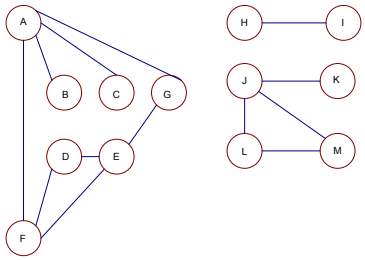
Граф

Граф е абстрактен тип данна, който представлява колекция от върхове и ребра.

Връх е прост обект, който има име и други свойства.

Ребро е връзка между два върха.



Граф с върхове A B C D E F G H I J K L M
и множество от ребра
AG AB AC LM JM JL JK ED FD HI FE AF GE

Приложение

- **карта на транспортни пътища – маршрути (връзки) между градовете (върхове): най-бързият път от София до Лондон със самолет, най-евтиният път от София до Лондон със самолет;**
- **електрически схеми – свързване с проводници на елементите (транзистори, резистори, кондензатори и т.н.): свързани ли са всички елементи в схемата, работоспособност на схемата;**
- **управление на заданията в производствен процес – задачите (върхове) се изпълняват в някаква последователност, която се представя чрез връзки: кога всяка задача трябва да се изпълни.**

Път от връх x **до връх** y **е списък от възлите, които са свързани чрез ребра в графа.**

Пример: Път от B **до** G **е списъкът** BAFEG**.**

Свързан граф е граф, в който има път от всеки възел до всеки друг възел в графа.

Несвързан граф е изграден от свързани компоненти.

Пример: Графът е изграден от три свързани компоненти.

Прост път е път, в който няма повтарящи се върхове.

Например: BAFEGAC **не е прост.**

Цикъл е прост път, с изключение на първия и последния възел, които съвпадат.

Пример: Пътят AFEGA **е цикъл.**

Дърво е граф без цикли.

Гора е група от несвързани дървета.

Покриващо дърво е подграф, който съдържа всички върхове, но само онези ребра, които формират дърво.

V **– брой на върховете в граф**

E **– брой на ребрата в граф**
E **се изменя от** 0 **до** $\frac{1}{2}V(V-1)$ **.**

Пълен граф е граф, в който присъстват всички ребра.

Разреден граф е граф с относително малък брой ребра (по-малко от $V \log V$ **).**

Плътен граф е граф с относително малък брой липсващи ребра.

Алгоритмите имат различна ефективност в зависимост от броя на ребрата в графа и неговата разреденост.

Видове графи:

1. **Ориентирани и неориентирани**
2. **Претеглени (на всяко ребро се присвоява число – тегло, представящо напр. разстояние или цена) и непретеглени**

Мрежа е ориентиран претеглен граф.

Физическо представяне

1. **Чрез матрица на съседство**
2. **Чрез структура на съседство**

Физическо представяне чрез матрица на съседство

Представяне чрез матрица на съседство – VxV **масив от логически стойности, като** adjmatrix[i][j] **се установява в** 1**, ако има ребро от връх** i **до връх** j **и** 0 **в противен случай.**

Всяко ребро се представя чрез две стойности adjmatrix[i][j] **и** adjmatrix[j][i]**.**

Приемаме, че има ребро от всеки връх към себе си – adjmatrix[i][i] = 1 **за** i **от** 1 **до** V**.**

Пример:

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	1	1	0	0	1	1	0	0	0	0	0	0
B	1	1	0	0	0	0	0	0	0	0	0	0	0
C	1	0	1	0	0	0	0	0	0	0	0	0	0
D	0	0	0	1	1	1	0	0	0	0	0	0	0
E	0	0	0	1	1	1	1	0	0	0	0	0	0
F	1	0	0	1	1	1	0	0	0	0	0	0	0
G	1	0	0	0	1	0	1	0	0	0	0	0	0
H	0	0	0	0	0	0	0	1	1	0	0	0	0
I	0	0	0	0	0	0	0	1	1	0	0	0	0
J	0	0	0	0	0	0	0	0	0	1	1	1	1
K	0	0	0	0	0	0	0	0	0	1	1	0	0
L	0	0	0	0	0	0	0	0	0	1	0	1	1
M	0	0	0	0	0	0	0	0	0	1	0	1	1

Логическо описание

Съединение на елементи от тип ЦЯЛО**, образуващи** матрица на съседство **с** V **върха и** E **ребра.**

```
тип ГРАФ = (V: ЦЯЛО;          {брой върхове}
            E: ЦЯЛО;          {брой ребра}
            масив adjmatrix[1:MAXV][1:MAXV]: ЦЯЛО)
```

