***Обходы деревьев. Деревья, часть 3.***

***Обход дерева*** – *это упорядоченная последовательность вершин дерева, в которой каждая* *вершина встречается только один раз*. При обходе все вершины дерева должны посещаться в определенном порядке. Существует несколько способов обхода всех вершин дерева. Рассмотрим четыре наиболее часто используемых способа обхода дерева:

1. *обход в прямом порядке;*
2. *симметричный обход;*
3. *обход в обратном порядке;*
4. *обход в ширину.*

*Обход в прямом порядке.*

*При таком обходе алгоритм вначале обрабатывает вершину, затем ее левый дочерний узел, а после правый дочерний узел*.

Рассмотрим дерево, представленное на рисунке *1*. и укажем обход его вершины в прямом порядке.

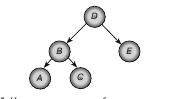


Рис. 1.

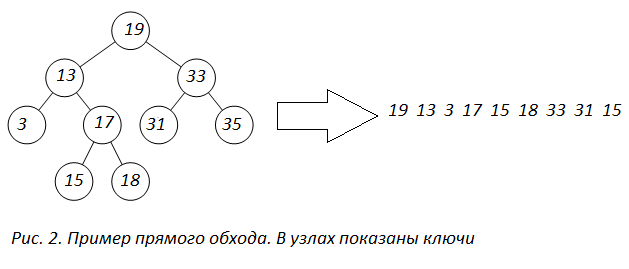
Как уже было сказано, первым делом алгоритм обратится к корню и выведет значение *D*. Затем он переместится к левой дочерней вершине корня, выведет *B* и рассмотрит уже ее левый дочерний узел, то есть *A*. Больше дочерних вершин нет, поэтому алгоритм вернется в узел *B* и проследует к его правой дочерней вершине, в нашем случае к *С*. У нее тоже нет потомков, следовательно, произойдет еще один возврат к вершине B.

Раз дочерних узлов у этой вершины больше не имеется, то программа поднимется вверх по дереву к корню *D* и проследует к его правому дочернему узлу *E*. Не обнаружив и у него потомков, алгоритм вновь обратится к корню, констатирует отсутствие любых других дочерних вершин и завершит обход. В результате порядок обхода будет выглядеть так: *D*, *B*, *A*, *C*, *E*.

Как видим из приведенного описания, алгоритм обхода будет следующим:

1. посещаем корень *R*;
2. рекурсивно обходим левое поддерево *R*;
3. рекурсивно обходим правое поддерево *R*.

На *рис*. *2* приведен еще один пример прямого обхода для дерева, каждому узлу которого поставлен в соответствие некоторый ключ.



С*имметричный обход.*

*При симметричном обходе алгоритм обрабатывает левый дочерний узел вершины, затем ее саму и только после этого правый дочерний узел*.

Работая с деревом, изображенным на рисунке *1*., программа начнет двигаться с корня, переместится сразу к левому дочернему узлу *B*, а через него к левому дочернему узлу *А*.

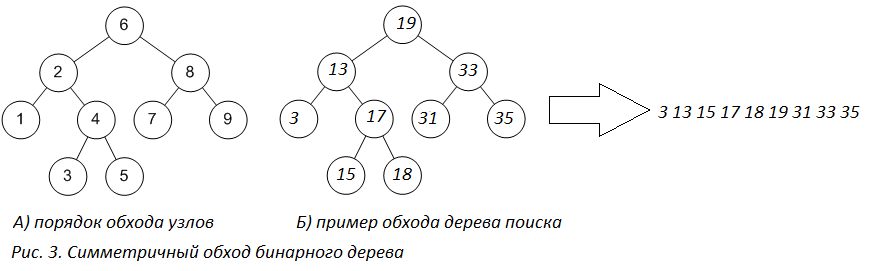
У этой вершины уже нет левого потомка, поэтому программа выведет *А* и, не обнаружив правого потомка, вернется к родительской вершине *B*, чтобы вывести ее. После этого алгоритм проследует к правому дочернему узлу *С*, у которого также отсутствует левый потомок. Выведя *С*, программа удостоверится, что нет и правого потомка, и снова вернется к родительской вершине *B*. Поскольку работа с левой частью дерева уже закончена, алгоритм поднимется к корню *D*, выведет его и обратится к правой части с дочерней вершиной *E*. У нее нет левого дочернего узла, значит, алгоритм выведет *E* и, не найдя правого дочернего узла, вернется к корню *D*. Итоговый порядок симметричного обхода будет таким: *A, B, C, D, E*.

