Teoretyczne podstawy informatyki

Repetytorium: informacja algorytmy, modele danych

Informatyka: mechanizacja abstrakcji

Zasadniczo jednak **informatyka jest**

- **nauką o *abstrakcji* ,** czyli nauką o tworzeniu właściwego modelu reprezentującego problem i wynajdowaniu odpowiedniej techniki mechanicznego jego rozwiązywania

- Informatycy tworzą abstrakcje rzeczywistych problemów w formach które mogą być rozumiane i przetwarzane w pamięci komputera

Abstrakcja

oznaczać będzie pewne uproszczenie, zastąpienie skomplikowanych i szczegółowych okoliczności występujących w świecie rzeczywistym zrozumiałym modelem umożliwiającym rozwiązanie naszego problemu. Oznacza to że **abstrahujemy od szczegółów** które nie maja wpływu lub mają minimalny wpływ na rozwiązanie problemu. Opracowanie odpowiedniego modelu ułatwia zajęcie się istotą problemu.

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

Informacja i zasady jej zapisu

Jednostka informacji: bit

Podstawową jednostką informacji jest bit, oznaczany też poprzez „b”(w ang. *kawałek*, skrót od **binary digit**, czyli cyfra dwójkowa).

- Bit jest to **podstawowa elementarna jednostka informacji**: wystarczająca do zakomunikowania jednego z co najwyżej dwóch jednakowo prawdopodobnych zdarzeń.
- Bit stanowi podstawę zapisu informacji w różnych typach pamięci komputera. Wszystkie inne jednostki składają się z jego wielokrotności.
- Bit przyjmuje jedną z dwóch wartości, które zwykle oznacza się jako **„0”** lub **„1”**. Jest to oznaczenie stosowane w matematyce (**wartość logiczna: „0” – fałsz, „1” - prawda**) oraz przy opisie informacji przechowywanej w pamięci komputera i opisie sposobów **kodowania informacji**.

Bajt

Jest to najmniejsza adresowalna jednostka informacji pamięci komputerowej, składająca się z bitów, w praktyce przyjmuje się że jeden bajt to **8 bitów** (zostało to uznane za standard w 1964 r.).

Jeden bajt może reprezentować zatem $2^8 = 256$ różnych wartości, które mogą oznaczać zapisywane informacje.

Bajt oznaczany jest poprzez **„B”**.

Stosowanie przedrostków *kilo, mega, giga* itp. jako do określania odpowiednich potęg liczby dwa **(np. 2^{10})** jest niezgodne z wytycznymi układu SI (np. *kilo* oznacza 1000, a nie 1024).

W celu odróżnienia przedrostków o mnożniku 1000 od przedrostków o mnożniku 1024 **(2^{10})**, w styczniu 1997 r. pojawiła się propozycja ujednoznacznienia opracowana przez **IEC** (ang. *International Electrotechnical Commission)* polegająca na dodawaniu litery "i" po symbolu przedrostka dwójkowego, oraz "bi" po jego nazwie.

Wielokrotności bajtów

Przedrostki dziesiętne (SI)			Przedrostki binarne (IEC)		
Nazwa	Symbol	Mnożnik	Nazwa	Symbol	Mnożnik
Kilobajt	kB / KB	$10^3 = 1000^1$	Kibibajt	KiB	$2^{10} = 1024^1$
Megabajt	MB	$10^6 = 1000^2$	Mebibajt	MiB	$2^{20} = 1024^2$
Gigabajt	GB	$10^9 = 1000^3$	Gibibajt	GiB	$2^{30} = 1024^3$
Terabajt	TB	$10^{12} = 1000^4$	Tebibajt	TiB	$2^{40} = 1024^4$

Systemy liczbowe w informatyce

Z racji reprezentacji liczb w pamięci komputerów za pomocą bitów, najbardziej naturalnym systemem w informatyce jest **system dwójkowy**.

Ze względu na specyfikę architektury komputerów, gdzie często najszybszy dostęp jest do adresów parzystych, albo podzielnych przez 4, 8 czy 16, często używany jest **szesnastkowy system liczbowy**. Sprawdza się on szczególnie przy zapisie dużych liczb takich jak adresy pamięci, zakresy parametrów itp.

System zmiennopozycyjny

- Każdą niezerową liczbę rzeczywistą reprezentujemy za pomocą przybliżenia wymiernego w postaci pary **(m, c)**, takich że:

$$m \in [-1, -\tfrac{1}{2}) \cup (\tfrac{1}{2}, 1]$$

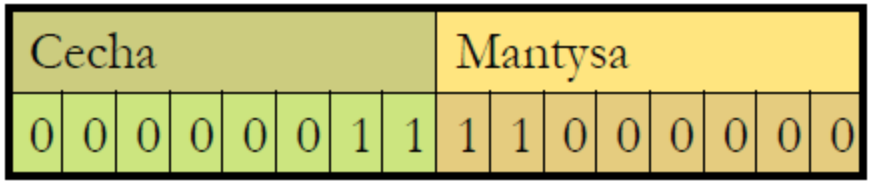
- m jest mantysą, zaś c jest cechą. Interpretacja takiej reprezentacji wyraża się wzorem:

$$x = m P^{c}$$

- Oczywiście w naszym przypadku P = **2** (podstawa systemu)
- System ten, umożliwia zapis liczb rzeczywistych z ustalonym **błędem względnym**.

System zmiennopozycyjny

- Liczba binarna zapisana w postaci cecha – mantysa na dwóch bajtach:



-

Tutaj: c = 3, m = - 1.

- W praktyce zwykle na cechę przeznaczamy jeden bajt, na mantysę minimum trzy bajty.
  - **Ilość bajtów przeznaczonych na cechę decyduje o zakresie.**
  - **Ilość bajtów przeznaczona na mantysę decyduje o błędzie.**
- Liczby w mantysie są kodowane w systemie znak – moduł.
- Zaś dla cechy w systemie uzupełnieniowym.

Błąd obliczeniowy

- **Błąd bezwzględny:** różnica między wartością zmierzona/obliczona a wartością dokładną

$$\Delta x = x - x_0$$

  - W **systemie stałopozycyjnym** obliczenia są wykonywane ze **stałym max błędem bezwzględnym**

- **Błąd względny:** błąd bezwzględny podzielony przez wartość dokładną

$$\delta x = \frac{\Delta x}{x_0}$$

  - W **systemie zmiennopozycyjnym** obliczenia są wykonywane ze **stałym max błędem względnym**

Struktury danych i algorytmy

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

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….

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 (np. stała albo **proporcjonalna do liczby danych**) 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), alg. najczęściej związany z działaniami na macierzach,
  - liczba powtórzeń jest nieznana (zależy od spełnienia pewnego warunku), alg. 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 z powrotami

- Metoda powrotów
  - Wymaga zapamiętania wszystkich wykonanych ruchów czy też wszystkich odwiedzonych stanów aby możliwe było cofanie posunięć.
  - Stanów mogą być tysiące lub miliony więc bezpośrednie zastosowanie metody powrotów, mogące doprowadzić do odwiedzenia wszystkich stanów, może być zbyt kosztowne.
  - Inteligentny wybór następnego posunięcia, **funkcja oceniająca**, może znacznie poprawić efektywność algorytmu.
    - Np. aby uniknąć przeglądania nieistotnych fragmentów przestrzeni stanów.

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.
- Błędy zaokrągleń, powstające przy reprezentacji liczb, a także przy wykonywaniu działań na nich rozwinęły się w samodzielna dziedzinę tzw. **analiza numeryczna**.

Efektywność algorytmu

- **Testy wzorcowe:**
  - Podczas porównywania dwóch lub więcej programów zaprojektowanych do wykonywania tego samego zadania, opracowujemy niewielki zbiór typowych danych wejściowych które mogą posłużyć jako **dane wzorcowe** (ang. benchmark).
  - Powinny być one reprezentatywne i zakłada się że program dobrze działający dla danych wzorcowych będzie też dobrze działał dla wszystkich innych danych.
  - Np. test wzorcowy umożliwiający porównanie algorytmów sortujących może opierać się na jednym **małym** zbiorze danych, np. zbiór pierwszych 20 cyfr liczby π; jednym **średnim**, np. zbiór kodów pocztowych województwa krakowskiego; oraz na **dużym** zbiorze takim jak zbiór numerów telefonów z obszaru Krakowa i okolic.
  - Przydatne jest też sprawdzenie jak algorytm działa dla ciągu już posortowanego (często działają kiepsko).

