Teoretyczne podstawy informatyki

Wykład 5:
Modele danych – wprowadzenie
Listy

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

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.

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 „pudelek", 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

- 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

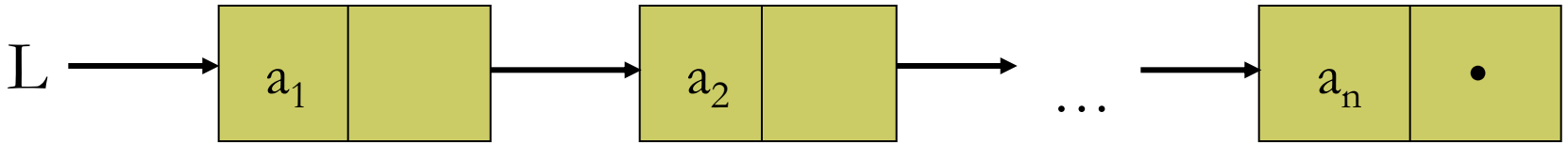
- Bardzo ważne są też **modele danych**, które **nie są** częścią języka programowania, takie jak listy, drzewa, grafy, zbiory.
- Np. w języku matematycznym, lista jest ciągiem **n** elementów, który zapisujemy jako **$(a_1,a_2, \ldots,a_n)$**.
  Do zbioru operacji wykonywanych na listach należą:
  -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

Modele danych a struktury danych

- Lista jest to abstrakcja matematyczna lub model danych.
- Lista jednokierunkowa to struktura danych:

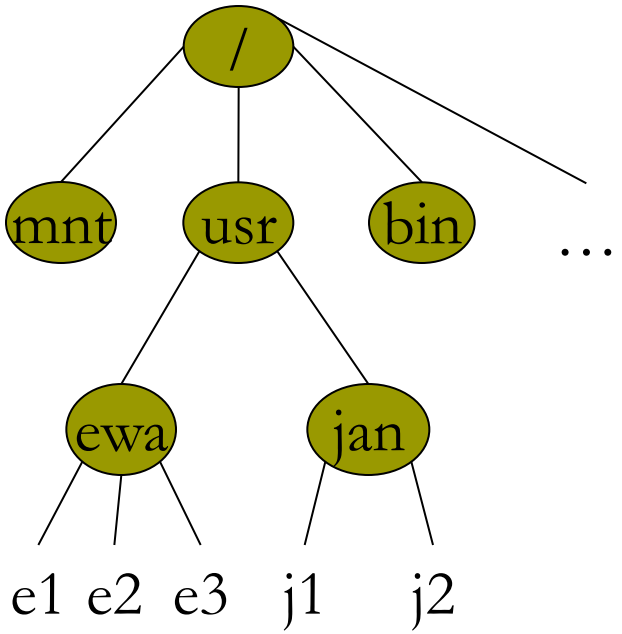
```
Typedef struct CELL *LIST;
struct CELL{
    int element;
    LIST next;
}
```

W niektórych językach (Lisp, Prolog) nie ma potrzeby stosowania (konstruowania) struktur danych do reprezentowania abstrakcyjnych list.



Modele danych w oprogramowaniu systemowym

- Modele danych możemy też spotkać w systemach operacyjnych i w aplikacjach. Zadaniem systemu operacyjnego jest zarządzanie i szeregowanie zasobów komputera. Model danych systemów operacyjnych Unix składa się z takich pojęć jak **pliki, katalogi** oraz **procesy.**
  - Dane jako takie są przechowywane w **plikach** (ang. files), które w systemie Unix reprezentowane są przez ciągi znaków.
  - Pliki są grupowane w ramach **katalogów** (ang. directories), będących zbiorami plików i (lub) innych katalogów.
  - Katalogi i pliki tworzą **drzewo** w którym pliki są liśćmi.



