# Drzewa

Definicja rekurencyjna drzewa

1) Pojedynczy wierzchołek jest drzewem.

2) Jeśli dane są drzewa $T_1, \dots T_k$ o korzeniach $r_1, \dots, r_k$ oraz pojedynczy wierzchołek $r$, to możemy stworzyć nowe drzewo $T$ o korzeniu w $r$ w ten sposób, że $r$ będzie ojcem wierzchołków $r_1, \dots, r_k$.

Ciąg wierzchołków $v_1, \dots, v_k$ nazywamy ścieżką w drzewie, jeśli $\forall i = 2, \dots, k$ wierzchołek $v_i$ jest synem wierzchołka $v_{i-1}$

Długością ścieżki jest liczba krawędzi w ścieżce (czyli w ścieżce $v_1, \dots, v_k$ liczba k-1).

Jeśli w drzewie istnieje ścieżka od wierzchołka $v$ do wierzchołka $u$, to mówimy, że $v$ jest przodkiem wierzchołka $u$, a $u$ jest potomkiem wierzchołka $v$.

Wierzchołek, który nie posiada synów nazywamy liściem.

Liczbę synów danego wierzchołka $v$ nazywamy stopniem danego wierzchołka i oznaczamy $d(v)$.

Stopień całego drzewa $d(T)$ to maksymalny ze stopni jego wierzchołków.

Wysokością wierzchołka w drzewie nazywamy długość najdłuższej ścieżki od tego wierzchołka do jednego z jego potomków.

Wysokością drzewa $h$ nazywamy wysokość jego korzenia.

**Def**. Drzewo pełne rzędu $k$ posiada stopień każdego wierzchołka nie będącego liściem równy $k$ i wszystkie liście są na tym samym poziomie.

Liczba wszystkich wierzchołków w drzewie pełnym stopnia $k$ o wysokości $h$ jest równa

$$1 + k + k^2 + \ldots + k^h = \frac{1 - k^{h+1}}{1 - k} = \frac{k^{h+1} - 1}{k - 1}, \quad k > 1.$$

Dla $k = 1 : h + 1$.

# Implementacje drzew

Implementacja **tablicowa**

```
class drzewo
{ int n; //n jest liczbą węzłów drzewa o węzłach 0,...,n-1
  int ojciec[MAX]; // ojciec[i]=j jeśli j jest ojcem i
                   // ojciec[k]=-1 jeśli k jest korzeniem
  public:
  drzewo(){};
  friend istream & operator >> (istream &, drzewo &);
  friend ostream & operator << (ostream &, drzewo);
  int StopienWezla(int);
  int StopienDrzewa();
  int LiczbaLisci();
  int WysokoscDrzewa(); // niełatwe

};

istream & operator >> (istream &we, drzewo &T)
{cout<<"Podaj liczbę węzłów w drzewie: ";
 we>>T.n;
 for(int i=0; i<n, i++)
  {cout<<"Podaj ojca węzła "<<i<<": ";
   we>>T.ojciec[i];
```

```
  }
 return we;
}

int drzewo::StopienWezla(int i)
{ int licznik=0;

 for(int j=0; j<n; j++)
 if(ojciec[j]==i) licznik++;

 return licznik;
}

int drzewo::LiczbaLisci()
{int licznik=0;

 for(j=0; j<n; j++)
  if(StopienWezla(j)==0) licznik++;

 return licznik;
}
```

Implementacja drzewa poprzez **listy synów**

```
struct wezel
{int klucz; //numer wierzchołka
wezel *nast;
}

class drzewo
{int n; // n jest liczbą węzłów drzewa o kluczach 0,..., n-1
 wezel *tab[MAX];

 public:
 drzewo(){};
 friend istream & operator >> (istream &, drzewo &);
 friend ostream & operator << (ostream &, drzewo);
 int StopienWezla(int i);
 int StopienDrzewa();
 int LiczbaLisci();
 int WysokoscDrzewa(); // niełatwe

};

istream & operator >> (istream &we, drzewo &T)
{ int ojciec;
  wezel *pom;

  cout<<"podaj liczbę węzłów w drzewie: ";
  we>>T.n;

  for(int i=0; i<n; i++)
  T.tab[i]=NULL;

  for(int i=0; i<n; i++)
  {cout<<"Podaj ojca węzła "<<i<<": ";
   we>> ojciec;
   pom=new wezel;//dodajemy nowy wezel na poczatek listy ojca
   if(pom!=NULL)
    {pom->klucz=i;
```

```
        pom->nast=tab[ojciec];
        tab[ojciec]=pom;;
      }
   }
 return we;
}

int drzewo::StopienWezla(int i)
{ wezel *pom;
  int licznik=0;

  pom=tab[i];
  while(pom!=NULL)
  {licznik++;
   pom=pom->nast;
  }
 return licznik;
}

ostream & operator << (ostream &wy, drzewo T)
{  // wypisujemy kolejne węzły i po dwukropku ich synów
  wezel *pom;
  for(int i=0; i<n, i++)
   {wy<<i<<": ";
    pom=tab[i];
    while(pom!=NULL)
     { wy<<pom->klucz<<", ";
       pom=pom->nast;
     }
   }
  return wy;
}
```