Дефиниране

```
#define MAXV <константа>     // максимален брой върхове
struct graph                 // ГРАФ
{
  int V;                     // брой върхове
  int E;                     // брой ребра
  int adjmatrix[MAXV][MAXV]; // матрица на съседство
};
typedef struct graph GRAPH;
```

Характеристики

1. **Представянето чрез матрица на съседство е подходящо за плътни графи – матрицата изисква V^2 елемента памет и V^2 стъпки само за инициализация.**
2. **Имената на върховете се представят чрез цели числа между 1 и V – бърз достъп до информацията във всеки възел. Използваме 1-буквени имена на върховете, като i^{тата} буква от азбуката съответства на цялото число i.**

Операции

1. **Създаване на граф**
2. **Обхождане на граф в дълбочина**

1. Създаване на граф

Алгоритъм

въведи V **и** E
за i **от** 1 **до** V **повтаряй**
  за j **от** 1 **до** V **повтаряй**
    adjmatrix[i][j] ← **0**
за i **от** 1 **до** V **повтаряй**
  adjmatrix[i][i] ← **1**
за k **от** 1 **до** E **повтаряй**
  въведи символ$_{връх1}$ и символ$_{връх2}$
  i ← конвертиран символ$_{връх1}$ в ЦЯЛО
  j ← конвертиран символ$_{връх2}$ в ЦЯЛО
  adjmatrix[i][j] ← **1**
  adjmatrix[j][i] ← **1**

```
// Конвертира 1-буквените имена на върхове на граф в
// цяло число – приемаме, че името на връх е главна буква
                ЦЯЛО      име на връх
int index (char c)
{
  return (c-'A'+1);
}

// Конвертира цяло число в 1-буквено име на връх на граф
                  име на връх     ЦЯЛО
char name (int i)
{
  return (i+'A'-1);
}
```

```
          ГРАФ
GRAPH create()
{
  GRAPH g;
  int i, j, k;
  char v1, v2;
  printf ("Въведи брой върхове = ");
  scanf ("%d", &g.V);
  printf ("Въведи брой ребра = ");
  scanf ("%d", &g.E);
  for (i=1; i<=g.V; i++)
    for (j=1;j <=g.V; j++)
      g.adjmatrix[i][j] = 0;
  for (i=1; i<=g.V; i++)
    g.adjmatrix[i][i] = 1;
```

```
  for (k=1; k<=g.E; k++)
  {
    printf ("Въведи ребро:\n");
    printf ("Връх 1: ");
    fflush (stdin);
    v1 = getchar();
    printf ("Връх 2: ");
    fflush (stdin);
    v2 = getchar();
    i = index (v1);
    j = index (v2);
    g.adjmatrix[i][j] = 1;
    g.adjmatrix[j][i] = 1;
  }
  return g;
}
```

2. Обхождане на граф в дълбочина

Масив val[V] – **съхранява реда, в който върховете се посещават.**

```
#define UNUSED 0          // върхът не е посетен
#define USED 1            // върхът е посетен
```

Алгоритъм

за k от 1 до V **повтаряй**
  val[k] ← **НЕПОСЕТЕН**
за k от 1 до V **повтаряй**
  ако val[k] **е НЕПОСЕТЕН**
    посети връх k

```
void search (GRAPH g, int val[])
{
  int k;
  for (k=1; k<=g.V; k++)          // инициализиране
    val[k] = UNUSED;
  for (k=1; k<=g.V; k++)
    if (val[k] == UNUSED)         // посещение на първия
      visit (g, val, k);          // непосетен връх
}
```

ГРАФ

масив за реда на посещение на върховете

посещение на връх k
val[k] ← **ПОСЕТЕН**
печат на името на посетения възел
за t от 1 до V **повтаряй**
  ако adjmatrix[k][t] **!= 0**
    ако val[t] **е НЕПОСЕТЕН**
      посети връх t

```
void visit (GRAPH g, int val[], int k)
{
  int t;
  val[k] = USED;
  printf("%c ", name(k));
  for (t=1; t<=g.V; t=t++)
    if (g.adjmatrix[k][t] != 0)        // съществува ребро
      if (val[t] == UNUSED)            // съседният връх t не е посетен
        visit (g, val, t);
}
```

ГРАФ

връх за посещение

масив за реда на посещение на върховете

Обхождане на граф в дълбочина
при следната последователност от въвеждане на ребрата:
AG AB AC LM JM JL JK ED FD HI FE AF GE.

A B C F D E G H I J K L M

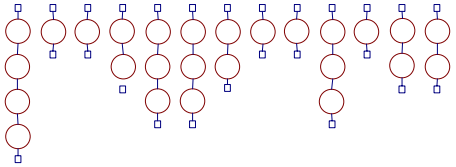
Физическо представяне чрез структура на съседство

Представяне чрез структура на съседство – всички върхове, свързани към даден връх, се изобразяват чрез списък на съседство за този връх.