Обратите внимание, что в выводе вершины располагаются в отсортированном порядке. Упорядоченные деревья как раз и строятся таким образом, чтобы симметричный обход дал на выходе отсортированные значения.

Запишем алгоритм симметричный обхода.

1. рекурсивно обходим левое поддерево;
2. посещаем корень R;
3. рекурсивно обходим правое поддерево.

Рассмотрим еще один пример, рисунок *3*.



И как мы уже говорили, для бинарного дерева поиска симметричный обход дает *сортировку ключей*. Если для любого узла *N* в его левом поддереве все ключи меньше, чем в *N*, а в правом поддерева все ключи больше, чем в *N*, то симметричный обход даст сортировку по возрастанию.

*Обход в обратном порядке.*

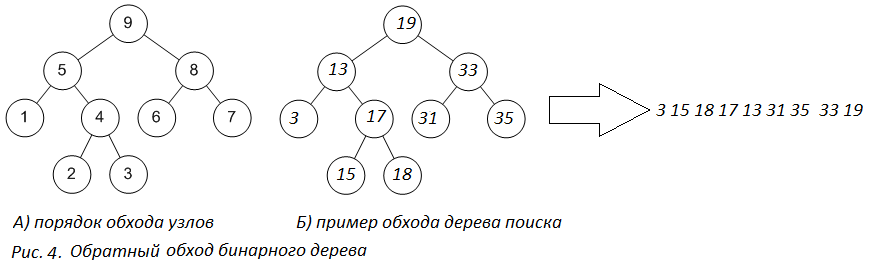
*В данном случае алгоритм обрабатывает сначала левый дочерний узел вершины, затем правый и только после саму вершину*.

Как и в предыдущих обходах, алгоритм начнет рассматривать дерево, изображенное на рисунке *1*., с корня, переместится в левую часть к дочернему узлу *B*, а через него к левому дочернему узлу *A*. Не найдя дальнейших потомков, он выведет *A*, вернется к родительской вершине *B* и перейдет к правому дочернему узлу *C*. У него тоже нет отходящих ветвей, поэтому программа выведет саму вершину *C* и снова обратится к родительскому узлу *B*.

Поскольку работа с дочерними вершинами на этом уровне закончена, алгоритм выведет *B*, поднимется к корню *D* и проследует к правому дочернему узлу *E*. Не обнаружив связанных с ним потомков, он выведет *E* и вернется к *D*. Раз изучение дочерних узлов окончено, остается вывести *D* и завершить обход. В целом порядок следования алгоритма будет таким: *A, C, B, E, D.*

Алгоритм обратного обхода.

1. рекурсивно обходим левое поддерево;
2. рекурсивно обходим правое поддерево;
3. посещаем корень.

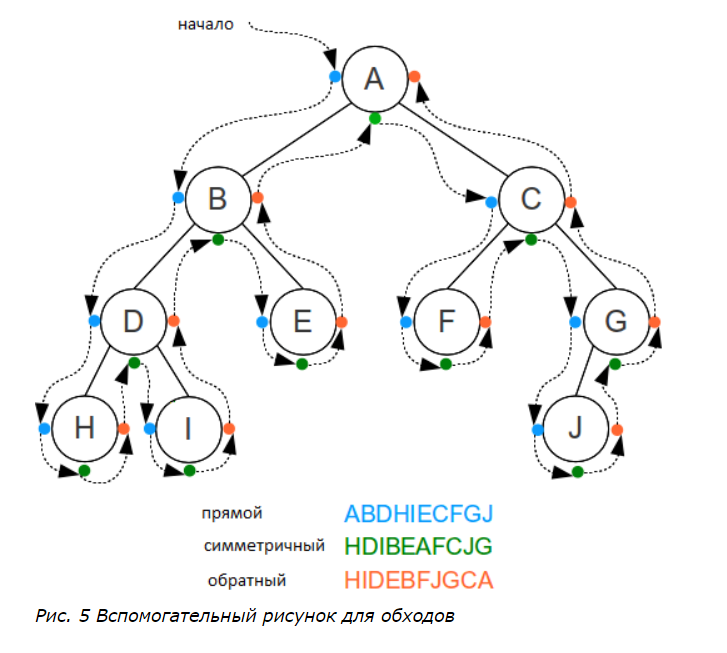


Если нам дано изображение дерева, и нужно найти его обходы, то может помочь следующая техника, рисунок 5. Обводим дерево огибающей замкнутой кривой (начинаем идти слева вниз и замыкаем справа вверх).