Implementacja drzewa - **Lewy syn - prawy brat**

```
struct wezel
{int klucz;
wezel *ojciec, *lewy_syn, *prawy_brat;
};

class drzewo
{wezel *korzen;
public:
...
...
...
};
```

**Drzewa binarne**

**Def**. Drzewo nazywamy binarnym, jeśli stopień każdego wierzchołka wynosi co najwyżej 2.

## Twierdzenie

$h+1 \leqslant n \leqslant 2^{h+1}-1$,
gdzie $h$ - wysokość drzewa i $n$ - liczba wierzchołków drzewa.

$n \leqslant 2^{h+1} - 1$
$2^{h+1} \geqslant n+1 \quad / \log_2$
$h+1 \geqslant \log_2(n+1)$
$h \geqslant \log_2(n+1) - 1$

## Wniosek

$\log_2(n+1) - 1 \leqslant h \leqslant n-1$

### Implementacja drzewa binarnego

Rozważamy drzewo binarne w postaci uporządkowanej (tzn. synowie danego węzła mają jednoznacznie określonego lewego i prawego syna).

```
struct wezel
{int klucz;
wezel *ojciec, *lewy, *prawy;
};

class drzewo2 //binarne
{wezel *korzen;
public:
...
void preorder(wezel *);
void preorder();
...
...
};
```

# Porządki wierzchołków w drzewach binarnych

Rekurencyjne metody uporządkowania wierzchołków w drzewie:

1) PREORDER
   A. wypisz wierzchołek,
   B. wypisz rekurencyjnie węzły lewego poddrzewa w porządku preorder
   C. wypisz rekurencyjnie węzły prawego poddrzewa w porządku preorder

2) POSTORDER
   B. wypisz rekurencyjnie węzły lewego poddrzewa w porządku postorder
   C. wypisz rekurencyjnie węzły prawego poddrzewa w porządku postorder
   A. wypisz wierzchołek,

3) INORDER
   B. wypisz rekurencyjnie węzły lewego poddrzewa w porządku inorder
   A. wypisz wierzchołek,
   C. wypisz rekurencyjnie węzły prawego poddrzewa w porządku inorder

Spostrzeżenie. Kolejność liści w każdym z tym trzech uporządkowań jest zawsze taka sama.

```
void drzewo2::preorder(wezel *v) //wypisuje węzły poddrzewa
{if(v!=NULL)                      // o korzeniu w węźle v
   {cout<<v->klucz<<", ";
    preorder(v->lewy);
    preorder(v->prawy);
    }
}
```

```
void drzewo2::preorder() //wypisuje węzły całego drzewa
{ preorder(korzen);
}
```

## Drzewa przeszukiwań binarnych (ang. BST - Binary Search Tree)

→ Wykorzystywane są do reprezentowania dynamicznego zbioru danych, którego elementy są identyfikowane za pomocą reprezentujących je kluczy.

Taka struktura może być wykorzystywana jako np. kolejka priorytetowa.

