Teoretyczne podstawy informatyki

Repetytorium:

Algorytmy

Modele danych

Automaty, wyrażenia regularne, gramatyki

Informatyka: mechanizacja abstrakcji

- **modele danych**: abstrakcje wykorzystywane do opisywania problemów
- **struktury danych:** konstrukcje języka programowania wykorzystywane do reprezentowania modelów danych. Przykładowo język C udostępnia wbudowane abstrakcje takie jak struktury czy wskaźniki, które umożliwiają reprezentowanie skomplikowanych abstrakcji takich jak grafy
- **algorytmy:** techniki wykorzystywane do otrzymywania rozwiązań na podstawie operacji wykonywanych na danych reprezentowanych przez abstrakcje modelu danych, struktury danych lub na inne sposoby

Struktury danych i algorytmy

- **Struktury danych** to **narzędzia** do reprezentowania informacji która ma być przetworzona przez program komputerowy,
- **Algorytmy** to **przepisy** wykonania czynności niezbędnych do jej przetworzenia.
- **Wybór algorytmu** do rozwiązania konkretnego problemu programistycznego pomaga w ustaleniu, jaką strukturę danych należałoby użyć, ale i odwrotnie – **wybrana struktura danych** ma ogromny wpływ na szczegóły realizacji i efektywności algorytmu.

Typy danych i struktury danych

Dane są to „obiekty" którymi manipuluje algorytm.
Te obiekty to nie tylko dane wejściowe lub wyjściowe (wyniki działania algorytmu), to również obiekty pośrednie tworzone i używane w trakcie działanie algorytmu.

Dane mogą by różnych **typów**, do najpospolitszych należą liczby (całkowite, dziesiętne, ułamkowe) i słowa zapisane w rozmaitych alfabetach.

Typy danych i struktury danych

Interesują nas sposoby w jaki algorytmy mogą organizować, zapamiętywać i zmieniać zbiory danych oraz sięgać do nich.

- Zmienne czyli „pudelka” w których chwilowo przechowujemy jakąś wartość,
- Wektory,
- Listy,
- Tablice czyli tabele (macierze), w których to możemy odwoływać się do indeksów,
- Kolejki i stosy,
- Drzewa, czyli hierarchiczne ułożenie danych,
- Zbiory…. Grafy…. Relacje….

Typy danych i struktury danych

W wielu zastosowaniach same struktury danych nie wystarczają.

Czasami potrzeba bardzo obszernych zasobów danych, stanowiących dla wielu algorytmów potencjalne dane wejściowe, a więc mające ustaloną strukturę i nadające się do odszukiwania i manipulowania nimi.

Nazywa się je **bazami danych** (relacyjne i hierarchiczne).

Kolejny krok to **bazy wiedzy**, których elementami są bazy danych, a które zawierają również informacje o związkach pomiędzy danymi.

Algorytmika

- Algorytm to „przepis postępowania” prowadzący do rozwiązania konkretnego zadania; zbiór poleceń dotyczących pewnych obiektów (danych) ze wskazaniem kolejności w jakiej mają być wykonane”.
- Jest jednoznaczną i precyzyjną specyfikacją kroków które mogą być wykonywane „mechanicznie”.
- W matematyce algorytm jest „pojęciem służącym do formułowania i badania rozstrzygalności problemów i teorii”
- Algorytm odpowiada na pytanie „jak to zrobić” postawione przy formułowaniu zadania. Istota algorytmu polega na rozpisaniu całej procedury na kolejne, możliwie elementarne kroki.
- **Algorytmiczne myślenie można kształtować niezależnie od programowania komputerów, chociaż każdy program komputerowy jest zapisem jakiegoś algorytmu.**

Sposoby zapisu algorytmu

- Najprostszy sposób zapisu to **zapis słowny**
  - Pozwala określić kierunek działań i odpowiedzieć na pytanie, czy zagadnienie jest możliwe do rozwiązania.
- Bardziej konkretny zapis to **lista kroków**
  - Staramy się zapisać kolejne operacje w postaci kolejnych kroków które należy wykonać.
- Budowa listy kroków obejmuje następujące elementy:
  - sformułowanie zagadnienia (zadanie algorytmu),
  - określenie zbioru danych potrzebnych do rozwiązania zagadnienia (określenie czy zbiór danych jest właściwy),
  - określenie przewidywanego wyniku (wyników): co chcemy otrzymać i jakie mogą być warianty rozwiązania,
  - zapis kolejnych ponumerowanych kroków, które należy wykonać, aby przejść od punktu początkowego do końcowego.
- Bardzo wygodny zapis to **zapis graficzny**, np:
  - Schematy blokowe i grafy.

Rodzaje algorytmów

Algorytmy można dzielić ze względu na czas działania.

- Algorytm liniowy:
  - Ma postać ciągu kroków (których jest liniowa ilość) które muszą zostać bezwarunkowo wykonane jeden po drugim.
  - Algorytm taki nie zawiera żadnych warunków ani rozgałęzień: zaczyna się od podania zestawu danych, następnie wykonywane są kolejne kroki wykonawcze, aż dochodzimy do wyniku

Rodzaje algorytmów

Algorytm z rozgałęzieniem:

- Większość algorytmów zawiera rozgałęzienia będące efektem sprawdzania warunków. Wyrażenia warunkowe umożliwiają wykonanie zadania dla wielu wariantów danych i rozważanie różnych przypadków.
- Powtarzanie różnych działań ma dwojaką postać:
  - liczba powtórzeń jest z góry określona (przed rozpoczęciem cyklu), najczęściej związany z działaniami na macierzach,
  - liczba powtórzeń jest nieznana (zależy od spełnienia pewnego warunku), najczęściej związany z obliczeniami typu iteracyjnego.

Algorytmy „dziel i zwyciężaj”

- Dzielimy problem na mniejsze części tej samej postaci co pierwotny.
- Teraz te pod-problemy dzielimy dalej na coraz mniejsze, używając tej samej metody, aż **rozmiar problemu** stanie się tak **mały**, że rozwiązanie będzie oczywiste lub będzie można użyć jakiejś innej efektywnej metody rozwiązania.
- Rozwiązania wszystkich pod-problemów muszą być połączone w celu utworzenia rozwiązania całego problemu.
- **Metoda zazwyczaj implementowana z zastosowaniem technik rekurencyjnych.**

Algorytmy oparte na programowaniu dynamicznym

- Można stosować wówczas, kiedy problem daje się podzielić na wiele pod-problemów, możliwych do zakodowania w jedno-, dwu- lub wielowymiarowej tablicy, w taki sposób że w pewnej określonej kolejności można je wszystkie (a więc i cały problem) efektywnie rozwiązać.

Jak obliczać ciąg Fibonacciego?

$$F(i) = \begin{cases} 1 & \text{jeśli } i = 1 \\ 1 & \text{jeśli } i = 2 \\ F(i-2)+F(i-1) & \text{jeśli } i > 2 \end{cases}$$

Aby obliczyć **F(n)**, wartość **F(k)**, gdzie **k<n** musimy wyliczyć **F(n-k)** razy.
Liczba ta rośnie wykładniczo.
Korzystnie jest więc zachować (zapamiętać w tablicy) wyniki wcześniejszych obliczeń (tu: **F(k)**).

Algorytmy z powrotami

- Przykładami tego typu algorytmów są gry.
  - Często możemy zdefiniować jakiś problem jako poszukiwanie jakiegoś rozwiązania wśród wielu możliwych przypadków.
  - Dana jest pewna przestrzeń stanów, przy czym stan jest to sytuacja stanowiąca rozwiązanie problemu albo mogąca prowadzić do rozwiązania oraz sposób przechodzenia z jednego stanu do drugiego.
  - Czasami mogą istnieć stany które nie prowadzą do rozwiązania.

Algorytmy wykorzystujące prawdopodobieństwo

- Jest bardzo wiele różnych typów algorytmów wykorzystujących prawdopodobieństwo.
- Jeden z nich to tzw. **algorytmy Monte-Carlo** które wykorzystują liczby losowe do zwracania albo wyniku pożądanego („prawda”), albo żadnego („nie wiem”). Wykonując algorytm **stałą** liczbę razy, możemy rozwiązać problem, dochodząc do wniosku, że jeśli żadne z tych powtórzeń nie doprowadziło nas do odpowiedzi „prawda”, to odpowiedzią jest „fałsz”.
  Odpowiednio dobierając liczbę powtórzeń, możemy dostosować prawdopodobieństwo niepoprawnego wniosku „fałsz” do tak niskiego poziomu, jak w danym przypadku uznamy za konieczne.
- **Nigdy** jednak nie osiągniemy prawdopodobieństwa popełnienia błędu na poziomie **zero**.

Co to są liczby losowe?

- **Wszystkie** generowane przez komputer **losowe** sekwencje są wynikiem działania specjalnego rodzaju algorytmu zwanego **generatorem liczb losowych** (ang. random number generator). Zaprojektowanie takiego algorytmu wymaga specjalistycznej wiedzy matematycznej.
  Przykład prostego generatora który całkiem dobrze sprawdza się w praktyce to tzw. "**liniowy generator kongurencyjny**".
  Wyznaczamy **stałe $a \geq 2$, $b \geq 1$, $x_0 \geq 0$** oraz **współczynnik $m > \max(a, b, x_0)$**.
  Możemy teraz wygenerować sekwencje liczb x_1, x_2, … za pomocą wzoru:

$$x_{n+1} = (a\, x_n + b) \bmod(m)$$

- Dla właściwych wartości stałych a, b, m oraz x_0, sekwencja wynikowa będzie wyglądała na losową, mimo ze została ona wygenerowana przy użyciu konkretnego algorytmu i na podstawie "jądra" x_0.
- **Dla szeregu zastosowań istotna jest odtwarzalność sekwencji liczb losowych.**

Algorytmy wykorzystujące prawdopodobieństwo

- Mamy pudełko w którym jest **n**-procesorów, nie mamy pewności czy zostały przetestowane przez producenta. Zakładamy że prawdopodobieństwo że procesor jest wadliwy (w nieprzetestowanym pudelku) jest 0.10.

- **Co możemy zrobić aby potwierdzić czy pudełko dobre?**
  - przejrzeć wszystkie procesory -> algorytm O(n)
  - losowo wybrać k procesorów do sprawdzenia -> algorytm O(1)
  - błąd polegałby na uznaniu ze pudełko dobre (przetestowane) jeżeli nie było takie.

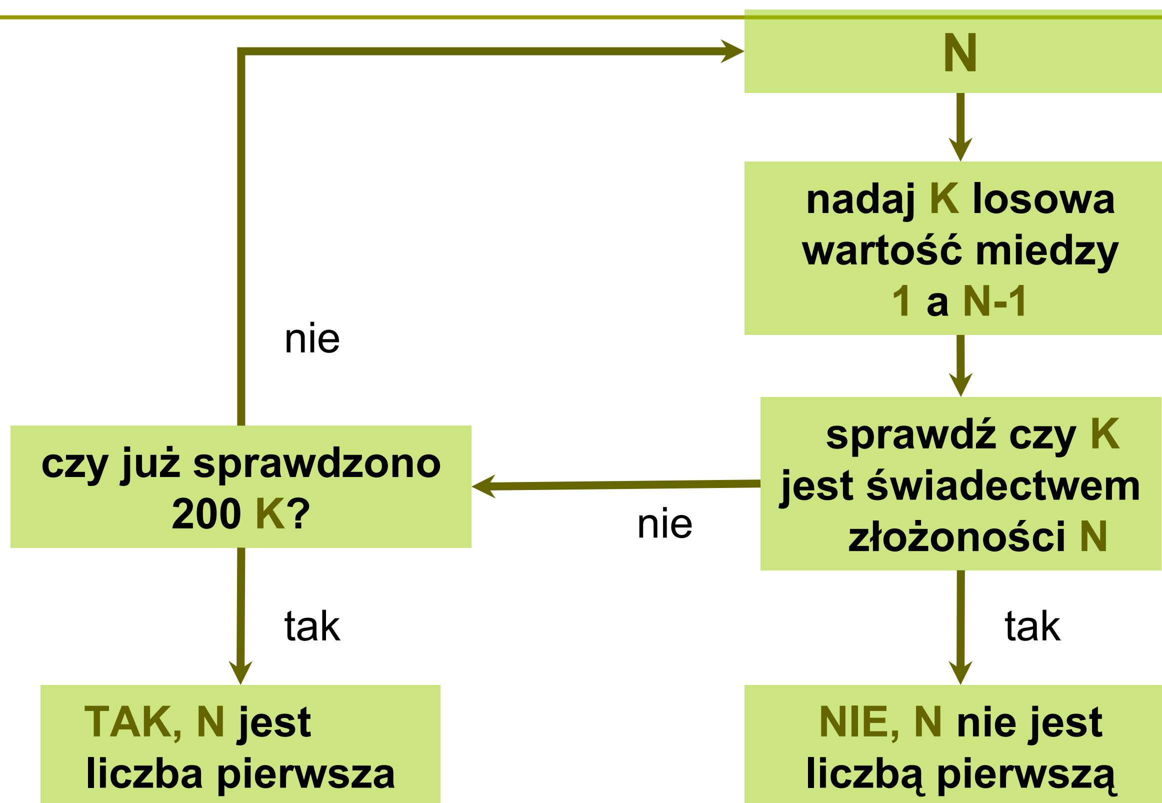
- Losujemy **k=131** procesorów.
  Jeżeli procesor jest dobry odpowiadamy „nie wiem". Prawdopodobieństwo ze „nie wiem" dla każdego z **k**-procesorów $\mathbf{(0.9)^k = (0.9)^{131} = 10^{-6}}$.
  $\mathbf{10^{-6}}$ to jest prawdopodobieństwo że pudełko uznamy za dobre choć nie było testowane przez producenta.
  Za cenę błędu = $\mathbf{10^{-6}}$, zamieniliśmy algorytm z ***O*(n)** na ***O*(1)**.
  Możemy regulować wielkość błędu/czas działania algorytmu zmieniając **k**.

Probabilistyczne algorytmy sprawdzania

„ Czy liczba N jest liczbą pierwszą ?”