Прямому обходу будет соответствовать порядок, в котором огибающая, двигаясь от корня дерева, впервые проходит рядом с узлами *слева*.

Для симметричного обхода порядок, в котором огибающая, двигаясь от корня, впервые проходит рядом с узлами *снизу*.

Для обратного обхода порядок, в котором огибающая, двигаясь от корня, впервые проходит рядом с узлами справа.

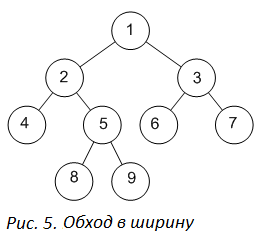


*Рис. 5. Геометрическая интерпретация обходов.*

*Обход в ширину.*

*Обход в ширину*: посещаем корень, затем узлы первого уровня, сыновья корня, слева направо. Затем - узлы второго уровня ("*внуки*" корня) слева направо и т.д.

Обход в ширину иногда называют *волновой алгоритм* - порядок обхода напоминает движение волны. На рисунке 6. показан обход дерева в ширину.

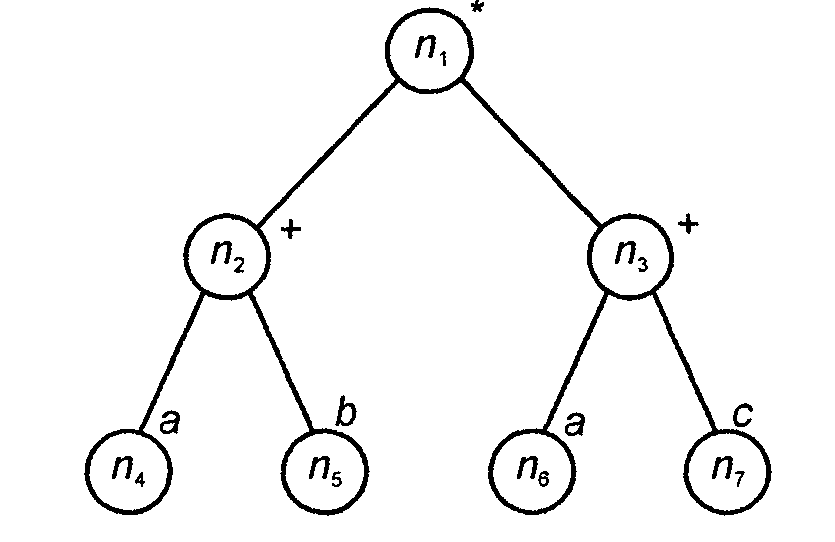


*Рис. 6. Обход в ширину.*

*Деревья и арифметические выражения*.

Часто бывает полезным сопоставить каждому узлу дерева *метку* (*label*) *или значение*. Дерево, у которого узлам сопоставлены метки, называется *помеченным деревом*. Метка узла — это не имя узла, а значение, которое "*хранится*" в узле, значение метки можно изменять, имя узла сохраняется постоянным. Рассмотрим бинарное дерево с метками, представляющее арифметическое выражение (*a+b)\*\_(a+c)*, рисунок *7*. Метки на рисунке проставлены рядом с соответствующими узлами . Правила соответствия меток деревьев элементам выражений следующие.

1. Метка каждого листа соответствует операнду и содержит его значение, например, узел  представляет операнд *а*.
2. Метка каждого внутреннего (родительского) узла соответствует оператору (операции). Пусть узел *n* помечен бинарным оператором , левый сын этого узла соответствует выражению , правый - . Тогда узел *п* и его сыновья представляют выражение .



*Рис. 7.* *Дерево выражения с метками*

Например, узел  представляет выражение *(a) + (b)* , т.е. *a + b*, Узел  представляет выражение *(а + b) \* (а + с)*, поскольку оператор *\** является меткой узла , выражения *а + b* и *а + с* представляются узлами  и  соответственно.

