ключ достигнутого узла также сравнивается с $p[i]$, и снова организуется

переход по левому и правому адресу и т.д. При нахождении незаполненного адреса связи ему присваивается ключ $p[i]$.

Пункт 3 повторяется до тех пор, пока не будут включены в дерево все элементы исходного массива.

Из рассмотренного алгоритма следует, что основным методом, используемым при построении двоичного дерева и решении других задач, является ***поиск***. В процессе поиска в упорядоченном бинарном дереве просматривается некоторый путь, начинающийся всегда в его корне. Искомое значение ключа q сравнивается со значением корня $p[1]$. Если $p[1]>q$, просмотр дерева продолжается по левой от корня ветви, иначе (если $p[1] <= q$) – по правой. Для произвольного узла с ключом $p[i]$ могут быть получены следующие результаты сравнения:

a) $p[i] = q$ – элемент, удовлетворяющий условию поиска найден, и поиск заканчивается по правой ветви.

b) $p[i] >q$ – производится переход к элементу, расположенному на левой ветви p(i).

c) $p[i] < q$ – производится переход к элементу, расположенному на правой ветви $p[i]$.

Поиск заканчивается, когда у какого-либо узла отсутствует адрес связи, необходимый для дальнейшего продолжения поиска.

Построение *B*+ дерева

При выполнении этой операции руководствуются основным правилом: изначально все узлы кроме последнего содержат один дополнительный элемент (то есть m+1 потомков в сумме), который будет использован для построения внутренних узлов.

Рассмотрим алгоритм на примере. Пусть количество потомков для каждого листового узла не более 4, то есть степень дерева $m = 5$. На вход подана последовательность чисел от 1 до 24. Разобьём их на блоки по приведенному правилу:

1|2|3|4|5 **6|7|8|9|10** **11|12|13|14|15** **16|17|18|19|20** **21|22|23|24**

Из слоя листовых узлов строим следующий уровень путём вынесения туда лишних (в данном случае пятых) элементов на уровень выше. У четырех из пяти блоков такие элементы есть. Это – 5, 10, 15 и 20. Полученный новый уровень разбиваем в соответствии с тем же правилом.

Пусть для внутренних (нелистовых) узлов будет задано ограничение на количество потомков, равное 2 (по свойствам 2 и 3 узлов *B*-дерева). Результат вынесения элементов на верхний уровень приведен ниже.

5|10|15 **20**

1|2|3|4 **6|7|8|9** **11|12|13|14** **16|17|18|19** **21|22|23|24**

Продолжаем процесс, пока не получим корень.

15

5|10 **20**

1|2|3|4 **6|7|8|9** **11|12|13|14** **16|17|18|19** **21|22|23|24**

Осталось добавить ссылки и получится готовое B-дерево.

Включение узла в двоичное дерево

Добавление элемента (ДОБАВИТЬ)
Дано: дерево *T* и пара (*K,V* – ключ и вершина).
Задача: добавить пару (*K, V*) в дерево *T*.

Алгоритм включения узла в дерево

1. Если дерево пусто, заменить его на дерево с одним корневым узлом ((*K,V*), ПУСТО, ПУСТО) и остановиться

   Иначе сравнить K с ключом корневого узла X:

   1.1. Если *K*>=*X*, рекурсивно добавить (*K,V*) в правое поддерево *T*

Иначе (если $K<X$) рекурсивно добавить (K,V) в левое поддерево T.

Включение узла в *B* дерево

Введём определение: *дерево потомков узла* – это поддерево, состоящее из самого узла и его потомков.

