TEORETYCZNE PODSTAWY INFORMATYKI:

POWTÓRKA CZ. I

20/01/2014

WFAiS UJ, Informatyka Stosowana
I rok studiów, I stopień

Informatyka

- Zasadniczo 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.

Informatyka:

W ramach tego wykładu omówiliśmy

- **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

Wykład 1:

Informacja i zasady jej zapisu

- Czym jest informacja?
  - Bity i Bajty
  - Kodowanie informacji
- Systemy zapisu liczb
  - System binarny (dwójkowy)
  - Sposoby kodowania: liczb naturalnych, całkowitych, rzeczywistych
  - Dlaczego pojawiają się błędy i zaokrąglenia
- Znaki i teksty
- Obrazy i dźwięki
- Kompresja i szyfrowanie

Wykład 2

Struktury danych i algorytmy

- Analiza algorytmów
- Typy danych i struktury danych
- Sposoby zapisu algorytmów
- Rodzaje algorytmów
- Schematy blokowe i algografy
- Wybór algorytmu

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.

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.

Algorytmy

- **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”.
- 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.
- Bardzo wygodny zapis to **zapis graficzny**
  - schematy blokowe i grafy.

Rodzaje algorytmów

Algorytm liniowy:

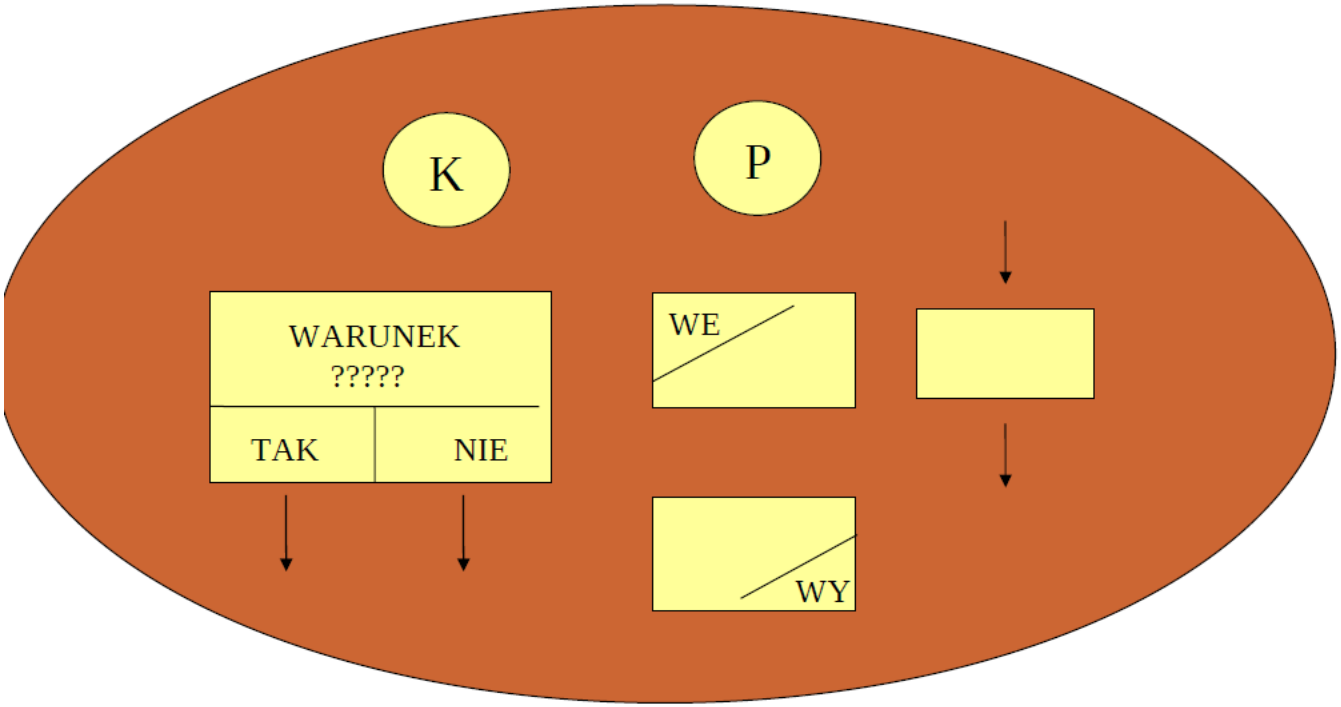
- Ma postać ciągu kroków których jest **liniowa ilość (np. stała albo proporcjonalna do liczby danych)** które muszą zostać bezwarunkowo wykonane jeden po drugim.
- Algorytm taki **nie zawiera żadnych warunków ani rozgałęzień**: zaczyna się od podania zestawu danych, następnie wykonywane są kolejne kroki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

Rodzaje algorytmów

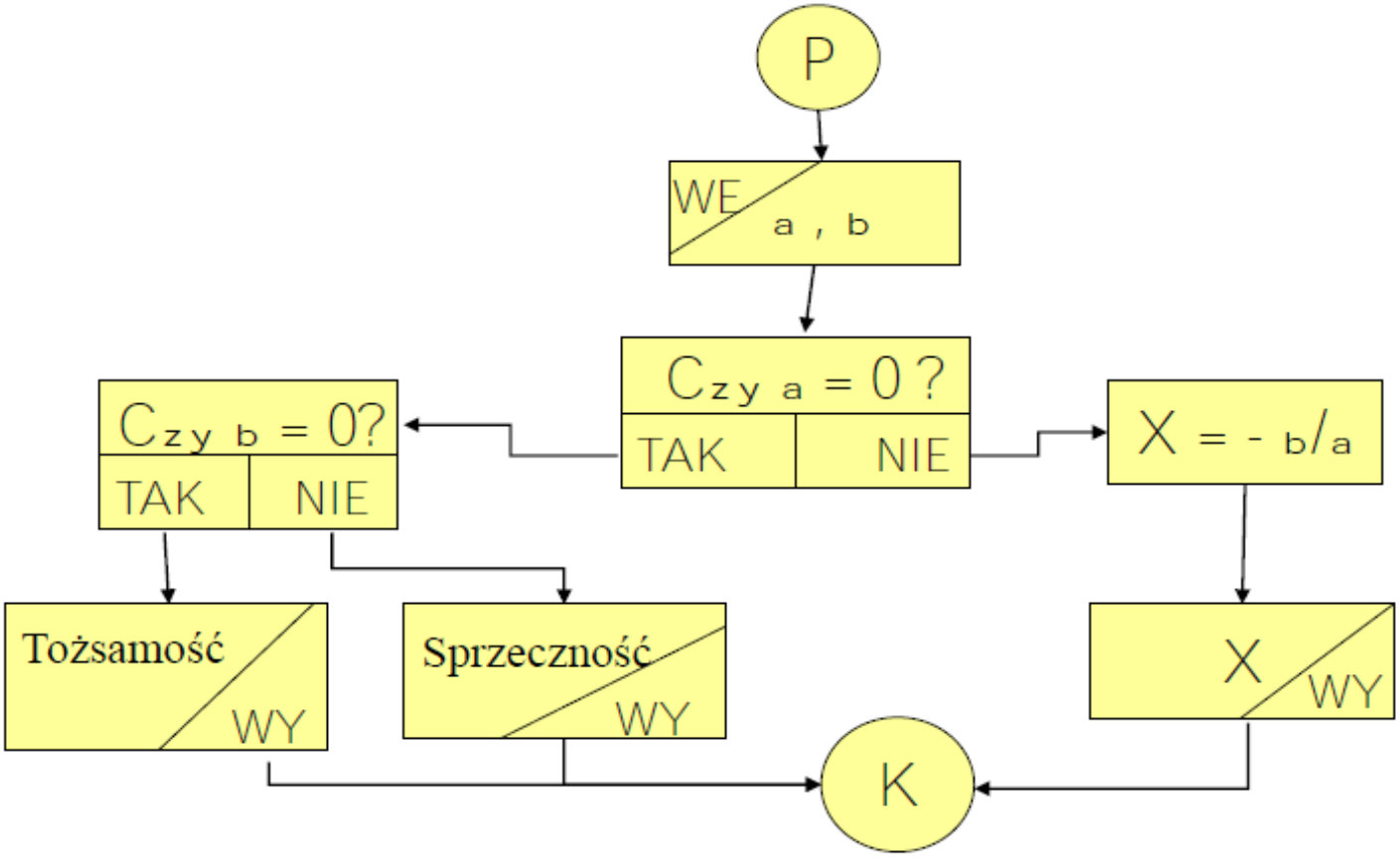
Algorytm z rozgałęzieniem:

- 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.