Списъкът на съседство за даден връх се представя чрез структурата линеен списък.

Началните възли на всеки списък се съхраняват в масив adj**, индексиран от върха.**

Пример:



Структура от списъци на съседство при следната последователност от въвеждане на ребрата: AG AB AC LM JM JL JK ED FD HI FE AF GE.

Логическо описание

Съединение на елементи от тип ВЪЗЕЛ, образуващи масив от линейни списъци на съседство за граф с V върха и E ребра.

тип ВЪЗЕЛ = (данна: ЦЯЛО; {данна}
следващ: *ВЪЗЕЛ; {връзка})

тип ГРАФ = (V: ЦЯЛО; {брой върхове}
E: ЦЯЛО; {брой ребра}
масив adj[1:MAXV]: *ВЪЗЕЛ)

Дефиниране

```
#define MAXV <константа>
struct node                    // възел
{
  int v;                       // данни
  struct node *next;           // връзка
};
typedef struct node NODE;
typedef NODE *LINK;
struct graph                   // ГРАФ
{
  int V;                       // брой върхове
  int E;                       // брой ребра
  LINK adj[MAXV];              // масив от линейни списъци
};                             // на съседство

typedef struct graph GRAPH;
```

Характеристики

1. **Представянето чрез структура на съседство е подходящо за разредени графи – необходимото пространство е O(V+E) за разлика от $O(V^2)$, необходимо за представянето чрез матрица на съседство.**
2. **Всяко ребро се представя два пъти отново – ребро между върховете i и j се представя като връх, съдържащ i в списъка на съседство на j и като връх, съдържащ j в списъка на съседство на i.**
3. **Важен е редът, в който се въвеждат ребрата.**

Операции

1. **Създаване на граф**
2. **Обхождане на граф в дълбочина**
   - **рекурсивен вариант**
   - **нерекурсивен вариант**
3. **Обхождане на граф в ширина**

1. Създаване на граф

Алгоритъм

въведи V **и** E
за i **от** 1 **до** V **повтаряй**
  adj[i] ← **ПРАЗНО**
за k **от** 1 **до** E **повтаряй**
  въведи $символ_{връх1}$ **и** $символ_{връх2}$
  i ← **конвертиран** $символ_{връх1}$ **в ЦЯЛО**
  j ← **конвертиран** $символ_{връх2}$ **в ЦЯЛО**
  t ← **нов връх**
  $данни_t$ ← i
  $следващ_t$ ← adj[j]
  adj[j] ← t
  t ← **нов връх**
  $данни_t$ ← j
  $следващ_t$ ← adj[i]
  adj[i] ← t

```
GRAPH create()                 ГРАФ
{  GRAPH g;
   int i, j, k;
   char v1, v2;
   LINK t;
   printf ("Въведи брой върхове = ");
   scanf ("%d", &g.V);
   printf ("Въведи брой ребра = ");
   scanf ("%d", &g.E);
   for (i=1; i<=g.V; i++)
     g.adj[i] = NULL;
   for (k=1; k<=g.E; k++)
   {
     printf ("Въведи ребро:\n");
     printf ("Връх 1: ");
     fflush (stdin);
     v1 = getchar();
```

```
    printf ("Връх 2: ");
    fflush (stdin);
    v2 = getchar();
    i = index (v1);
    j = index (v2);

    t = (LINK) malloc (sizeof (NODE));
    t->v = i;                       // връх i се добавя
    t->next = g.adj[j];             // в списъка на съседство на
    g.adj[j] = t;                   // връх j

    t = (LINK) malloc (sizeof(NODE));
    t->v = j;                       // връх j се добавя
    t->next = g.adj[i];             // в списъка на съседство на
    g.adj[i] = t;                   // връх i
  }
  return g;
}
```

2. Обхождане на граф в дълбочина

– рекурсивен вариант

Масив val[V] – **съхранява реда, в който върховете се посещават.**

```
#define UNUSED 0        // върхът не е посетен
#define USED 1          // върхът е посетен
```

Алгоритъм

за k от 1 до V **повтаряй**
  val[k] ← **НЕПОСЕТЕН**
за k от 1 до V **повтаряй**
  ако val[k] **е НЕПОСЕТЕН**
    посети връх k

ГРАФ — масив за реда на посещение на върховете

```
void search (GRAPH g, int val[])
{
  int k;
  for (k=1; k<=g.V; k++)          // инициализиране
    val[k] = UNUSED;
  for (k=1; k<=g.V; k++)
    if (val[k] == UNUSED)         // посещение на първия
      visit (g, val, k);          // непосетен връх
}
```