Вначале определим функцию, которая добавляет элемент с ключом K к дереву потомков узла X. После выполнения функции во всех пройденных узлах, кроме, может быть, самого узла X, будет меньше $2t - 1$, но не меньше $t - 1$ ключей. Величина t определяется из формулы:

$$m = 2(t - 1).$$

Алгоритм добавления элемента к дереву потомков

1. Если X – не лист
   1.1. Определяем интервал, где должен находиться K. Пусть Y – соответствующий сын.
   1.2. Рекурсивно добавляем K к дереву потомков Y.
   1.3. Если узел Y полон, то есть содержит $2t - 1$ ключей, расщепляем его на два. Узел $Y1$ получает первые $t - 1$ ключей Y, и первые t его потомков, а узел $Y2$ – последние $t - 1$ ключей Y и последние t его потомков. Средний из ключей узла Y попадает в узел X, а указатель на Y в узле X заменяется указателями на узлы $Y1$ и $Y2$.
2. Если X – лист, просто добавляем туда ключ K.

Теперь сформулируем алгоритм добавления элемента с ключом K ко всему дереву. Буквой R обозначим корневой узел.

Алгоритм добавления элемента к B дереву

1. Добавим K к дереву потомков R.
2. Если в результате R содержит $2t - 1$ ключей, расщепляем его на два. Узел $R1$ получает первые $t - 1$ из ключей R и первые t его потомков, а $R2$ – последние $t - 1$ из ключей R и последние t его потомков. Средний из ключей узла R попадает во вновь

созданный узел, который становится корневым. Узлы $R1$ и $R2$ становятся его потомками.

Зачастую добавление элемента приводит к полному или частичному сдвигу всех узлов дерева. Таким образом, эта операция является одной из наиболее трудоемких.

Удаление узла из бинарного дерева

Удаление узла (УДАЛИТЬ)
Дано: дерево T с корнем n и ключом K.
Задача: удалить из дерева T узел с ключом K (если такой есть).

Алгоритм удаления узла из дерева

1. Если дерево T пусто, остановиться

   Иначе сравнить K с ключом X корневого узла n.

   2.1. Если $K>X$, рекурсивно удалить K из правого поддерева T

   2.2. Иначе если $K<X$, рекурсивно удалить K из левого поддерева T.

   2.3. Если $K=X$, то необходимо рассмотреть два случая.

   2.3.1. Если одного из детей нет, то значения второго ребёнка m ставим вместо соответствующих значений корневого узла, затирая его старые значения, и освобождаем память, занимаемую узлом m.

   2.3.2. Если оба потомка присутствуют, то

   2.3.2.1. найдём узел m, являющийся самым левым узлом правого поддерева;

   2.3.2.2. скопируем значения полей (ключ, значение) узла m в соответствующие поля узла n.

   2.3.2.3. у предка узла m заменим ссылку на узел m ссылкой на правого потомка узла m (который может быть равен ПУСТО).

2.3.2.4. освободим память, занимаемую узлом *m* (на него теперь никто не указывает, а его данные были перенесены в узел *n*).

Удаление узла из *B* дерева

При удалении узла (ключа) могут возникнуть две проблемы:

a) Удалённый элемент является разделителем потомков внутри какого-либо нелистового узла;

b) После удаления в узле может остаться менее допустимого m/2 числа потомков.

Для того чтобы устранить эти проблемы, требуется перебалансировка дерева. Она может быть выполнена после удаления узла (ключа) или до него. Рассмотрим первый вариант.

Алгоритм удаления узла из B дерева

1. Если корень одновременно является листом (в дереве всего один узел), удаляем ключ из этого узла.
2. В противном случае находим узел, содержащий ключ, запоминая путь к нему. Пусть этот узел – *X*.
3. Если *X* – лист, удаляем оттуда ключ.
   - 3.1. Если в узле *X* осталось не меньше $t - 1$ ключей, останавливаемся. Иначе проверяем количество ключей в следующем, а потом в предыдущем узле.
   - 3.2. Если следующий узел есть, и в нём не менее *t* ключей, добавляем в *X* ключ-разделитель между ним и следующим узлом, а на его место ставим первый ключ следующего узла, после чего останавливаемся.
   - 3.3. В противном случае, если есть предыдущий узел, и в нём не менее *t* ключей, добавляем в *X* ключ-разделитель между ним и предыдущим узлом, а на его место ставим последний ключ предыдущего узла, после чего останавливаемся.