- W połowie lat 70-tych odkryto **dwa bardzo eleganckie probabilistyczne algorytmy** sprawdzające, czy liczba jest pierwsza. Były one jednymi z pierwszych rozwiązań probabilistycznych dla trudnych problemów algorytmicznych. Wywołały fale badań które doprowadziły do probabilistycznych rozwiązań wielu innych problemów.
- Oba algorytmy wykonują się w czasie wielomianowym (niskiego stopnia), zależnym od liczby cyfr w danej liczbie **N** (czyli ***O*** **(log N)**).
- Oba algorytmy są oparte na losowym szukaniu pewnych rodzajów potwierdzeń lub **świadectw złożoności** liczby N.
- Po znalezieniu takiego świadectwa algorytm może się bezpiecznie zatrzymać z odpowiedzią **„nie, N nie jest liczbą pierwszą”,** ponieważ istnieje bezdyskusyjny dowód że **N** jest liczbą złożoną.
- Poszukiwanie musi być przeprowadzone w taki sposób aby w pewnym rozsądnym czasie algorytm mógł przerwać szukanie odpowiadając, że **N** jest liczbą pierwszą z bardzo małą szansą omyłki.
- Trzeba zatem znaleźć dająca się **szybko sprawdzać** definicje świadectwa złożoności.

„Czy liczba N jest liczbą pierwszą ?



jeśli to jest odpowiedź to jest to prawda z błędem $1/2^{200}$

jeśli to jest odpowiedź to jest to prawda

Świadectwa złożoności (zarys)

- Każda liczba parzysta poza 2 to jest złożona
- Jeżeli suma cyfr liczby jest podzielna przez 3 to liczba jest złożona (iteracyjny prosty algorytm liniowo zależny od liczby cyfr)
- **Test pierwszości Fermata:**
  - jeśli **n** jest liczbą pierwszą oraz **k** jest dowolna liczba całkowita **(1, n-1)**, to $k^{n-1} \equiv 1 \pmod{n}$.
  - natomiast jeśli **n** jest liczbą złożoną (z wyjątkiem kilku złych liczb złożonych – liczb Carmichael'a) oraz jeśli **k** wybierzemy losowo z przedziału **(1, n-1)** to prawdopodobieństwo tego że $k^{n-1} \not\equiv 1 \pmod{n}$ jest mniejsze niż ½.
  - Zatem liczby złożone (poza liczbami Carmichael'a) spełniają warunek testu dla danego **k** z prawdopodobieństwem nie mniejszym niż ½.
- **Test pierwszości Solovay-Strassena:**
  - jeśli **k** i **n** nie maja wspólnych dzielników (co by było świadectwem złożoności) policz: $X = k^{(n-1)/2} \pmod{n}$, **Y = Js(n,k) (symbol Jacobiego),** jeśli $X \neq Y$ to **k** jest świadectwem złożoności liczby **n**.
  - dla tego testu nie ma 'złych' liczb złożonych.

Wybór algorytmu

- Regułą jest że należy implementować algorytmy najprostsze, które wykonują określone zadanie.
- Prosty algorytm to
  - łatwiejsza implementacja, czytelniejszy kod
  - łatwość testowania
  - łatwość pisania dokumentacji,….
- Jeśli program ma działać wielokrotnie, jego wydajność i wykorzystywany algorytm stają się bardzo ważne. W ogólności, efektywność wiąże się z czasem potrzebnym programowi na wykonanie danego zadania. Istnieją również inne zasoby, które należy niekiedy oszczędnie wykorzystywać w pisanych programach:
  - ilość przestrzeni pamięciowej wykorzystywanej przez zmienne
  - generowane przez program obciążenie sieci komputerowej
  - ilość danych odczytywanych i zapisywanych na dysku

Wybór algorytmu

- **Zrozumiałość** i **efektywność** to są często sprzeczne cele. Typowa jest sytuacja w której programy efektywne dla dużej ilości danych są trudniejsze do napisania/zrozumienia.
- Np. sortowanie przez wybieranie (łatwy, nieefektywny dla dużej ilości danych) i sortowanie przez scalanie (trudniejszy, dużo efektywniejszy).
- Zrozumiałość to pojecie względne, natomiast **efektywność** można obiektywnie zmierzyć.
- **Metodyka:** testy wzorcowe, analiza złożoności obliczeń

Złożoność obliczeniowa i asymptotyczna

- Złożoność obliczeniowa:
  - Jest to miara służąca do porównywania efektywności algorytmów.
  - Mamy dwa kryteria efektywności:
    - czas,
    - pamięć.
- Do oceny efektywności stosujemy jednostki logiczne wyrażające związek miedzy **rozmiarem danych *N*** (wielkość pliku lub tablicy) a **ilością czasu *T*** potrzebną na ich przetworzenie.
- Funkcja wyrażająca zależność miedzy ***N*** a ***T*** jest zwykle bardzo skomplikowana, a jej obliczenie ma znaczenie jedynie w odniesieniu do dużych rozmiarów danych
- Przybliżona miara efektywności to tzw. złożoność asymptotyczna.

Notacja „wielkie *O* ” (wprowadzona przez P. Bachmanna w 1894r)

Definicja:

f(n) jest *O*(g(n)), jeśli istnieją liczby dodatnie c i N takie że:

$f(n) < c \cdot g(n)$ dla wszystkich $n \geq N$.

Przykład:

$f(n) = n^2 + 100n + \log_{10} n + 1000$

możemy przybliżyć jako:

$f(n) \approx n^2 + 100n + O(\log_{10} n)$

albo jako:

$f(n) \approx O(n^2)$

Notacja **„wielkie *O*”** ma kilka pozytywnych własności które możemy wykorzystać przy szacowaniu efektywności algorytmów.

Notacja Ω i Θ

- Notacja „wielkie *O*" odnosi się do górnych ograniczeń funkcji.
- Istnieje symetryczna definicja dotycząca dolnych ograniczeń:

Definicja
f(n) jest Ω(g(n)), jeśli istnieją liczby dodatnie c i N takie że, f(n) $\geq$ c g(n) dla wszystkich n $\geq$ N.

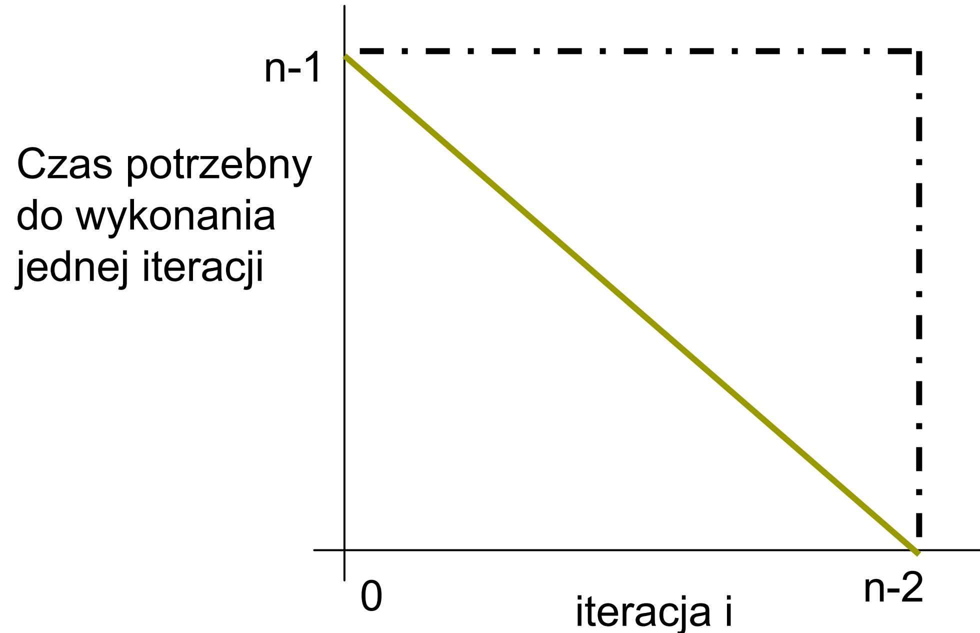
Równoważność
f(n) jest Ω(g(n)) wtedy i tylko wtedy, gdy g(n) jest *O*(f(n))

Definicja
f(n) jest Θ(g(n)), jeśli istnieją takie liczby dodatnie c_1, c_2 i N takie że, $c_1 g(n) \leq f(n) \leq c_2 g(n)$ dla wszystkich n $\geq$ N.

Przykłady rzędów złożoności

- Algorytmy można klasyfikować ze względu na złożoność czasową lub pamięciową. W związku z tym wyróżniamy wiele klas algorytmów.
  - **Algorytm stały:** czas wykonania pozostaje taki sam niezależnie od ilości przetwarzanych elementów.
  - **Algorytm kwadratowy:** czas wykonania wynosi $\boldsymbol{O}(n^2)$.
  - **Algorytm logarytmiczny:** czas wykonania wynosi $\boldsymbol{O}(\log n)$.
  - itd …
- **Analiza złożoności algorytmów jest niezmiernie istotna** i nie można jej lekceważyć argumentując potencjalną szybkością obliczeń komputera. Nie sposób jej przecenić szczególnie w kontekście doboru struktury danych.

Proste lub precyzyjne ograniczenie



Górne ograniczenie czasu niezbędnego do wykonania wszystkich iteracji wynosi:

$$O\left(\sum_{n=0}^{n-2} (n-i-1) \right) = O(n(n-1)/2)$$

Iteracja

- Programy i algorytmy wykorzystują iteracje do wielokrotnego wykonywania określonych zadań bez konieczności definiowania ogromnej liczby pojedynczych kroków, np. w przypadku zadania
  - *wykonaj dany krok 1000 razy.*
- Najprostszym sposobem wielokrotnego wykonania sekwencji operacji jest wykorzystanie konstrukcji iteracyjnej, jaką jest instrukcja **for** lub **while** w języku C.

Rekurencja

- Zagadnieniem blisko związanym z powtórzeniami (iteracją) jest rekurencja (ang. recursion) – technika, w której definiuje się pewne pojęcie bezpośrednio lub pośrednio na podstawie tego samego pojęcia.
- Np. można zdefiniować pojęcie lista stwierdzeniem:
  - *lista jest albo pusta, albo jest sklejeniem elementu i listy*
- Definicje rekurencyjne są szeroko stosowane do specyfikacji gramatyk języków programowania (patrz następne wykłady).

Definicja rekurencyjna

- Definicja rekurencyjna składa się z dwóch części.
  - W pierwszej, zwanej **podstawową** lub **warunkiem początkowym**, są wyliczone elementy podstawowe, stanowiące części składowe wszystkich pozostałych elementów zbioru.
  - W drugiej części, zwanej **krokiem indukcyjnym**, są podane reguły umożliwiające konstruowanie nowych obiektów z elementów podstawowych lub obiektów zbudowanych wcześniej.

- Reguły te można stosować wielokrotnie, tworząc nowe obiekty.

Metody rozwiązywania rekurencji

- **Metoda podstawiania:**
  - zgadujemy oszacowanie, a następnie dowodzimy przez indukcję jego poprawność.
- **Metoda iteracyjna:**
  - przekształcamy rekurencję na sumę, korzystamy z technik ograniczania sum.
- **Metoda uniwersalna::**
  - stosujemy oszacowanie na rekurencję mające postać **$T(n) = a\,T(n/b) + f(n)$**, gdzie **$a \geq 1$**, **$b > 1$**, a **$f(n)$** jest daną funkcją.

Twierdzenie o rekurencji uniwersalnej

- Niech **$a \geq 1$** i **$b > 1$** będą stałymi, niech **f(n)** będzie pewną funkcją i niech **T(n)** będzie zdefiniowane dla nieujemnych liczb całkowitych przez rekurencje

  $$T(n) = a\, T(n/b) + f(n)$$

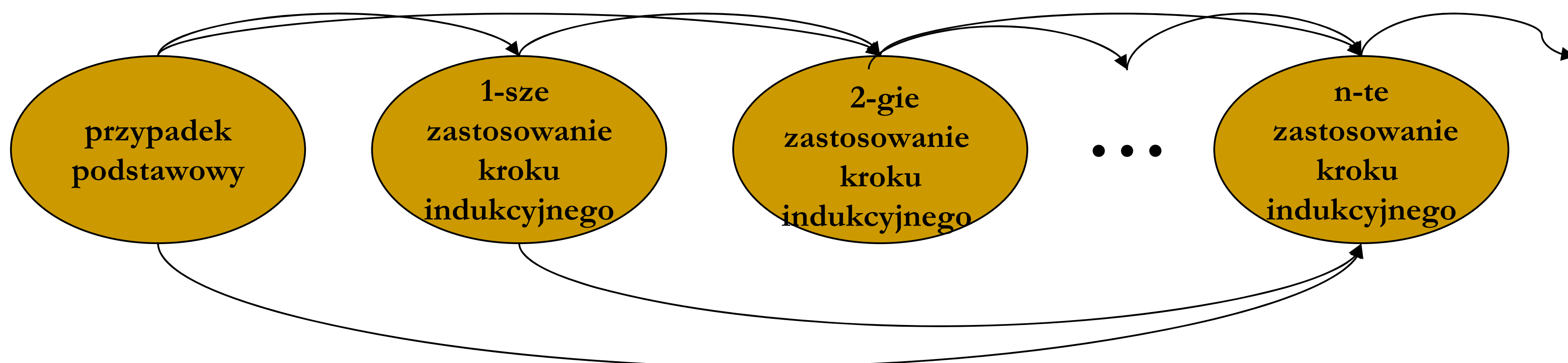
  gdzie **(n/b)** oznacza najbliższą liczbę całkowitą do wartości dokładnej **n/b**.
- Wtedy funkcja **T(n)** może być ograniczona asymptotycznie w następujący sposób:
  - Jeśli $f(n) = O(n^{\log_b a-\varepsilon})$ dla pewnej stałej $\varepsilon>0$, to $T(n) = \Theta(n^{\log_b a})$.
  - Jeśli $f(n) = \Theta(n^{\log_b a})$ to $T(n) = \Theta(n^{\log_b a} \log n)$.
  - Jeśli $f(n) = n^{\log_b a+\varepsilon})$ dla pewnej stałej $\varepsilon>0$ i jeśli $af(n/b) \leq cf(n)$ dla pewnej stałej $c<1$ i wszystkich dostatecznie dużych **n**, to $T(n) = Q(f(n))$.