Часто при обходе деревьев составляется список не имен узлов, а их меток. В случае дерева выражений при прямом обходе получаем известную *префиксную форму* *записи* выражений, где оператор предшествует и левому, и правому операндам. Как мы знаем, префиксная форма для выражения имеет вид , где  и  префиксные формы для выражений  и . Например, при прямом обходе узлов (точнее, меток) дерева, получаем префиксное выражение *\* + ab +ac.*

Обратный обход меток дерева выражений дает так называемое *постфиксное* (или *польское*) представление выражений. Выражение в постфиксной форме имеет вид , где  и  - постфиксные формы для выражений  и  соответственно. Например, постфиксная форма выражения для нашего дерева имеет вид ab + ac + \*.

При симметричном обходе дерева выражений получим так называемую инфиксную форму выражения, которая совпадает с привычной "стандартной" формой записи выражений. Для дерева на рис. *7* инфиксное выражение запишется как *а + b \* а + с*.

Во всех трех формах записей выражений скобки не используются.

Обходы деревьев иногда называют соответственно как *префиксный обход*, *постфиксный обход и инфиксный обход*.

Умение обходить дерево дает возможность легко устанавливать *изоморфность* двух деревьев. Проходя два дерева одновременно, можно проверять наличие вершины у одного дерева при условии наличия ее у другого. При одновременном прохождении двух деревьев обработка вершин будет означать проверку обоих деревьев на наличие соответствующих вершин. В данном случае для прохождения деревьев представляется разумным использовать префиксный обход, так как это позволяет не интересоваться наличием сыновей, если один из родителей отсутствует.

***Гомоморфизм графов.***

Рассмотрим два графа ** и **. Функция *f* из графа ** в граф ** называется *гомоморфизмом* из  в  и обозначается , если она обладает следующими свойствами.

1. Если , то . ().

2. Если , то . ().

3. Если вершины *u* и *v* инцидентны ребру *е* графа , то вершины *f(u)* и

*f(v*) инцидентны ребру *f(е)* графа .

Справедливы следующие теоремы.

*Теорема 1*. Если функция *f* - гомоморфизм из  в  то *f(G) -* подграф *(f(V), f(E))* графа .

*Теорема 2*. Если граф *G* связный и *f* - гомоморфизм, то граф *f(G)* связный.

*Теорема 3.*Если граф *G* полный и *f* - гомоморфизм, то *f(G)* полный.

Многие свойства графа *G* не являются инвариантными относительно *f.*

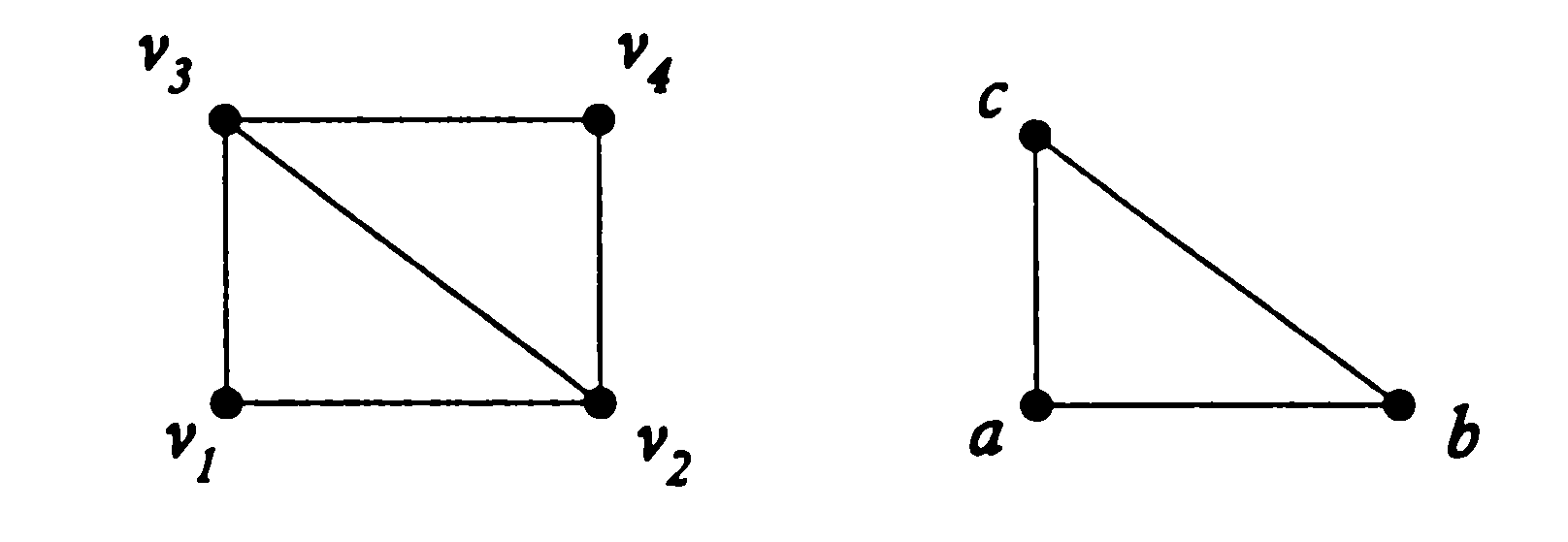
Гомоморфизм  является *изоморфизмом*, если  и представляют собой *взаимно однозначные соответствия*. Если  изоморфизм, то  и  называются *изоморфными*.