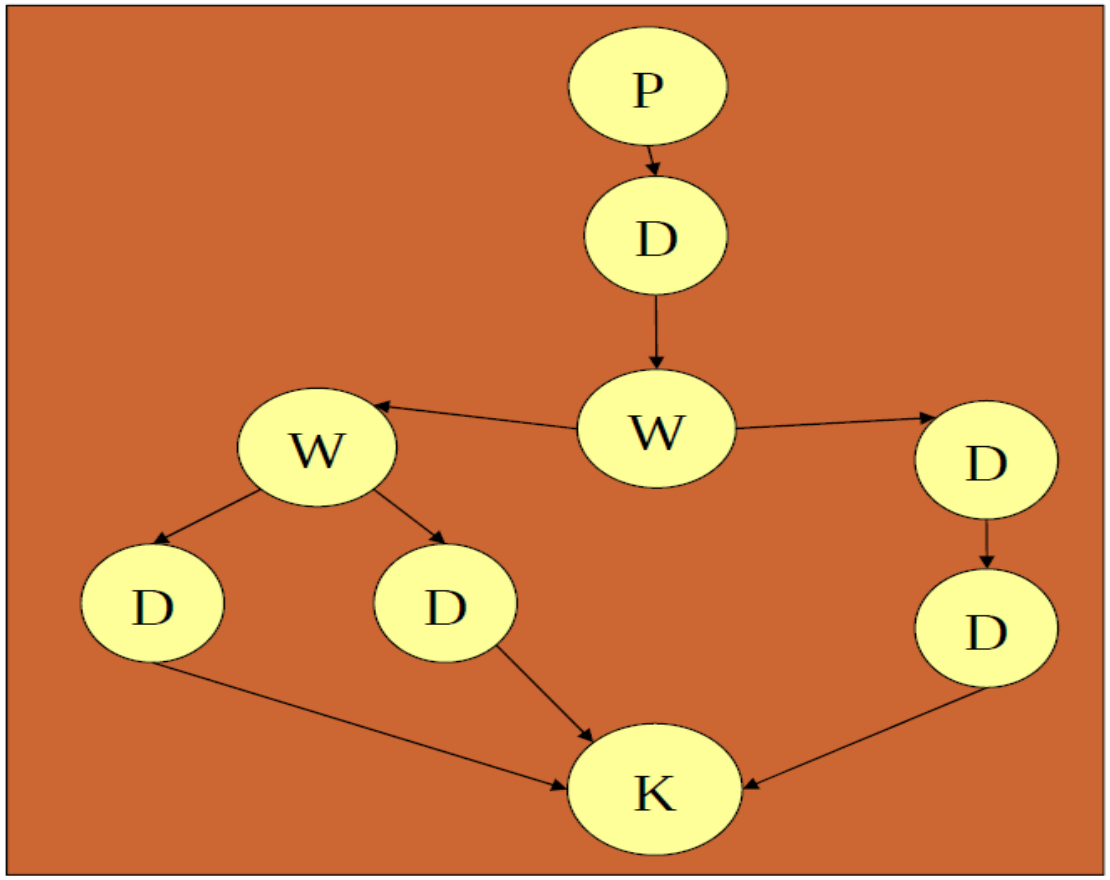
Schematy blokowe i algografy



Schemat blokowy rozwiązania równania liniowego



Graf algorytmu rozwiązania równania liniowego



- P – początek
- K – koniec
- D – działanie
- W – warunek

Schemat blokowy czy graf ?

- **Graf** to tylko **schemat kontrolny** służący do **sprawdzenia algorytmu**.
  - Brak informacji o wykonywanych operacjach
- **Schemat blokowy** służy jako **podstawa do tworzenia programów**.

Algorytmy: „dziel i zwyciężaj”

- Metoda: „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.
- Ten typ algorytmów zazwyczaj jest implementowany z zastosowaniem **technik rekurencyjnych**.

Algorytmy: „dziel i zwyciężaj”

- **Jak znaleźć minimum ciągu liczb?**
  - Dzielimy ciąg na dwie części, znajdujemy minimum w każdej z nich, bierzemy minimum z obu liczb jako minimum ciągu.

- **Jak sortować ciąg liczb?**
  - Dzielimy na dwie części, każdą osobno sortujemy a następnie łączymy dwa uporządkowane ciągi (scalamy).

Algorytmy oparte na programowaniu dynamicznym

- Można stosować wówczas, kiedy problem daje się podzielić na wiele pod-problemów, których rozwiązania są możliwe 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 obliczeń rośnie wykładniczo.
- **Korzystnie jest więc zachować (zapamiętać w tablicy) wyniki wcześniejszych obliczeń (F(k)).**

Algorytmy z powrotami

- Często możemy zdefiniować jakiś problem jako poszukiwanie rozwiązania wśród wielu możliwych przypadków.
- Dane:
  - Pewna przestrzeń stanów, przy czym stan jest to sytuacją stanowiąca rozwiązanie problemu albo mogąca prowadzić do rozwiązania
  - Sposób przechodzenia z jednego stanu do drugiego.
  - Mogą istnieć stany które nie prowadzą do rozwiązania.

Przykładami tego typu algorytmów są gry.

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.**

Wybór algorytmu

- 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
  - mniej obliczeń to lepsza dokładność numeryczna (zaokrąglenia)

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 „dzielenie i scalanie” (trudniejszy, dużo efektywniejszy).
- Zrozumiałość to pojęcie względne, natomiast **efektywność** można obiektywnie zmierzyć**: testy wzorcowe, analiza złożoności obliczeń.**

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.

Wykład 3

Złożoność obliczeniowa algorytmów

- Notacja *„wielkie 0”*
- Notacja Ω i Θ
- *Algorytm Hornera*
- *Przykłady rzędów złożoności*
- *Klasy złożoności algorytmów*
- *Funkcje niewspółmierne*
- *Analiza czasu działania algorytmu*
  - *Instrukcje proste; instrukcje warunkowe; bloki instrukcji*
- *Efektywność algorytmu*

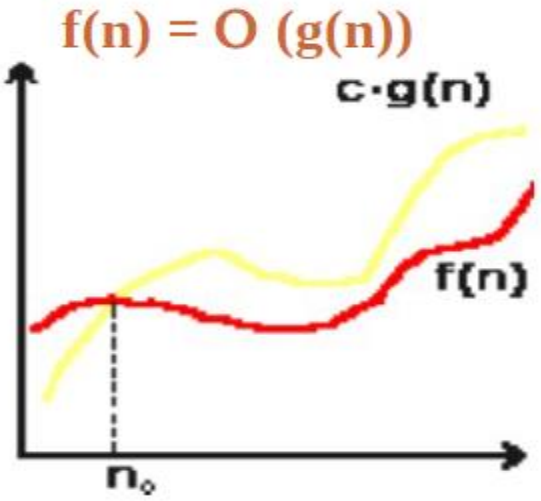
Złożoność obliczeniowa

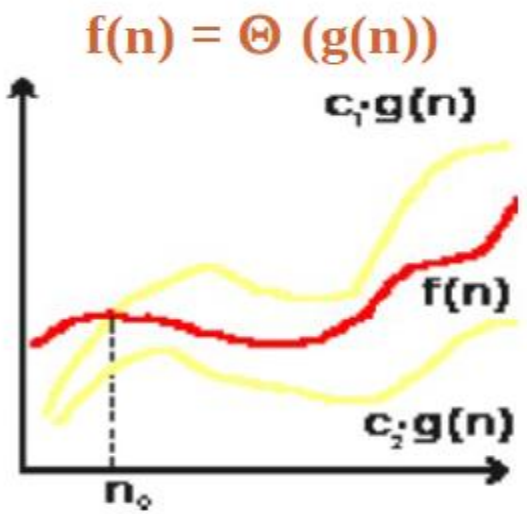
- **Złożoność obliczeniowa**:
  - Jest to miara służąca do porównywania efektywności algorytmów.
  - Mamy dwa kryteria efektywności:
    - Czas,
    - Pamięć
- Do oceny efektywności stosujemy jednostki logiczne wyrażające związek miedzy rozmiarem danych N (wielkość pliku lub tablicy) a ilością czasu T potrzebną na ich przetworzenie.