3.4. Наконец, если и с предыдущим ключом не получилось, объединяем узел X со следующим или предыдущим узлом, и в объединённый узел перемещаем ключ, разделяющий два узла. При этом в родительском узле может остаться только $t - 2$ ключа. Тогда, если это не корень, выполняем аналогичную процедуру с ним.

3.5. Если в результате дошли до корня, и в нём осталось от 1 до $t - 1$ ключей, остановка, потому что корень может иметь и меньше $t - 1$ ключей.

3.6. Если в корне не осталось ни одного ключа, исключаем корневой узел, а его единственный потомок делаем новым корнем дерева.

4. Если X – не лист, а K – его i-й ключ, удаляем самый правый ключ из поддерева потомков i-го сына X, или, наоборот, самый левый ключ из поддерева потомков $(i + 1)$-го сына X.

Обход дерева

Обход дерева может выполняться одним из следующих способов:

1) «сверху вниз» или *префиксный обход* (функция_прямого_вызова) — обойти всё дерево по порядку (вершина, левое поддерево, правое поддерево);

2) «снизу вверх» или *постфиксный обход* (функция_обратного_вызова) — обойти всё дерево по порядку (левое поддерево, правое поддерево, вершина);

3) «симметричный обход» или *инфиксный обход* (функция_симметричного_вызова) — обойти всё дерево по порядку (левое поддерево, вершина, правое поддерево).

Эти процедуры, судя по названию и описанию, имеют рекурсивный характер. Их алгоритмы можно описать следующим образом.

ИНФИКСНЫЙ_ОБХОД позволяет обойти все узлы дерева в порядке возрастания ключей и применить к каждому узлу заданную

пользователем функцию обратного вызова. Эта функция обычно работает только с парой (*K*,*V*), хранящейся в узле. Операция ИНФИКСНЫЙ_ОБХОД реализуется рекурсивным образом: сначала она запускает себя для левого поддерева, потом выполняет функцию обратного вызова для корня, а затем вызывает себя для правого поддерева.

Инфиксный (симметричный) обход дерева

Дано: дерево *T* и функция *f*
Задача: применить *f* ко всем узлам дерева *T* в порядке возрастания ключей

Алгоритм обхода дерева

1. Если дерево пусто, остановиться.
   1.1. Иначе
      1.1.1. Рекурсивно обойти правое поддерево *T*.
      1.1.2. Применить функцию *f* к корневому узлу.
      1.1.3. Рекурсивно обойти левое поддерево *T*.

В простейшем случае, функция f может выводить значение пары (*K*,*V*). При использовании операции ИНФИКСНЫЙ_ОБХОД будут выведены все пары в порядке возрастания ключей. Если же использовать ПРЕФИКСНЫЙ_ОБХОД, то пары будут выведены в порядке, соответствующем описанию дерева.

Поиск элемента в дереве

В общем случае для выполнения этой операции могут использоваться различные условия. В зависимости от способа их задания и организации поиска различают большое количество видов таких операций. Наибольшее распространение получили следующие виды поиска по простому условию:

d) совпадению,
e) близости снизу

f) близости сверху.

Первый вид предполагает нахождение элемента с заданным значением ключа *K*. Если такого ключа нет, то операция закончилась безуспешно.

При *поиске по близости* операция всегда заканчивается успешно. Если он выполняется снизу, то результатом будет элемент, ключ которого является ближайшим меньшим искомого. При поиске по близости сверху результат представляет собой элемент с ключом, ближайшим большим искомого.

Поиск элемента двоичного дерева

Рассмотрим алгоритм поиска по совпадению.

Поиск элемента (ИСКАТЬ)

Дано: дерево *T* и ключ *K*.