Таким образом, изоморфизм, по сути, является *переименованием* вершин и ребер графа *G*, которое сохраняет свойство гомоморфности, так что если вершины *u* и *v* инцидентны ребру *е* графа *G*, то вершины *f(u)* и *f(v)* инцидентны ребру *f(е)* графа *G'*.

Очевидно, что практически все свойства графов инвариантны относительно изоморфизма. Отсюда следует, что простейший способ показать неизоморфность двух графов - установить свойство, которым обладает один граф не обладает другой.

Рассмотрим примеры.

*Пример 1, рисунок 1.*



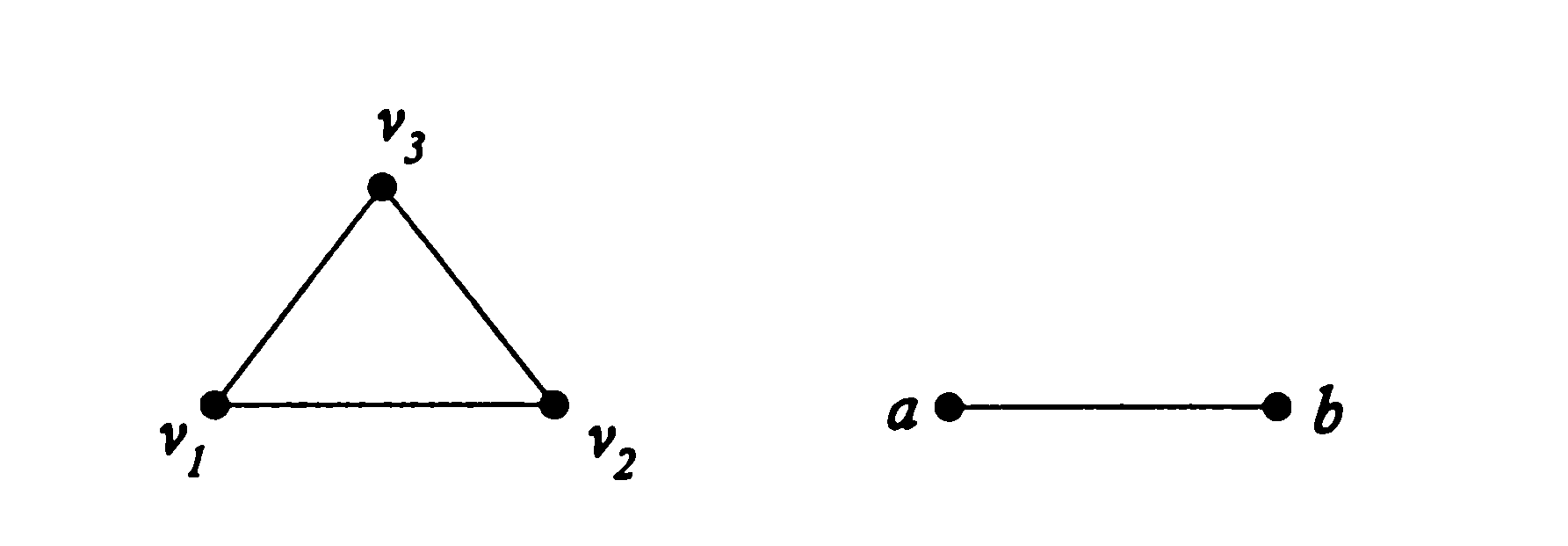
*Рис. 1.*

Построим гомоморфизм.