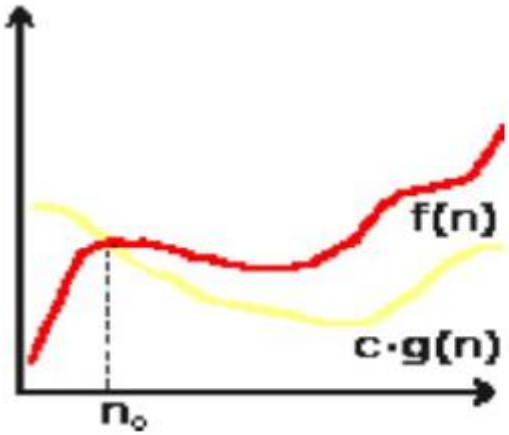
Złożoność asymptotyczna

- 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 O, Ω i Θ







$f(n) = \Omega\ (g(n))$

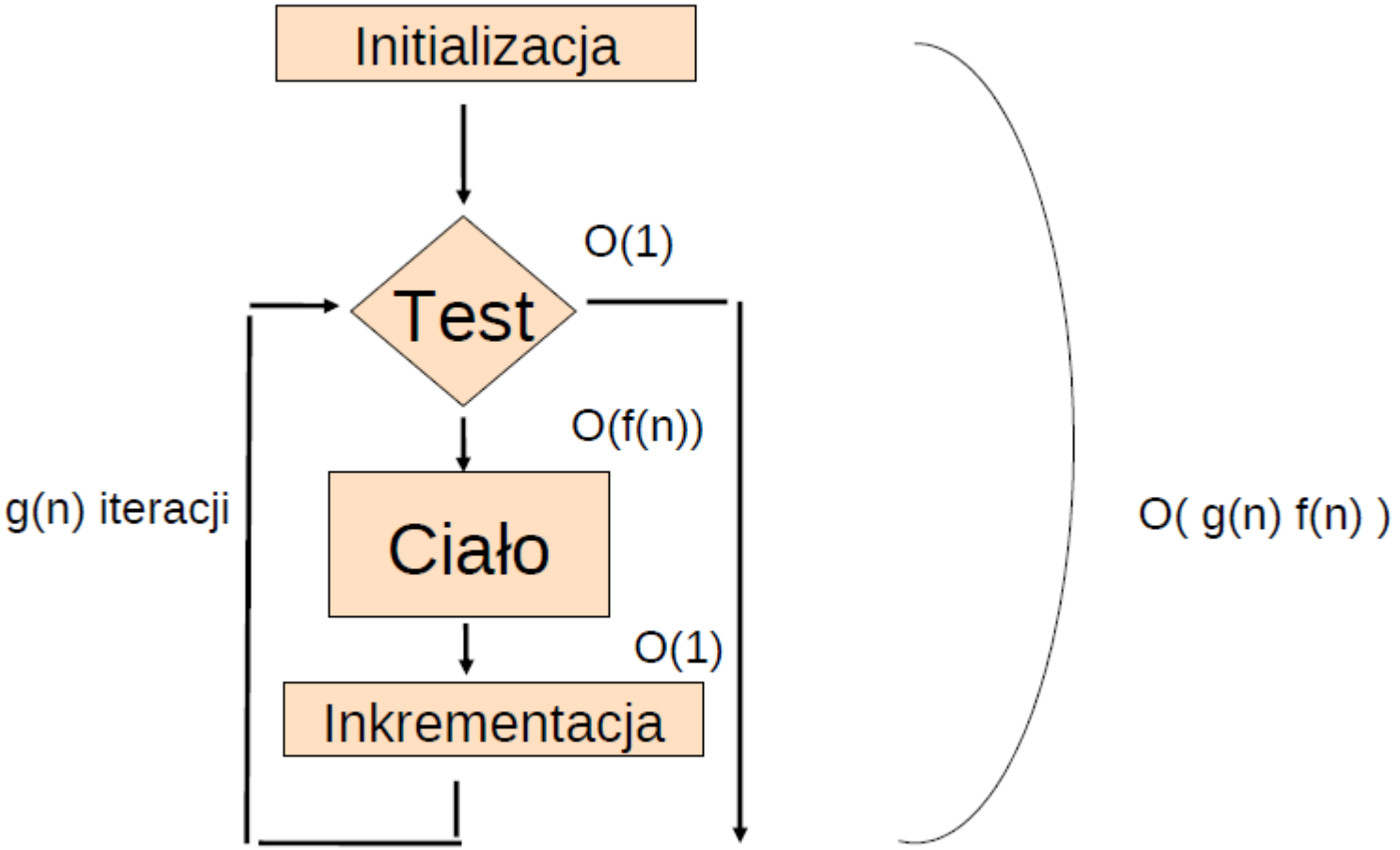
Przykłady rzędów złożoności

- 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 $O(n^2)$.
  - **Algorytm logarytmiczny:** czas wykonania wynosi $O(\log n)$.
  - itd …
- **Analiza złożoności algorytmów** jest niezmiernie istotna i nie można jej lekceważyć argumentując potencjalną szybkością obliczeń komputera. Nie sposób jej przecenić szczególnie zastanawiając się nad doborem struktury danych.

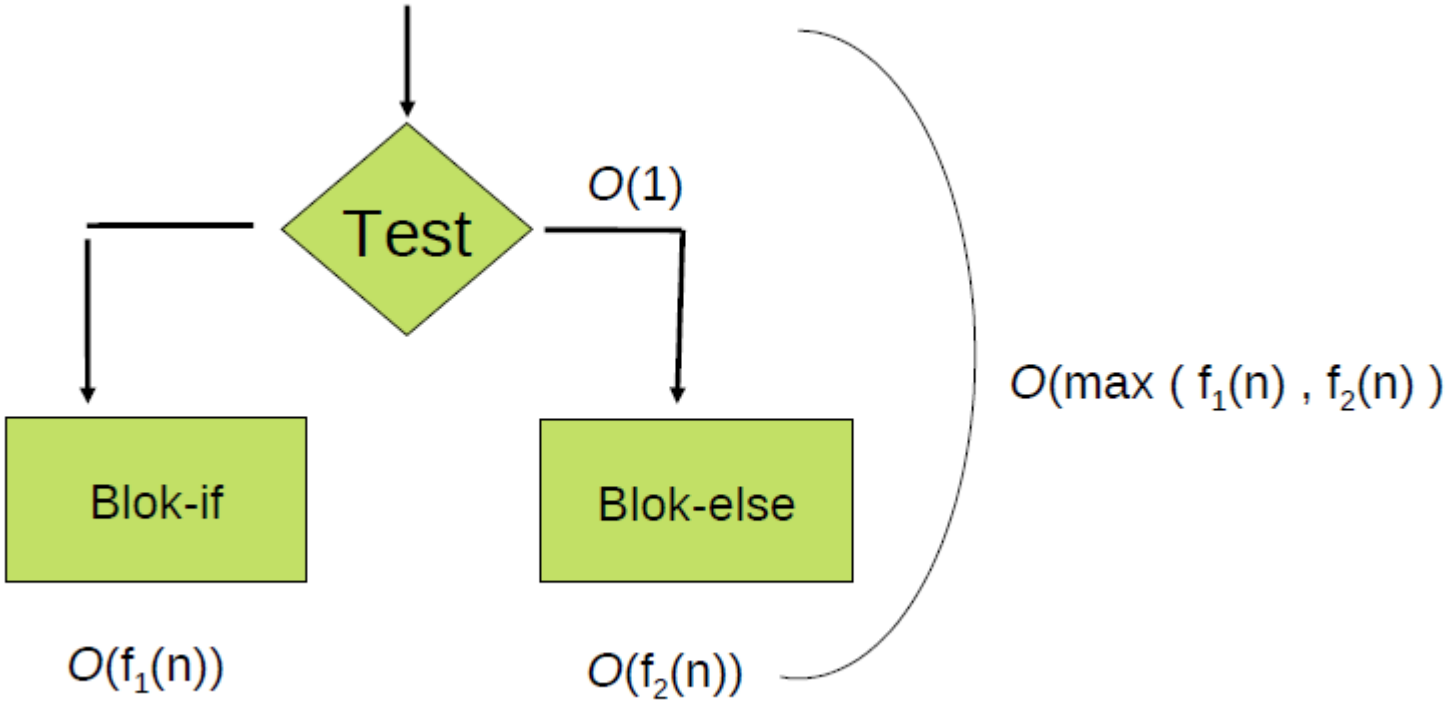
Czas działania instrukcji prostych

- Przyjmujemy zasadę że czas działania pewnych prosty operacji na danych wynosi **O(1)**, czyli jest niezależny od rozmiaru danych wejściowych.
  - **Operacje arytmetyczne**, np. (+), (-)
  - **Operacje logiczne** (&&)
  - **Operacje porównania** (<=)
  - **Operacje dostępu do struktur danych**, np. indeksowanie tablic (A[i])
  - **Proste przypisania**, np. kopiowanie wartości do zmiennej.
  - Wywołania **funkcji bibliotecznych**, np. **scanf** lub **printf**
- Każdą z tych operacji można wykonać za pomocą pewnej (niewielkiej) liczby rozkazów maszynowych.