Indukcja

- Niech **S(n)** będzie dowolnym twierdzeniem dotyczącym liczby całkowitej **n**. W najprostszej formie dowodu indukcyjnego (**indukcja częściowa**) twierdzenia **S(n)** dowodzi się dwóch faktów:
  - **Przypadku podstawowego:** za który często przyjmuje się twierdzenie **S(0)**. Przypadkiem podstawowym może jednak być równie dobrze **S(k)** dla dowolnej liczby całkowitej **k**. Dowodzi się wówczas prawdziwości twierdzenia **S(n)** dla $n \geq k$.
  - **Kroku indukcyjnego:** gdzie dowodzi się, że dla wszystkich $n \geq 0$ (lub wszystkich $n \geq k$), prawdziwość **S(n)** implikuje prawdziwość **S(n+1)**.

Definicje indukcyjne (raz jeszcze)

- W definicji indukcyjnej (zwanej tez rekursywną) definiuje się jedną lub więcej klas reprezentujących ściśle powiązane ze sobą obiekty (lub fakty) na bazie tych samych obiektów.
- **Definicja rekurencyjna powinna zawierać:**
  - jedną lub więcej reguł podstawowych, z których niektóre definiują pewne obiekty proste,
  - jedną lub więcej reguł indukcyjnych, za pomocą których definiuje się większe obiekty na bazie mniejszych z tego samego zbioru.



Modele danych

- **Modele danych są to abstrakcje wykorzystywane do opisywania problemów.**
- Każdą koncepcje matematyczną można opisać za pomocą modelu danych.
  W informatyce wyróżniamy zazwyczaj dwa aspekty:
  - Wartości które nasz obiekt może przyjmować.
    Przykładowo wiele modeli danych zawiera obiekty przechowujące wartości całkowitoliczbowe. Ten aspekt modelu jest **statyczny**; określa bowiem wyłącznie grupę wartości przyjmowanych przez obiekt.
  - Operacje na danych.
    Przykładowo stosujemy zazwyczaj operacje dodawania liczb całkowitych. Ten aspekt modelu nazywamy **dynamicznym**; określa bowiem metody wykorzystywane do operowania wartościami oraz tworzenia nowych wartości.
- **Badanie modeli danych, ich właściwości oraz sposobów właściwego ich wykorzystania stanowi jedno z podstawowych zagadnień informatyki.**

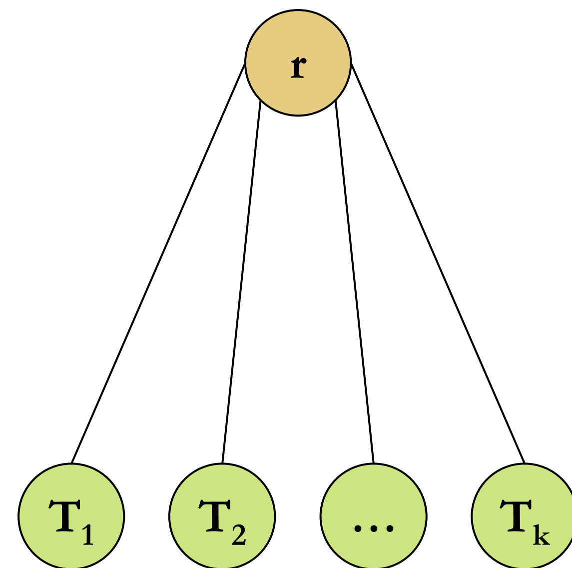
Model danych oparty na drzewach

- Istnieje wiele sytuacji w których przetwarzane informacje mają strukturę hierarchiczną lub zagnieżdżoną, jak drzewo genealogiczne lub diagram struktury organizacyjnej.

- Abstrakcje modelujące strukturę hierarchiczną nazywamy **drzewem** – jest to jeden z najbardziej podstawowych modeli danych w informatyce.

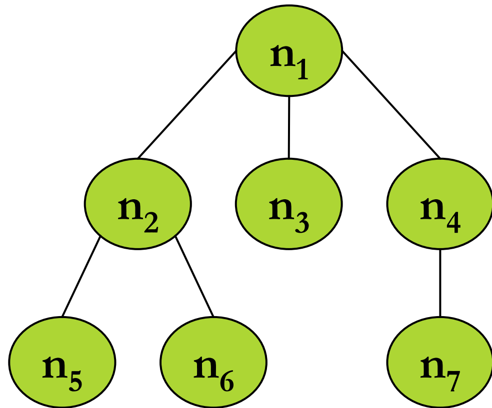
Rekurencyjna definicja drzew

- **Podstawa**: Pojedynczy węzeł **n** jest drzewem. Mówimy że **n** jest korzeniem drzewa złożonego z jednego węzła.
- **Indukcja:** Niech **r** będzie nowym węzłem oraz niech $T_1, T_2, \ldots, T_k$ będą drzewami zawierającymi odpowiednio korzenie $c_1, c_2, \ldots, c_k$. Załóżmy że żaden węzeł nie występuje więcej niż raz w drzewach $T_1, T_2, \ldots, T_k$, oraz że **r**, będący „nowym" węzłem, nie występuje w żadnym z tych drzew. Nowe drzewo **T** tworzymy z węzła **r** i drzew $T_1, T_2, \ldots, T_k$ w następujący sposób:
  - węzeł **r** staje się korzeniem drzewa **T**;
  - dodajemy **k** krawędzi, po jednej łącząc **r** z każdym z węzłów $c_1, c_2, \ldots, c_k$, otrzymując w ten sposób strukturę w której każdy z tych węzłów jest dzieckiem korzenia **r**. Inny sposób interpretacji tego kroku to uczynienie z węzła **r** rodzica każdego z korzeni drzew $T_1, T_2, \ldots, T_k$.



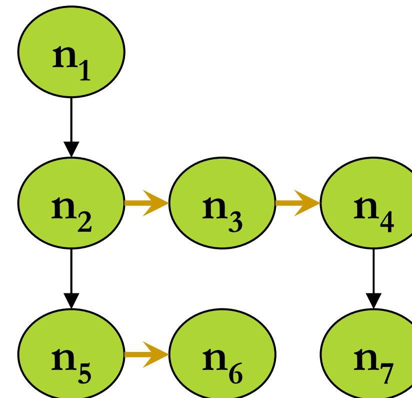
Reprezentacje drzewa

Drzewo złożone z 7 węzłów



info – etykieta
leftmostChild – informacja o węźle
rightSibling – część listy jednokierunkowej dzieci rodzica tego węzła

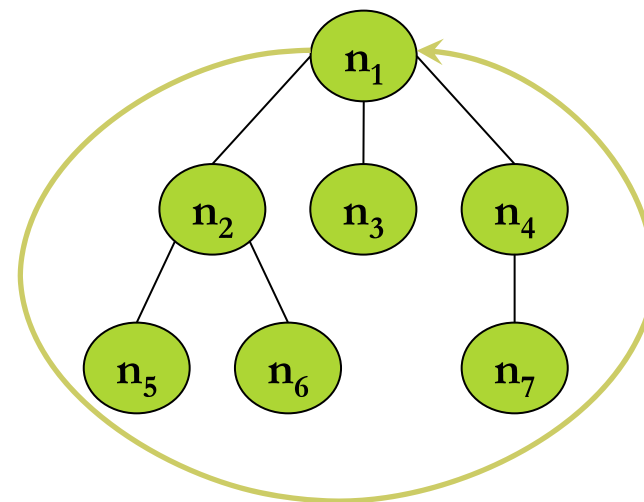
Reprezentacja skrajnie lewy potomek-prawy element siostrzany




```
typedef struct NODE *pNODE;
struct NODE{
        int info;
        pNODE leftmostChild, rightSibling;
};
```

Rekurencja w drzewach

- Użyteczność drzew wynika z liczby możliwych operacji rekurencyjnych, które możemy na nich wykonać w naturalny i jasny sposób (chcemy drzewa przeglądać).
- Prosta rekurencja zwraca etykiety węzłów w **porządku wzdłużnym** (ang. **pre-order listing**), czyli: korzeń, lewe poddrzewo, prawe poddrzewo.
- Inną powszechnie stosowaną metodą do przeglądania węzłów drzewa jest tzw. **przeszukiwanie wsteczne** (ang. **post-order listing**), czyli lewe poddrzewo, prawe poddrzewo, korzeń.



Przeszukiwanie wszerz

Drzewa przeszukiwania binarnego

- Jest to zaetykietowane drzewo binarne dla którego etykiety należą do zbioru w którym możliwe jest zdefiniowanie relacji mniejszości.
- Dla każdego węzła **x** spełnione są następujące własność:
  - wszystkie węzły w lewym poddrzewie maja etykiety mniejsze od etykiety węzła **x**
  - wszystkie w prawym poddrzewie maja etykiety większe od etykiety węzła **x**.
- **Wyszukiwanie elementu:**
  - **Podstawa:**
    - Jeśli drzewo **T** jest puste, to na pewno nie zawiera elementu **x**.
    - Jeśli **T** nie jest puste i szukana wartość **x** znajduje się w korzeniu, drzewo zawiera **x**.
  - **Indukcja:**
    - Jeśli **T** nie jest puste, ale nie zawiera szukanego elementu **x** w korzeniu, niech **y** będzie elementem w korzeniu drzewa **T**.
    - Jeśli $x<y$, szukamy wartości **x** tylko w lewym poddrzewie korzenia **y**.
    - Jeśli $x>y$, szukamy wartości **x** tylko w prawym poddrzewie korzenia **y**.
- **Własność drzewa przeszukiwania binarnego gwarantuje, że szukanej wartości x na pewno nie ma w poddrzewie, którego nie przeszukujemy.**

Model danych oparty na zbiorach

- Zbiór jest najbardziej podstawowym modelem danych w matematyce.
- Wszystkie pojęcia matematyczne, od drzew po liczby rzeczywiste można wyrazić za pomocą specjalnego rodzaju zbioru.
- Jest wiec naturalne że jest on również podstawowym modelem danych w informatyce.
- **Dotychczas wykorzystaliśmy to pojecie mówiąc o**
  - **zdarzeniach w przestrzeni probabilistycznej,**
  - **słowniku,** który także jest rodzajem zbioru na którym możemy wykonywać tylko określone operacje: wstawiania, usuwania i wyszukiwania

Podstawowe definicje

- W matematyce pojęcie **zbioru** nie jest zdefiniowane wprost.
- Zamiast tego, podobnie jak punkt czy prosta w geometrii, **zbiór jest zdefiniowany za pomocą swoich własności**.
- W szczególności istnieje pojecie **przynależności,** które jest sensowne tylko i wyłącznie dla zbiorów. Jeśli **S** jest **zbiorem** oraz **x** jest **czymkolwiek**, zawsze możemy odpowiedzieć na pytanie

  „Czy x należy do zbioru S?"
- Zbiór **S** składa się więc z wszystkich takich elementów **x**, dla których **x** należy do zbioru **S**.

Podstawowe definicje

- **Notacja:**
  - Wyrażenie $\mathbf{x} \in \mathbf{S}$ oznacza, że element **x** należy do zbioru **S**.
  - Jeśli elementy $\mathbf{x_1, x_2, \ldots, x_n}$ należą do zbioru **S**, i żadne inne, to możemy zapisać:
    $\mathbf{S = \{x_1, x_2, \ldots, x_n\}}$
  - Każdy **x** musi być inny, nie możemy umieścić w zbiorze żadnego elementu dwa lub więcej razy. Kolejność ułożenia elementów w zbiorze jest jednak całkowicie dowolna.
  - Zbiór pusty, oznaczamy symbolem $\varnothing$, jest zbiorem do którego nie należą żadne elementy. Oznacza to że $\mathbf{x} \in \varnothing$ jest zawsze fałszywe.

Podstawowe definicje

- **Definicja za pomocą abstrakcji:**
  - Wyliczenie elementów należących do zbioru nie jest jedynym sposobem jego definiowania. Bardzo wygodne jest wyjście od definicji że istnieje zbiór **S** oraz że jego elementy spełniają własność **P**, tzn. **{x : x ∈ S oraz P(x) }** czyli „zbiór takich elementów **x** należących do zbioru **S**, które spełniają własność **P**.
- **Równość zbiorów:**
  - Dwa zbiory są **równe** (czyli są tym samym zbiorem), jeśli zawierają **te same elementy.**
- **Zbiory nieskończone:**
  - Zwykle wygodne jest przyjęcie założenia że zbiory są skończone. Czyli że istnieje pewna skończona liczba **N** taka, że nasz zbiór zawiera dokładnie **N** elementów. Istnieją jednak również zbiory nieskończone np. liczb naturalnych, całkowitych, rzeczywistych, itd.

43

Operacje na zbiorach

- Operacje często wykonywane na zbiorach:
  - **Suma:** dwóch zbiorów **S** i **T**, zapisywana **S ∪ T**, czyli zbiór zawierający elementy należące do zbioru **S** lub do zbioru **T**.
  - **Przecięcie (iloczyn):** dwóch zbiorów **S** i **T**, zapisywana **S ∩ T**, czyli zbiór zawierający należące elementy do zbioru **S** i do zbioru **T**.
  - **Różnica:** dwóch zbiorów **S** i **T**, zapisywana **S \ T**, czyli zbiór zawierający tylko te elementy należące do zbioru **S**, które nie należą do zbioru **T**.
- Jeżeli **S** i **T** są zdarzeniami w przestrzeni probabilistycznej,
  - suma, przecięcie i różnica mają naturalne znaczenie,
  - **S ∪ T** jest zdarzeniem polegającym na zajściu zdarzenia **S** lub **T**,
  - **S ∩ T** jest zdarzeniem polegającym na zajściu zdarzenia **S** i **T**,
  - **S \ T** jest zdarzeniem polegającym na zajściu zdarzenia **S** ale nie **T**,
  - Jeśli **S** jest zbiorem obejmującym całą przestrzeń probabilistyczna, **S \ T** jest dopełnieniem zbioru T.