посещение на връх k
val[k] ← **ПОСЕТЕН**
печат на името на посетения възел
за t от първия до последния връх в списъка adj[k]
повтаряй
  ако val[t] **е НЕПОСЕТЕН**
    посети връх t

ГРАФ — връх за посещение — масив за реда на посещение на върховете

```
void visit (GRAPH g, int val[], int k)
{
  LINK t;
  val[k] = USED;
  printf ("%c ", name(k));
  for (t = g.adj[k]; t != NULL; t = t->next)
    if (val[t->v] == UNUSED)
      visit (g, val, t->v);
}
```

2. Обхождане на граф в дълбочина

– нерекурсивен вариант – използва стек, който съхранява върховете, които са докоснати, но все още не са посетени.

Масив val[V] – **съхранява реда, в който върховете се посещават.**

val[i] = UNUSED	**върхът не е посетен**
val[i] = -1	**върхът е в стека**
val[i] = 1:V	**върхът е посетен**

ГРАФ — масив за реда на посещение на върховете — СТЕК

```
void search (GRAPH g, int val[], STACK s)
{
  int k;
  for (k=1; k<=g.V; k++)
    val[k] = UNUSED;
  for (k=1; k<=g.V; k++)
    if (val[k] == UNUSED)
      visit (g, val, s, k);
}
```

посещение на връх k
включи възел k във върха на стека
докато стекът стане празен повтаряй
  изключи възел k от върха на стека
  val[k] ← **ПОСЕТЕН**
  печат на името на посетения възел
  за t от първия до последния връх в списъка adj[k]
  повтаряй
    ако val[t] **е НЕПОСЕТЕН**
      включи възел t във върха на стека
      val[t] ← **-1**

ГРАФ — СТЕК — връх за посещение

```
void visit (GRAPH g, int val[], STACK s, int k)
{
  LINK t;
  push (&s,k);
  while (s.top != NULL)
  {
    pop (&s, &k);
    val[k] = USED;
    printf ("%c ", name(k));
    for (t = g.adj[k]; t != NULL; t = t->next)
        if (val[t->v] == UNUSED)
        {
          push (&s, t->v);
          val[t->v] = -1;
        }
  }
}
```

масив за реда на посещение на върховете

3. Обхождане на граф в ширина – използва опашка, която съхранява върховете, които са докоснати, но все още не са посетени.

ГРАФ — масив за реда на посещение на върховете — ОПАШКА

```
void search (GRAPH g, int val[], QUEUE q)
{
  int k;
  for (k=1 ;k <= g.V; k++)
    val[k] = UNUSED;
  for (k=1; k<=g.V; k++)
    if (val[k] == UNUSED)
        visit (g, val, q, k);
}
```

посещение на връх k
включи възел k в опашката
докато опашката стане празна
  изключи възел k от опашката
  val[k] ← **ПОСЕТЕН**
  печат на името на посетения възел
  за t от първия до последния връх в списъка adj[k]
  повтаряй
    ако val[t] **е НЕПОСЕТЕН**
      включи възел t в опашката
      val[t] ← **-1**

ГРАФ — ОПАШКА — връх за посещение

```
void visit (GRAPH g, int val[], QUEUE q, int k)
{  LINK t;
  put (&q, k);
  while (!(q.front == NULL && q.rear == NULL))
  {
    get (&q, &k);
    val[k] = USED;
    printf ("%c ", name(k));
    for (t = g.adj[k]; t != NULL; t = t->next)
        if (val[t->v] == UNUSED)
        {
          put (&q, t->v);
          val[t->v] = -1;
        }
  }
}
```

масив за реда на посещение на върховете

Задача: **Абстрактен тип данни ГРАФ**

1. **Реализирайте граф чрез матрица на съседство.**
2. **Напишете функция** print()**, която отпечатва граф, представен чрез матрица на съседство.**

Алгоритъм

```
за i от 1 до V повтаряй
  отпечатай името на връх i
за i от 1 до V повтаряй
  отпечатай името на връх i
  за j от 1 до V повтаряй
    отпечатай admatrix[i][j]
```

3. **Отпечатайте графа, като използвате функцията** print()**.**

4. **Реализирайте граф чрез структура на съседство.**
5. **Напишете функция** print()**, която отпечатва граф, представен чрез структура на съседство.**

Алгоритъм

```
за i от 1 до V повтаряй
  отпечатай името на връх i
  докато adj[i] е различно от ПРАЗНО повтаряй
    отпечатай името на върха в adj[i]
    t ← следващия елемент на adj[i]
    adj[i] ← t
```

6. **Отпечатайте графа, като използвате функцията** print()**.**