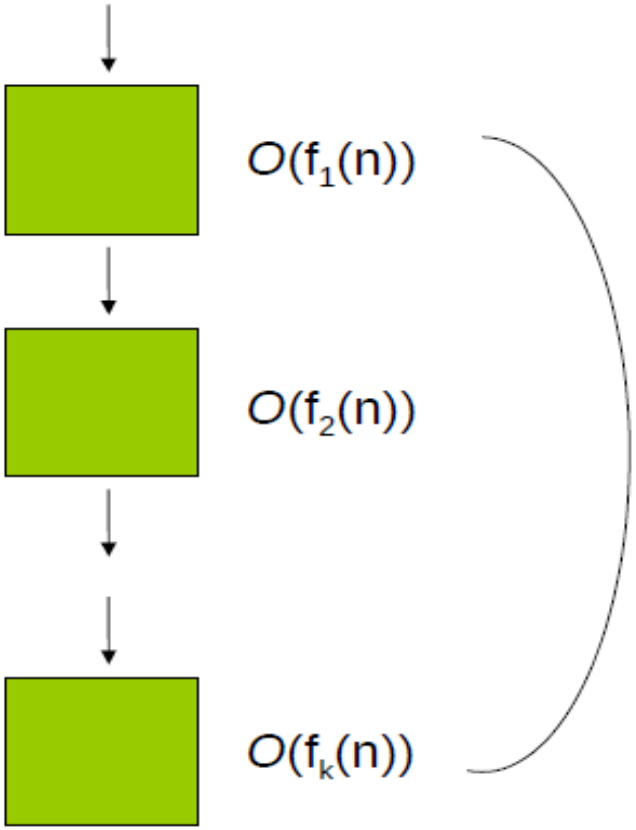
Czas działania pętli „for”



Czas działania instrukcji „if”



Czas działania bloku instrukcji



$O(f_1(n)+f_2(n)+...+ f_k(n))$

lub

$O($ największa $f_i(n))$

Uwagi końcowe

- Na wybór najlepszego algorytmu dla tworzonego programu wpływa wiele czynników, najważniejsze to:
  - **prostota,**
  - **łatwość implementacji**
  - **efektywność**
- Do problemu systematycznej analizy czasu działania programu powrócimy jeszcze na wykładzie za kilka tygodni…

Wykład 4 – część I

Kombinatoryka

- Wariacje z powtórzeniami
- Permutacje
- Wariacje bez powtórzeń
- Kombinacje
- Łączenie reguł kombinatorycznych

Wykład 4 – część II

Prawdopodobieństwo i algorytmy probabilistyczne

- Co to jest teoria prawdopodobieństwa
- Podstawowe pojęcia:
  - reguła sum, reguła iloczynów
  - prawdopodobieństwa warunkowe
- Przykład z kartami
- Analiza probabilistyczna
- Liczby losowe, generatory liczb losowych
- Algorytmy wykorzystujące prawdopodobieństwo
  - Czy pudełko jest wadliwe?
  - Czy liczba N jest liczba pierwszą?

Teoria prawdopodobieństwa

- Teoria prawdopodobieństwa, szeroko stosowana we współczesnej nauce, ma również wiele zastosowań w informatyce, np.:
  - szacowanie czasu działania programów dla przypadków ze średnimi, czyli typowymi danymi wejściowymi,
  - wykorzystanie do projektowania algorytmów „podejmujących decyzje” w niepewnych sytuacjach, np. najlepsza możliwa diagnoza medyczna na podstawie dostępnej informacji,
  - algorytmy typu Monte Carlo,
  - różnego rodzaju symulatory procesów,
  - prawie zawsze „prawdziwe” 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**.

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 jest 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 $(0.9)^k = (0.9)^{131} = 10^{-6}$.
  10^{-6} to jest prawdopodobieństwo że pudełko uznamy za dobre choć nie było testowane przez producenta.
  Za cenę błędu = 10^{-6}, zamieniliśmy algorytm z O(n) na O(1).
  Możemy regulować wielkość błędu/czas działania algorytmu zmieniając k.

„ 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 ?”

Wykład 5 – część I

Iteracja

Rekurencja

Indukcja

- Iteracja
- Rekurencja
- Indukcja
- Algorytmy sortujące
- Rozwiazywanie rekurencji

Iteracja

- Źródłem potęgi komputerów jest zdolność do **wielokrotnego wykonywania (powtarzania) tego samego zadania** lub jego różnych wersji.

  Iteracja = „powtarzanie”

- W informatyce z pojęciem iteracji (ang. iteration) można się spotkać przy różnych okazjach. Wiele zagadnień związanych z modelami danych, np. listami, opiera się na powtórzeniach typu:
  - ***lista jest albo pusta, albo składa się z jednego elementu poprzedzającego inny, kolejny element itd....***

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.

Definicja rekurencyjna

Rekurencyjna definicja funkcji silnia !

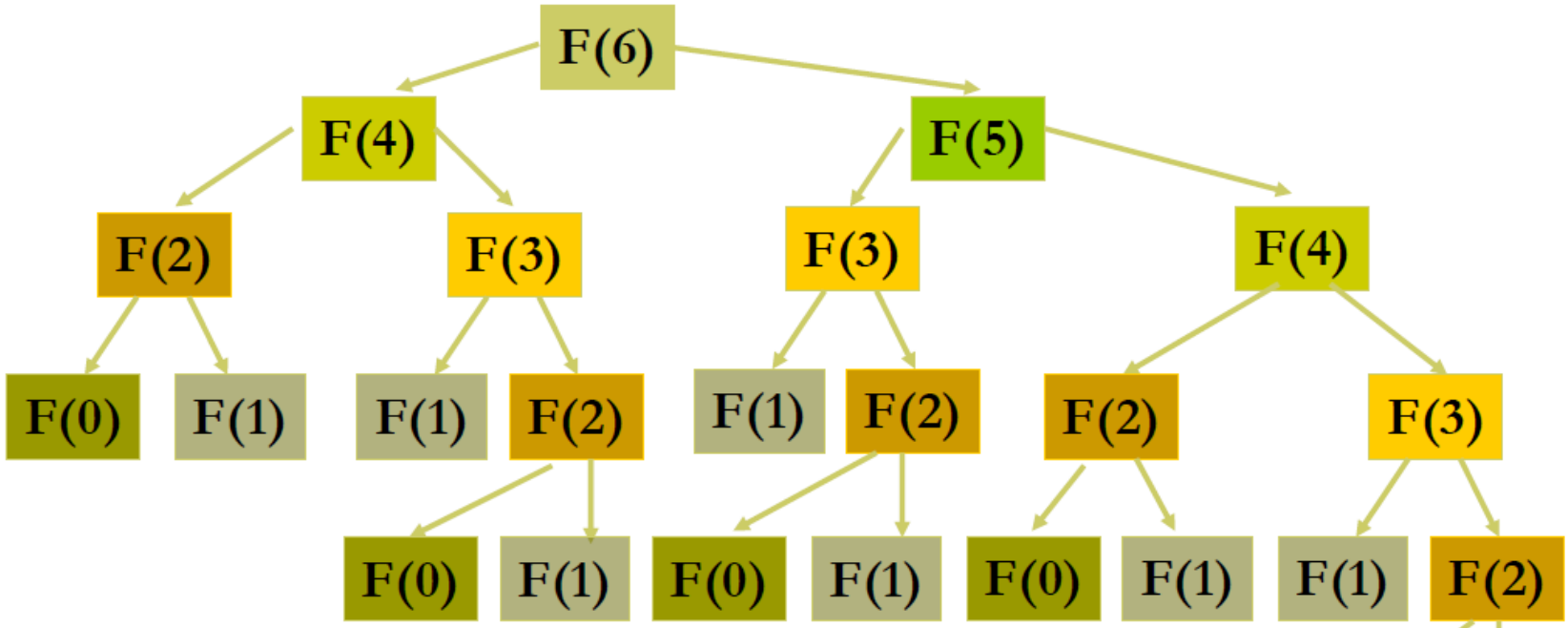
$$n! = \begin{cases} 1, & \text{jeśli } n = 0 \text{ (podstawa)} \\ n \bullet (n-1)! & \text{jeśli } n > 0 \text{ (indukcja)} \end{cases}$$