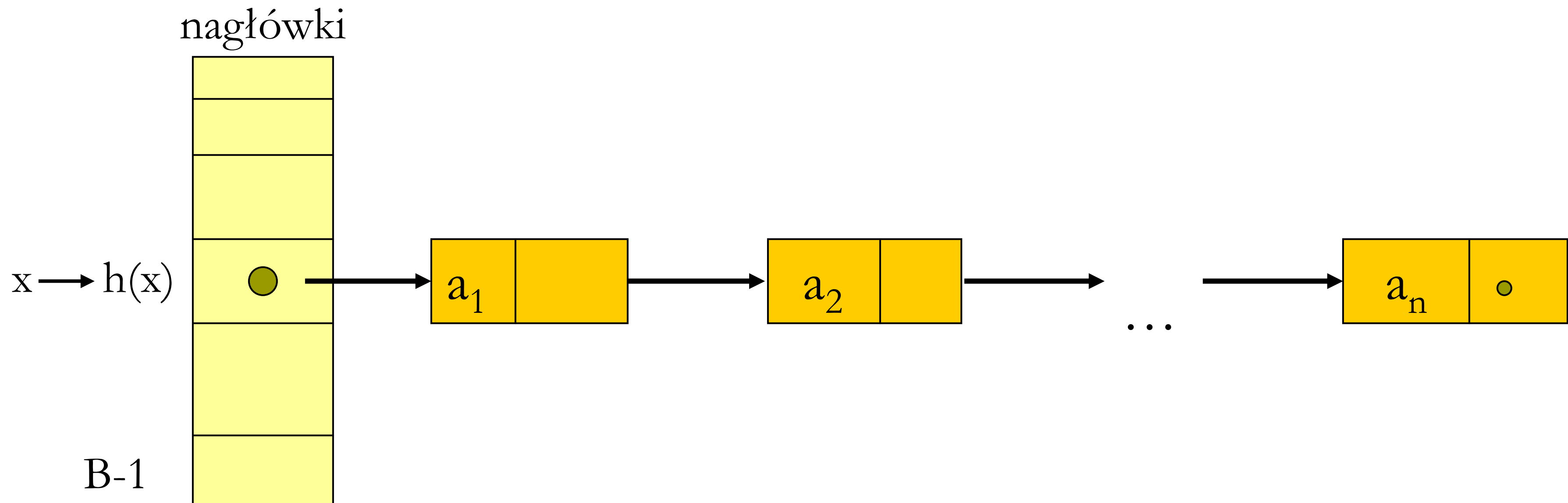
Implementacja zbiorów oparta na wektorze własnym

- Definiujemy **uniwersalny zbiór U** w którym zawierają się wszystkie zbiory na których będziemy przeprowadzać operacje. Np. talia kart (zbiór 52 kart) jest uniwersalny dla różnych możliwych zbiorów kart.
- **Porządkujemy elementy zbioru U** w taki sposób, by każdy element tego zbioru można było związać z **unikatową „pozycją ”,** będącą liczbą całkowitą od **0** do **n-1** (gdzie **n** jest liczba elementów w zbiorze uniwersalnym). Liczba elementów w zbiorze **S** jest **m**.
- Wówczas, **zbiór S** zawierający się w zbiorze **U**, możemy **reprezentować za pomocą wektora własnego** złożonego z zer i jedynek – dla każdego elementu **x** należącego do zbioru **U**, jeśli **x** należy także do zbioru **S**, odpowiadająca temu elementowi pozycja zawiera wartość **1**; jeśli **x** nie należy do **S**, na odpowiedniej pozycji mamy wartość **0**.

Struktura danych tablicy mieszającej

- Reprezentacja słownika oparta o wektor własny, jeśli tylko możliwa, umożliwiłaby bezpośredni dostęp do miejsca w którym element jest reprezentowany.
- Nie możemy jednak wykorzystywać zbyt dużych zbiorów uniwersalnych ze względu na pamięć i czas inicjalizacji.
- Np. słownik dla słów złożonych z co najwyżej 10 liter.
  Ile możliwych kombinacji: $\mathbf{26^{10}+26^{9}+ \ldots + 26 = 10^{14}}$ możliwych słów.
  Faktyczny słownik: to tylko około $\mathbf{10^{6}}$.
  Co robimy?
- Grupujemy, każda grupa to jedna komórka z „nagłówkiem” + lista jednokierunkowa z elementami należącymi do grupy.
- **Taka strukturę nazywamy tablicą mieszająca (ang. hash table)**

Struktura danych tablicy mieszającej



- Istnieje **funkcja mieszająca** (ang. **hash function**), która jako argument pobiera element **x** i zwraca liczbę całkowitą z przedziału **0 do B-1**, gdzie **B** jest liczbą komórek w tablicy mieszającej.
- Wartością zwracaną przez **h(x)** jest komórka, w której umieszczamy element **x**.
- Ważne aby funkcja **h(x)** „mieszała”, tzn. aby komórki zawierały tę samą przybliżoną liczbę elementów.

Implementacja zbiorów

- **Listy jednokierunkowe, wektory własne** oraz **tablice mieszające** to trzy najprostsze sposoby reprezentowania zbiorów w języku programowania.
- **Listy jednokierunkowe** oferują największą elastyczność w przypadku większości operacji na zbiorach, nie zawsze są jednak rozwiązaniem najbardziej efektywnym.
- **Wektory własne** są najszybszym rozwiązaniem dla pewnych operacji, mogą jednak być wykorzystywane tylko w sytuacjach, gdy zbiór uniwersalny jest mały.
- Często złotym środkiem są **tablice mieszające**, które zapewniają zarówno oszczędne wykorzystanie pamięci, jak i satysfakcjonujący czas wykonania operacji.

Relacyjny model danych

- Relacyjny model danych wykorzystuje pojęcie relacji (ang. **relation**) które jest bardzo mocno związane z przedstawioną wcześniej definicją z teorii zbiorów, jednak różni się w kilku szczegółach:
  - W relacyjnym modelu danych informacja jest przechowywana w tabelach.
  - Kolumny tabeli maja nadane konkretne nazwy i są atrybutami relacji.
  - Każdy wiersz w tabeli jest nazywany krotką i reprezentuje jeden podstawowy fakt.
  - Pojęcie relacji odwołuje się do każdej krotki.

Atrybuty relacji: Zajęcia, StudentID, Ocena

Krotki to:
(CS101, 12345, 5.0)
(CS101, 67890, 4.0)
…

zajęcia	student ID	ocena
CS101	12345	5.0
CS101	67890	4.0
EE200	12345	3.0
EE200	22222	4.5
CS101	33333	2.0
PH100	67890	3.5

Reprezentowanie relacji

- Podobnie jak w przypadku zbiorów istnieje wiele różnych sposobów reprezentowania relacji za pomocą struktur danych.
- **Tabela** postrzegana jako zbiór wierszy **powinna być zbiorem struktur** zawierających pola odpowiadające nazwom kolumn.

```
struct ZSO {
        char Zajecia[5];
        int StudentID;
        char Ocena[3]; }
```

zajęcia	student ID	ocena
CS101	12345	5.0
CS101	67890	4.0
EE200	12345	3.0
EE200	22222	4.5
CS101	33333	2.0
PH100	67890	3.5

- Sama **tabela** może być reprezentowana za pomocą:
  - **tablicy struktur tego typu**
  - **listy jednokierunkowej złożonej z takich struktur.**
- Możemy identyfikować jeden lub więcej atrybutów jako „dziedzinę” relacji i traktować pozostałe atrybuty jako przeciwdziedzinę.

Schemat bazy danych

Zajęcia	Dzień	Godzina
CS101	Pn	9.00
CS101	S	9.00
EE200	Pt	8.30
EE200	W	13.00
CS101	Pt	9.00
PH100	C	8.15

Zajęcia	Wymagania
CS101	CS100
EE200	EE005
EE200	CS100
CS120	CS101
CS121	CS120
CS205	CS101
CS206	CS121
CS206	CS205

Zajęcia	Student ID	Ocena
CS101	12345	5.0
CS101	67890	4.0
EE200	12345	3.0
EE200	22222	4.5
CS101	33333	2.0
PH100	67890	3.5

Zajęcia	Klasa
CS101	Aula
EE200	Hala
PH100	Laborat

Projektowanie

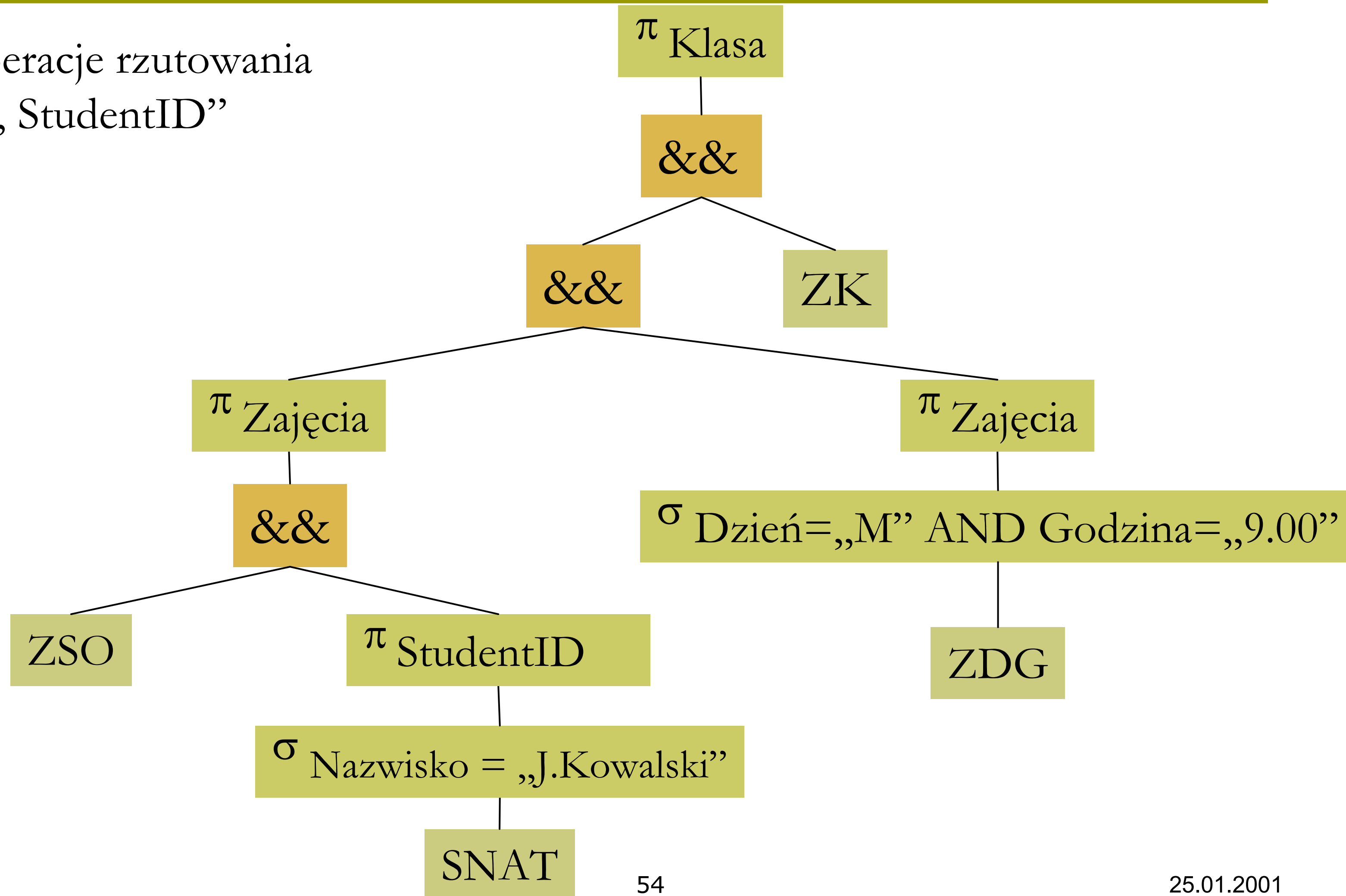
- **Projektowanie I : wybór schematu baz danych**
  - rozdzielamy informacje budując kilka relacji (krotek) zamiast umieszczać je w jednej dużej krotce,
  - nie należy rozdzielać atrybutów reprezentujących powiązane ze sobą informacje.
- **Projektowanie II : wybór klucza**
  - jeden z ważniejszy aspektów projektowania bazy danych,
  - nie istnieje „jedyna” właściwa metoda wybierania klucza.
- **Projektowanie III: wybór indeksu głównego**
  - ma zdecydowany wpływ na szybkość z jaka możemy wykonywać „typowe” zadanie.
- **Projektowanie IV: kiedy tworzyć indeks drugorzędny?**
  - utworzenie ułatwia wykonywanie operacji wyszukiwania krotki dla danej wartości jednej lub więcej składowych,
  - każdy indeks drugorzędny wymaga dodatkowego czasu wstawiania i usuwania informacji z relacji.

Algebra relacyjna

- Algebra relacyjna jest wysoko poziomową notacją definiowania operacji zapytań dotyczących jednej lub wielu relacji.
  Głównymi operacjami tej algebry są: **suma, przecięcie, różnica, selekcja, rzutowanie i złączenie.**
  Jest silną notacją wyrażania zapytań bez podawania szczegółów dotyczących planowanych operacji na otrzymanych danych.
- Istnieje wiele sposobów efektywnego implementowania operacji złączenia.
- Optymalizacja wyrażeń algebry relacyjnej może w znaczący sposób skrócić czas wyznaczania ich wartości, jest więc istotnym elementem wszystkich języków opartych na algebrze relacyjnej wykorzystywanych w praktyce do wyrażania zapytań.
- Istnieje wiele sposobów skracania czasu obliczania danego wyrażenia. Najlepsze efekty przynosi przenoszenie operacji selekcji w dół drzewa wyrażenia.

„Gdzie przebywa J. Kowalski w poniedziałek o 9-tej rano?”

