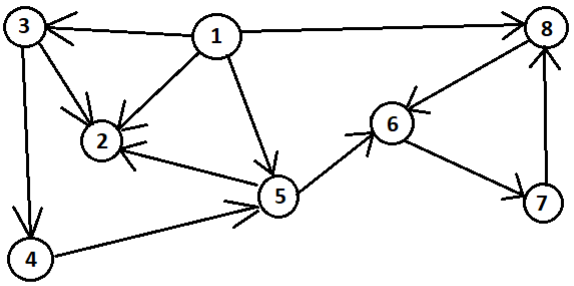
## Представление графов списками смежных вершин

Наиболее распространенным способом представления графов в программировании являются списки смежных вершин. Этот способ представляет ребра графа в виде массива в котором каждой вершине сопоставлен список смежных с ней вершин. Например, граф на рисунке может быть представлен в виде следующего массива списков

| | |
|---|---|
| 1 | 2,3,5,8 |
| 2 | |
| 3 | 2,4 |
| 4 | 5 |
| 5 | 2,6 |
| 6 | 7 |
| 7 | 8 |
| 8 | 6 |




Наиболее простым и удобным способом хранения такой структуры данных в C++ является вектор из векторов целых чисел `vector<vector<int>>`. Например, рассмотрим следующую программу.

```
int n=100,a=10,b=13;
vector<vector<int>> G;
G.resize(n);
G[a].push_back(b);
```

В двух первых строчках этой программы объявляются переменные, далее, создается вектор из пустых векторов, который представляет граф без ребер с номерами вершин от 0 до 99. В последней строке добавляется одно ориентированное ребро из вершины 10 в вершину 13. Если граф является неориентированным, то следует добавить и обратное ребро.

```
G[b].push_back(a);
```

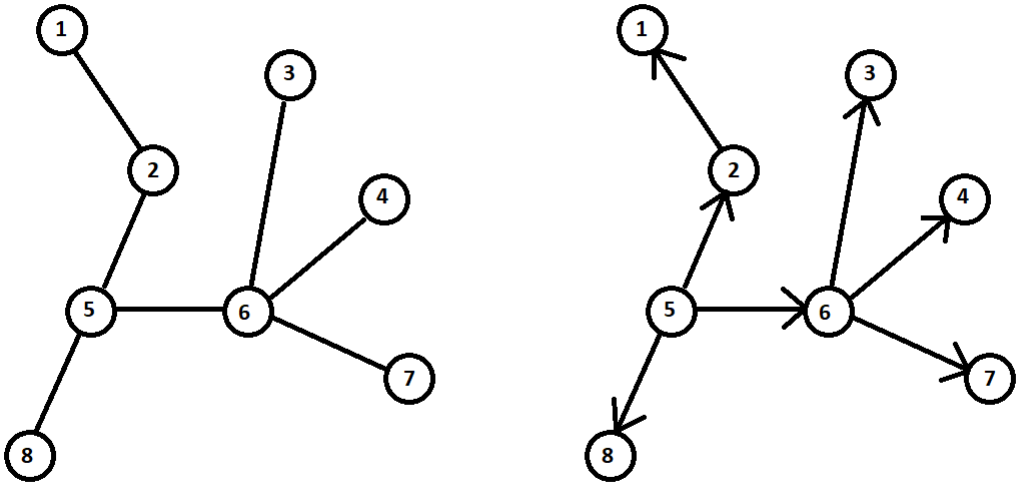
Граф может быть взвешенным. Это означает, что каждому ребру приписано некоторое число, например стоимость или время прохода по этому ребру. В этом случае потребуется также хранить вес ребра, в результате, каждый элемент списка превратится в пару из номера смежной вершины и ребра.

```
int n=100,a=10,b=13,v=80;
vector<vector<pair<int,int>>> G;
G.resize(n);
G[a].push_back({b,v});
```

Здесь в последней строке добавляется одно ориентированное ребро из вершины 10 в вершину 13 с весом 80.