Задача: проверить, есть ли узел с ключом *K* в дереве *T*, и если да, то вернуть ссылку на этот узел.

Алгоритм поиска элемента двоичного дерева по ключу

1. Если дерево пусто, сообщить, что узел не найден, и остановиться.
2. Иначе сравнить *K* со значением ключа корневого узла *X*.
   - 2.1. Если $K=X$, выдать ссылку на этот узел и остановиться
     - 2.1.1. Иначе, если $K>X$, рекурсивно искать ключ *K* в правом поддереве *T*.
     - 2.1.2. Иначе (если $K<X$) рекурсивно искать ключ *K* в левом поддереве *T*.

Поиск элемента *B* дерева

Благодаря структуре этого дерева и постоянно поддерживаемому балансу, поиск осуществляется за логарифмическое время от количества элементов. Сам алгоритм описывается следующим образом. Пусть ищется значение *X* в заданном *B*-дереве.

Алгоритм поиска элемента B дерева

1. Начинаем от корня
2. Если *X* –один из ключей текущего узла, то СТОП, ключ найден.
   Иначе находим пару соседних ключей *Y, Z*, таких что *X* лежит в интервале от *Y* до *Z*.
3. Переходим к потомку между *Y* и *Z*.
4. Возврат к шагу 2.
5. Если дошли до листа и *X* нет среди ключей, то СТОП, ключ и элемент не найден.

Порядок выполнения лабораторной работы

Подготовка к работе

Подготовка предполагает выполнение следующих этапов.

1. Знакомство со всеми разделами руководства.

2. Разработка алгоритма и программы построения двоичного дерева поиска.

3. Разработка алгоритма и программы построения B+ дерева.

Последовательность выполнения лабораторной работы

1. Разработать алгоритм и программу построения двоичного дерева поиска:
   a) Получить у преподавателя задание на величину диапазона значений ключей и размер их массива *n*.
   b) Сформировать массив ключей, значения которых задаются с помощью датчика случайных чисел.
   c) По заданию преподавателя упорядочить значения ключей по возрастанию (убыванию).
   d) Приняв за корень элемент с ключом из середины массива (ключ с номером *n* / 2), построить двоичное дерево поиска.

Информационные поля узлов дерева можно оставить пустыми.

e) Вывести значения ключей по уровням дерева.

2. Разработать алгоритм и программу построения B+ дерева:
   a) Получить у преподавателя задание на количество ярусов, величину диапазона значений ключей и размер их массива *n* на каждом ярусе.
   b) Сформировать массив ключей, значения которых задаются с помощью датчика случайных чисел.
   c) По заданию преподавателя упорядочить значения ключей по возрастанию (убыванию).
   d) Приняв за корень элемент с ключом из середины массива (ключ с номером *n* / 2), построить B+ дерево. Информационные поля узлов дерева можно оставить пустыми.
   e) Вывести значения ключей по уровням дерева.
3. Разработать алгоритм и программу поиска заданного преподавателем типа (по совпадению, интервалу или близости) на двоичном и B+ дереве.
4. Оценить сложность разработанных алгоритмов.

Содержание отчета о выполненной работе

Отчет должен содержать следующее.

- Название и цель работы, а также исходные данные.
- Пошаговые алгоритмы построения двоичного и B+ дерева, а также тексты их программ.
- Распечатки результатов, полученных с помощью ЭВМ.
- Пошаговые алгоритмы поиска на двоичном и B+ дереве, а также тексты их программ.
- Выводы о сложности разработанных алгоритмов.