Usuń operacje rzutowania „Zajęcia, StudentID”

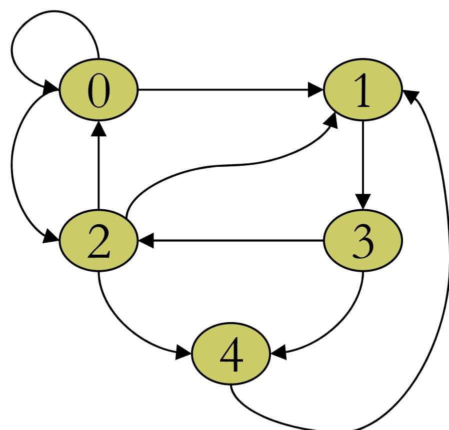


Graf

- Graf to jest relacja binarna.
- Dla grafów mamy ogromne możliwości wizualizacji jako **zbiór punktów** (zwanych **wierzchołkami**) **połączonych liniami lub strzałkami** (nazwanych **krawędziami**). Pod tym względem **graf stanowi uogólnienie drzewiastego modelu danych**.
- Podobnie jak drzewa, grafy występują w różnych postaciach: grafów **skierowanych** i **nieskierowanych** lub **etykietowanych** i **niezaetykietowanych**.
- Grafy są przydatne do analizy szerokiego zakresu problemów: obliczanie odległości, znajdowanie cykliczności w relacjach, reprezentacji struktury programów, reprezentacji relacji binarnych, reprezentacji automatów i układów elektronicznych.

Podstawowe pojęcia

- **Graf skierowany (**ang. *directed graph***)**
  Składa się z następujących elementów:
  - Zbioru **V** wierzchołków (ang. *nodes, vertices*)
  - Relacji binarnej **E** na zbiorze **V**. Relacje **E** nazywa się zbiorem krawędzi (ang. *edges*) grafu skierowanego. Krawędzie stanowią zatem pary wierzchołków **(u,v)**.



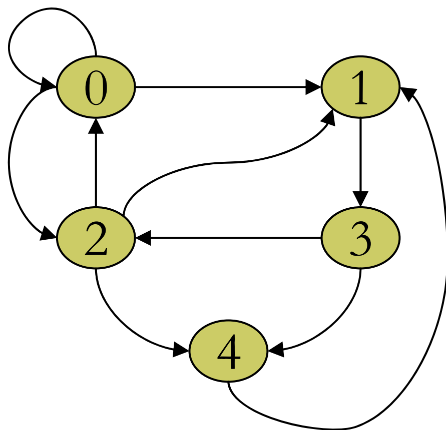
V = {0,1,2,3,4}
E = { (0,0), (0,1), (0,2), (1,3), (2,0), (2,1), (2,4), (3,2), (3,4), (4,1) }

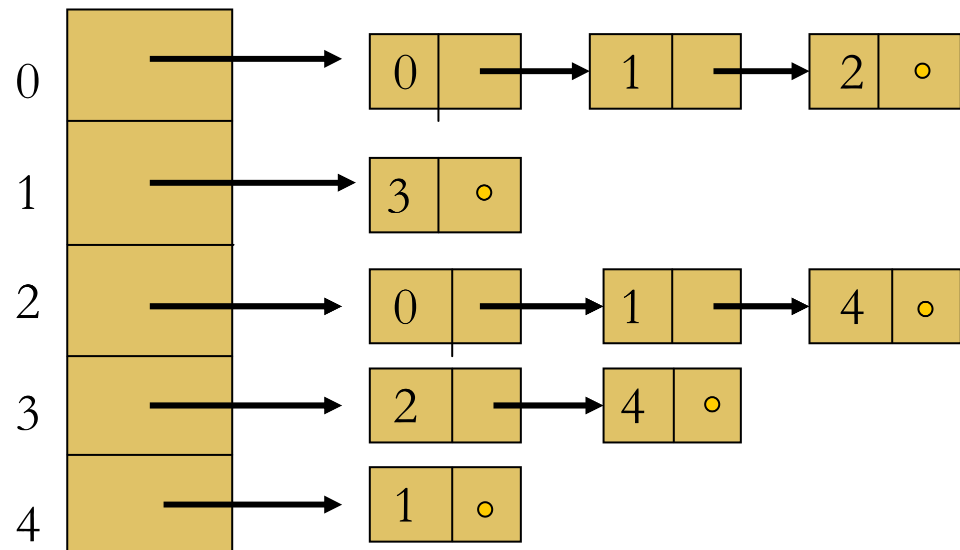
Sposoby implementacji grafów

- Istnieją **dwie standardowe metody reprezentacji grafów**.
  - Pierwsza z nich, **listy sąsiedztwa** (ang. *adjacency lists*), jest, ogólnie rzecz biorąc, podobna do implementacji relacji binarnych.
  - Druga, **macierze sąsiedztwa** (ang. *adjacency matrices*), to nowy sposób reprezentowania relacji binarnych, który jest bardziej odpowiedni dla relacji, w przypadku którym liczba istniejących par stanowi znaczącą część całkowitej liczby par, jakie mogłyby teoretycznie istnieć w danej dziedzinie.
- Wierzchołki są ponumerowane kolejnymi liczbami całkowitymi **0,1,....., MAX-1** lub oznaczone za pomocą innego adekwatnego typu wyliczeniowego (używamy poniżej typu **NODE** jako synonimy typu wyliczeniowego).
- Wówczas można skorzystać z podejścia opartego na wektorze własnym.
- Element **successors[u]** zawiera wskaźnik do listy jednokierunkowej wszystkich bezpośrednich następników wierzchołka **u**. Następniki mogą występować w dowolnej kolejności na liście jednokierunkowej.

Reprezentacja grafu za pomocą list sąsiedztwa

- Listy sąsiedztwa zostały posortowane wg. kolejności, ale następniki mogą występować w **dowolnej kolejności** na odpowiedniej liście sąsiedztwa.



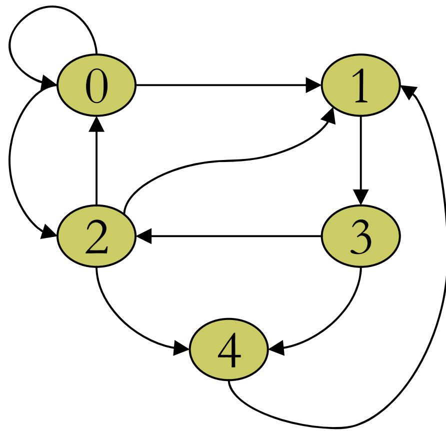


Reprezentacja grafu za pomocą macierzy sąsiedztwa

- Tworzymy dwuwymiarową tablicę;

  BOOLEAN vertices[MAX][MAX];

  w której element **vertices[u][v]** ma wartość **TRUE** wówczas, gdy istnieje krawędź **(u, v)**, zaś **FALSE**, w przeciwnym przypadku.



	0	1	2	3	4
0	1	1	1	0	0
1	0	0	0	1	0
2	1	1	0	0	1
3	0	0	1	0	1
4	0	1	0	0	0

Porównanie macierzy sąsiedztwa z listami sąsiedztwa.

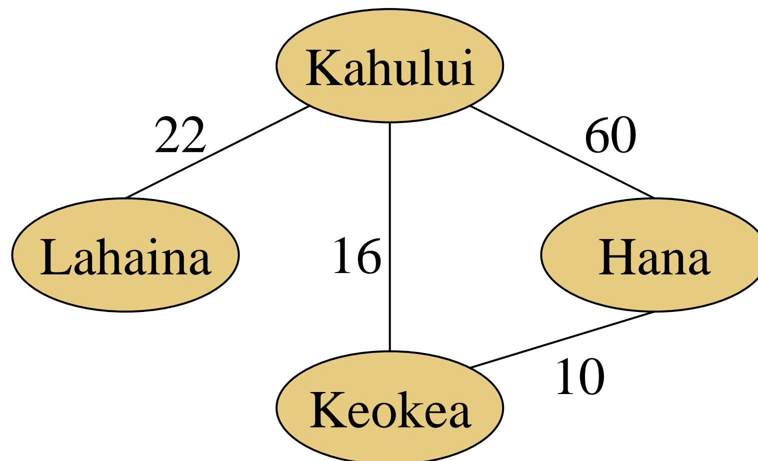
- **Macierze sąsiedztwa** są preferowanym sposobem reprezentacji grafów wówczas, gdy **grafy są gęste** (ang. *dense*), to znaczy, kiedy liczba krawędzi jest bliska maksymalnej możliwej ich liczby.
- Dla grafu skierowanego o **n** wierzchołkach maksymalna liczba krawędzi wynosi $\mathbf{n^2}$.
- Jeśli **graf jest rzadki** (ang. *sparse*) to reprezentacja oparta na listach sąsiedztwa może pozwolić zaoszczędzić pamięć.
- Istotne różnice miedzy przedstawionymi reprezentacjami grafów są widoczne już przy wykonywaniu prostych operacji.

Preferowany sposób reprezentacji:

OPERACJA	GRAF GESTY	GRAF RZADKI
Wyszukiwanie krawędzi	Macierz sąsiedztwa	Obie
Znajdowanie następników	Obie	Lista sąsiedztwa
Znajdowanie poprzedników	Macierz sąsiedztwa	Obie

Spójna składowa grafu nieskierowanego

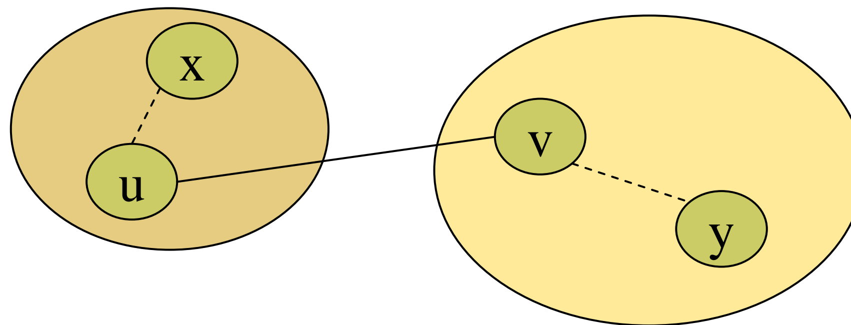
- Każdy graf nieskierowany można podzielić na jedna lub większą liczbę **spójnych składowych** (ang. *connected components*).
- Każda spójna składowa to taki zbiór wierzchołków, że dla każdych dwóch z tych wierzchołków istnieje łącząca je ścieżka. Jeżeli graf składa się z jednej spójnej składowej to mówimy że jest **spójny** (ang. *connected*).



To jest graf spójny

Algorytm wyznaczania spójnych składowych

- Chcemy określić spójne składowe grafu G. Przeprowadzamy rozumowanie indukcyjne.
- **Podstawa:**
  - Graf $\mathbf{G_o}$ zawiera jedynie wierzchołki grafu **G** i żadnej jego krawędzi. Każdy wierzchołek stanowi odrębną spójną składową .
- **Indukcja:**
  - Zakładamy, że znamy już spójne składowe grafu G_i po rozpatrzeniu pierwszych **i** krawędzi, a obecnie rozpatrujemy **(i+1)** krawędź **{u, v}**.
    - jeżeli wierzchołki **u, v** należą do jednej spójnej składowej to nic się nie zmienia
    - jeżeli do dwóch różnych, to łączymy te dwie spójne składowe w jedną.

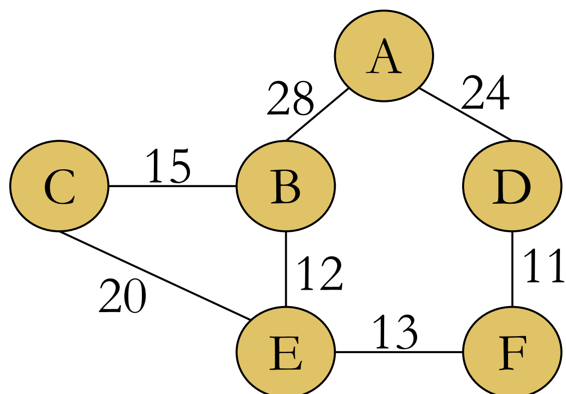


Znajdowanie minimalnego drzewa rozpinającego

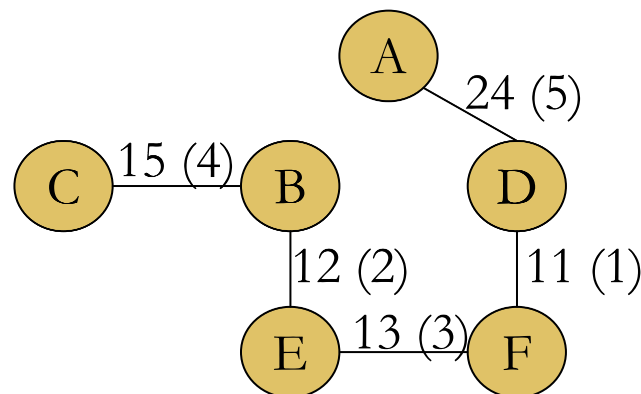
- **Algorytm Kruskala** jest dobrym przykładem **algorytmu zachłannego** (ang. *greedy algorithm*), w przypadku którego podejmowany jest szereg decyzji, z których każdą stanowi wybranie opcji najlepszej w danym momencie. Lokalnie podejmowane decyzje polegają w tym przypadku na wyborze krawędzi dodawanej do formowanego drzewa rozpinającego.
- Za każdym razem wybierana jest krawędź o najmniejsze wartości etykiety, która nie narusza definicji drzewa rozpinającego, zabraniającej utworzenia cyklu.
- Dla algorytmu Kruskala można wykazać, że jego rezultat jest optymalny globalnie, to znaczy że daje on w wyniku drzewo rozpinające o minimalnej wadze.
- Czas wykonania algorytmu jest ***O*(m log m)** gdzie **m** to jest większa z wartości liczby wierzchołków i liczby krawędzi.

Znajdowanie minimalnego drzewa rozpinającego

- Istnieje wiele algorytmów. Jeden z nich to algorytm Kruskala, który stanowi proste rozszerzenie algorytmu znajdowania spójnych składowych. Wymagane zmiany to:
  - należy rozpatrywać krawędzie w kolejności zgodnej z rosnącą wartością ich etykiet,
  - należy dołączyć krawędź do drzewa rozpinającego tylko w takim wypadku gdy jej końce należą do dwóch różnych spójnych składowych.