BST to takie drzewo binarne, w którym klucze uporządkowane są według następującej reguły:

klucz(u) $\leqslant$ klucz (v) dla każdego *u* znajdującego się w lewym poddrzewie wierzchołka *v*.

klucz(v) < klucz (u) dla każdego węzła *u* znajdującego się w prawym poddrzewie wierzchołka *v*.

```
struct wezel
{
      typ_klucza klucz;
      wezel *ojciec, *lewy, *prawy;
};

class BST
{
    wezel *korzen;
    public:
    BST();
    wezel* minimum();
    wezel* maksimum();
    wezel *znajdz(typ_klucza);
    void wstaw(typ_klucza);
    wezel* nastepnik(wezel*);
    wezel* poprzednik(wezel*);
    void usun(wezel*);
    ~BST();
};

 BST::BST(){
   korzen=NULL;
 }

 wezel* BST::minimum()
{ //zwraca wskaźnik do węzła z minimalną wartością klucza
    wezel *pom;
    pom=korzen;
    if(pom==NULL) return pom;
    while(pom->lewy!=NULL) pom=pom->lewy;
    return pom;
 }

 wezel* BST::znajdz(typ_klucza x)
{
 //wyszukuje pierwsze wystąpienie w drzewie klucza
```

```
// o wartości x lub zwraca NULL gdy nie ma takiego
// elementu w drzewie

    wezel *pom;
    pom=korzen;
    while(pom!=NULL && pom->klucz!=x){
        if(x<=pom->klucz) pom=pom->lewy;
        else pom=pom->prawy;
    }
    return pom;
}

  void BST::wstaw(typ_klucza x)
{//nowy element jest zawsze wstawiany jako liść
    wezel *p, *q, *r;
    p=new wezel;
    if(p!=NULL){
        p->klucz=x;
        p->lewy=p->prawy=NULL;
        q=korzen;
        r=NULL;
        while(q!=NULL)
        {
            r=q;
            if(x<=q->klucz) q=q->lewy;
            else q=q->prawy;
        }
        if(r==NULL){   //drzewo puste
            korzen=p;
            korzen->ojciec=NULL;
        }
        else {
            if(x<=r->klucz){
                r->lewy=p;
                p->ojciec=r;
            }
            else{
                r->prawy=p;
                p->ojciec=r;
            }
        }
    }
  }
```

Spostrzeżenie. Węzły (klucze) w drzewie BST w porządku INORDER tworzą zawsze ciąg niemalejący.

```
wezel *BST::nastepnik(wezel *w)
 {
//zwraca wskaźnik do elementu, który jest następnikiem
//w porządku INORDER elementu wskazywanego przez wskaźnik w

   wezel *p, *q;
   p=w;
   if(p->prawy!=NULL){
      q=p->prawy;
      while(q->lewy!=NULL)
         q=q->lewy;
      return q;
   }
   else{
```

```
    while(p->ojciec!=NULL && p->ojciec->prawy==p)
      p=p->ojciec;
    return p->ojciec;
  }

wezel* BST::poprzednik(wezel *w)
{
  //zwraca wskaźnik do elementu, który jest poprzednikiem
  //w porządku INORDER elementu wskazywanego przez wskaźnik w.
  wezel *p,*q;
   p=w;
   if(p->lewy!=NULL){
     q=p->lewy;
     while(q->prawy!=NULL)
       q=q->prawy;
     return q;
   }
   else{
     while(p->ojciec!=NULL && p->ojciec->lewy==p)
       p=p->ojciec;
     return p->ojciec;
   }
}

}
```

W celu usunięcia węzła rozważamy 3 przypadki:

1) usuwamy węzeł, który nie ma synów
2) usuwamy węzeł, który ma dokładnie jednego syna
3) usuwamy węzeł, który ma dwóch synów

```
void BST::usun(wezel *w){
  wezel *s, *t;
  //s będzie wskaźnikiem do elementu, który
  //"fizycznie" będziemy usuwać
  if(w->lewy==NULL || w->prawy==NULL) s=w;
  else s=poprzednik(w);
  if(s->lewy!=NULL) t=s->lewy;
  else{
    if(s->prawy!=NULL) t=s->prawy;
    else t=NULL;
  }
  if(t!=NULL) t->ojciec=s->ojciec;
  if(s->ojciec==NULL) korzen=t;
  else{
    if(s->ojciec->prawy==s) s->ojciec->prawy=t;
    else s->ojciec->lewy=t;
  }
  w->klucz=s->klucz;
  delete s;
}
```

Zad. domowe. Napisać funkcje:

```
int wysokosc(wezel *w);
int wysokosc();
void potomkowie(wezel *w);
void przodkowie(wezel *w);
int CzyPotomek(wezel *p, wezel *w); //czy p jest potomkiem w
int CzyPrzodek (wezel *p, wezel *w); //czy p jest przodkiem w
```

**Spostrzeżenie**.
Podstawowe operacje na losowym drzewie BST (tzn. wstaw, usun, znajdz, ...) mają pesymistyczną złożoność O($h$) a w konsekwencji $O(n)$ (bo $h = O(n)$).