Типы алгоритмов, исследуемых в лабораторной работе 1

1. Составить программу, которая формирует одномерный массив из *n*случайных чисел. Определить среднее арифметическое этих чисел. Значение *n* меняется в пределах от 10 до 50 миллионов.
2. Составить программу, которая формирует одномерный массив из*n*случайных чисел. Получить из него два новых массива: один из четных чисел, а другой из нечетных. Значение *n* меняется в пределах от 10 до 50 миллионов.
3. Составить программу, которая формирует одномерный массив из*n*случайных чисел. Определить сумму отрицательных чисел и отдельно сумму остальных. Значение *n* меняется в пределах от 10 до 50 миллионов.
4. Составить программу, которая формирует одномерный массив из*n*случайных чисел. Определить количество четных чисел и количество нечетных чисел. Значение *n* меняется в пределах от 10 до 50 миллионов.
5. Составить программу, которая формирует одномерный массив из*n*случайных чисел. Отдельно определить произведение четных чисел, и произведение нечетных чисел. Значение *n*меняется в пределах от 10 до 50 миллионов.
6. Составить программу, которая формирует матрицу из *n*n*случайных чисел. Определить произведение чисел, лежащих на главной диагонали матрицы. Значение *n* меняется в пределах от 5 до 10 тысяч.
7. Составить программу, которая формирует матрицу из *n*n*случайных чисел. Определить произведение чисел, лежащих на побочной диагонали матрицы. Значение *n* меняется в пределах от 5 до 10 тысяч.

8. Составить программу, которая формирует матрицу из *n***n*случайных чисел. Определить сумму чисел, лежащих выше главной диагонали матрицы. Значение *n* меняется в пределах от 5 до 10 тысяч.

9. Составить программу, которая формирует матрицу из n*n случайных чисел. Определить произведение всех чисел в матрице. Значение *n*меняется в пределах от 5 до 10 тысяч.

10. Составить программу, которая формирует матрицу из n*n случайных чисел. Определить сумму отрицательных чисел и отдельно сумму остальных. Значение *n* меняется в пределах от 5 до 10 тысяч.

11. Составить программу, которая формирует матрицу из n*n случайных чисел. Определить количество четных чисел и количество нечетных. Значение *n* меняется в пределах от 5 до 10 тысяч.

Приложение 2

Типы алгоритмов, исследуемых в лабораторной работе 2

1. Разработать алгоритм и программу простого линейного поиска с циклом For. В качестве исходных данных использовать строку текста, из которой необходимо выделить слова. Аргумент поиска – слово.
2. Разработать алгоритм и программу ускоренного линейного поиска. В качестве исходных данных использовать строку текста, из которой необходимо выделить слова. Аргумент поиска – слово.
3. Разработать алгоритм и программу дихотомического поиска. В качестве исходных данных использовать массив целых чисел, который вводится с клавиатуры. Аргумент поиска – число.
4. Разработать алгоритм и программу дихотомического поиска. В качестве исходных данных использовать массив целых чисел, который формируется с помощью датчика случайных чисел с диапазоном от 0 до 100. Аргумент поиска – число.
5. Разработать алгоритм и программу интерполирующего поиска. В качестве исходных данных использовать массив целых чисел, который вводится с клавиатуры. Аргумент поиска – число.
6. Разработать алгоритм и программу интерполирующего поиска. В качестве исходных данных использовать массив целых чисел, который формируется с помощью датчика случайных чисел с диапазоном от 0 до 100. Аргумент поиска – число.
7. Разработать алгоритм и программу простого линейного поиска с циклом For. В качестве исходных данных использовать строку текста, из которой необходимо выделить слова. Затем слова упорядочить по алфавиту. Аргумент поиска – слово.
8. Разработать алгоритм и программу ускоренного линейного поиска. В качестве исходных данных использовать строку текста, из

которой необходимо выделить слова. Затем слова упорядочить по алфавиту. Аргумент поиска – слово.

Приложение 3

Типы алгоритмов, исследуемых в лабораторной работе 3

Разработать следующие алгоритмы и программы с использованием рекурсии.