Efektywność algorytmu

- **Czas działania:**
  - Oznaczamy przez funkcje **T(n)** liczbę jednostek czasu, które zajmuje wykonanie programu lub algorytmu w przypadku problemu o rozmiarze **n**.
  - Funkcje te nazywamy **czasem działania**. Dość często czas działania zależy od konkretnych danych wejściowych, nie tylko ich rozmiaru. W takim przypadku, funkcje **T(n)** definiuje się jako **najmniej korzystny przypadek** z punktu widzenia kosztów czasowych. Inną wyznaczaną wielkością jest też **czas średni**, czyli średni dla różnych danych wejściowych.

Modele danych

Modele danych

- **Modele danych są to abstrakcje wykorzystywane do opisywania problemów.**
- 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.

Modele danych języków programowania

- Każdy język programowania zawiera **własny model danych**, który zazwyczaj istotnie różni się od modeli oferowanych przez inne języki.
- **Podstawowa zasada** realizowana przez większość języków programowania w odniesieniu do modeli danych określa, że **każdy program ma dostęp do „pudełek”**, które traktujemy jako obszary pamięci.
  - Każde „pudełko” ma swój typ, np. int, char.
  - Wartości przechowywane w pudełkach nazywamy często obiektami danych.
  - Możemy teraz nadawać nazwy wykorzystywanym pudełkom. W ogólności nazwa jest dowolnym wyrażeniem wskazującym na pudełko.

Modele danych języków programowania

- Bardzo ważne są też **modele danych**, które **nie są** częścią języka programowania, takie jak **listy, drzewa, grafy, zbiory**.
- Np. w języku matematycznym, **lista** jest ciągiem **n** elementów, który zapisujemy jako **$(a_1,a_2,\ldots,a_n)$**. Do zbioru operacji wykonywanych na listach należą:
  - tworzenie listy,
  - wstawianie nowego elementu do listy,
  - usuwanie elementu z listy,
  - łączenie list.

Modele danych języków programowania

- Podstawowe typy danych w **języku programowania C** to:
  - liczby całkowite, liczby zmiennoprzecinkowe,
  - znaki,
  - tablice,
  - struktury,
  - wskaźniki.
- Wszystkie te pojęcia to **statyczne elementy modelu danych**.
- Dopuszczalne operacje na tych danych to:
  - typowe operacje arytmetyczne na liczbach całkowitych i zmiennoprzecinkowych,
  - operacje dostępu do elementów tablic i struktur,
  - oraz wyłuskiwanie wskaźników czyli znajdywanie obiektów przez nie wskazywanych.
- Te operacje to **dynamiczne elementy modelu danych.**

Modele danych języków programowania

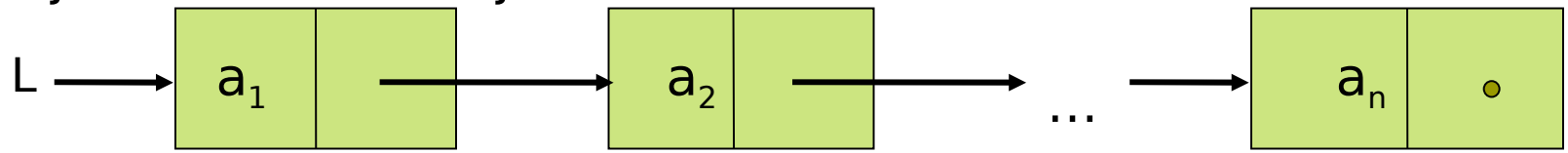
- Bardzo ważne są też **modele danych**, które **nie są** częścią języka programowania, takie jak **listy, drzewa, grafy, zbiory**.
- Np. w języku matematycznym, **lista** jest ciągiem **n** elementów, który zapisujemy jako **$(a_1,a_2,\ldots,a_n)$**. Do zbioru operacji wykonywanych na listach należą:
  - tworzenie listy,
  - wstawianie nowego elementu do listy,
  - usuwanie elementu z listy,
  - łączenie list.

LISTY

Listy należą do najbardziej podstawowych modeli danych wykorzystywanych w programach komputerowych.

Struktura danych → lista jednokierunkowa

- Najprostszym sposobem **implementacji listy** jest wykorzystanie jednokierunkowej listy komórek. Każda z komórek składa się z dwóch pól, jedno zawiera element listy, drugie zawiera wskaźnik do następnej komórki listy jednokierunkowej.



Lista jednokierunkowa **L = (a_1, a_2, …, a_n)**.

- Dla każdego elementu istnieje dokładnie jedna komórka; element a_i znajduje się w polu **i**-tej komórki.
- Wskaźnik w **i**-tej komórce wskazuje na **i+1** komórkę, dla **i= 1, 2, …, n-1.**
- Wskaźnik w ostatniej komórce jest równy **NULL** i oznacza koniec listy.
- Poza listą wykorzystujemy wskaźnik **L**, który wskazuje na pierwsza komórkę listy. Gdyby lista była pusta **L = NULL**.
- Dla każdej komórki znamy wskaźnik następnej *(ang.* **next***)*.

Podstawowa terminologia

- **Operacje na listach, możemy:**
  - ***sortować listę*** czyli formalnie zastępować daną listę inną listą która powstaje przez wykonanie permutacji na liście oryginalnej,
  - ***dzielić listę*** na podlisty,
  - ***scalać*** podlisty,
  - ***dodawać element*** do listy,
  - ***usuwać element*** z listy,
  - ***wyszukać element*** w liście.

Lista jednokierunkowa

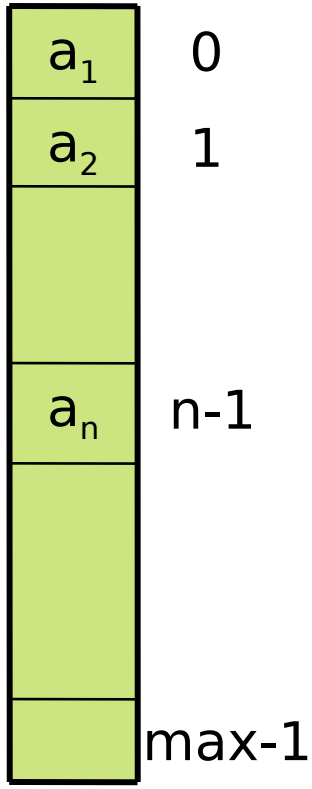
Wstawianie, wyszukiwanie, usuwanie:

Lista	Wstawianie	Usuwanie	Wyszukanie
Brak Duplikatów	$n/2 \rightarrow n$	$n/2 \rightarrow n$	$n/2 \rightarrow n$
Duplikaty	0	m	$m/2 \rightarrow m$
Lista posortowana	n/2	n/2	n/2

n ilość elementów w słowniku (oraz liście bez duplikatów)
m ilość elementów w liście z duplikatami
n/2→m oznacza że **średnio** przeszukujemy **n/2** przy pomyślnym wyniku oraz **m** przy niepomyślnym.
Zobaczymy w następnym wykładzie że dla implementacji słownika w postaci drzewa przeszukiwania binarnego operacje wymagają średnio ***O*(log n)**.

Lista oparta na tablicy

- Innym powszechnie znanym sposobem implementowania listy jest tworzenie struktury złożonej z dwóch komponentów:
  - **tablicy** przechowującej elementy listy L (musimy zadeklarować maksymalny wymiar),
  - **zmiennej** przechowującej liczbę elementów znajdującej się aktualnie na liście, oznaczmy ją przez n.
- **Implementacja list oparta na tablicy** jest z wielu powodów bardziej wygodna niż oparta na liście jednokierunkowej.
- **Wada:** konieczność zadeklarowania maksymalnej liczby elementów.
- **Zaleta:** możliwość przeszukiwania binarnego jeżeli lista była posortowana.