**Def**.
Niech **losowo skonstruowane drzewo BST** o $n$ kluczach będzie drzewem powstającym przez wykonanie ciągu operacji wstaw(...) dla $n$ kluczy pojawiających się w losowej kolejności do początkowo pustego drzewa. Zakładamy, że każda z $n!$ permutacji kluczy jest jednakowo prawdopodobna.

**Lemat**.
Wysokość losowo skonstruowanego drzewa BST o $n$ węzłach wynosi $\Theta(\log n)$.

**Wniosek**.
Podstawowe operacje na losowym drzewie BST (tzn. wstaw, usun, znajdz, ...) mają średnią złożoność O($\log n$).

# Drzewa AVL

→ są to drzewa BST o dodatkowej własności:
dla każdego węzła wysokość jego poddrzew różni się co najwyżej o 1.

Lemat
Wysokość drzewa AVL o $n$ węzłach wynosi $h = \Theta(\log n)$.

Dowód
Maksymalna liczba węzłow o drzewie binarnym o wysokości $h$ wynosi $2^{h+1} - 1$.
Niech $m_h$ oznacza minimalną liczbę węzłów w drzewie AVL o wysokości $h$.
$m_h = m_{h-1} + m_{h-2} + 1$

$y_n = y_{n-1} + y_{n-2} + 1$

$y_n - y_{n-1} - y_{n-2} = 1$ równanie rekurencyjne liniowe niejednorodne.

Rozwiązujemy najpierw równanie rekurencyjne jednorodne

$y_n - y_{n-1} - y_{n-2} = 0$

$y_n = \lambda^n$

$\lambda^n - \lambda^{n-1} - \lambda^{n-2} = 0 \qquad / : \lambda^{n-2}$

$\lambda^2 + \lambda - 1 = 0$

$\lambda_1 = \frac{1+\sqrt{5}}{2} \vee \lambda_2 = \frac{1-\sqrt{5}}{2}$

Dostajemy dwa rozwiązania bazowe: $y_n^1 = (\frac{1+\sqrt{5}}{2})^n$ oraz $y_n^2 = (\frac{1-\sqrt{5}}{2})^n$

Rozwiązanie ogólne równania jednorodnego jest kombinacją liniową rozwiązań bazowych.

$y_n = C_1(\frac{1+\sqrt{5}}{2})^n + C_2(\frac{1-\sqrt{5}}{2})^n$

Metodą przewidywania znajdujemy rozwiązanie szczególne równania niejednorodnego.
$y_* = A$

Wstawiamy $y_*$ do równania niejednorodnego.

$A - A - A = 1$

$A = -1$

$y_* = -1$

Rozwiązanie ogólne rozwiązania niejednorodnego jest sumą rozwiązania ogólnego równania jednorodnego i rozwiązania szczególnego.

$y_n = C_1(\frac{1-\sqrt{5}}{2})^n + C_2(\frac{1+\sqrt{5}}{2})^n - 1$

Znajdujemy wartość stałych $C_1$ i $C-2$ wykorzystując warunki początkowe.

$y_0 = m_0 = 1 \quad y_1 = m_1 = 2$

$y_n = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n - 1$

$y_n = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n - 1 \approx \frac{1}{\sqrt{5}}(1.61)^n - \frac{1}{\sqrt{5}}(-0.61)^n - 1 \approx \frac{1}{\sqrt{5}}(1.61)^n$

$m_h \leqslant n \leqslant 2^{h+1} - 1$

$2^{h+1} \geqslant n+1$

$h \geqslant \log_2(n+1) - 1$ (**)

$\frac{1}{\sqrt{5}}(1.61)^h \leqslant n \ \ /\log_{1.61}$

$h \leqslant \log_{1.61}\sqrt{5}n) = \frac{\log_2(\sqrt{5}n)}{\log_2 1.61} = \frac{\log_2\sqrt{5}+\log_2 n}{\log_2 1.61}$ (*)

(*), (**) $\Rightarrow \log_2(n+1) \leqslant h \leqslant \frac{\log_2\sqrt{5}+\log_2 n}{\log_2 1.61} \Rightarrow h = \Theta(\log_2 n)$

**Wniosek**
Wszystkie operacje podstawowe na drzewie AVL da się wykonać w pesymistycznym czasie O($\log n$)

Zad. Co to jest drzewo AVL? Podaj wzór rekurencyjny na minimalną liczbę węzłów w drzewie AVL o wysokości $h$. Oblicz minimalną liczbę węzłów jaką może mieć drzewo AVL o wysokości równej 4 i narysuj jedno z tych drzew.