Rekurencyjna definicja ciągu Fibonacciego?

$$F(n) = n \qquad \text{jeśli } n < 2$$

$$F(n) = F(n-2)+F(n-1) \qquad \text{jeśli } n \geq 2$$

Rekurencja dla ciągu Fibonacciego



Obliczenia są bardzo nieefektywne bo funkcja F() jest wywoływana 25 razy dla obliczenia 7-go elementu. Powtarzamy wiele razy te same obliczenia!

Rekurencja czy iteracja?

- Każdy **problem mający rozwiązanie rekurencyjne daje się także rozwiązać w sposób iteracyjny**, choć jego rozwiązanie iteracyjne może być mniej czytelne w porównaniu z rekurencyjnym, a niekiedy wręcz sztuczne.
- **Rekurencja może być ponadto symulowana** w sposób iteracyjny, przy użyciu struktur danych zwanych stosami.
- **Programy rekurencyjne są często mniejsze i łatwiejsze do zrozumienia** od ich iteracyjnych odpowiedników.
- Co ważniejsze, niektóre problemy (szczególnie niektóre problemy wyszukiwania) są znacznie łatwiejsze do rozwiązania za pomocą programów rekurencyjnych.

Indukcja

- Zagadnieniem również związanym z iteracja i rekurencja jest indukcja (ang. induction):
  - **technika stosowana w matematyce do dowodzenia, że twierdzenie S(n) jest prawdziwe dla wszystkich nieujemnych liczb całkowitych n lub, uogólniając, dla wszystkich liczb całkowitych ≥ od pewnego ograniczenia dolnego.**

Indukcja

- Zagadnieniem również związanym z iteracja i rekurencja jest indukcja (ang. induction):
  - **technika stosowana w matematyce do dowodzenia, że twierdzenie S(n) jest prawdziwe dla wszystkich nieujemnych liczb całkowitych n lub, uogólniając, dla wszystkich liczb całkowitych ≥ od pewnego ograniczenia dolnego.**

Wykład 5 – część II

Elementy technik sortowania

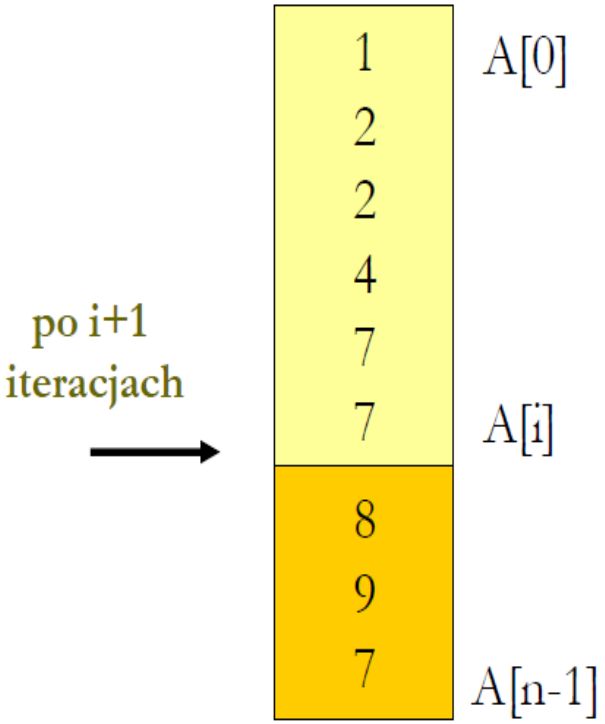
- Algorytmy iteracyjne: sortowanie przez wybieranie
- Algorytmy rekurencyjne: sortowanie przez dzielenie i scalanie

Elementy technik sortowania

- Najprostszym sposobem wielokrotnego wykonania sekwencji operacji jest wykorzystanie **konstrukcji iteracyjnej** (instrukcje **for, while** w języku C).
- **Przykład:**
  - Przypuśćmy że mamy listę liczb całkowitych (7, 4, 2, 8, 9, 7, 7, 2, 1).
  - Sortujemy tę listę (w porządku niemalejącym) permutując ją do postaci (1, 2, 2, 4, 7, 7, 7, 8, 9).
  - Należy zauważyć, że sortowanie nie tylko porządkuje wartości, tak że każda jest równa lub mniejsza kolejnej liczbie z listy, ale także zachowuje liczbę wystąpień każdej wartości.
- **Algorytm sortujący** pobiera na wejściu dowolna listę i zwraca jako wynik listę posortowaną. Każdy element występujący w liście pierwotnej występuje również w liście posortowanej.

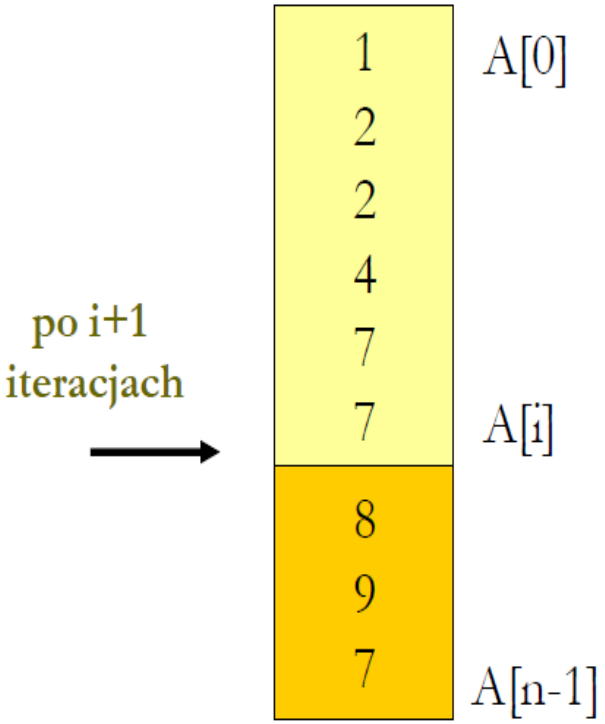
Sortowanie przez wybieranie – iteracyjny algorytm sortujący

- Mamy tablicę **A** zawierającą **n** liczb całkowitych które chcemy posortować w porządku niemalejącym. Można to zrobić wielokrotnie powtarzając krok:
  - wyszukaj najmniejszy element nieposortowanej części tablicy
  - wymień go z elementem znajdującym się na pierwszej pozycji nieposortowanej części tablicy
- **Pierwsza iteracja:** wybiera najmniejszy element w A[0, n-1], zamienia z elementem na pozycji A[0];
- **Druga iteracja:** wybiera najmniejszy element w A[1, n-1], zamienia z elementem na pozycji A[1];
- **Trzecia iteracja:** …
- **I-ta iteracja wymaga przejrzenia (n-i) elementów**.



Sortowanie przez wybieranie – rekurencyjny algorytm sortujący

- Mamy tablicę A zawierająca n liczb całkowitych które chcemy posortować w porządku niemalejącym.
- Można to robić rekurencyjnie
  - wybieramy najmniejszy element z reszty tablicy A (czyli z A[i, …, n-1]),
  - wymieniamy wybrany w poprzednim kroku element z elementem A[i],
  - sortujemy resztę tablicy czyli A[i+1, …, n-1].
- **Podstawa:**
  - Jeśli i = n-1, to pozostaje do posortowania jedynie ostatni element tablicy. Ponieważ pojedynczy element jest zawsze posortowany nie trzeba podejmować żadnych działań.
- **Indukcja:**
  - Jeśli i < n-1, to należy znaleźć najmniejszy element w tablicy A[i, …, n-1], wymienić go z elementem A[i] i rekurencyjnie posortować tablice A[i+1,...,n-1].
- Kompletny algorytm realizujący powyższą rekurencję rozpoczyna się od i=0.