Как мы уже говорили, многие свойства графа *G* не являются инвариантными относительно гомоморфизма *f*. Например, в данном случае вершине  соответствует вершина *с*, но , а .

*Пример 2, рисунок 2.*



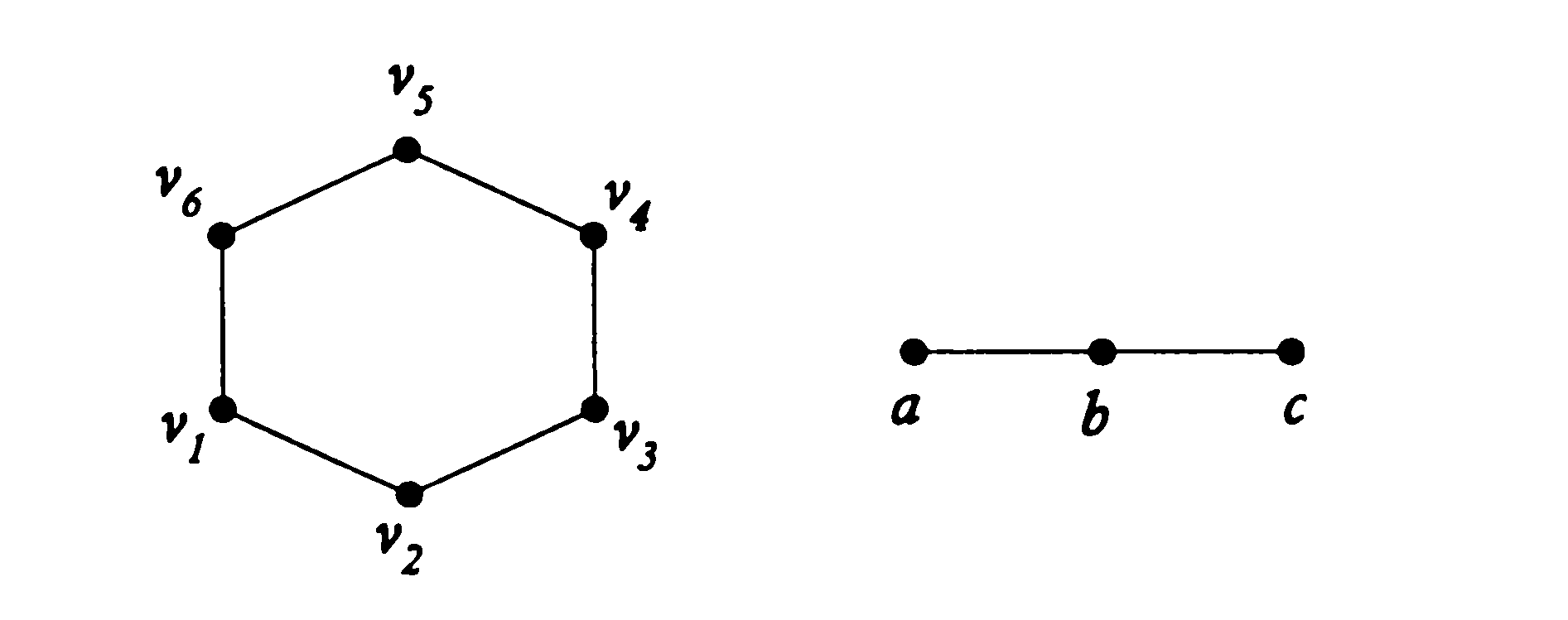
*Рис. 2.*

*Гомоморфизм не существует*. Не выполняется *3* свойство гомоморфизма. Предположим, что мы построили гомоморфизм следующим образом:



Тогда по *3* свойству гомоморфизма ребру графа *G* должна соответствовать петля *(a, a)* графа *G'*, но у графа *G'* нет такой петли и это и никакое другое построение не является гомоморфизмом.