1. Линейного поиска целочисленного значения ключа в заданном массиве и вывода этого массива.
2. Линейного поиска слова в заданном массиве и вывода этого массива.
3. Дихотомического поиска целочисленного значения ключа в заданном массиве и вывода этого массива.
4. Интерполирующего поиска целочисленного значения ключа в заданном массиве и вывода этого массива.
5. Вычисления целой степени целого числа.
6. Вычисления целой степени вещественного числа.
7. Перевода целого числа, введенного с клавиатуры, в двоичную систему счисления.
8. Перевода целого числа, введенного с клавиатуры, в систему счисления с основанием q.
9. Ввода одномерного массива и линейного поиска целочисленного значения ключа в нем.
10. Ввода одномерного массива слов и линейного поиска заданного слова в нем.
11. Ввода одномерного массива и дихотомического поиска целочисленного значения ключа в нем.
12. Ввода одномерного массива и интерполирующего поиска целочисленного значения ключа в нем.

Приложение 4

Типы алгоритмов, исследуемых в лабораторной работе 4

1. Составить две программы, которые реализуют алгоритмы простой сортировки «пузырьком» и вставками. Исходные данные задавать с помощью датчика случайных чисел.

2. Составить две программы, которые реализуют алгоритмы простой сортировки «пузырьком» и выбором. Исходные данные задавать с помощью датчика случайных чисел.

3. Составить две программы, которые реализуют алгоритмы простой сортировки «пузырьком» и шейкером. Исходные данные задавать с помощью датчика случайных чисел.

4. Составить две программы, которые реализуют алгоритмы простой сортировки «пузырьком» и слиянием. Исходные данные задавать с помощью датчика случайных чисел.

5. Составить две программы, которые реализуют алгоритмы простой сортировки «пузырьком» и быстрой сортировки. Исходные данные задавать с помощью датчика случайных чисел.

6. Составить две программы, которые реализуют алгоритмы простой сортировки «пузырьком» и сортировки Шелла. Исходные данные задавать с помощью датчика случайных чисел.

7. Составить две программы, которые реализуют алгоритмы простой сортировки «пузырьком» и методом Боуза- Нельсона. Исходные данные задавать с помощью датчика случайных чисел.

8. Составить две программы, которые реализуют алгоритмы ускоренной сортировки «пузырьком» и вставками. Исходные данные задавать с помощью датчика случайных чисел.

9. Составить две программы, которые реализуют алгоритмы ускоренной сортировки «пузырьком» и выбором. Исходные данные задавать с помощью датчика случайных чисел.

10. Составить две программы, которые реализуют алгоритмы ускоренной сортировки «пузырьком» и шейкером. Исходные данные задавать с помощью датчика случайных чисел.

11. Составить две программы, которые реализуют алгоритмы ускоренной сортировки «пузырьком» и слиянием. Исходные данные задавать с помощью датчика случайных чисел.

12. Составить две программы, которые реализуют алгоритмы ускоренной сортировки «пузырьком» и быстрой сортировки. Исходные данные задавать с помощью датчика случайных чисел.

13. Составить две программы, которые реализуют алгоритмы ускоренной сортировки «пузырьком» и сортировки Шелла. Исходные данные задавать с помощью датчика случайных чисел.

14. Составить две программы, которые реализуют алгоритмы ускоренной сортировки «пузырьком» и методом Боуза- Нельсона. Исходные данные задавать с помощью датчика случайных чисел.

Приложение 5

Типы деревьев, исследуемых в лабораторной работе 5

Вариант 1

1. Построить двоичное дерево, содержащее n = 15 узлов. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 100.
2. Построить B+ дерево, содержащее n = 15 узлов и имеющее степень m = 5. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 100.
3. Обеспечить обход деревьев «сверху вниз».
4. Выполнить поиск значения ключа по совпадению.

Вариант 2

1. Построить двоичное дерево, содержащее n = 16 узлов. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 150.
2. Построить B+ дерево, содержащее n = 16 узлов и имеющее степень m = 5. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 150.
3. Обеспечить обход деревьев «снизу вверх».
4. Выполнить поиск значения ключа по близости снизу.