Podobnie jak drzewa AVL definiujemy drzewa czerwono-czarne, B-drzewa, 2-3 drzewa.

# Grafy

$G = (V, E)$ - uporządkowana para zbiorów

$V$ - zbiór wierzchołków
Liczba wierzchołków nazywana jest **rzędem** grafu.

$E$ - zbiór krawędzi

$E \subset \{\{u, v\} : u, v \in V\}$

Często zamiast $\{u, v\}$ piszemy krócej $uv$.

Liczba krawędzi nazywana jest **rozmiarem** grafu.

$N(u) = \{v \in V : uv \ \in E\}$ - sąsiedztwo wierzchołka $u$

$d(u) = |N(u)|$ – stopień wierzchołka $u$

Przykład.
$V = \{0, 1, \ldots, 5\}$
$E = \{\{0, 1\}, \{0, 2\}, \{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 5\}\}$

# Implementacja grafów

1. **Macierz sąsiedztwa**
$V = \{0, 1, \ldots, n-1\}$
macierz $A \in M_{n\times n}$
$A[i][j] = \begin{cases} 1 & \{i, j\} \in E \\ 0 & \{i, j\} \notin E \end{cases}$
Macierz A jest macierzą symetryczną.

```
class graf
{int n;
int A[MaxRzad][MaxRzad];
public:
graf();
int LiczbaWierzcholkow();
int LiczbaKrawedzi();
int MaksymalnyStopien();
int Minimalny stopien();
istream & operator >>(istream &, graf &);
ostream & operator <<(ostream &, graf);
};

graf::graf()
{n=0;
for(int u=0; u<MAxRzad; u++)
 for(int v=0; v<MaxRzad; v++)
  A[u][v]=0;
}

istream & operator >>(istream &we, graf &G)
{cout<<"Podaj liczbę wierzchołków grafu: ";
 we>>G.n;
 cout<<"Podaj krawędzie grafu (jako pary liczb naturalnych)";
```

```
 cout<<"Jeśli chcesz zakończyć podawanie krawędzi wpisz parę -1 -1.";
 we>>u>>v;
 while(u>=0 && v>=0)
 {G.A[u][v]=1;
  G.A[v][u]=1;
  we>>u>>v;
 }
 return we;
}

int graf::LiczbaKrawedzi()
{int m=0;
 for(int u=0; u<n; u++)
   for(int v=u+1; v<n; v++)
     m+=A[u][v];

 return m;
}
```

2. **Lista sąsiedztw**

```
struct wierzcholek
{int nr;
 wierzcholek *nast;
};

class graf
{int n;
wierzcholek *tab[MaxRzad];
public:
graf();
int LiczbaWierzcholkow();
int LiczbaKrawedzi();
int MaksymalnyStopien();
int MinimalnyStopien();
istream & operator >>(istream &, graf &);
ostream & operator <<(ostream &, graf);
};

graf::graf()
{n=0;
for(int u=0; u<MAxRzad; u++)
 tab[u]=NULL;
}

ostream & operator <<(ostream &wy, graf G)
{for(int u=0; u<G.n; u++)
 wy<<u<<": ";
 wierzcholek *pom;
 pom=G.tab[u];
 while(pom!=NULL)
  {wy <<pom->nr<<", ";
   pom=pom->nast;
  }
 return wy;
}

istream & operator >>(istream &we, graf &G)
{cout<<"Podaj liczbę wierzchołków grafu: ";
 we>>G.n;
 cout<<"Podaj krawędzie grafu (jako pary liczb naturalnych)";
```

```
 cout<<"Jeśli chcesz zakończyć podawanie krawędzi wpisz parę -1 -1.";
 we>>u>>v;
 while(u>=0 && v>=0)
 {wierzchołek *pom;
  pom=new wierzcholek;
  if(pom!=NULL)  //dodajemy v do listy sąsiadów u
    {pom->nr=v;
     pom->nast=G.tab[u];
     G.tab[u]=pom;
    }
  pom=new wierzcholek;
  if(pom!=NULL)  //dodajemy u do listy sąsiadów v
    {pom->nr=u;
     pom->nast=G.tab[v];
     G.tab[v]=pom;
    }

  we>>u>>v;
 }
 return we;
}
```