*Пример 3, рисунок 3.*



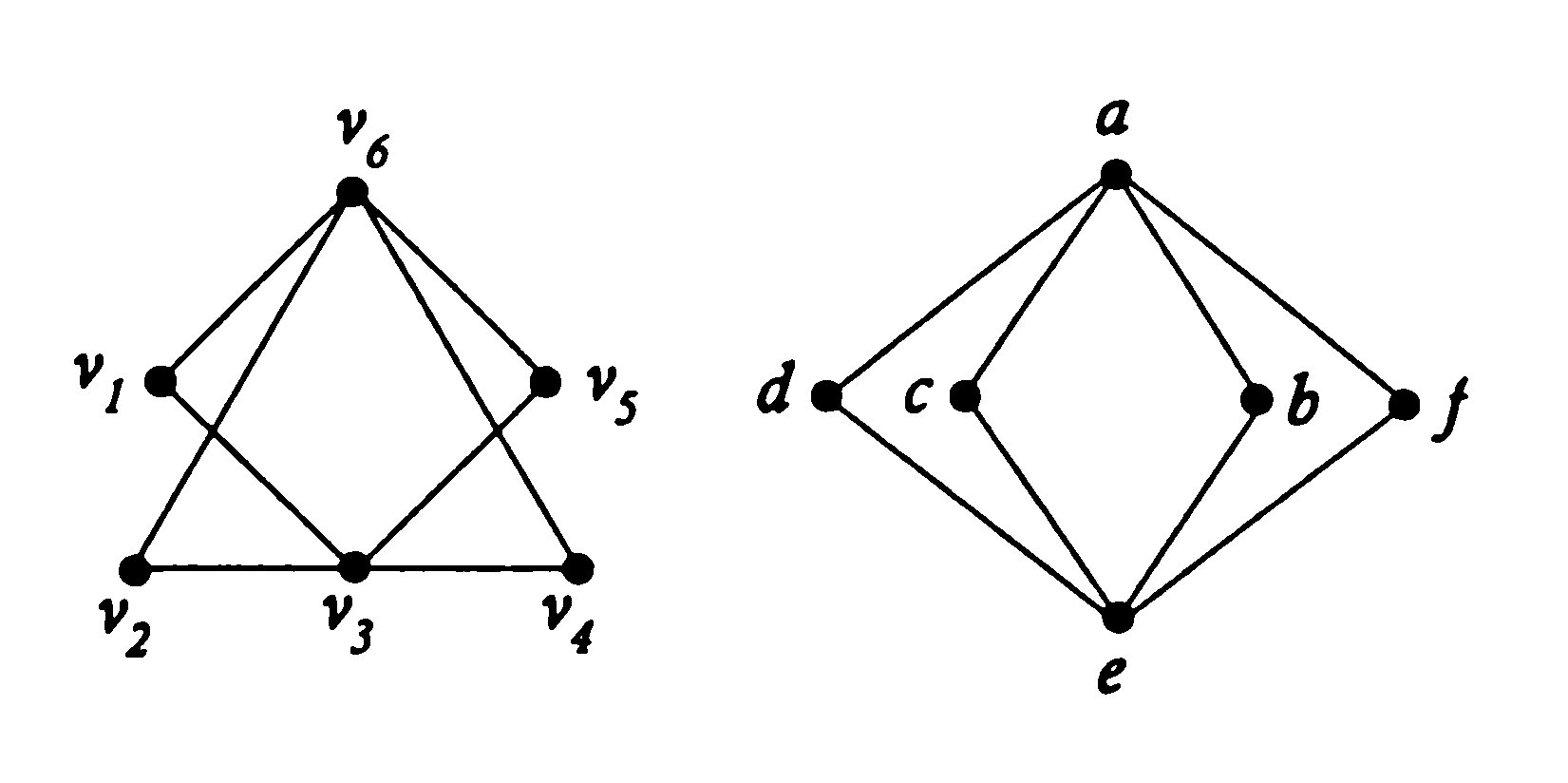
*Рис. 3.*

Построим гомоморфизм.



Гомоморфизм не инъективный, не сюръективный.