## Деревья

Существует несколько определений дерева. Обычно деревом называют связный неориентированный граф без циклов. В этом случае дерево хранится в памяти как обыкновенный неориентированный граф. Вместе с тем, часто в дереве выделяют специальную вершину — корень дерева, и задают направления ребер так, чтобы они задавали направление в сторону противоположную корню. Такое дерево называется корневым или подвешенным и хранится в памяти как ориентированный граф.




На рисунке слева, дерево представлено как неориентированный граф. Справа аналогичное корневое дерево с корнем в вершине 5. Для хранения корневого дерева, может использоваться более компактная форма хранения в виде списка непосредственных предков. Список непосредственных предков — это последовательность из $n$ чисел $a_1, a_2, \ldots, a_n$. Если значение $a_i$ равно нулю, то вершина с номером $i$ является корнем. Иначе, оно означает, что существует ребро из вершины $a_i$ в вершину $i$. Например, для дерева на правом рисунке список непосредственных предков будет выглядеть следующим образом.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 2 | 5 | 6 | 6 | 0 | 5 | 6 | 5 |

В памяти или файле будут храниться только значения `2␣5␣6␣6␣0␣5␣6␣5`.

В неориентированном дереве можно выбрать произвольную вершину, объявить ее корнем, и придать ребрам дерева правильную ориентацию. Эту операцию называют подвешиванием дерева.

## Обход в глубину

Алгоритм обхода в глубину или dfs (Depth-first search) является наиболее распространенным способом перечисления вершин графа. Алгоритм является рекурсивным. Параметром функции является номер некоторой вершины графа. Ее мы называем текущей вершиной. Функция выполняет рекурсивный вызов для каждой вершины смежной с текущей.

Наиболее просто этот алгоритм реализуется для корневого дерева.

```
vector<vector<int>> G;
void dfs(int v) {
  for (int c:G[v])
    dfs(c);
}
```

Здесь предполагается, что переменная `G` является глобальной и хранит представление дерева в виде списков смежности. Первый вызов функции `dfs` выполняется для вершины, являющейся корнем. Представленная функция пока не решает никаких задач и нуждается в доработке для каждого конкретного случая.

Если дерево является неориентированным, то алгоритм немного усложняется. Это происходит от того, что требуется отбросить то ребро, по которому алгоритм вошел в вершину. Этот вариант функции будет содержать два формальных параметра: номер текущей и номер предшествующей вершины.

```
vector<vector<int>> G;
```

```
void dfs(int v,int p) {
  for (int c:G[v])
    if (c!=p)
      dfs(c,v);
}
```

Первый вызов может иметь вид `dfs(v,-1)`, где `v` — любая из вершин. Второй параметр может быть любым числом не равным номеру какой-нибудь вершины.

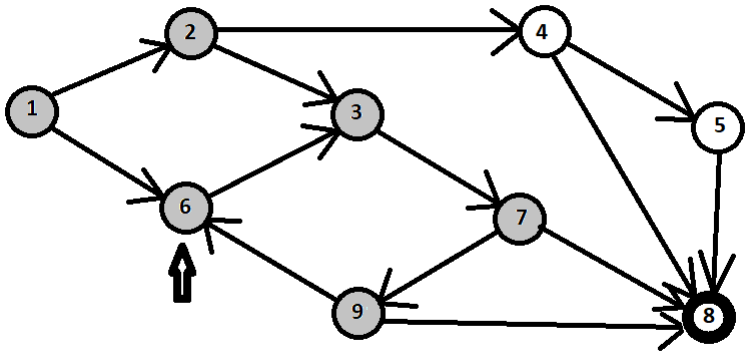
Для применения `dfs` к неориентированным графам произвольного вида требуется реализовать алгоритм так, чтобы не произошло его зацикливания при попадании в ранее посещенную вершину. Для этого потребуется глобальный вектор пометок. При входе в некоторую вершину она будет помечаться как посещенная. При переборе вершин пометки будут проверяться.

```
vector<vector<int>> G;
vector<int> P;
void dfs(int v) {
  P[v]=1;
  for (int c:G[v])
    if (P[c]==0)
      dfs(c);
}
```

Предполагается, что до запуска `dfs` вектор пометок `P` заполнен нулями. После завершения алгоритма вектор содержит единицы только для тех вершин, которые находятся в одной области связности с вершиной, для которой был сделан первый вызов `dfs`. Таким образом, этот метод можно использовать для поиска областей связности в неориентированном графе.