Stos

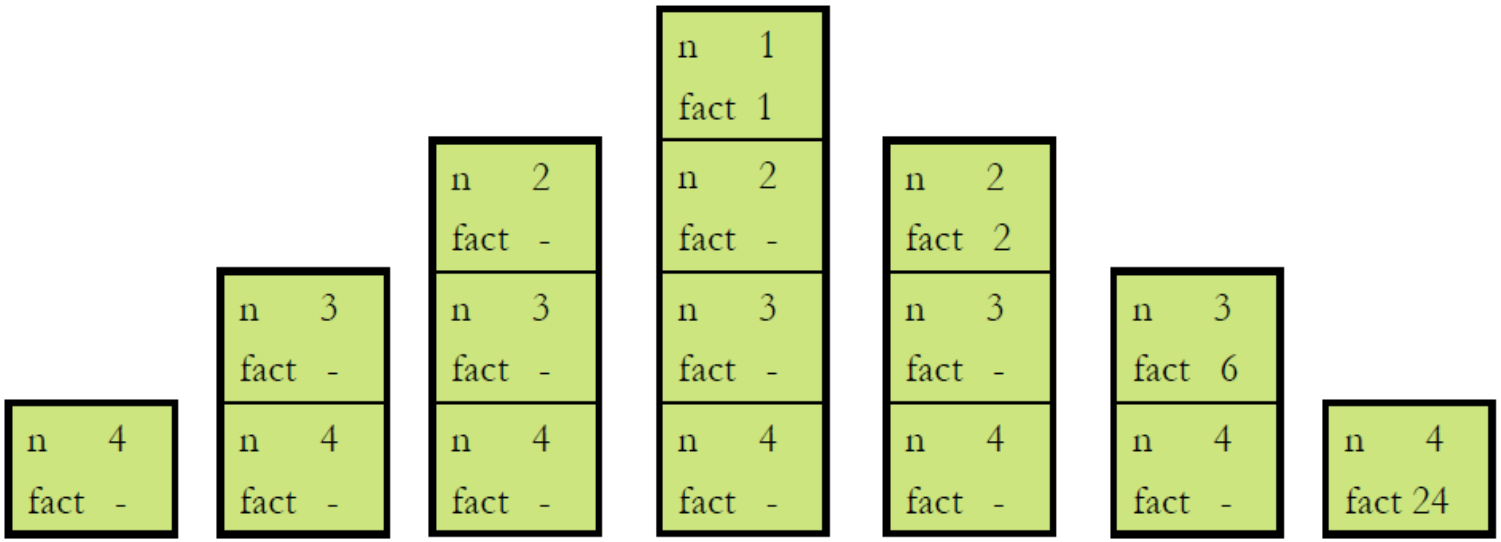
Abstrakcyjny typ danych
= stos
Implementująca struktura danych
= lista jednokierunkowa, lista oparta na tablicach

- **Stos:** Sekwencja elementów $a_1, a_2, \ldots, a_n$ należących do pewnego typu.
- **Operacje wykonywane na stosie:**
  - kładziemy element na szczycie stosu (ang. **push**)
  - zdejmujemy element ze szczytu stosu (ang. **pop**)
  - czyszczenie stosu – sprawienie że stanie się pusty (ang. **clear**)
  - sprawdzenie czy stos jest pusty (ang. **empty**)
  - sprawdzenie czy stos jest pełny

  Każda z operacji jest **$T(n) = O(1)$.**
- Stos jest wykorzystywany **„w tle”** do implementowania funkcji rekurencyjnych.

Wykorzystanie stosu w implementacji wywołań funkcji

- Stos czasu wykonania przechowuje rekordy aktywacji dla wszystkich istniejących w danej chwili aktywacji.
- Wywołując funkcje kładziemy rekord aktywacji „na stosie”.
- Kiedy funkcja kończy swoje działanie, zdejmujemy jej rekord aktywacji ze szczytu stosu, odsłaniając tym samym rekord aktywacji funkcji która ją wywołała.



Kolejka

Abstrakcyjny typ danych
= kolejka
Implementująca struktura danych
= lista jednokierunkowa, lista oparta na tablicach

- **Kolejka:** sekwencja elementów a_1, a_2, ..., a_n należących do pewnego typu.
- **Operacje wykonywane na kolejce:**
  - dołączenie elementu do końca kolejki (ang. **push**)
  - usunięcie element z początku kolejki (ang. **pop**)
  - czyszczenie kolejki – sprawienie że stanie się pusta (ang. **clear**)
  - sprawdzenie czy kolejka jest pusta (ang. **empty**)

Każda z operacji jest **T(n) = *O*(1).**

Więcej abstrakcyjnych typów danych...

Abstrakcyjny typ danych	Abstrakcyjna implementacja	Struktura danych
Słownik	Drzewa przeszukiwania binarnego	Struktura lewe dziecko – prawe dziecko
Kolejka priorytetowa	Zrównoważone drzewo częściowo uporządkowane	Kopiec
Słownik	Lista	1. Lista jednokierunkowa 2. Tablica mieszająca
Stos	Lista	1. Lista jednokierunkowa 2. Tablica
Kolejka	Lista	1. Lista jednokierunkowa 2. Tablica cykliczna

ZBIORY

- Zbiór jest najbardziej podstawowym modelem danych w matematyce.
- Wszystkie pojęcia matematyczne, od drzew po liczby rzeczywiste można wyrazić za pomocą specjalnego rodzaju zbioru.

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

- **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”**

Operacje na zbiorach

- Jeżeli **S** i **T** są **zdarzeniami w przestrzeni probabilistycznej**, to 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**.

Zbiory a listy

- Istotne różnice miedzy pojęciem zbiór $S = \{x_1, x_2, \ldots, x_n\}$ a listą $L = (x_1, x_2, \ldots, x_n)$:
  - Kolejność elementów w zbiorze jest nieistotna (a dla listy jest istotna).
  - Elementy należące do listy mogą się powtarzać (a dla zbioru nie mogą).

Implemetacja zbiorów na listach

- **Zbiory jako posortowane listy**
  - Operacje wykonujemy znacznie szybciej jeżeli elementy są posortowane. Za każdym razem porównujemy ze sobą tylko dwa elementy (po jednym z każdej listy).
  - Wyznaczenie sumy, przecięcia czy różnicy **zbiorów o rozmiarach m i n** wymaga czasu **O(m+n).**
  - Jeżeli listy nie były pierwotnie posortowane to sortowanie list zajmuje **O(m log m + n log n).**
  - Operacja ta może nie być szybsza niż **O(m n)** jeśli ilość elementów list jest bardzo różna.
  - **Jeżeli liczby m i n są porównywalne to O(m log m + n log n) < O(m n)**

Implementacja 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**.

Przykład z kartami

- Zbiór wszystkich kart koloru trefl

  1111111111110000000000000000000000000000000000
- Zbiór wszystkich figur

  000000000011100000000011100000000011100000000111
- Poker w kolorze kier (as, walet, dama, król)

  000000000000000000000000100000000011100000000000
- Każdy element zbioru kart jest związany z unikatową pozycją.

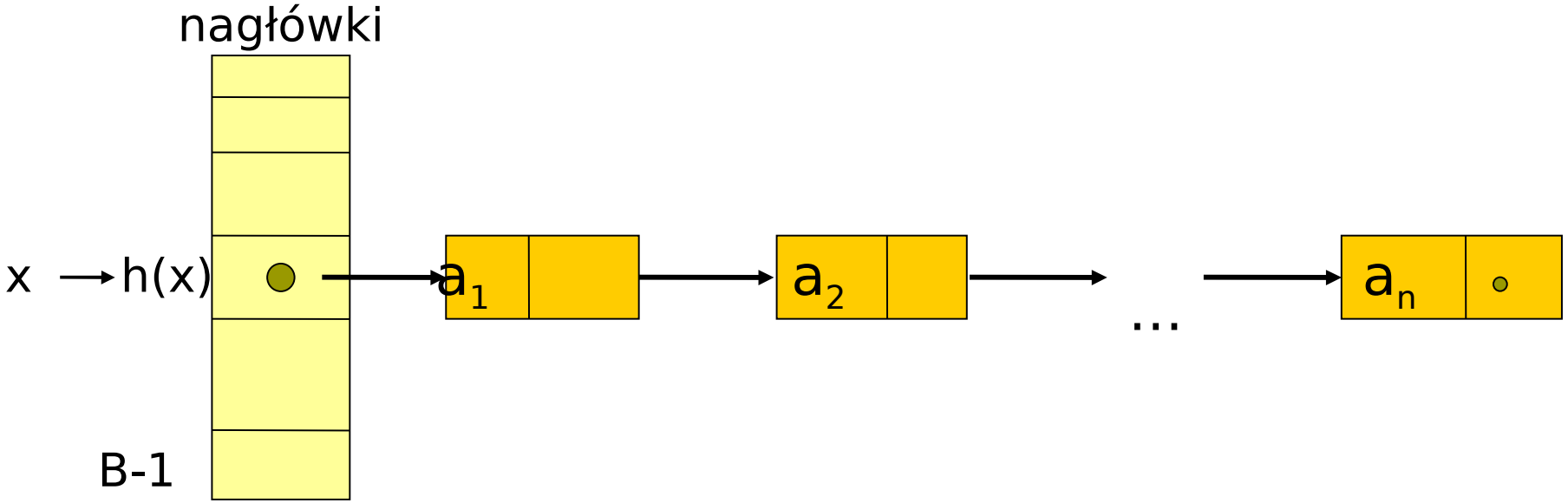
Implementacja oparta na wektorze własnym

- Czas potrzebny na wykonanie operacji sumy, przecięcia i różnicy jest ***O*(n)**.
- Jeśli przetwarzane zbiory są dużą częścią zbioru uniwersalnego to jest to dużo lepsze niż ***O*(m log m)** (posortowanie listy) lub ***O*(m²)** (nieposortowane listy).
- Jeśli **m << n** to jest to oczywiście nieefektywne.
- Ta implementacja również niepraktyczna jeżeli wymaga zbyt dużego **U**.