# Przeglądanie wierzchołków grafu

### Przeglądanie grafu wgłąb (DFS - depth first search)

Pseudokod:

```
przegladanie_wglab(G)
{
  for(każdy wierzchołek v w G)  //trzeba zadeklarować
    odwiedzony[v]=0;            //globalnie tablicę
  for(każdy wierzchołek v w G)  //int odwiedzony[MaxRzad];
    if(odwiedzony[v]==0)
      DFS(G,v);
}

DFS(G,v)
{
  odwiedzony[v]=1;
  wypisz(v);
  for(każdy sąsiad u wierzchołka v w G)
    if(!odwiedzony[u])
      DFS(u);
  }
```

Przeglądanie wgłąb w implementacji poprzez macierz sąsiedztwa

```
int odwiedzony[MaxRzad];
void graf::przegladanie_wglab()
   { for(v=0; v<n; v++)
     odwiedzony[v]=0;

     for(v=0; v<n; v++)
       if(odwiedzony[v]==0)
         DFS(v);
   }
```

```
void graf::DFS(int v)
   {
     odwiedzony[v]=1;
     cout<<v<<", ";
     for(u=0; u<n; u++)
      if(A[v][u]==1) //u jest sąsiadem v
        if(odwiedzony[u]==0) //u jest nieodwiedzony
         DFS(u);
    }
```

Ćw.
Napisz funkcję spojny(), która sprawdzi czy graf jest spójny.

```
void graf::spojny()
{DFS(0);
 for(int v=0; v<n; v++)
  if(odwiedzony[v]==0) return 0;

 return 1;
}
```

Ćw. Napisz funkcję LiczbaSkladowychSpojnych() (która zwróci liczbę składowych spójnych grafu) oraz funkcję Wypisz-SkladoweSpojne(), która wypisze wierzchołki kolejnych składowych spójnych (np. skladowa 1: 0, 1, 4, ..., składowa 2: 2, 3, ...)

```
int odwiedzony[MaxRzad];
int k=0;
void graf::przegladanie_wglab2()
   { for(v=0; v<n; v++)
     odwiedzony[v]=0;

     for(v=0; v<n; v++)
        if(odwiedzony[v]==0)
        { k++;
         DFS2(v);
        }
   }

void graf::DFS2(int v)
   {
     odwiedzony[v]=k;
     for(u=0; u<n; u++)
      if(A[v][u]==1) //u jest sąsiadem v
        if(odwiedzony[u]==0) //u jest nieodwiedzony
         DFS2(u);
    }

int graf::LiczbaSkladowychSpojnych()
   { przegladanie_wglab2();
     return k;
    }

void graf::WypiszSkladoweSpojne()
   { przegladanie_wglab2();
     for(int i=1; i<=k; i++)
      { cout<<"Składowa "<<i<<": ";
        for(int j=0; j<n; j++)
         if(odwiedzony[j]==k) cout<<j<<", ";
```

```
        cout<<endl;
      }

    }
```

**Przeglądanie grafu wszerz (BFS - breadth first search)**

Pseudokod.

```
przegladanie_wszerz(G)
  {
  for(każdy wierzchołek v w G)
    odwiedzony[v]=0;
  for(każdy wierzchołek v w G)
    if(odwiedzony[v]==0)
      BFS(G,v);
}



BFS(G,v)
{
  odwiedzony[v]=1;
  wstaw_do_kolejki(v);
  while(!kolejka_pusta())
   {
    v=pobierz_z_kolejki(); //i usuń z kolejki
    wypisz(v);
    for(każdy sąsiad u wierzchołka v w G)
      if(!odwiedzony[u])
       {
        odwiedzony[u]=1;
        wstaw_do_kolejki(u);
       }
    }
}
```