Этот же прием используется для обхода неориентированных графов. Но теперь вводятся три вида пометок. Вершины, которые не были пройдены алгоритмом называем белыми и помечаем числом 0. Вершины для которых `dfs` уже завершил свою работу называем черными и помечаем числом 2. Все остальные, а именно, вершины в которые `dfs` уже вошел, но не завершил работу называем серыми и помечаем числом 1.

```
vector<vector<int>> G;
vector<int> P;
void dfs(int v) {
  P[v]=1;
  for (int c:G[v])
    if (P[c]==0)
      dfs(c);
  P[v]=2;
}
```



Алгоритм `dfs` перечислял вершины графа, изображенного на рисунке в следующем порядке: 1␣2␣3␣7␣8␣9␣6, текущая вершина 6. Вершины с номерами 4 и 5 еще не были пройдены, поэтому они белые. Вершина 8 была пройдена и алгоритм уже завершил работу для этой вершины, поэтому она черная. Для остальные вершин рекурсивная функция уже начала их обработку, но еще не закончила, поэтому они серые.

Все серые вершины образуют путь от начальной до текущей вершины, поэтому наличие перехода к серой вершине означает наличие цикла в графе.

## Некоторые алгоритмы

### Диаметр дерева

Диаметром связного графа называется максимум из кратчайших расстояний между всеми парами вершин. Проще всего найти диаметр у дерева, поскольку в дереве каждая пара вершин связана единственным путем. Одним из простых способов нахождения диаметра является следующий. Выбираем произвольную вершину $x$ и находим наиболее удаленную от нее вершину $y$. Далее выбираем $y$ и вновь находим наиболее удаленную от нее вершину $z$. Вершины $y$ и $z$ будут концами диаметра.

Для реализации этого алгоритма потребуется модифицировать `dfs`, добавив в него еще один параметр — расстояние от корня. Тогда, заголовок функции может выглядеть так

```
pair<int,int> dfs(int v,int p,int len)
```

Функция возвращает пару в которой первый элемент равен длине пути, а второй — номеру самой дальней вершины. Для пар определен оператор сравнения, поэтому при реализации можно использовать функцию `max`.

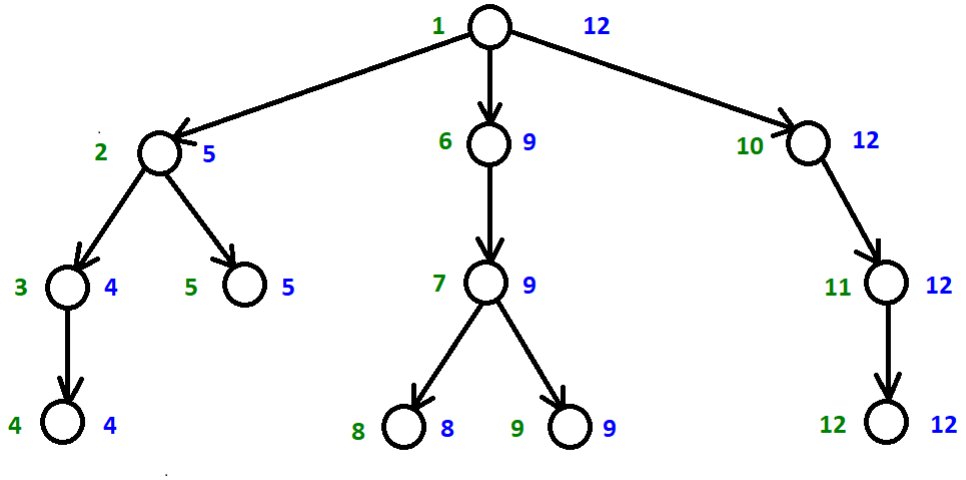
### Топологическая сортировка

Топологической сортировкой называется такое упорядочивание вершин ориентированного ациклического графа $v_1, v_2, \ldots, v_n$, при котором для любых номеров $i$ и $j$ выполняется следующее свойство: если $i < j$ то не существует пути из $v_j$ в $v_i$. Эта задача элементарно решается с помощью алгоритма `dfs`. Заметим, что алгоритм оканчивает обработку некоторой вершины $v$ в тот момент, когда все вершины достижимые из нее уже обработаны. Таким образом, вывод номера вершины в момент выхода из нее позволяет получить последовательность обратную к требуемой. Для получения ответа полученную последовательность остается только развернуть.