*Пример 4, рисунок 4.*

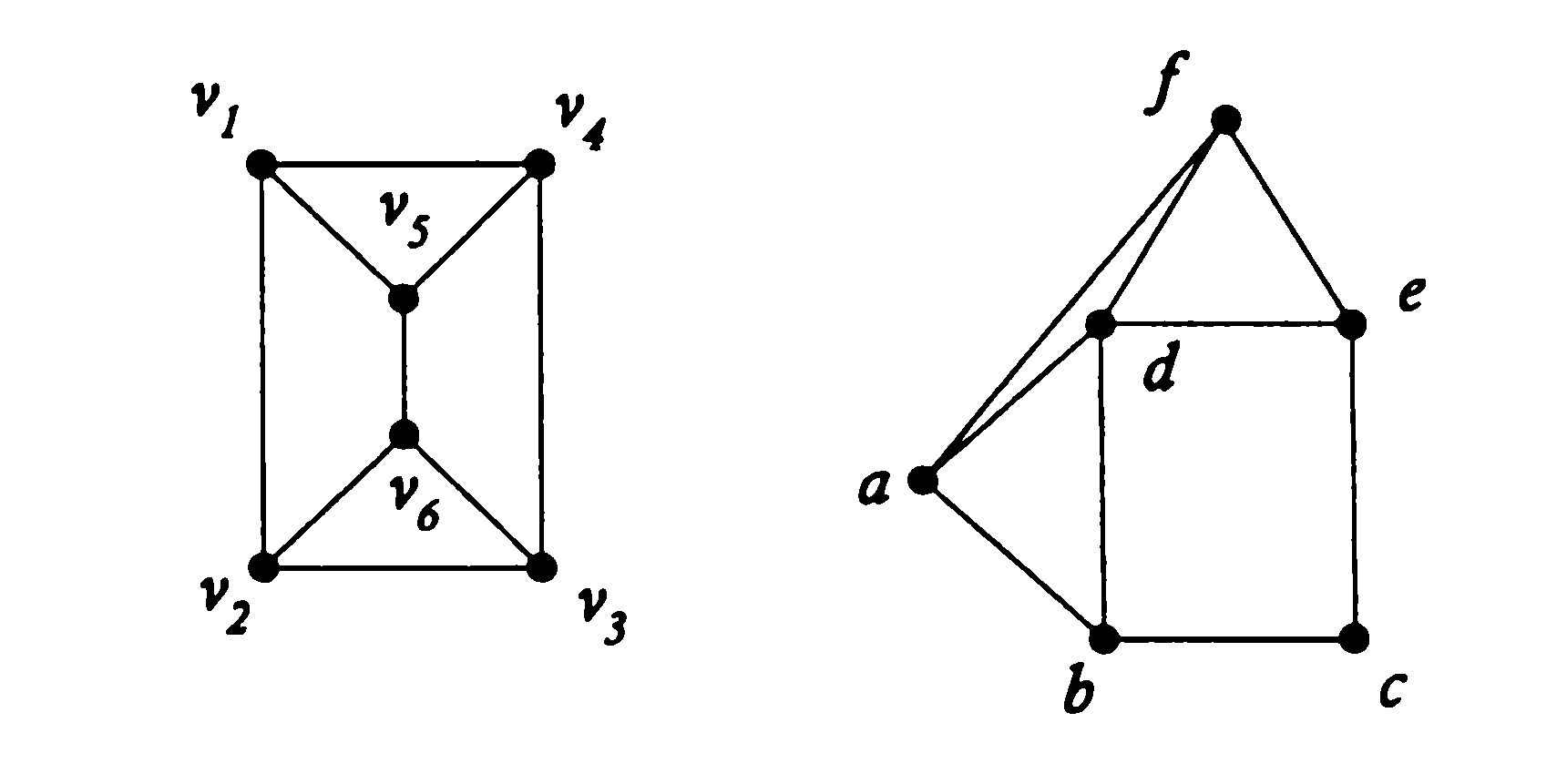


*Рис. 4.*

Изоморфизм.



*Пример 5, рисунок 5.*



Графы не изоморфны, вершина *d* имеет степень *4*, ни одна из вершин первого графа не имеет степень *4*;

***Дополнение.***

Умение обходить дерево дает возможность легко устанавливать *изоморфность* двух деревьев. Проходя два дерева одновременно, можно проверять наличие вершины у одного дерева при условии наличия ее у другого. При одновременном прохождении двух деревьев обработка вершин будет означать проверку обоих деревьев на наличие соответствующих вершин. В данном случае для прохождения деревьев представляется разумным использовать префиксный обход, так как это позволяет не интересоваться наличием сыновей, если один из родителей отсутствует.