Przeglądanie wszerz w implementacji poprzez macierz sąsiedztwa

```
int odwiedzony[MaxRzad];
void graf::przegladanie_wszerz()
  { for(v=0; v<n; v++)
    odwiedzony[v]=0;

    for(v=0; v<n; v++)
      if(odwiedzony[v]==0)
        BFS(v);
  }

#include "kolejka.h"
//w pliku kolejka.h mamy implementację kolejki
void graf::BFS(int v)
  { kolejka K;
    odwiedzony[v]=1;
    K.wstaw(v);
    while(K.pusta()==0)
     {
      v=K.zwroc();
      K.usun();
      cout<<v<<", ";
```

```
        for(u=0; u<n; u++)
        if(A[v][u]==1) //(*)
          if(odwiedzony[u]==0)
            {
              odwiedzony[u]=1;
             K.wstaw(u);
            }
       }
   }
```

**Przeglądanie wszerz w implementacji poprzez listy sąsiedztw**
W miejsce (*) wystarczy wstawić:

```
void graf::CzyKrawedz(int v, int u)
{wierzcholek *pom;
 pom=tab[v];
 while(pom!=NULL)
   {if (pom->nr==u) return 1;
    pom=pom->nast;
   }
 return 0;
}
```

Twierdzenie.
Złożoność algorytmów przeglądania wgłąb i wszerz: $O(m+n)(=O(n^2))$, gdzie $m$ ($\leqslant \frac{n(n-1)}{2}$) jest liczbą krawędzi w grafie.

**Cykl Eulera**

**Cyklem** (obchodem) **Eulera** w grafie $G$ nazywamy taką drogę, rozpoczynającą się i kończącą w tym samym wierzchołku, która zawiera wszystkie krawędzie grafu $G$ dokładnie 1 raz.

Graf, który ma cykl Eulera nazywamy **grafem eulerowskim**.

**Twierdzenie. (Euler, 1736)**
Graf $G$ jest eulerowski $\Leftrightarrow$ gdy $G$ jest spójny i wszystkie wierzchołki tego grafu mają stopnie parzyste.

Ćw. Napisz fukcję składową **eulerowski()** klasy graf (w obu implementacjach), która zwróci 1, jeśli graf jest eulerowski i zwróci 0 w przeciwnym przypadku.

Algorytm (łatwy do zapamiętania z bardzo kiepską złożonością)

Rozpocznij od dowolnego wierzchołka.
Z aktualnego wierzchołka przejdź do sąsiedniego wierzchołka dowolną krawędzią (która nie jest mostem) i usuń tę krawędź z grafu.

**Mostem** w grafie nazywamy taką krawędź, której usunięcie spowoduje, że liczba składowych spójnych w grafie wzrośnie.

Uwaga.
Sprawdzenie, czy krawędź jest mostem można dokonać w czasie $O(m)$ a więc sprawdzanie dla wszystkich $m$ krawędzi czy są one mostami spowoduje, że złożoność powyższego algorytmu jest aż $O(m^2)$.

```
cykl_Eulera()
{  //zakładamy, że G=(V,E) ma cykl Eulera
     STOS=∅// stos pomocniczy
     CE=∅// stos z kolejnymi wierzchołkami cyklu Eulera
      wybierz dowolny wierzchołek v grafu G i wstaw go na STOS
      while(STOS ≠∅)
         {
```

```
        v=szczyt_STOSU;
        if (N(v) ≠ ∅)  //N(v) - zbiór sąsiadów v w grafie G
          {
            u=pierwszy wierzchołek z N(v);
            wstaw wierzchołek u na STOS;
            usun krawędź uv z grafu G //E = E \ {u,v};
          }
         else //gdy N(v) = ∅
         {
         usuń v ze STOSU
         wstaw v na CE
         }
      }
}
```

```
#include "stos.h"
void graf::cykl_Eulera()
{   //zakładamy, że G = (V,E) ma cykl Eulera
     stos STOS, CE;
     STOS.wstaw(0);
     while(!STOS.pusty())
       {
        v=STOS.zwroc();
        int u=0;
         while (u<n)
           {if (A[v][u]==1)
               {STOS.wstaw(u);
                A[v][u]=0; A[u][v]=0;
                break;
               }
            u++;
            }
         if (u==n) //tzn. N(v) = ∅
          {STOS.usun();
           CE.wstaw(v);
          }
        }
}
```

Twierdzenie.
Złożoność algorytmu znajdowania cyklu Eulera jest $\theta(m)$.