Struktura danych tablicy mieszającej

- Każda komórka składa się z listy jednokierunkowej, w której przechowujemy wszystkie elementy zbioru wysłanego do tej komórki przez funkcje mieszającą.
- Aby odnaleźć element **x** obliczamy wartość **h(x)**, która wskazuje na numer komórki.
- Jeśli tablica mieszająca zawiera element **x**, to możemy go znaleźć przeszukując listę która znajduje się w tej komórce.
- Tablica mieszająca pozwala na wykorzystanie reprezentacji zbiorów opartej na liście (wolne przeszukiwanie), ale dzięki podzieleniu zbioru na **B** komórek, czas przeszukiwania jest **~ 1/B** potrzebnego do przeszukiwania całego zbioru.
- W szczególności może być nawet ***O*(1)**, czyli taki jak w reprezentacji zbioru opartej na wektorze własnym.

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.

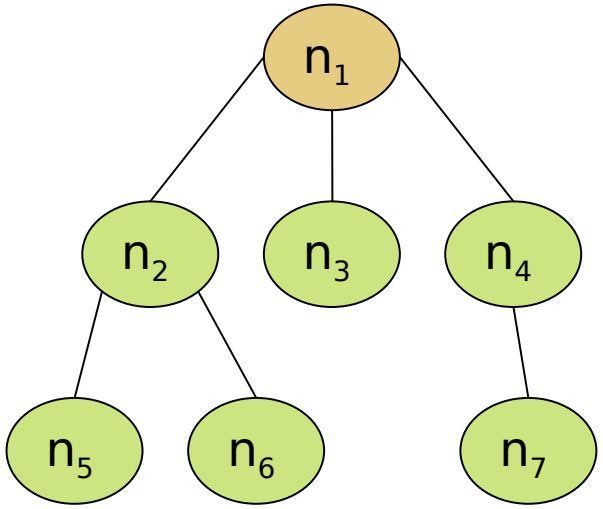
Drzewa

- 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.

Podstawowa terminologia

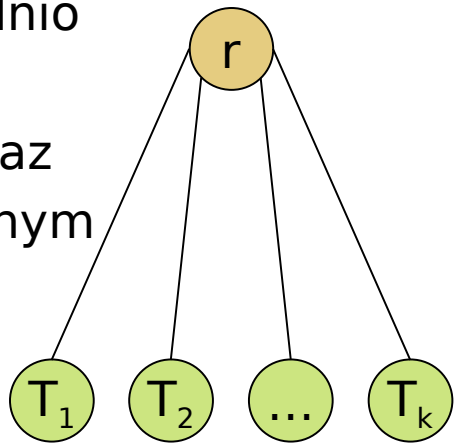
- **Drzewa** są zbiorami punktów, zwanych **węzłami** lub **wierzchołkami**, oraz połączeń, zwanych **krawędziami**.
- Krawędź łączy dwa różne węzły.
- Aby struktura zbudowana z węzłów połączonych krawędziami była drzewem musi spełniać pewne warunki:
  - W każdym drzewie wyróżniamy jeden węzeł zwany **korzeniem n_1** (ang. **root**)
  - Każdy węzeł **c** nie będący korzeniem jest połączony krawędzią z innym węzłem zwanym **rodzicem p** (ang. **parent**) węzła **c**. Węzeł **c** nazywamy także dzieckiem (ang. **child**) węzła **p**.
  - Każdy węzeł **c** nie będący korzeniem ma dokładnie jednego rodzica.
  - Każdy węzeł ma dowolną liczbę dzieci.
  - Drzewo jest **spójne** (ang. **connected**) w tym sensie że jeżeli rozpoczniemy analizę od dowolnego węzła **c** nie będącego korzeniem i przejdziemy do rodzica tego węzła, następnie do rodzica tego rodzica, itd., osiągniemy w końcu korzeń.



n_1 = rodzic n_2, n_3, n_4
n_2 = rodzic n_5, n_6
n_6 = dziecko n_2

Rekurencyjna definicja drzew

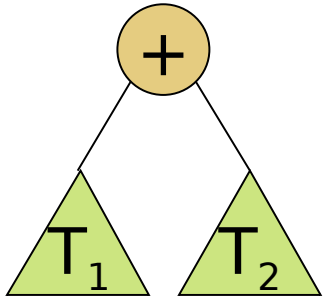
- **Podstawa:** Pojedynczy węzeł n jest drzewem. Mówimy że n jest korzeniem drzewa złożonego z jednego węzła.
- **Indukcja:** Niech r będzie nowym węzłem oraz niech T_1, T_2, …, T_k będą drzewami zawierającymi odpowiednio korzenie c_1, c_2, …, c_k. Załóżmy że żaden węzeł nie występuje więcej niż raz w drzewach T_1, T_2, …, 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, …, 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, …, 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, …, T_k.



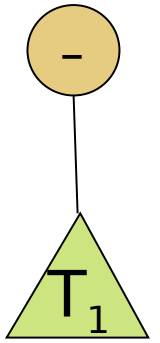
Drzewa zaetykietowane i drzewa wyrażeń.

- **Drzewo zaetykietowane** to takie w którym z każdym węzłem drzewa związana jest jakaś etykieta lub wartość. Możemy reprezentować wyrażenia matematyczne za pomocą drzew zaetykietowanych.
- Definicja drzewa zaetykietowanego dla wyrażeń arytmetycznych zawierających operandy dwuargumentowe +,-,•,/ oraz operator jednoargumentowy -.
  - **Podstawa:** Pojedynczy operand niepodzielny jest wyrażeniem.

    Reprezentujące go drzewo składa się z pojedynczego węzła, którego etykietą jest ten operand.
  - **Indukcja:** Jeśli $\mathbf{E_1}$ oraz $\mathbf{E_2}$ są wyrażeniami reprezentowanymi odpowiednio przez drzewa $\mathbf{T_1}$, $\mathbf{T_2}$, wyrażenie $\mathbf{(E_1 + E_2)}$ reprezentowane jest przez drzewo którego korzeniem jest węzeł o etykiecie **+**. Korzeń ten ma dwoje dzieci, którego korzeniami są odpowiednio korzenie drzew $\mathbf{T_1}$, $\mathbf{T_2}$.

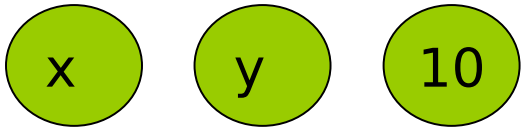


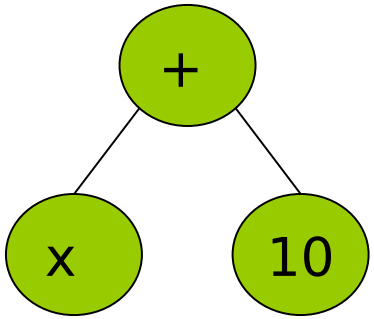
$E_1 + E_2$



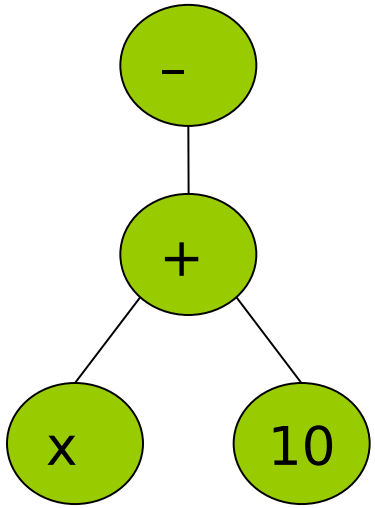
$(- E_1)$

Konstrukcja drzew wyrażeń

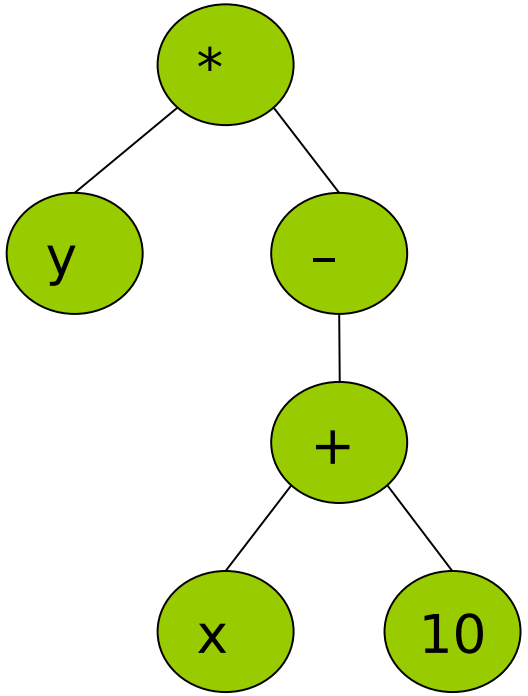




(x + 10)