### Проверка принадлежности вершины поддереву

Пусть задано корневое дерево и запросы, в которых требуется определить, верно ли что некоторая вершина $x$ принадлежит поддереву, корнем которого является вершина $y$. Дерево известно заранее и не изменяется со временем. Тогда, возможно сделать предобработку следующего вида, которая позволит отвечать на каждый из запросов со сложностью $O(1)$.

Представим себе работу `dfs`, как движение со временем. Каждый переход по ребру вниз занимает 1 секунду, переход рпо ребру вверх можно считать мгновенным. Запустим `dfs` и сохраним в два вектора время входа в каждую вершину и время выхода. Легко увидеть, какое условие позволит правильно отвечать на запросы указзанного вида.




На рисунке зеленым цветом обозначено время входа в вершину, а синим — время выхода.

Реализация `dfs` будет проще, если переменная с текущем временем будет глобальной.

### Обход в ширину

Еще одним вариантом перечисления всех вершин графа является обход в ширину или bfs (breadth-first search). Этот алгоритм перечисляет вершины графа в порядке их удаленности от начальной вершины. В ходе работы он использует очередь для хранения номеров вершин и пометки для пройденных вершин.

До начала работы основного цикла в очередь помещается начальная вершина и помечается как пройденная. Основной цикл выполняется, пока очередь не пуста. Из очереди извлекается вершина, просматриваются все смежные с ней непомеченные вершины, помечаются и добавляются в очередь.

Алгоритм можно использовать для поиска областей связности или для поиска кратчайших по количеству ребер путей.

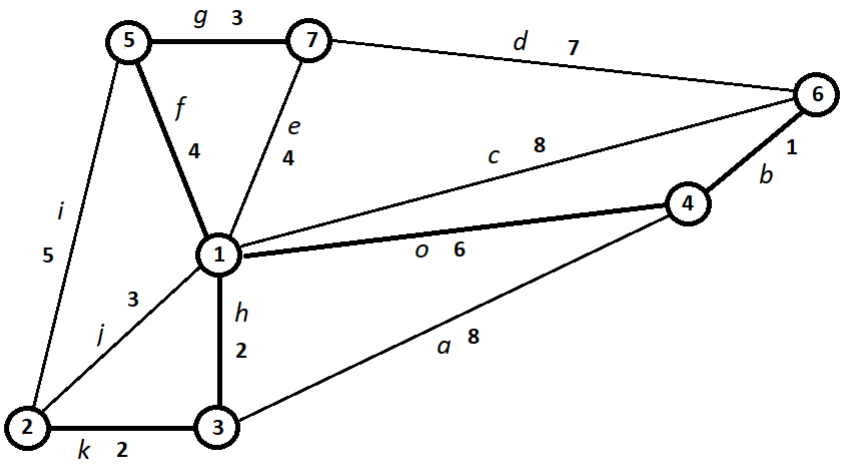
### Поиск кратчайшего остова алгоритмом Прима

Пусть задан связный взвешенный неориентированнный граф. Остовом называется такое подмножество ребер этого графа, которое образует дерево. Минимальным называется такой остов, сумма весов которого является наименьшей.

Алгоритм Прима позволяет находить кратчайший остов со сложностью $O(m \log m)$, где $m$ — количество ребер графа. Он использует приоритетную очередь из ребер и вектор пометок. Каждое ребро представляется парой, где первый элемент является его весом, а второй элемент может быть либо номером ребра, либо содержать номера инцидентных вершин. Если требуется найти только вес остова, то достаточно хранить номер только одной вершины. Приоритетная очередь должна быть настроена для выбора ребер с минимальным весом.