Вариант 3

1. Построить двоичное дерево, содержащее n = 12 узлов. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 80.
2. Построить B+ дерево, содержащее n = 12 узлов и имеющее степень m = 5. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 80.

3. Обеспечить симметричный обход деревьев.
4. Выполнить поиск значения ключа по близости сверху.

Вариант 4

1. Построить двоичное дерево, содержащее n = 20 узлов. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 180.
2. Построить B+ дерево, содержащее n = 20 узлов и имеющее степень m = 5. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 180.
3. Обеспечить обход деревьев «сверху вниз».
4. Выполнить поиск значения ключа по совпадению.

Вариант 5

1. Построить двоичное дерево, содержащее n = 20 узлов. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 180.
2. Построить B+ дерево, содержащее n = 20 узлов и имеющее степень m = 5. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 180.
3. Обеспечить обход деревьев «снизу вверх».
4. Выполнить поиск значения ключа по близости снизу.

Вариант 6

1. Построить двоичное дерево, содержащее n = 20 узлов. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 200.
2. Построить B+ дерево, содержащее n = 20 узлов и имеющее степень m = 5. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 200.
3. Обеспечить симметричный обход деревьев.
4. Выполнить поиск значения ключа по близости сверху.

Вариант 7

1. Построить двоичное дерево, содержащее n = 18 узлов. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 160.
2. Построить B+ дерево, содержащее n = 18 узлов и имеющее степень m = 5. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 160.
3. Обеспечить обход деревьев «сверху вниз».
4. Выполнить поиск значения ключа по совпадению.

Вариант 8

1. Построить двоичное дерево, содержащее n = 14 узлов. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 120.
2. Построить B+ дерево, содержащее n = 14 узлов и имеющее степень m = 5. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 120.
3. Обеспечить обход деревьев «снизу вверх».
4. Выполнить поиск значения ключа по близости снизу.

Вариант 9

1. Построить двоичное дерево, содержащее n = 15 узлов. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 140.
2. Построить B+ дерево, содержащее n = 15 узлов и имеющее степень m = 5. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 140.
3. Обеспечить симметричный обход деревьев.
4. Выполнить поиск значения ключа по близости сверху.

Вариант 10

1. Построить двоичное дерево, содержащее n = 20 узлов. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 200.
2. Построить B+ дерево, содержащее n = 20 узлов и имеющее степень m = 5. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 200.
3. Обеспечить обход деревьев «сверху вниз».
4. Выполнить поиск значения ключа по совпадению.

Вариант 11

1. Построить двоичное дерево, содержащее n = 18 узлов. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 160.
2. Построить B+ дерево, содержащее n = 18 узлов и имеющее степень m = 5. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 160.
3. Обеспечить обход деревьев «снизу вверх».
4. Выполнить поиск значения ключа по близости снизу.

Вариант 12

1. Построить двоичное дерево, содержащее n = 16 узлов. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 120.
2. Построить B+ дерево, содержащее n = 16 узлов и имеющее степень m = 5. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 120.
3. Обеспечить симметричный обход деревьев.
4. Выполнить поиск значения ключа по близости сверху.

Вариант 13

1. Построить двоичное дерево, содержащее n = 12 узлов. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 100.
2. Построить B+ дерево, содержащее n = 12 узлов и имеющее степень m = 5. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 100.
3. Обеспечить обход деревьев «сверху вниз».
4. Выполнить поиск значения ключа по совпадению.

Вариант 14

1. Построить двоичное дерево, содержащее n = 18 узлов. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 160.
2. Построить B+ дерево, содержащее n = 18 узлов и имеющее степень m = 5. Значения ключей в узлах задавать с помощью датчика случайных чисел с диапазоном D от 0 до 160.
3. Обеспечить обход деревьев «снизу вверх».
4. Выполнить поиск значения ключа по близости снизу.

ОГЛАВЛЕНИЕ