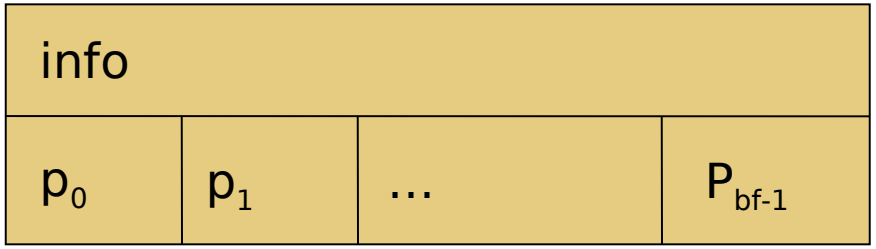
(- (x + 10))



(y * - (x + 10))

Reprezentacja tablicowa

- Jednym z najprostszych sposobów reprezentowania drzewa jest wykorzystanie dla każdego węzła struktury składającej się z pola lub pól reprezentujących etykietę oraz tablicy wskaźników do dzieci tego węzła.



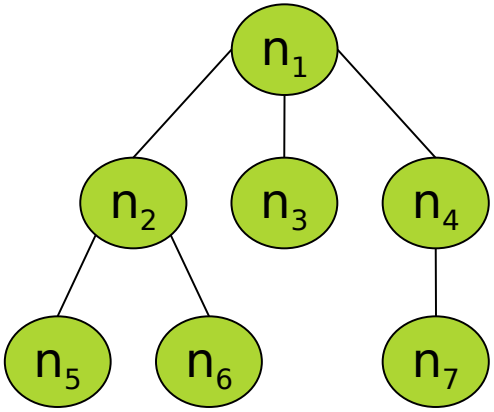
- **Info** reprezentuje **etykietę węzła**.
- Stała **bf** jest **rozmiarem tablicy wskaźników.** Reprezentuje maksymalną liczbę dzieci dowolnego węzła, czyli **czynnik rozgałęzienia** (ang. **branching factor**).
- **i**-ty element tablicy reprezentującej węzeł zawiera wskaźnik do **i**-tego dziecka tego węzła.
- Brakujące połączenia możemy reprezentować za pomocą wskaźnika pustego **NULL**.

Reprezentacje drzewa

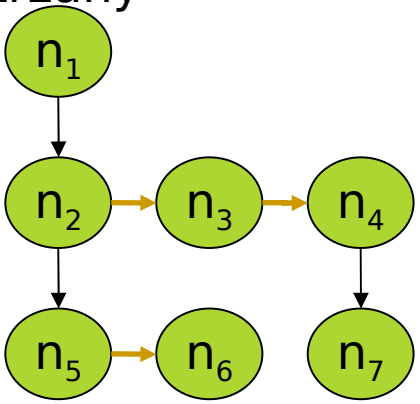
- W reprezentacji drzew zwanej **skrajnie lewy potomek-prawy element siostrzany** (ang. left-most-child-right-sibling), w każdym węźle umieszczamy jedynie wskaźniki do skrajnie lewego dziecka; węzeł nie zawiera wskaźników do żadnego ze swoich pozostałych dzieci.
- Aby odnaleźć drugi i wszystkie kolejne dzieci węzła **n**, tworzymy listę jednokierunkowa tych dzieci w której każde dziecko **c** wskazuje na znajdujące się bezpośrednio po jego prawej stronie dziecko węzła **n**.
- Wskazany węzeł nazywamy **prawym elementem siostrzanym** węzła **c**.

Reprezentacje drzewa

Drzewo złożone z 7 węzłów



Reprezentacja skrajnie lewy potomek-prawy element siostrzany



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

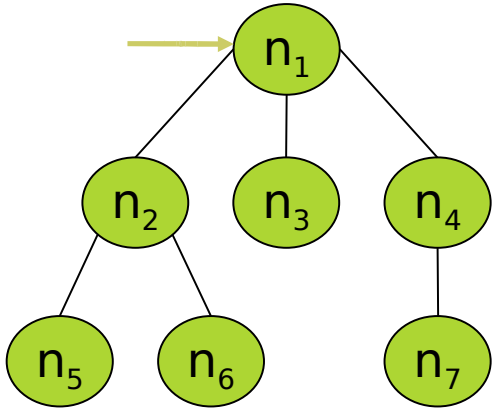
```
typedef struct NODE *pNODE;
struct NODE{
        int info;
        pNODE leftmostChild, rightSibling;
};
```

Reprezentacje drzewa

- **Reprezentacja oparta na tablicy wskaźników** umożliwia nam dostęp do **i-tego dziecka** dowolnego węzła w czasie ***O*(1)**. Taka reprezentacja wiąże się jednak ze znacznym marnotrawstwem przestrzeni pamięciowej, jeśli tylko kilka węzłów ma wiele dzieci. W takim wypadku większość wskaźników w tablicy **children** będzie równa **NULL**.
- **Reprezentacja skrajnie lewy potomek-prawy element siostrzany** wymaga mniejszej przestrzeni pamięciowej. Nie wymaga również istnienia maksymalnego czynnika rozgałęzienie węzłów. Możemy reprezentować węzły z dowolna wartością tego czynnika, nie modyfikując jednocześnie struktury danych.

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.

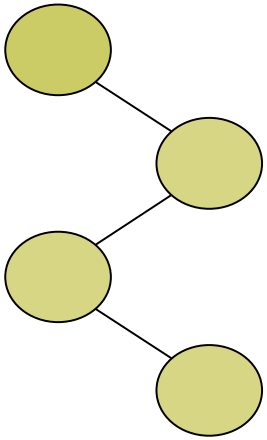


Drzewa binarne

- W drzewie binarnym węzeł może mieć co najwyżej dwoje bezpośrednich potomków.
- **Rekurencyjna definicja** drzewa binarnego:
- **Podstawa:**
  - Drzewo puste jest drzewem binarnym.
- **Indukcja:**
  - Jeśli **r** jest węzłem oraz $\mathbf{T_1}$, $\mathbf{T_2}$ są drzewami binarnymi, istnieje drzewo binarne z korzeniem **r**, lewym poddrzewem $\mathbf{T_1}$ i prawym poddrzewem $\mathbf{T_2}$. Korzeń drzewa $\mathbf{T_1}$ jest lewym dzieckiem węzła **r**, chyba że $\mathbf{T_1}$ jest drzewem pustym. Podobnie korzeń drzewa $\mathbf{T_2}$ jest prawym dzieckiem węzła **r**, chyba że $\mathbf{T_2}$ jest drzewem pustym.
  - Większość terminologii wprowadzonej przy okazji drzew stosuje się oczywiście też do drzew binarnych.
- **Różnica:** drzewa binarne wymagają rozróżnienia lewego od prawego dziecka, zwykle drzewa tego nie wymagają. Drzewa binarne to **NIE** są zwykle drzewa, w których węzły mogą mieć co najwyżej dwójkę dzieci.

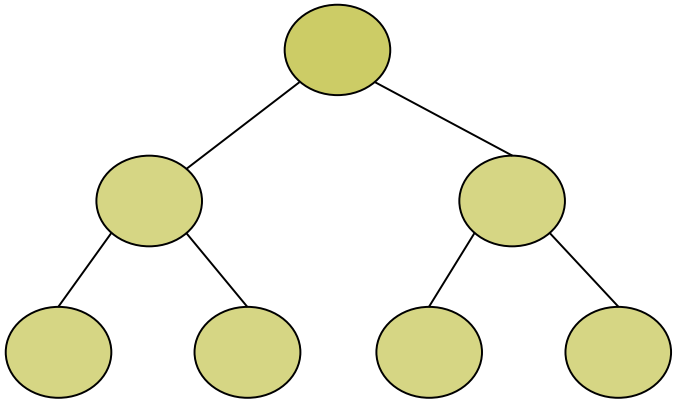
Drzewa binarne

Zdegenerowane drzewo binarne



Wysokość drzewa złożonego z k-węzłów to k-1.
Czyli h = *O*(k).
Operacje insert, delete, find wymagają średnio *O*(k).