Sortowanie przez „dzielenie i scalanie” – rekurencyjny algorytm sortujący

- Najlepszy opis **sortowania przez scalanie** opiera się na rekurencji i ilustruje równocześnie bardzo korzystne zastosowanie techniki „dziel i zwyciężaj”.
- Listę **(a_1, a_2, a_3, …, a_n)** sortuje się **dzieląc** na **dwie listy** o dwukrotnie mniejszych rozmiarach. Następnie obie listy są sortowane osobno.
  Aby zakończyć proces sortowania oryginalnej listy n-elementów, **obie listy zostają scalone** przy pomocy specjalnego algorytmu.
- **Scalanie:**
  - Prostym sposobem scalania dwóch list jest analiza od ich początków. W każdym kroku należy znaleźć mniejszy z dwóch elementów będących aktualnie na czele list, wybrać go jako kolejny element łączonej listy i usunąć go z “pierwotnej listy”, wskazując na kolejny pierwszy element. W przypadku równych pierwszych elementów można dodawać je do łączonej listy w dowolnej kolejności.

Sortowanie przez „dzielenie i scalanie” – rekurencyjny algorytm sortujący

- **Podstawa:**
  - Jeśli lista do posortowania jest pusta lub jednoelementowa, zostaje zwrócona ta sama lista – jest ona już posortowana.

- **Krok indukcyjny:**
  - Jeżeli lista ma nie mniej niż 2 elementy to podziel listę na dwie (np. elementy o parzystym indeksie i elementy o nieparzystym indeksie).
    Posortuj każdą z dwóch list osobno i scal.

Wykład 5 – część III

Rozwiązywanie rekurencji

- Rekurencja dla algorytmu „dziel i zyciężaj”
- Rozwiązywanie rekurencji
  - Metoda podstawiania
  - Metoda iteracyjna
    - Drzewa rekursji
  - Metoda uniwersalna

Czas działania programu

- Dla konkretnych danych wejściowych jest wyrażony liczba wykonanych prostych (elementarnych) operacji lub “kroków”. Jest dogodne zrobienie założenia że operacja elementarna jest maszynowo niezależna.
- Każde wykonanie i-tego wiersza programu jest równe c_i, przy czym c_i jest stałą.
- **Kiedy algorytm zawiera rekurencyjne wywołanie samego siebie, jego czas działania można często opisać zależnością rekurencyjna** (rekurencja) wyrażającą czas dla problemu rozmiaru n za pomocą czasu dla podproblemów mniejszych rozmiarów.
- **Możemy wiec użyć narzędzi matematycznych aby rozwiązać rekurencje i w ten sposób otrzymać oszacowania czasu działania algorytmu.**

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.

Wykład 6 – część I

Modele danych

- Abstrakcja, modele danych a struktury danych
- Modele danych
  - języków programowania
  - w oprogramowaniu systemowym
  - w edytorach tekstów
  - układów komputerowych
  - Języka C
- Bazy danych i bazy wiedzy

Abstrakcja

- Abstrakcja
  - Oznacza **uproszczenie**, zastąpienie skomplikowanych i szczegółowych okoliczności występujących w świecie rzeczywistym zrozumiałym **modelem** umożliwiającym rozwiązanie naszego **problemu**.
  - Oznacza to, że **„abstrahujemy” od szczegółów**, które nie mają wpływu lub mają minimalny wpływ na rozwiązanie problemu.
  - Opracowanie odpowiedniego modelu umożliwia zajęcie się istotą problemu.

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 a struktury danych

- **Modele danych to abstrakcje** wykorzystywane do opisywania problemów.
- **Struktury danych to reprezentacja danego modelu danych**, którą musimy skonstruować w sytuacji gdy język programowania nie ma wbudowanej tej reprezentacji.
- Konstruujemy **strukturę danych** za pomocą **abstrakcji obsługiwanych przez ten język**.

Wykład 6 – część II

Listy

- Podstawowa terminologia
- Lista jednokierunkowa
- Słownik
- Lista dwukierunkowa, cykliczna, lista z duplikatami
- Lista oparta na tablicy
- Stos
- Kolejka

Podstawowa terminologia

Lista

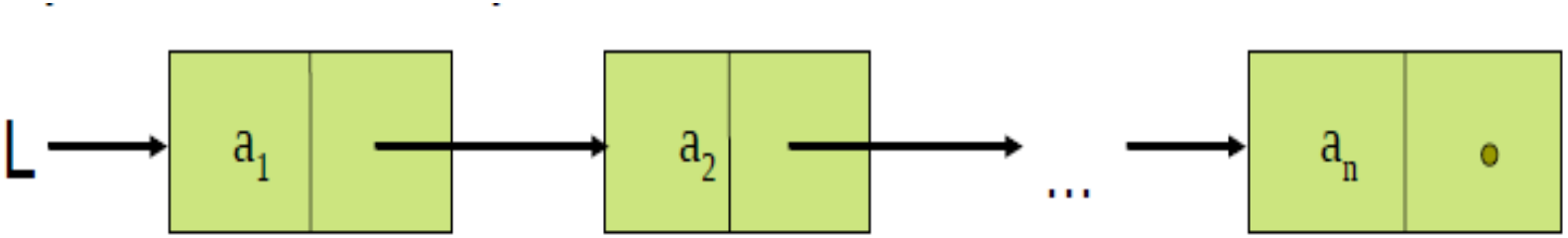
- Jest to skończona sekwencja zera lub większej ilości elementów.
- Jeśli wszystkie te elementy należą do typu T, to w odniesieniu do takiej struktury używamy sformułowania **„lista elementów T”.**
- Możemy więc mieć **listę liczb całkowitych, listę liczb rzeczywistych, listę struktur, listę list liczb całkowitych**, itd. Oczekujemy że elementy listy należą do jednego typu, ale ponieważ może być on unią różnych typów to to ograniczenie może być łatwo pominięte.
- Często przedstawiamy listę jako $(a_1, a_2, \ldots, a_n)$ gdzie symbole a_i reprezentują kolejne elementy listy.
- Listą może być też ciąg znaków.

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

- 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.
- Jeżeli mówimy o konkretnej implementacji, to oznacza że dyskutujemy „strukturę danych”
- Lista jednokierunkowa **$L = (a_1, a_2, a_3, \ldots, a_n)$**



Stos

- **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.

Kolejka

- **Kolejka:** Sekwencja elementów $a_1, a_2, \ldots, a_n$ 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

Wykład 7 – część I

Modele danych: zbiory

- Podstawowe definicje
- Operacje na zbiorach
- Prawa algebraiczne
- Struktury danych
  - Lista jednokierunkowa
  - Wektor własny
  - Tablica mieszająca

Zbiór

- **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 pojęcie mówiąc o 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 pojęcie **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 $x \in S$ oznacza, że element x należy do zbioru S.
  - Jeśli elementy $x_1, x_2, \ldots, x_n$ należą do zbioru S i żadne inne, to możemy zapisać:
    $S = \{x_1, x_2, \ldots, x_n\}$
  - 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 $x \in \varnothing$ 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 ϵ 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.

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**.

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**.