На каждом шаге алгоритм расширяет остов, увеличивая его на одну вершину и одно ребро. Вершины, которые добавляются в остов, помечаются. После добавления вершины, в очередь добавляются все ребра, выходящие из нее, и, ведущие к вершинам, которые не входят в уже построенную часть остова.

На первом шаге в остов добавляется одна произвольная вершина, на последующих шагах выбирается непомеченная вершина к которой ведет ребро минимального веса из уже построенной части остова. Все такие ребра на предшествующих шагах уже были добавлены в приоритетную очередь, поэтому требуемое ребро извлекается из ее начала. Следует учитывать, что очередь может содержать ребра ведущие в вершины остова. Такие ребра должны извлекаться из очереди и отбрасываться.




| шаг | вершины остова | ребра остова | очередь |
|---|---|---|---|
| 1 | 1 | | (2,h),(3,j),(4,f),(4,e),(6,o),(8,c) |
| 2 | 1, 3 | h | (2,k),(3,j),(4,f),(4,e),(6,o),(8,c),(8,a) |
| 3 | 1, 3, 2 | h, k | (3,j),(4,f),(4,e),(5,i),(6,o),(8,c),(8,a) |
| 4 | 1, 3, 2, 5 | h, k, f | (3,g),(4,e),(5,i),(6,o),(8,c),(8,a) |
| 5 | 1, 3, 2, 5, 7 | h, k, f, g | (4,e),(5,i),(6,o),(7,d),(8,c),(8,a) |
| 6 | 1, 3, 2, 5, 7, 4 | h, k, f, g, o | (1,b), (7,d),(8,c),(8,a) |
| 7 | 1, 3, 2, 5, 7, 4, 6 | h, k, f, g, o, b | (7,d),(8,c),(8,a) |

На первом шаге в остов добавляется вершина 1, а в очередь все ребра инцидентные этой вершине.

На втором шаге из очереди извлекается ребро `k` и добавляется в остов. В очередь добавляются только те ребра, которые не ведут в вершины из построенной части остова.

Обратите внимание, что после третьего и пятого шага в голове очереди находятся ребра, которые соединяют вершины остова. Такие ребра просто удаляются.

На четвертом шаге в очереди два ребра с минимальным весом. Можно выбрать любое из этих ребер. Полученные остовы будут различаться, но их суммарный вес будет одинаковым.

### Поиск кратчайшего пути алгоритмом Дейкстры

Пусть задан неориентированный взвешенный граф, причем веса всех ребер являются неотрицательными. Требуется найти длины кратчайших путей от некоторой заданной начальной вершины до всех остальных вершин. Длина пути вычисляется как сумма весов ребер на этом пути. Для решения этой задачи можно применить алгоритм Дейкстры, который в чем-то похож на алгоритм Прима.

Все вершины разбиваются на два множества: те, до которых уже известен кратчайший путь (помеченные вершины), и те, до которых кратчайший путь пока не известен (непомеченные). Поддерживается вектор длин кратчайших путей, обладающий следующим свойством: на каждом шаге $i$-тый элемент вектора содержит длину кратчайшего пути до вершины с номером $i$, проходящего только через помеченные вершины.

На каждом шаге алгоритма выбирается вершина второго множества с наименьшей длиной пути и переносится в первое множество. После этого выполняется операция релаксации — пересчета кратчайших путей с учетом вновь помеченной вершины.

```
if cost(i,j)+len(i)<len(j) then
  len(j)<-cost(i,j)+len(i)
```

Здесь $i$ — номер выбранной вершины, $j$ — номер вершины смежной с $i$, $cost(i,j)$ — вес ребра из $i$ в $j$, $len(i)$ и $len(j)$ — элементы вектора кратчайших путей.

Для выбора вершины с наименьшей длиной пути можно использовать структуру данных как в алгоритме Прима.

До начала работы вектор кратчайших длин инициализируется специальным образом. Длину пути до начальной вершины считаем равной нулю, все остальные длины считаем равными плюс бесконечности.

Алгоритм останавливает работу, когда структура данных для поиска вершины с минимальной длиной пути станет пустой. Если в этот момент некоторые элементы вектора кратчайших длин остаются равными плюс бесконечности, то это означает, что соответствующие вершины являются недостижимыми.