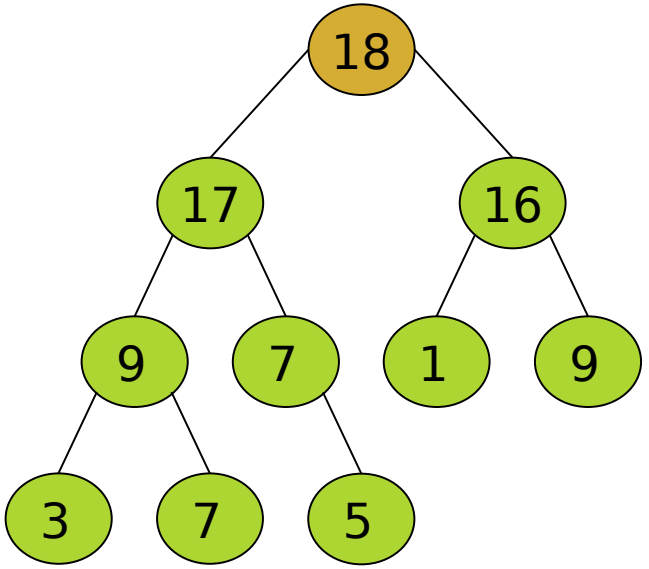
Pełne drzewo binarne



Drzewo o wysokości h ma $k=2^{h+1}-1$ węzłów.
Czyli h = *O*(log k).
Operacje insert, delete, find wymagają średnio *O*(log k).

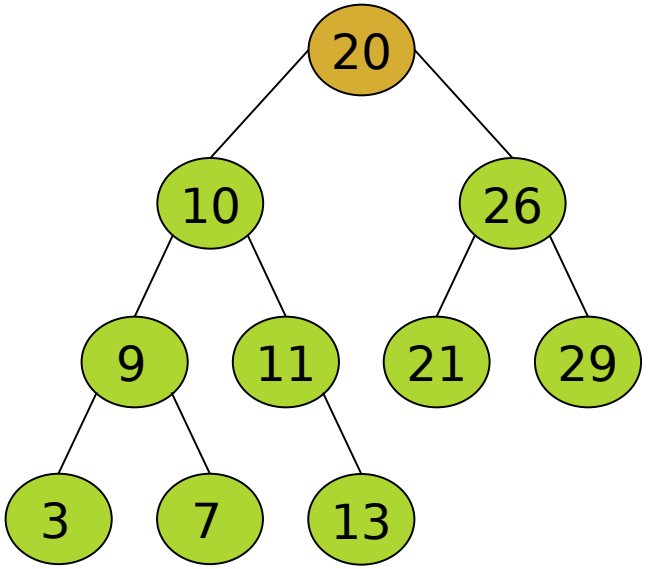
Drzewa binarne częściowo uporządkowane

- Jest to **zaetykietowane drzewo binarne** o następujących własnościach:
  - Etykietami węzłów są elementy z przypisanymi priorytetami; priorytet może być wartością elementu lub przynajmniej jednego z jego komponentów.
  - Element przechowywany w węźle musi mieć co najmniej tak duży priorytet jak element znajdujący się w dzieciach tego węzła. Element znajdujący się w korzeniu dowolnego poddrzewa jest wiec największym elementem tego poddrzewa.



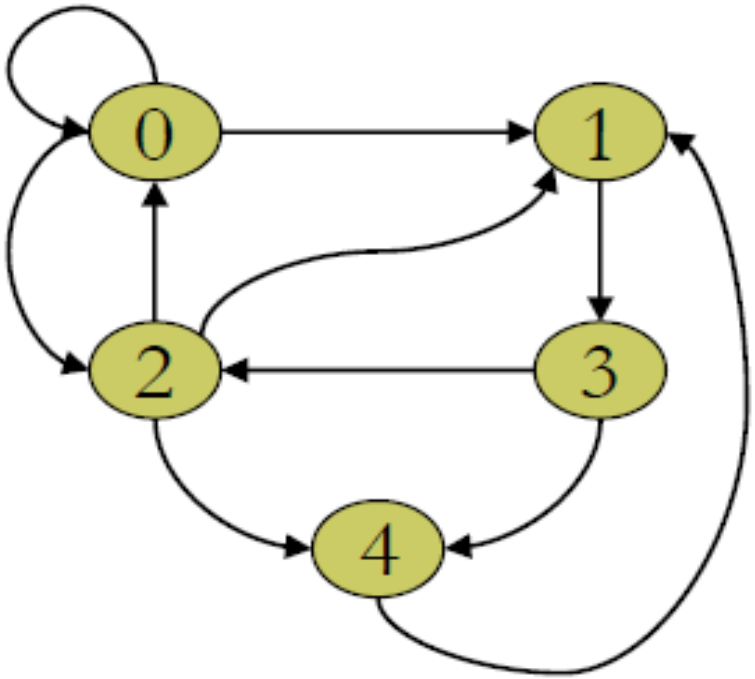
Drzewa przeszukiwania binarnego

- 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 własności**:
  - wszystkie węzły w lewym poddrzewie maja etykiety mniejsze od etykiety węzła **x**
  - wszystkie w prawym poddrzewie maja etykiety większe od etykiety węzła **x**.



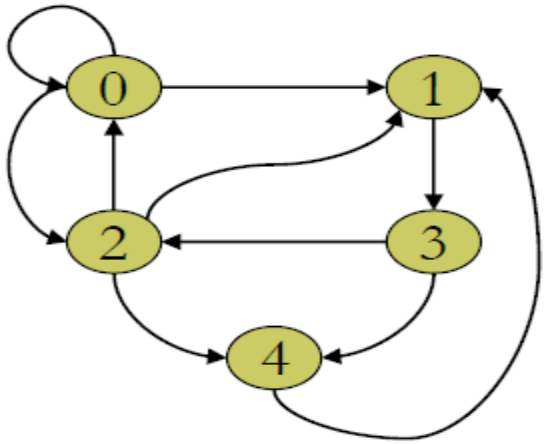
Graf

- **Graf to jest relacja binarna.**



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)**.



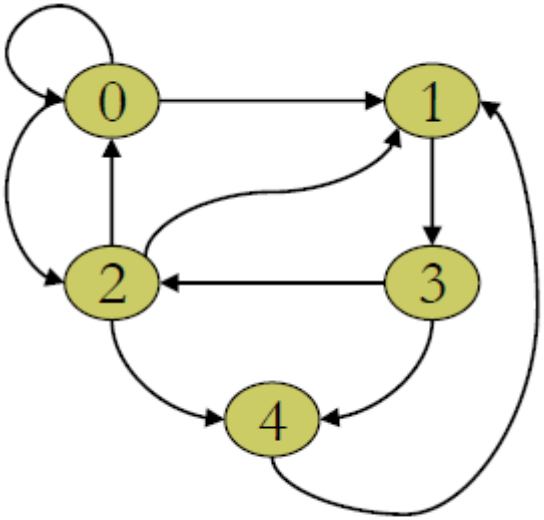
V = {0,1,2,3,4}
E = { (0,0), (0,1), (0,2), (1,3),
(2,0), (2,1), (2,4), (3,2),
(3,4), (4,1) }

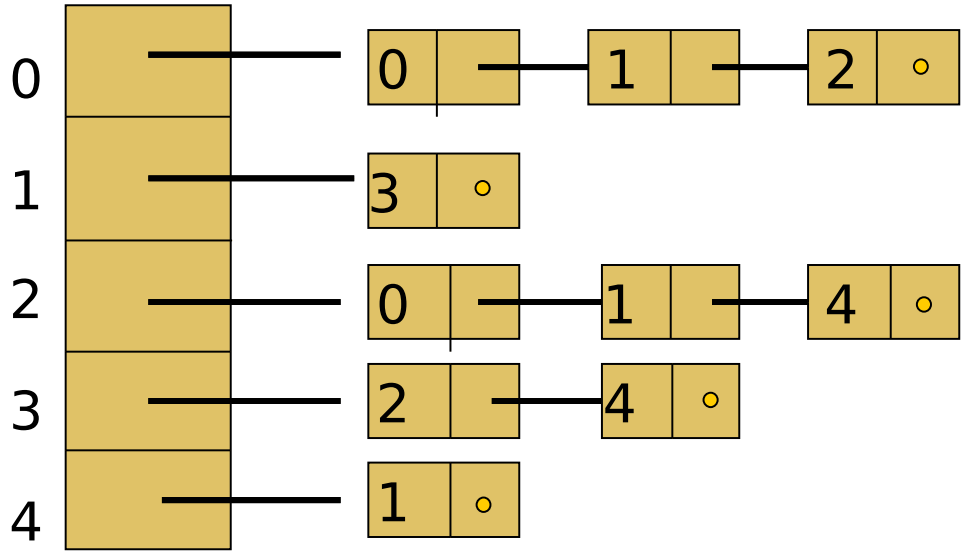
Sposoby implementacji grafów

- 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.