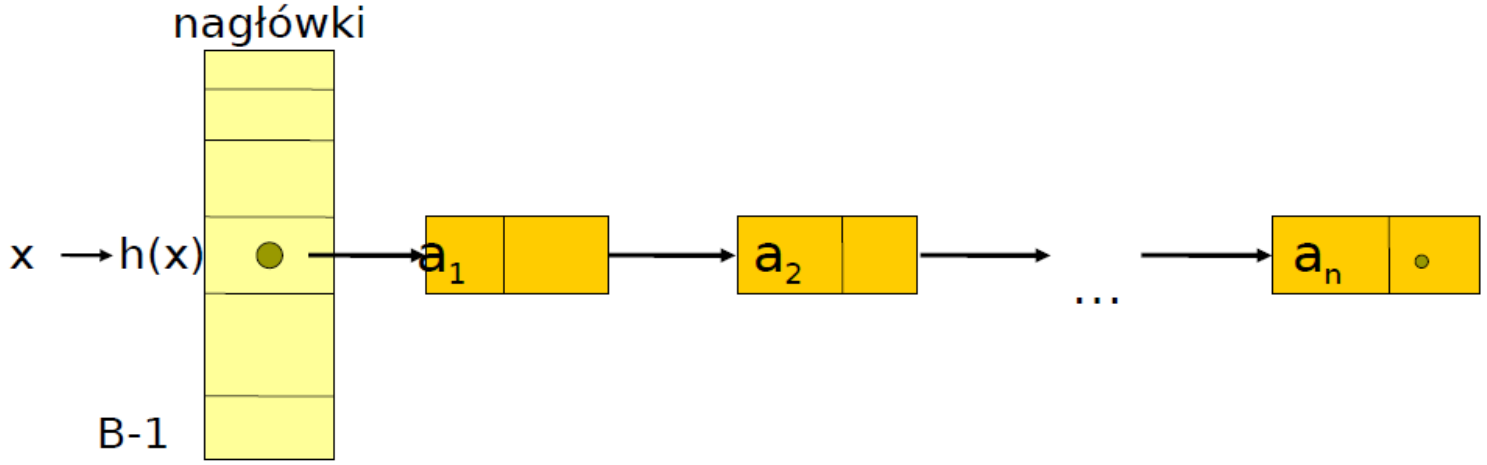
Przykład z kartami

- Zbiór wszystkich kart koloru trefl
  - 11111111111100000000000000000000000000000000000000
- Zbiór wszystkich figur
  - 00000000001110000000001110000000001110000000000111
- Poker w kolorze kier (as, walet, dama, król)
  - 00000000000000000000001000000001110000000000000
- Każdy element zbioru kart jest związany z unikatową pozycją.

Struktura danych oparta na 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: **$26^{10}+26^{9}+ \ldots + 26 = 10^{14}$** możliwych słów.
  Faktyczny słownik: to tylko około **10^{6}**.
  Co robimy?
- **Grupujemy**, każda grupa to jedna komórka z „nagłówkiem” + lista jednokierunkowa z elementami należącymi do grupy.
- **Taką strukturę nazywamy tablicą mieszająca (ang. hash table)**

Struktura danych tablicy mieszającej



- Istnieje **funkcja mieszającą** (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 komórką, w której umieszczamy element **x**.
- Ważne aby funkcja **h(x)** „mieszała”, tzn. aby komórki zawierały tę samą przybliżoną liczbę elementów.

Podsumowanie

- **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.

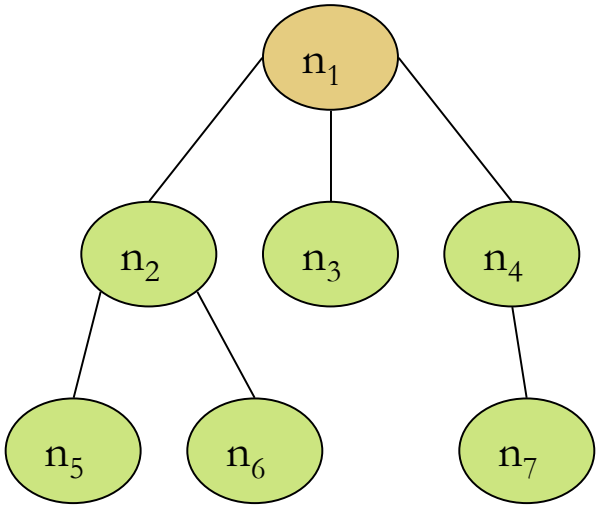
Wykład 7 – część II

Modele danych: drzewa

- Podstawowa terminologia
- Rekurencyjna definicja
- Drzewa zaetykietowane
  - Drzewa wyrażeń
- Struktura danych dla drzew
  - Reprezentacje drzewa
  - Rekurencja w drzewach
- Drzewa binarne
- Drzewa przeszukiwania binarnego
- Drzewa binarne częściowo uporządkowane
- Kolejka priorytetowa
- Zrównoważone drzewa częściowo uporządkowane i kopce
  - Sortowanie przez kopcowanie

Podstawowa terminologia

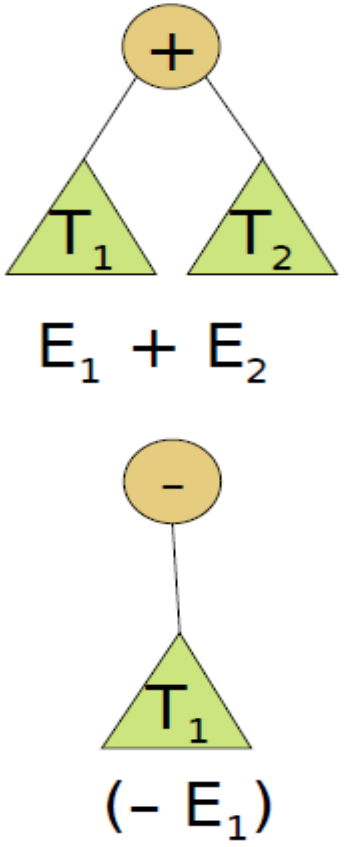
- Aby **struktura** zbudowana z węzłów połączonych krawędziami była **drzewem** musi spełniać pewne warunki:
  - W każdym drzewie wyróżniamy jeden węzeł zwany **korzeniem** n_1 (ang. root)
  - Każdy węzeł c nie będący korzeniem jest połączony krawędzią z innym **węzłem** zwanym rodzicem p (ang. parent) 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.



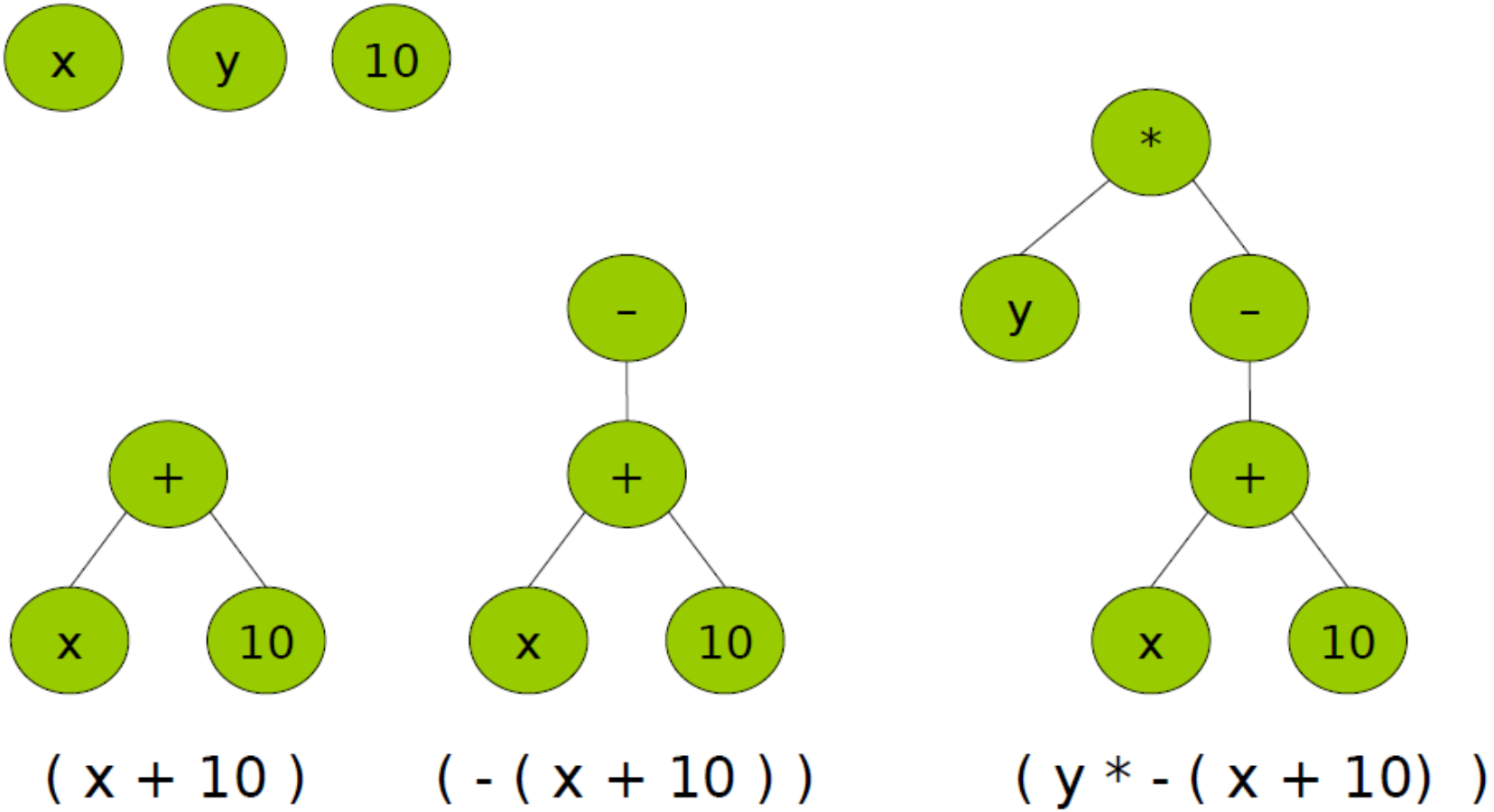
n_1 = rodzic n_2, n_3, n_4
n_2 = rodzic n_5, n_6
n_6 = dziecko n_2

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.