Graf nieskierowany



Minimalne drzewo rozpinające

(w nawiasach podano kolejność dodawanych krawędzi)

Znajdowanie minimalnego drzewa rozpinającego

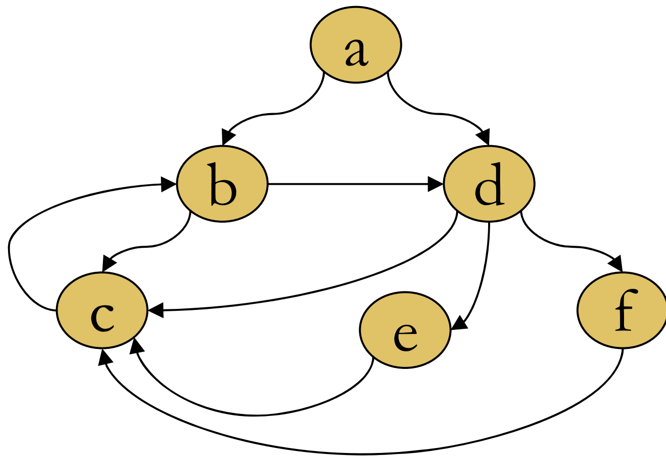
- **Algorytm Kruskala** jest dobrym przykładem **algorytmu zachłannego** (ang. *greedy algorithm*), w przypadku którego podejmowany jest szereg decyzji, z których każdą stanowi wybranie opcji najlepszej w danym momencie. Lokalnie podejmowane decyzje polegają w tym przypadku na wyborze krawędzi dodawanej do formowanego drzewa rozpinającego.
- Za każdym razem wybierana jest krawędź o najmniejsze wartości etykiety, która nie narusza definicji drzewa rozpinającego, zabraniającej utworzenia cyklu.
- Dla algorytmu Kruskala można wykazać, ze jego rezultat jest optymalny globalnie, to znaczy że daje on w wyniku drzewo rozpinające o minimalnej wadze.
- Czas wykonania algorytmu jest ***O*(m log m)** gdzie **m** to jest większa z wartości liczby wierzchołków i liczby krawędzi.

Algorytm przeszukiwania w głąb

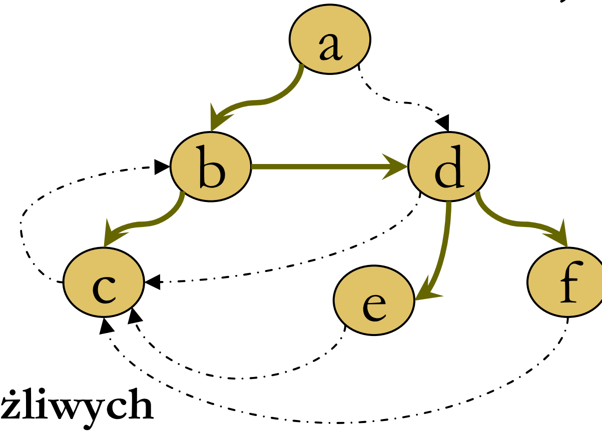
- Jest to podstawowa metoda badania grafów skierowanych.
- Bardzo podobna do stosowanych dla drzew, w których startuje się od korzenia i rekurencyjnie bada wierzchołki potomne każdego odwiedzonego wierzchołka.
- Trudność polega na tym że w grafie mogą pojawiać się cykle… Należy wobec tego znaczyć wierzchołki już odwiedzone i nie wracać więcej do takich wierzchołków.
- Z uwagi na fakt, że w celu uniknięcia dwukrotnego odwiedzenia tego samego wierzchołka jest on odpowiednio oznaczany, graf w trakcie jego badania zachowuje się podobnie do drzewa.
- W rzeczywistości można narysować drzewo, którego krawędzie rodzic-potomek będą niektórymi krawędziami przeszukiwanego grafu G.
- Takie drzewo nosi nazwę **drzewa przeszukiwania w głąb** (ang. *depth-first-search*) dla danego grafu.

Algorytm przeszukiwania w głąb

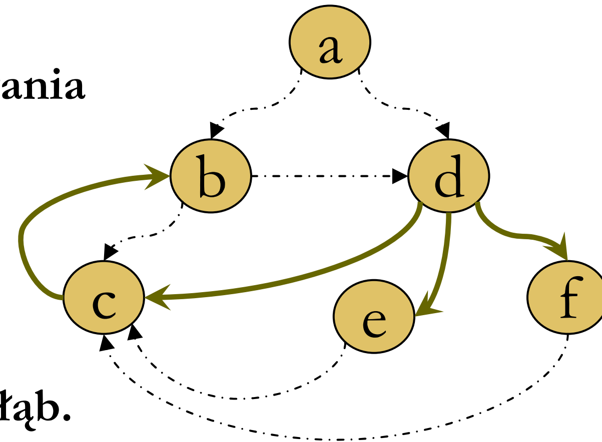
Las przeszukiwania:
dwa drzewa o korzeniach a, d



Graf skierowany



Jedno z możliwych drzew przeszukiwania



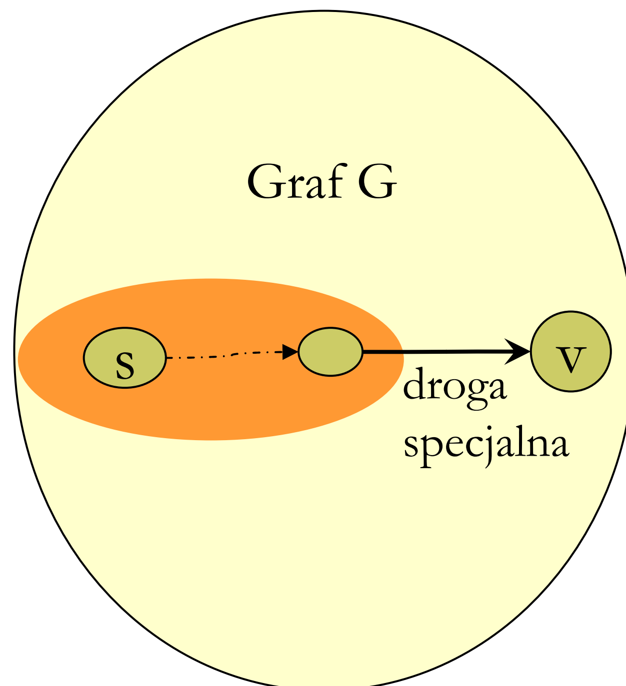
Las przeszukiwania w głąb.

Podsumowanie informacji o algorytmach grafowych

PROBLEM	ALGORYTM(Y)	CZAS WYKONANIA
Minimalne drzewo rozpinające	Algorytm Kruskala	O(m log n)
Znajdowanie cykli	Przeszukiwanie w głąb	O(m)
Uporządkowanie topologiczne	Przeszukiwanie w głąb	O(m)
Osiągalność w przypadku pojedynczego źródła	Przeszukiwanie w głąb	O(m)
Spójne składowe	Przeszukiwanie w głąb	O(m)
Najkrótsza droga dla pojedyncz. źródła	Algorytm Dijskry	O(m log n)
Najkrótsza droga dla wszystkich par	Algorytm Dijskry	O(m n log n)
	Algorytm Floyda	$O(n^3)$

Algorytm Dijkstry znajdowania najkrótszych dróg.

- Traktujemy wierzchołek **s** jako wierzchołek źródłowy. W etapie pośrednim wykonywania algorytmu w grafie **G** istnieją tzw. **wierzchołki ustalone** (ang. *settled*), tzn. takie dla których znane są odległości minimalne. W szczególności zbiór takich wierzchołków zawiera również wierzchołek **s**.
- Dla nieustalonego wierzchołka **v** należy zapamiętać długość **najkrótszej drogi specjalnej** (ang. *special path*) czyli takiej która rozpoczyna się w wierzchołku źródłowym, wiedzie przez ustalone wierzchołki, i na ostatnim etapie przechodzi z obszaru ustalonego do wierzchołka **v**.



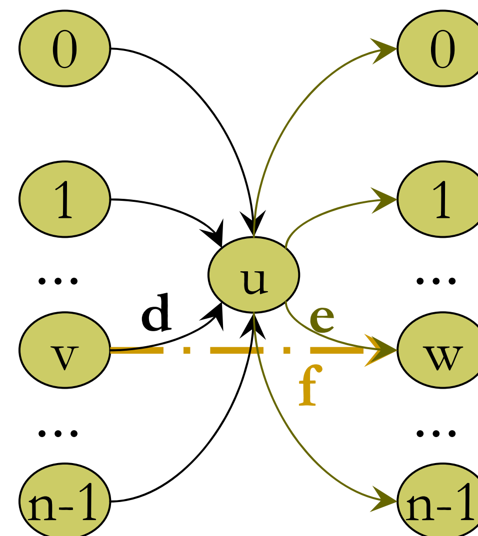
Algorytm Floyda-Warshalla znajdowania najkrótszych dróg.

- Podstawa algorytmu jest działanie polegające na rozpatrywaniu po kolei każdego wierzchołka grafu jako **elementu centralnego** (ang. *pivot*).
- Kiedy wierzchołek **u** jest elementem centralnym, staramy się wykorzystać fakt, że **u** jest wierzchołkiem pośrednim miedzy wszystkimi parami wierzchołków.
- Dla każdej pary wierzchołków, na przykład **v** i **w**, jeśli suma etykiet krawędzi **(v, u)** oraz **(u, w)** (na rysunku **d+e**) , jest mniejsza od bieżąco rozpatrywanej etykiety **f** krawędzi wiodącej od **v** do **w**, to wartość **f** jest zastępowana wartością **d+e**.

```
Node u, v, w;

for (v = 0; w < MAX; v++)
     for (w=0; w < MAX; w++)
          dist[v][w] = edge[v][w];
for (u=0; u< MAX; u++)
     for (v=0; v< MAX; v++)
          for (w=0; w<MAX; w++)
               if( dist[v][u]+dist[u][w] < dist[v][w])
               dist[v][w] = dist [v][u] + dist [u][w];
```

edge[v][w] –etykieta krawędzi, wierzchołki numerowane



Wzorce i automaty

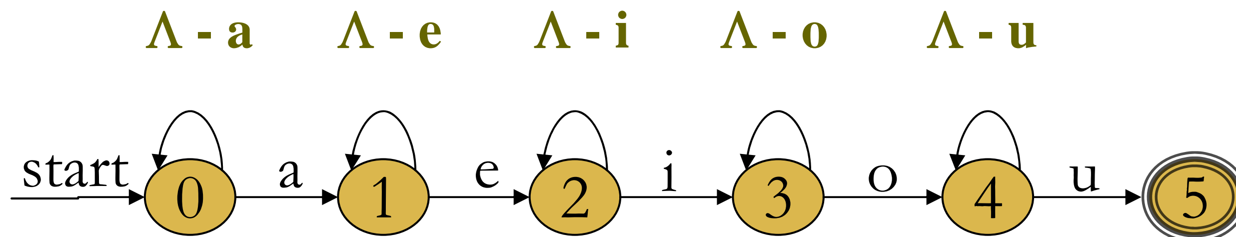
- Problematyka wzorców stanowi bardzo rozwiniętą dziedzinę wiedzy. Nosi ona nazwę teorii automatów lub teorii języków, a jej podstawowe definicje i techniki stanowią istotną część informatyki.
- Poznamy trzy równoważne opisy wzorców:
    1. oparty na teorii grafów, polegać będzie na wykorzystaniu ścieżek w grafie szczególnego rodzaju który nazwiemy automatem.
    2. o charakterze algebraicznym, wykorzystujący notacje wyrażeń regularnych.
    3. oparty o wykorzystanie definicji rekurencyjnych, nazwany gramatyką bezkontekstową.

Grafy reprezentujące maszyny stanów

- Działanie omówionego programu można reprezentować za pomocą **grafu**, którego **wierzchołki określają stany programu**.
- I odwrotnie: program można zaprojektować przez zaprojektowanie w pierwszej kolejności grafu, a następnie mechaniczne przetłumaczenie takiego grafu na program (ręcznie lub przy pomocy istniejących narzędzi programistycznych.
- Koncepcja działania automatu jest prosta. Zakłada się że **automat pobiera listę znaków zwaną ciągiem wejściowym** (ang. input sequence).
- Jego działanie rozpoczyna się w stanie początkowym, tuż przed odczytaniem pierwszego znaku wejściowego. W zależności od tego jaki jest to znak, automat wykonuje przejście – do tego samego lub innego stanu.
- **Przejścia są zdefiniowane przez graf automatu**. Następnie automat odczytuje drugi znak i wykonuje kolejne przejście i tak dalej.

Automat rozpoznający ciągi liter posiadające podciąg ''aeiou''.

- **Stany programu są reprezentowane za pomocą grafu skierowanego**, którego krawędzie etykietuje się zbiorami znaków. Krawędzie takie nazywa się **przejściami** (ang. *transition*).
- Niektóre wierzchołki są zaznaczone jako **stany końcowe** (ang. *accepting states*). Osiągnięcie takiego stanu oznacza znalezienie poszukiwanego wzorca i jego akceptację.
- Jeden z wierzchołków jest zawsze określany jako **stan początkowy** (ang. *start state*) – stan w którym następuje rozpoczęcie procesu rozpoznawania wzorca. Wierzchołek taki oznacza się przez umieszczenie prowadzącej do niego strzałki która nie pochodzi od innego wierzchołka. Graf posiadający opisywaną postać nosi nazwę **automatu skończonego** (ang. *finite automaton*) lub po prostu automatu.

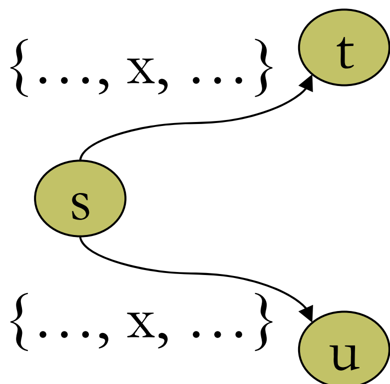


Automaty deterministyczne

- Automaty które dotychczas omówiliśmy posiadają pewną istotną własność.
- Dla każdego stanu **s** oraz dowolnego znaku wejściowego **x** istnieje co najwyżej jedno przejście ze stanu **s**, którego etykieta zawiera znak **x**.
- O tego rodzaju automatach mówimy że są **deterministyczne** (ang. *deterministic*).
- Symulacja automatu deterministycznego dla danej sekwencji wejściowej jest bardzo prosta. W każdym stanie **s** dla kolejnego znaku wejściowego **x** należy rozpatrzyć każdą etykietę przejść ze stanu **s**. Jeżeli uda się znaleźć przejście, którego etykieta zawiera znak **x**, to przejście to określa właściwy stan następny. Jeżeli żadna nie zawiera znaku **x**, to automat „zamiera" i nie może przetwarzać dalszych znaków wejściowych.