Reprezentacja grafu za pomocą list sąsiedztwa

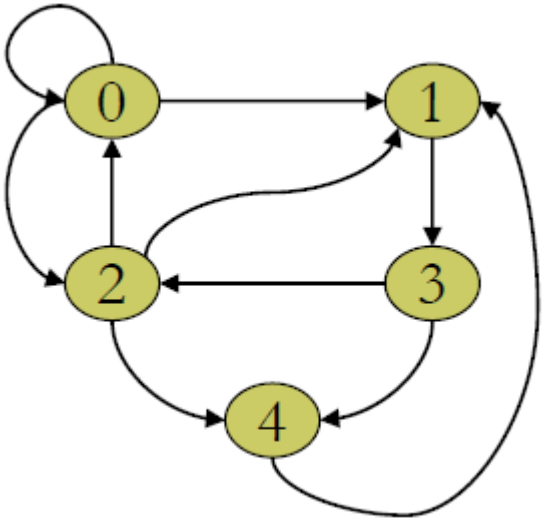
- Listy sąsiedztwa zostały posortowane wg. kolejności, ale następniki mogą występować w **dowolnej kolejności** na odpowiedniej liście sąsiedztwa.





Reprezentacja grafu za pomocą macierzy sąsiedztwa

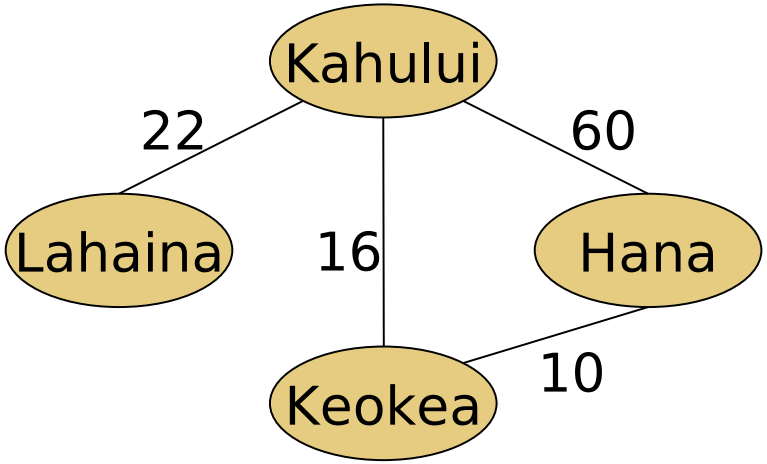
- Tworzymy dwuwymiarową tablicę;
  BOOLEAN vertices[MAX][MAX];
  w której element **vertices[u][v]** ma wartość **TRUE** wówczas, gdy istnieje krawędź **(u, v)**, zaś **FALSE**, w przeciwnym przypadku.



	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

Spójna składowa grafu nieskierowanego

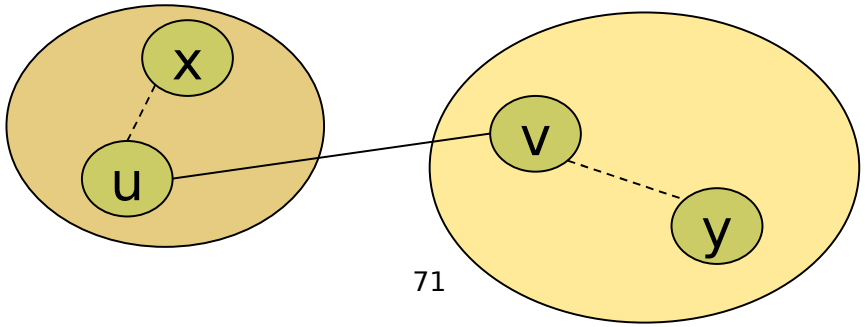
- Każdy graf nieskierowany można podzielić na jedna lub większą liczbę **spójnych składowych** (ang. *connected components*).
- Każda spójna składowa to taki zbiór wierzchołków, że dla każdych dwóch z tych wierzchołków istnieje łącząca je ścieżka. Jeżeli graf składa się z jednej spójnej składowej to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

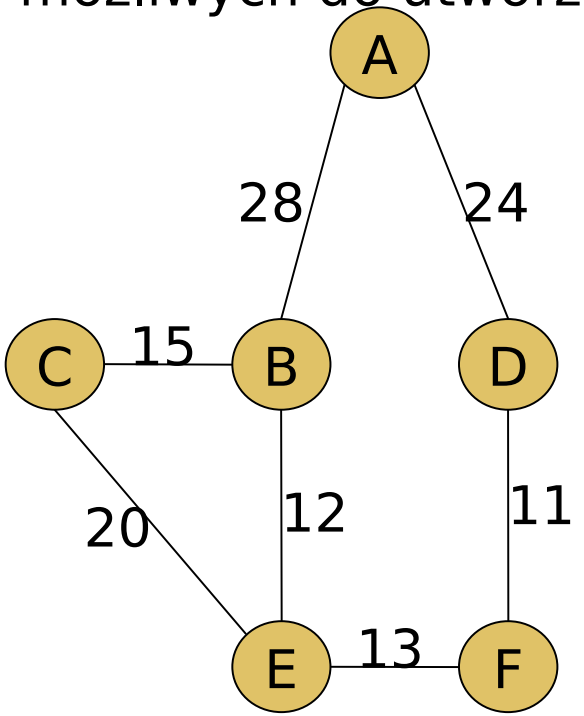


Minimalne drzewa rozpinające

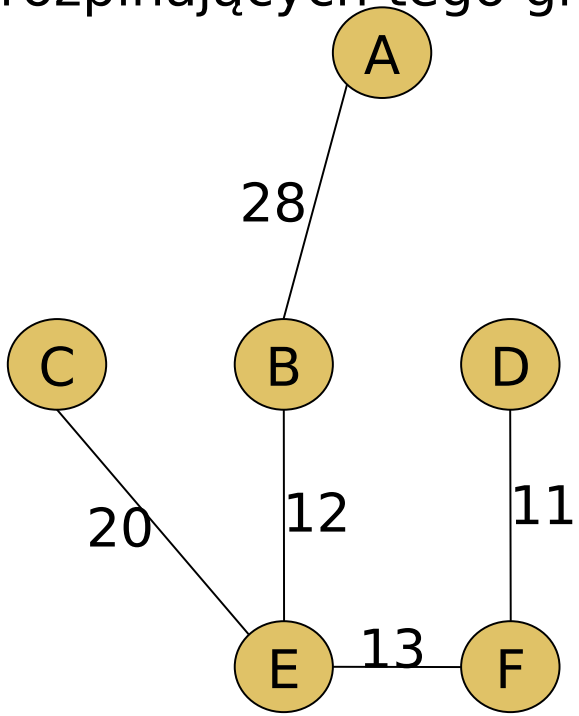
- **Drzewo rozpinające** (ang. *spanning tree*) grafu nieskierowanego **G** stanowi **zbiór wierzchołków tego grafu** wraz z podzbiorem jego krawędzi, takich że:
  - **łączą one wszystkie wierzchołki**, czyli istnieje droga miedzy dwoma dowolnymi wierzchołkami która składa się tylko z krawędzi drzewa rozpinającego.
  - **tworzą one drzewo nie posiadające korzenia**, nieuporządkowane. Oznacza to że nie istnieją żadne (proste) cykle.
- Jeśli graf **G** stanowi pojedynczą spójną składową to drzewo rozpinające zawsze istnieje.

Minimalne drzewa rozpinające

- **Minimalne drzewo rozpinające** (ang. *minimal spanning tree*) to drzewo rozpinające, w którym **suma etykiet jego krawędzi jest najmniejsza** ze wszystkich możliwych do utworzenia drzew rozpinających tego grafu.