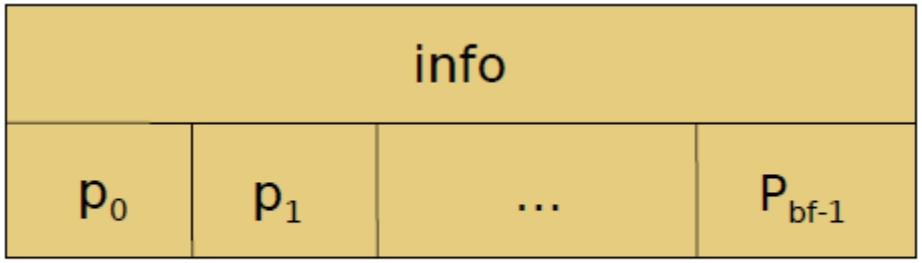
Konstrukcja drzew wyrażeń



Tablica wskaźników jako reprezentacja drzewa

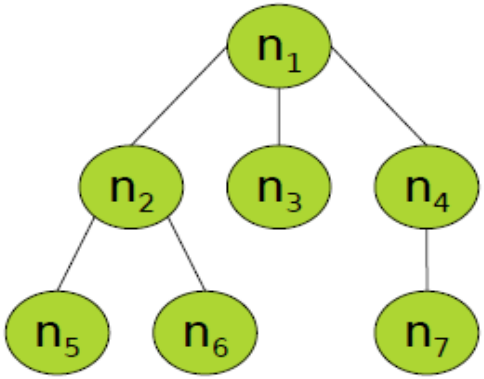
- Jednym z najprostszych sposobów reprezentowania drzewa jest wykorzystanie dla każdego węzła struktury składającej się z pola lub pól reprezentujących etykietę oraz tablicy wskaźników do dzieci tego węzła.
- Info reprezentuje etykietę węzła.
- Stała bf jest rozmiarem tablicy wskaźników. Reprezentuje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

```
typedef struct NODE *pNODE
struct NODE{
    int info;
    pNODE children[BF];
};
```



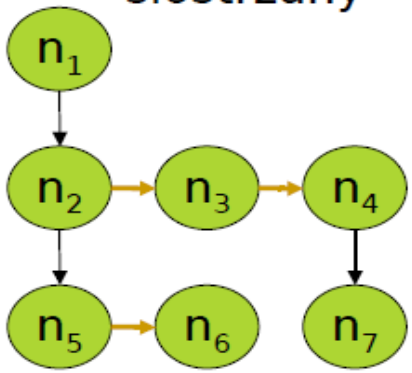
Reprezentacje drzewa

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



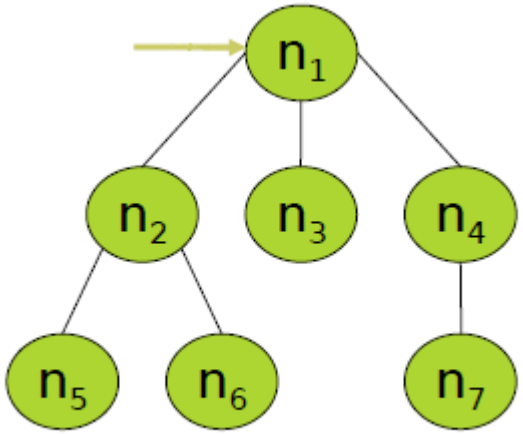

```
typedef struct NODE *pNODE;
struct NODE{
        int info;
        pNODE leftmostChild, rightSibling;
};
```

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.

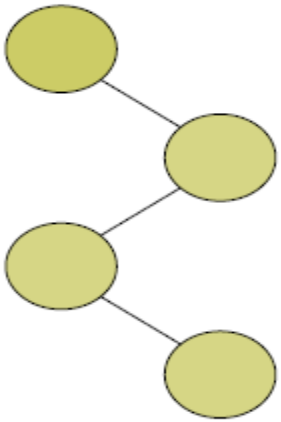
Rekurencja w drzewach

- 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.



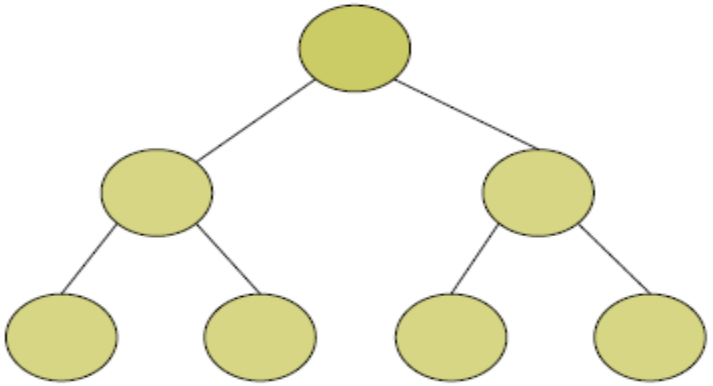
Drzewa binarne

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



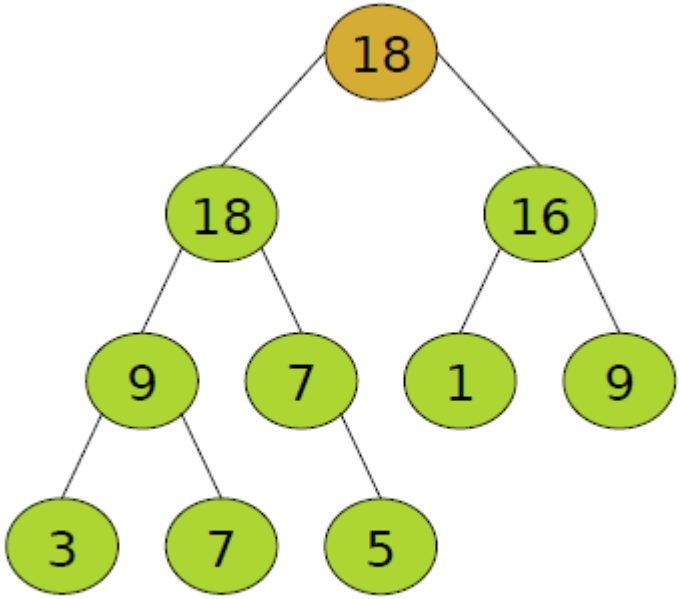
Drzewo o wysokości h ma $k=2^{h+1}-1$ węzłów.
Czyli h = O(log k).
Operacje insert, delete, find wymagają średnio O(log k).

Drzewa 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.

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.



Podsumowanie

- Ważnym modelem danych reprezentującym **informacje hierarchiczne** są **drzewa**.
- Do implementowania drzew możemy **wykorzystać wiele różnych struktur danych** (także takich) które wymagają połączenia tablic ze wskaźnikami. **Wybór struktury danych zależy od operacji wykonywanych na drzewie**.
- Dwiema najważniejszymi reprezentacjami węzłów drzewa są **skrajnie lewy potomek-prawy element siostrzany** oraz **tree** (tablica wskaźników do dzieci).
- Drzewa nadają się doskonale do stosowania na nich algorytmów i dowodów rekurencyjnych.
- **Drzewo binarne** jest jednym z wariantów modelu drzewa, w którym każdy węzeł ma (opcjonalne) lewe i prawe dziecko.
- **Drzewo przeszukiwania binarnego** jest zaetykietowanym drzewem binarnym, które spełnia **„własność drzewa przeszukiwania binarnego”**.