Automaty niedeterministyczne

- **Automaty niedeterministyczne** (ang. *nondeterministic*) mogą (ale nie muszą) posiadać dwa (lub więcej) przejścia z danego stanu zawierające ten sam symbol.
- Warto zauważyć, że automat deterministyczny jest równocześnie automatem niedeterministycznym, który nie posiada wielu przejść dla jednego symbolu.
- Automaty niedeterministyczne nie mogą być bezpośrednio implementowane za pomocą programów, ale stanowią przydatne pojęcie abstrakcyjne.



W momencie podjęcia próby symulacji automatu niedeterministycznego w przypadku ciągu wejściowego składającego się ze znaków $\mathbf{a_1, a_2, \ldots, a_k}$ może okazać się, że ten sam ciąg etykietuje wiele ścieżek. Wygodnie jest przyjąć, że automat niedeterministyczny akceptuje taki ciąg wejściowy, jeżeli co najmniej jedna z etykietowanych ścieżek prowadzi do stanu akceptującego.

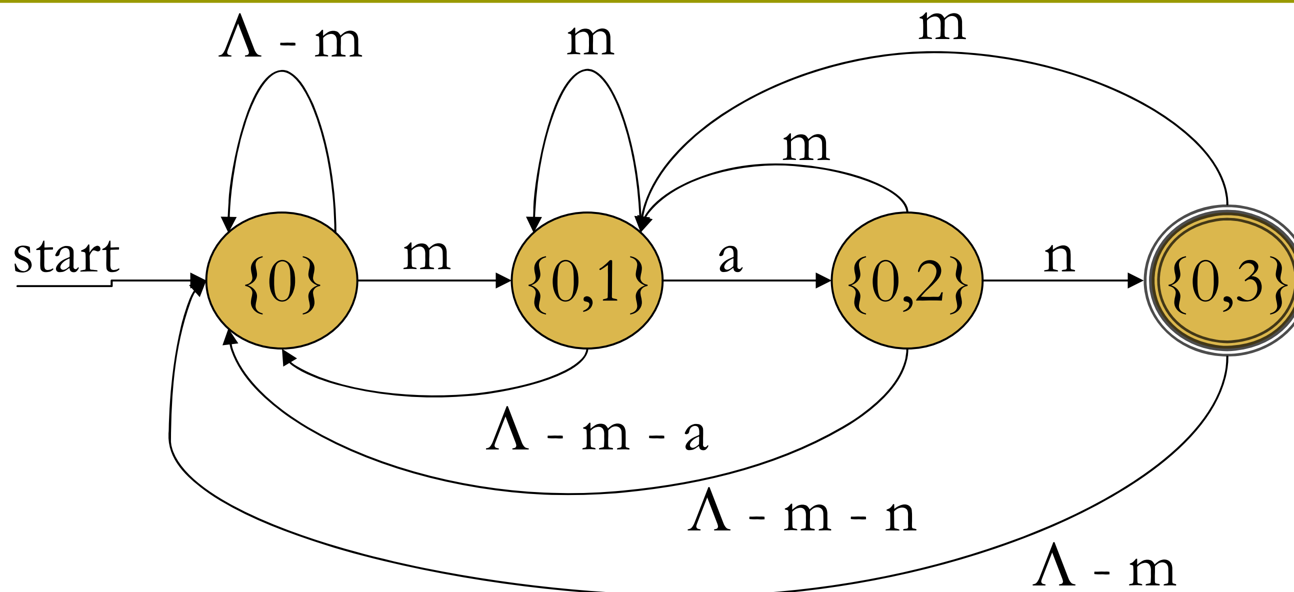
niedeterminizm = „zgadywanie”

Jak zastąpić automat niedeterministyczny deterministycznym?

- Automat niedeterministyczny zastępujemy deterministycznym przez skonstruowanie **równoważnego** automatu deterministycznego. Technika ta nosi nazwę **konstrukcji podzbiorów** (ang. *subset construction*).
- **Równoważność automatów:**
  - Mówimy, że automat **A** jest **równoważny** (ang. *equivalent*) automatowi **B**, jeżeli akceptują ten sam zbiór ciągów wejściowych.
    Innymi słowy, jeżeli $a_1a_2\ldots a_k$ jest dowolnym ciągiem symboli, to spełnione są dwa następujące warunki:

1. Jeżeli istnieje ścieżka zaetykietowana jako $a_1a_2\ldots a_k$ wiodąca od stanu początkowego automatu **A** do pewnego stanu akceptującego automatu **A**, to istnieje również ścieżka zaetykietowana $a_1a_2\ldots a_k$ wiodąca od stanu początkowego automatu **B** do stanu końcowego tego automatu.
2. Jeżeli istnieje ścieżka zaetykietowana jako $a_1a_2\ldots a_k$ wiodąca od stanu początkowego automatu **B** do pewnego stanu akceptującego automatu **B**, to istnieje również ścieżka zaetykietowana $a_1a_2\ldots a_k$ wiodąca od stanu początkowego automatu **A** do stanu końcowego tego automatu.

Automat deterministyczny D



Konstrukcje automatu można uznać za skończoną. Przejścia ze stanu **{0,3}** nie prowadza do żadnego stanu automatu **D** którego jeszcze nie sprawdziliśmy.
Stan **{0}** oznacza że odczytany ciąg nie kończy się żadnym przedrostkiem wyrazu **man**, stan **{0,1}** że kończy się sekwencją **m**, stan **{0,2}** że kończy się sekwencją **ma**, stan **{0,3}** że kończy się sekwencja **man**.

Raz jeszcze automaty

- Automat skończony jest to oparty na modelu grafowym sposób określania wzorców. Występują jego dwie odmiany: **automat deterministyczny** oraz **automat niedeterministyczny**.
- Istnieje możliwość konwersji automatu deterministycznego na program rozpoznający wzorce.
- Istnieje możliwość konwersji automatu niedeterministycznego na automat deterministyczny rozpoznający te same wzorce dzięki konstrukcji podzbiorów.

- Istnieje bliski związek miedzy automatami deterministycznymi a programami, które wykorzystują pojecie stanu w celu odróżnienia ról, jakie odgrywają różne części programu. Zaprojektowanie automatu deterministycznego stanowi często dobry sposób opracowania takiego programu. Może to być trudne zadanie, często łatwiej zaprojektować automat niedeterministyczny, a następnie za pomocą konstrukcji podzbiorów zamienić go na jego odpowiednik deterministyczny.

Wyrażenia regularne

- **Wyrażenia regularne** (ang. *regular expressions*) stanowią algebraiczny sposób definiowania wzorców.
- Wyrażenia regularne stanowią analogię do algebry wyrażeń arytmetycznych oraz do algebry relacyjnej.
- Zbiór wzorców które można wyrazić w ramach algebry wyrażeń regularnych odpowiada dokładnie zbiorowi wzorców, które można opisać za pomocą automatów.

Operandy wyrażeń regularnych

- Wyrażenia regularne posiadają pewne rodzaje operandów niepodzielnych (ang. *atomic operands*). Poniżej lista:
  1. Znak
  2. Symbol ε
  3. Symbol ∅
  4. Zmienna która może być dowolnym wzorcem zdefiniowanym za pomocą wyrażenia regularnego.
- **Wartość wyrażenia regularnego jest wzorcem** składającym się ze zbioru ciągów znaków, który często **określa się mianem języka** (ang. *language*).
- Język określony przez wyrażenia regularne **E** oznaczony będzie jako **L(E)** lub określany jako **język wyrażenia E**.

Języki operandów niepodzielnych

- Języki operandów niepodzielnych definiuje się w następujący sposób.
  1. Jeżeli **x** jest dowolnym znakiem, to wyrażenie regularne **x** oznacza język **{x}**, to znaczy **L(x) = {x}**.
     Należy zauważyć, że taki język jest zbiorem zawierającym jeden ciąg znakowy.
     Ciąg ten ma długość **1** i jedyna pozycja tego ciągu określa znak **x**.
  2. **L(ε) = {ε}**. Specjalny symbol **ε** jako wyrażenie regularne oznacza zbiór, którego jedynym ciągiem znakowym jest ciąg pusty, czyli ciąg o długości **0**.
  3. **L(∅) = ∅** . Specjalny symbol **∅** jako wyrażenie regularne oznacza zbiór pusty ciągów znakowych.
- Istnieją trzy operatory w odniesieniu do wyrażeń regularnych.
- Można je grupować przy użyciu nawiasów, podobnie jak ma to miejsce w przypadku innych znanych algebr.
- Definiuje się prawa kolejności działań oraz prawa łączności, które pozwalają na pomijanie niektórych par nawiasów – tak jak w przypadku wyrażeń arytmetycznych.

Operatory wyrażeń regularnych

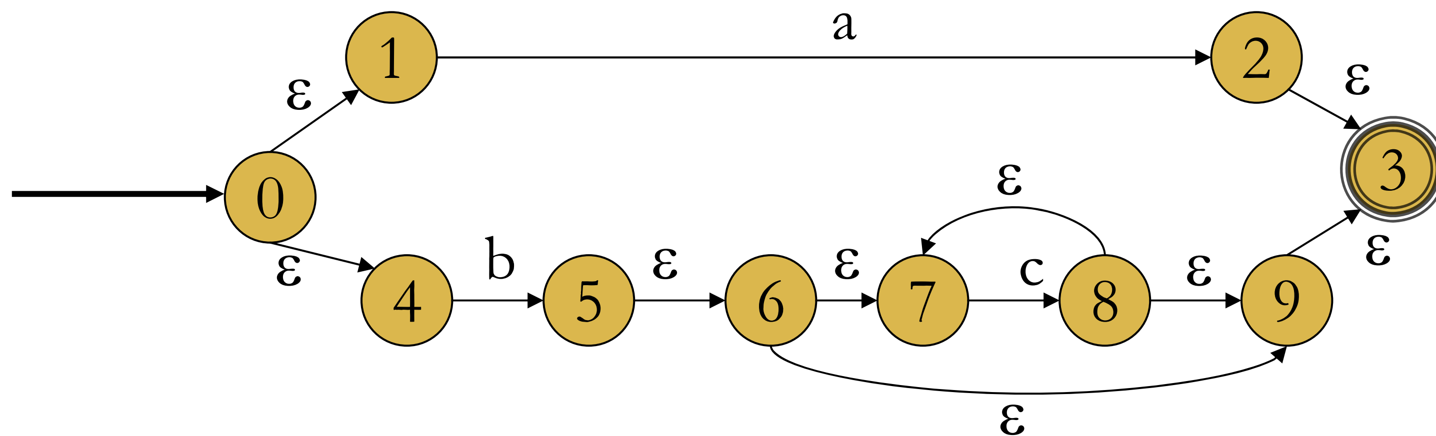
- **Suma:**
  - Symbol sumy (ang. *union*) oznacza się za pomocą symbolu **|** . Jeżeli **R** i **S** są dwoma wyrażeniami regularnymi, to **R | S** oznacza sumę języków określanych przez **R** i **S**. To znaczy **L(R|S) = L(R) ∪ L(S)**.
  - **L(R) i L(S) są zbiorami ciągów znakowych, notacja sumowania jest uzasadniona.**
- **Złożenie:**
  - Operator złożenia (ang. *concatenation*) nie jest reprezentowany przez żaden odrębny symbol.
  - Jeżeli **R** i **S** są wyrażeniami regularnymi to **RS** oznacza ich złożenie. **L(RS)**, czyli język określony przez **RS**, jest tworzony z języków **L(R)** i **L(S)** w sposób następujący:
    - Dla każdego ciągu znakowego **r** należącego do **L(R)** oraz każdego ciągu znakowego **s** należącego do **L(S)**, ciąg **rs**, czyli złożenie ciągów **r** i **s**, należy do **L(RS)**.
  - **Złożenie dwóch list takich jak ciągi znaków, jest wykonywane przez pobranie po kolei elementów pierwszej z nich i uzupełnienie ich po kolei elementami drugiej listy.**

Operatory wyrażeń regularnych

- **Domknięcie:**
  - Operator domknięcia (ang. *closure*), jest to operator jednoargumentowy przyrostkowy. Domknięcie oznacza się za pomocą symbolu *****, tzn. **R*** oznacza domkniecie wyrażenia regularnego **R**. Operator domknięcia ma najwyższy priorytet.
  - Efekt działania operatora domknięcia można zdefiniować jako „określenie występowania zera lub większej liczby wystąpień ciągów znaków w R”.
  - Oznacza to że **L(R*)** składa się z:
    1. Ciągu pustego **ε**, który można interpretować jako brak wystąpień ciągów znaków w **L(R)**.
    2. Wszystkich ciągów znaków języka **L(R)**. Reprezentują one jedno wystąpienie ciągów znaków w **L(R)**.
    3. Wszystkich ciągów znaków języka **L(RR)**, czyli złożenia języka **L(R)** z samym sobą. Reprezentują one dwa wystąpienia ciągów znaków z **L(R)**.
    4. Wszystkich ciągów znaków języka **L(RRR), L(RRRR)** i tak dalej, które reprezentują trzy, cztery i więcej wystąpień ciągów znaków z **L(R)**.
  - Nieformalnie można napisać: **R* = e | R | RR | RRR | …**
  - **Wyrażenie po prawej stronie to nie jest wyrażeniem regularnym ponieważ zawiera nieskończoną liczbę wystąpień operatora sumy. Wszystkie wyrażenia regularne są tworzone ze skończonej liczby wystąpień operatorów.**

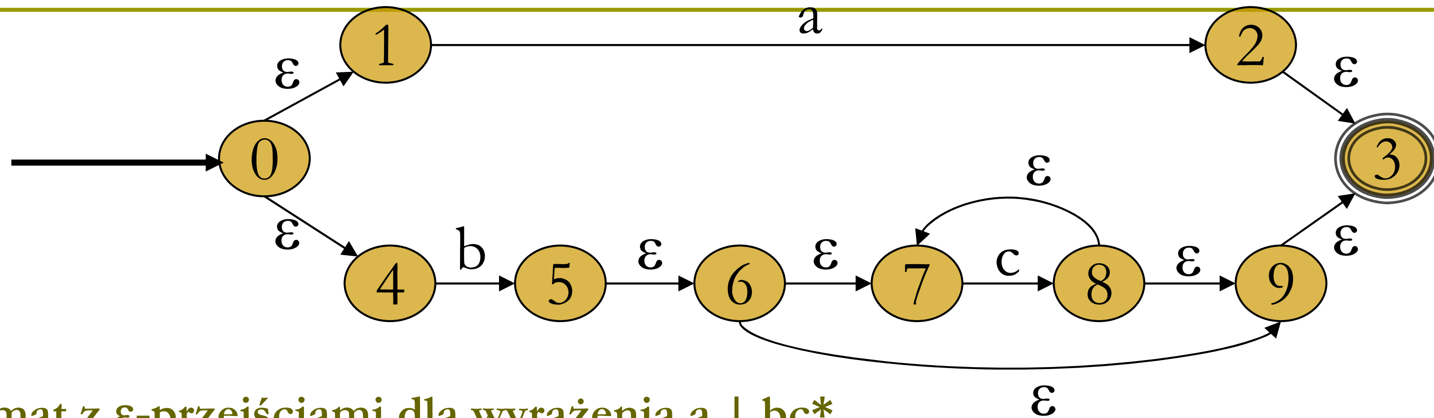
Automaty z epsilon przejściami

- Należy rozszerzyć notacje używaną w przypadku automatów w celu umożliwienia opisu krawędzi **posiadających etykietę ε**. Takie automaty wciąż akceptują ciąg znaków **s** wtedy i tylko wtedy, gdy ścieżka zaetykietowana ciągiem **s** wiedzie od stanu początkowego do stanu akceptującego. **Symbol ε**, ciąg pusty, **jest „niewidoczny”** w ciągach znaków, stąd w czasie konstruowania etykiety danej ścieżki w efekcie usuwa się wszystkie symbole **ε** i używa tylko rzeczywistych znaków.