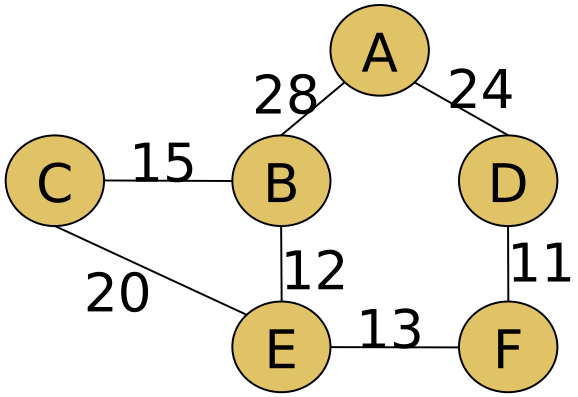
Graf nieskierowany



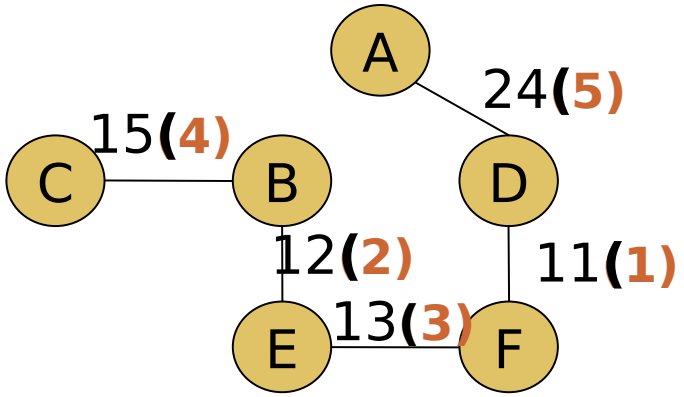
Drzewo rozpinające

Algorytm Kruskala

- Najbardziej znany 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



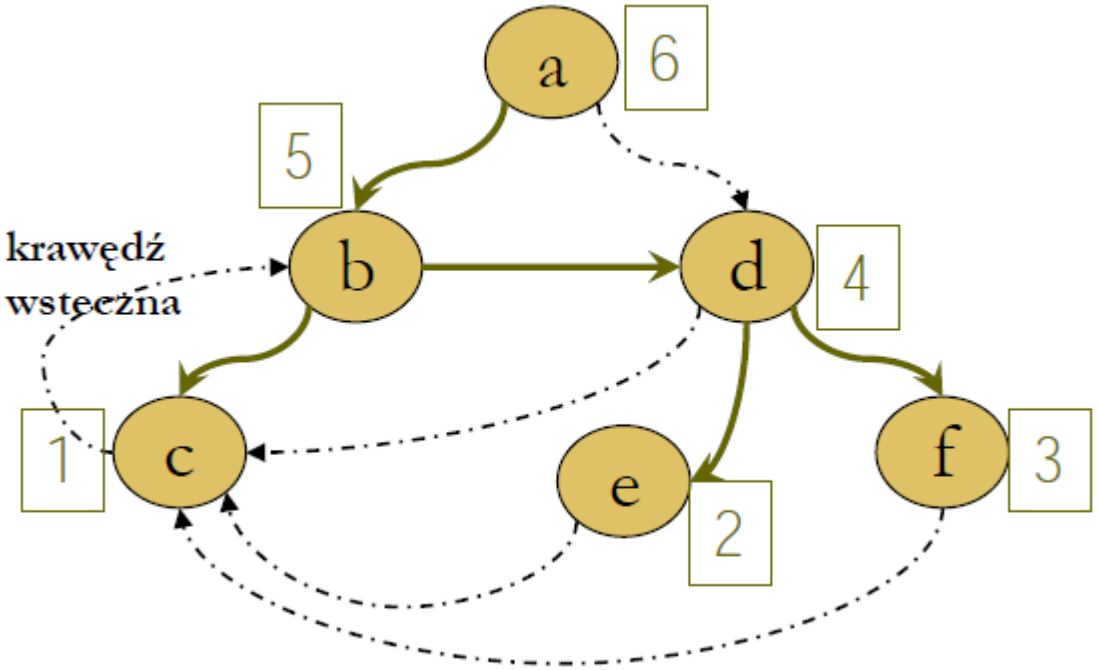
Minimalne drzewo rozpinające

(w nawiasach podano kolejność dodawanych krawędzi)

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 ze w grafie mogą pojawiać się cykle... Należy wobec tego **znaczyć wierzchołki już odwiedzone i nie wracać więcej** do takich wierzchołków.
- W celu uniknięcia dwukrotnego odwiedzenia tego samego wierzchołka jest on odpowiednio oznaczany więc **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.

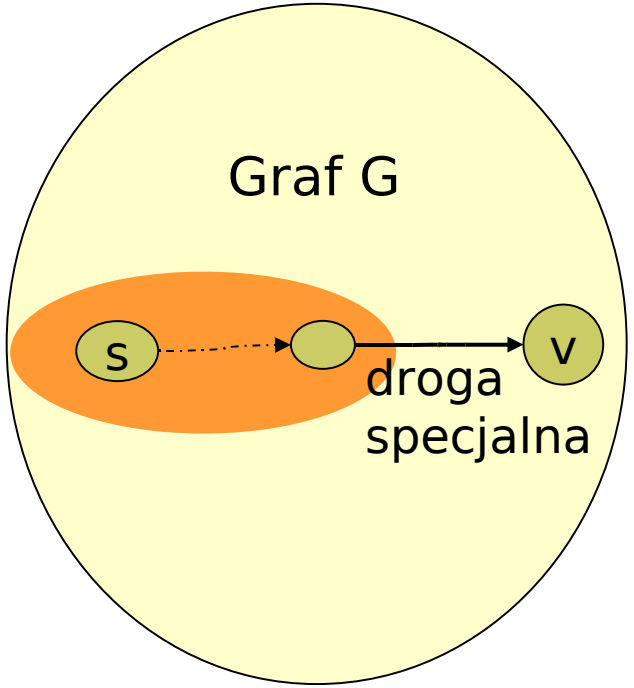
Drzewo przeszukiwania w głąb



- Po (podczas) konstruowaniu drzewa przeszukiwania w głąb można ponumerować jego wierzchołki w **kolejności wstecznej** (ang. *post-order*).

Algorytm Dijkstry

- 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 Dijkstry

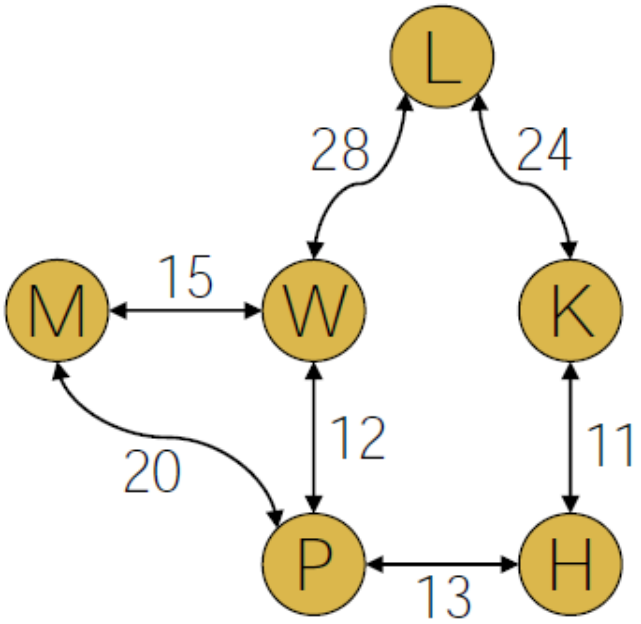
- Dla każdego wierzchołka u zapamiętujemy wartość **dist(u)**. Jeśli **u** jest wierzchołkiem ustalonym, to **dist(u)** jest długością najkrótszej drogi ze źródła do wierzchołka **u**. Jeśli **u** nie jest wierzchołkiem ustalonym, to **dist(u)** jest długością drogi specjalnej ze źródła do **u**.
- Na czym polega **ustalanie wierzchołków**:
  - znajdujemy wierzchołek **v** który jest nieustalony ale posiada najmniejszą **dist(v)** ze wszystkich wierzchołków nieustalonych
  - przyjmujemy wartość **dist(v)** za minimalną odległość z **s** do **v**
  - dostosowujemy wartości wszystkich **dist(u)** dla innych wierzchołków, które nie są ustalone, wykorzystując fakt, że wierzchołek **v** jest już ustalony.
  - Czyli porównujemy stare **dist(u)** z wartością **dist(v) +etykieta(v, u)** jeżeli taka **(v, u)** krawędź istnieje.

Czas wykonania algorytmu jest ***O*(m log n)**.

Algorytm Dijkstry

Etapy wykonania algorytmu Dijkstry

	ETAPY ustalania wierzchołków				
MIASTO	(1)	(2)	(3)	(4)	(5)
H	0*	0*	0*	0*	0*
P	13	13	13*	13*	13*
M	INF	INF	33	33	33*
W	INF	INF	25	25*	25*
L	INF	35	35	35	35
K	11	11*	11*	11*	11*



Algorytmy grafowe

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	$O(n^3)$