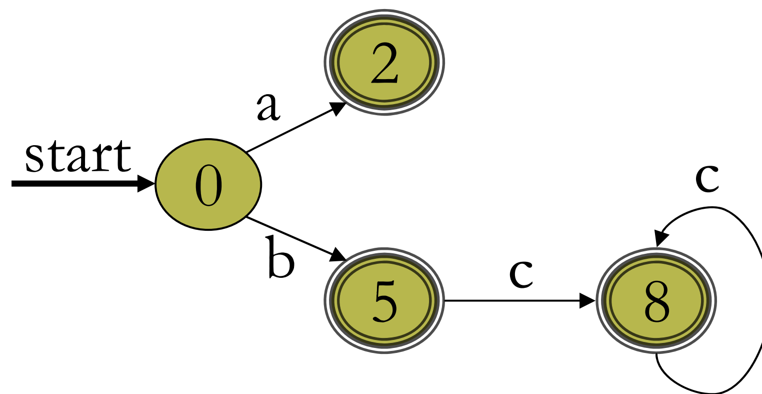


Automat z ε-przejściami dla wyrażenia a | bc*

Automaty z epsilon przejściami



Automat z ε-przejściami dla wyrażenia a | bc*

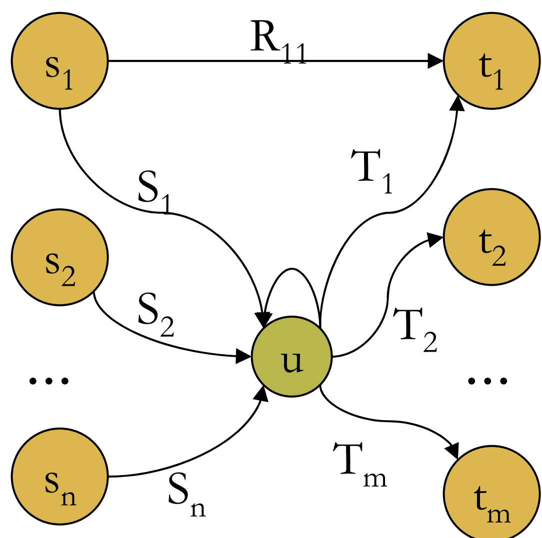


Automat skonstruowany na podstawie eliminacji ε -przejść.

Automat akceptuje wszystkie ciągi języka L (a | bc*).

Konstrukcja eliminacji stanów.

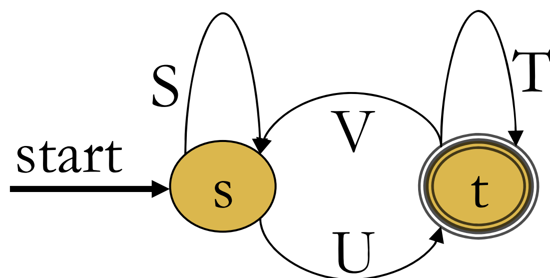
- Kluczowym etapem konwersji z postaci automatu na wyrażenie regularne jest eliminacja stanów. Chcemy wyeliminować stan **u**, ale chcemy zachować etykiety krawędzi występujące w postaci wyrażeń regularnych, tak aby zbiór etykiet ścieżek miedzy dowolnymi pozostałymi stanami nie uległ zmianie.
- Poprzedniki stanu **u** to $s_1, s_2, \ldots, s_n$ zaś następniki stanu **u** to $t_1, t_2, \ldots, t_m$ (mogą też istnieć stany wspólne).



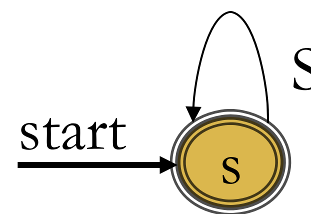
Zbiór ciągów znaków etykietujących ścieżki wiodące z wierzchołków s_i do wierzchołka **u**, włącznie z ścieżkami biegnącymi kilkakrotnie wokół pętli $u \to u$, oraz z wierzchołka **u** do wierzchołka t_j, jest opisany za pomocą wyrażenia regularnego $S_i U^* T_j$.
Po eliminacji wierzchołka **u** należy zastąpić etykietę R_{ij}, czyli etykietę krawędzi $s_i \to t_j$ przez etykietę $R_{ij} \mid S_i U^* T_j$.

Redukcja zupełna automatu

- W celu otrzymania wyrażenia regularnego określającego wszystkie ciągi znaków akceptowane przez automat **A** i żadne inne, należy rozpatrzyć po kolei każdy stan akceptujący **t** automatu **A**.
- Każdy ciąg znaków akceptowany przez automat **A** jest akceptowany dlatego, że etykietuje on ścieżkę wiodąca ze stanu początkowego **s** do pewnego stanu akceptującego **t**.



Automat zredukowany do dwóch stanów.
Wyrażenie regularne: S*U(T | VS*U)*



Automat posiadający jedynie stan początkowy.
Wyrażenie regularne: S*

Gramatyki

- Gramatyka bezkontekstowa wykorzystuje model funkcji rekurencyjnych.
- Jednym z ważnych zastosowań gramatyk są specyfikacje języków programowania. Gramatyki stanowią zwięzłą notacje opisu ich składni.
- Gramatyki posiadają większe możliwości w zakresie opisu języków niż wyrażenia regularne. Gramatyki oferują co najmniej tak samo duże możliwości opisu języków, jak wyrażenia regularne przez przedstawienie sposobu symulowania wyrażeń regularnych za pomocą gramatyk. Istnieją jednakże języki które można wyrazić za pomocą gramatyk, ale nie można za pomocą wyrażeń regularnych.

Jednoznaczne gramatyki wyrażeń

Konstrukcja jednoznacznej gramatyki polega na zdefiniowaniu trzech kategorii syntaktycznych o następującym znaczeniu:

<Czynnik> - generuje wyrażenia, które nie mogą zostać rozdzielone, to znaczy czynnik jest albo pojedynczym operandem, albo dowolnym wyrażeniem umieszczonym w nawiasie.

< Składnik> - generuje iloczyn lub iloraz czynników. Pojedynczy czynnik jest składnikiem, a więc stanowi ciąg czynników rozdzielonych operatorami * lub /. Przykładami składników są 12 lub 12/3*45.

<Wyrażenie> - generuje różnicę lub sumę jednego lub większej liczby składników. Pojedynczy składnik jest wyrażeniem, a więc stanowi sekwencję składników rozdzielonych operatorami + lub -.
Przykładami wyrażeń są 12, 12/3*45 lub 12+3*45-6.

(1) <W> → <W> + <S> | <W> - <S> | <S>
(2) <S> → <S> * <Cz> | <S> / <Cz> | <Cz>
(3) <Cz> → (<W>) | <L>
(4) <L > → <L><C> | <C>
(5) <C> → 0 | 1 | ...... | 9

gramatyka jednoznaczna wyrażeń arytmetycznych

Analizator składniowy

Gramatyka

(1) <I> → w c <I>
(2) <I> → { <D>
(3) <I> → s ;
(4) <D> → <I> <D>
(5) <D> → }

Tabela analizy składniowej

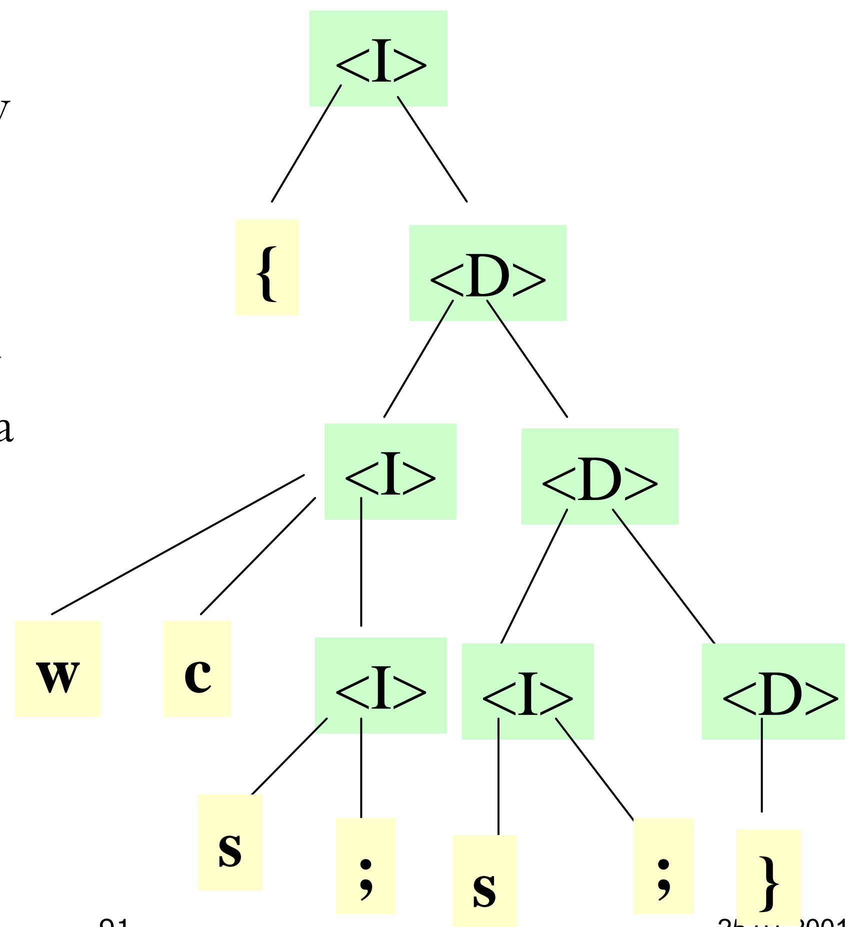
	w	c	{	}	s	;	ENDM
<I>	1		2		3		
<D>	4		4	5	4		

Postać gramatyki przedstawionej powyżej umożliwia jej analizę składniową za pomocą schodzenia rekurencyjnego lub za pomocą analizy składniowej opartej na tabeli.
Metoda zamiany gramatyki na analizator składniowy to algorytm pozwalający stwierdzić, czy dany ciąg znaków należy do pewnego języka

Drzewo rozbioru

Drzewa analizy składniowej (drzewa rozbioru) stanowią formę reprezentacji, która przedstawia strukturę ciągu znaków zgodną z daną gramatyką. Niejednoznaczność – to problem, który pojawia się w sytuacji gdy ciąg znaków posiada dwa lub więcej odrębnych drzew analizy składniowej, przez co nie posiada unikatowej struktury zgodnie z daną gramatyką

Pełne drzewo rozbioru dla analizy składniowej dla ciągu: {w c s ; s ; } ENDM



Konstrukcja gramatyki dla wyrażenia regularnego: a | bc*

1. Tworzymy kategorie syntaktyczne dla trzech symboli, które pojawiają się w tym wyrażeniu:

   $< A > \to a \qquad < B > \to b \qquad < C > \to c$

2. Tworzymy gramatykę dla c*: $< D > \to < C > < D > \mid \varepsilon$

   Wówczas $L(<D>) = L(<C>))^* = c^*$

3. Tworzymy gramatykę dla bc*: $< E > \to < B > < D >$

4. Tworzymy gramatykę dla całego wyrażenia regularnego a | bc*:

   $< F > \to < A > \mid < E >$

końcowa postać gramatyki

$<F> \to <A> \mid <E>$
$<E> \to <B> <D>$
$<D> \to <C> <D> \mid \varepsilon$
$<A> \to a$
$<B > \to b$
$<C> \to c$

Język posiadający gramatykę ale nie posiadający wyrażenia regularnego

Język E będzie zbiorem znaków składających się z jednego lub większej liczby symboli 0, po których występuje ta sama liczba symboli 1, to znaczy:

E = { 01, 0011, 000111, ….}

W celu opisania ciągów znaków języka E można użyć przydatnej notacji opartej na wykładnikach. Niech s^n, gdzie s jest ciągiem znaków, zaś n liczba całkowitą, oznacza ss….s (n razy), to znaczy s złożone ze sobą n razy.

Wówczas:

$E = \{0^1 1^1, 0^2 1^2, 0^3 1^3, \ldots\}$ lub $E = \{0^n 1^n \mid n \geq 1\}$

Język E można zapisać za pomocą gramatyki:

$$\langle S \rangle \to 0 \langle S \rangle 1$$
$$\langle S \rangle \to 0\ 1$$

Ponieważ nie istnieją żadne inne ciągi znaków możliwe do utworzenia na podstawie tych dwóch produkcji, E = L(<S>).