Modele danych w oprogramowaniu systemowym

- **Procesy** są pojedynczymi wykonaniami programów.
  Procesy pobierają zero lub więcej strumieni wejściowych i produkują zero lub więcej strumieni wyjściowych. W systemach Unix procesy mogą składać się z **potoków** (ang. pipes), kiedy to wynik jednego procesu może zasilać wejście kolejnego procesu. Efekt takiego połączenia procesów można traktować jako jeden duży proces z własnym wejściem i wyjściem.
  - Przykład:

```
ls | grep file
```

- Istnieje wiele innych aspektów działania systemu operacyjnego, np. sposób zarządzania bezpieczeństwem danych oraz interakcja z użytkownikiem.
- **Dość łatwo można zauważyć że model danych systemu operacyjnego różni się od modeli danych języków programowania.**

Model danych w edytorach tekstu

- Każdy model danych wbudowany w taki edytor wiąże się z pojęciami ciągów tekstowych oraz operacjami charakterystycznymi dla redagowania tekstu.
- Model zawiera więc zazwyczaj pojęcie **wierszy** (ang. lines), które podobnie jak większość plików są ciągami znaków. Jednak w przeciwieństwie do plików wiersze mogą się wiązać ze swoimi numerami. Mogą być także grupowane w większe jednostki zwane **akapitami.**
- **Operacje na wierszach** można zazwyczaj stosować dla wszystkich zawartych w nich elementów, nie tylko dla ich początku, jak w przypadku najbardziej powszechnych operacji na plikach.
- Typowy edytor wykorzystuje również pojęcie wiersza bieżącego oraz bieżącej pozycji w danym wierszu. Wykonywane przez edytor operacje zawierają rozmaite modyfikacje wierszy, takie jak usuwanie i wstawianie znaków, usuwanie lub tworzenie nowych wierszy, poszukiwanie określonych ciągów znaków, itd.

Modele danych układów komputerowych

- Model danych opisujący układy komputerowe, zwany **logiką wnioskowania**, jest najbardziej przydatnym narzędziem w projektowaniu komputerów.
- Komputery składają się z komponentów elementarnych zwanych bramkami (ang. gates). Każda bramka ma jedno lub więcej wejść i jedno wyjście; na wejściu i wyjściu dopuszczalne są tylko dwie wartości: **0** lub **1**. Bramka wykonuje prostą funkcje – np. koniunkcje (bramka **AND**).
- Na pewnym poziomie abstrakcji projektowanie komputera jest procesem, w którym decyduje się o sposobie połączenia bramek tak, by możliwe było efektywne wykonywanie na nim prostych operacji.

x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

Sumator jednobitowy

- Aby wykonać instrukcje przypisania **a = b + c** w języku **C**, komputer wykonuje dodawanie za pomocą układu zwanego sumatorem (ang. adder). W komputerze wszystkie liczby są zapisywane w notacji binarnej wykorzystującej dwie cyfry, **0** i **1** (zwane **cyframi binarnymi** lub **bitami**). Mając kilka bramek możemy zbudować układ zwany **sumatorem jednobitowym** (ang. one bit adder). Dwa bity wejściowe, **x** i **y**, oraz wejściowy bit przeniesienia, **c**, są sumowane. Efektem tej operacji jest bit sumy oraz wyjściowy bit przeniesienia (ang. carry out) **d**.
- Przykład:
  - **dz** to łącznie dwubitowa liczba binarna wyrażający łączną liczbę danych wejściowych (**x, y, c**) mających wartość 1,
  - **d** = bit przeniesienia,
  - **z** = bit sumy.

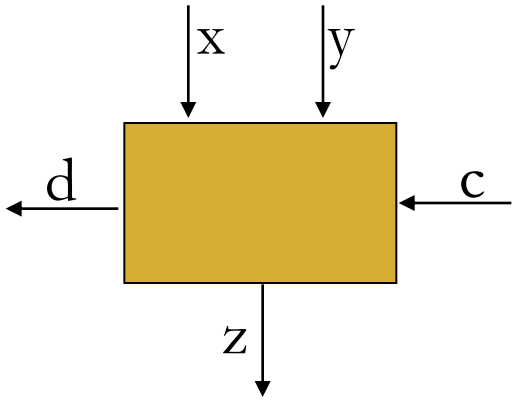


Tabela prawdy:

x	y	c	d	z
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
1	0	0	0	1
0	1	1	1	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Modele danych języka C

- Statyczna część modelu danych w języku C to system typów opisujący wartości, które mogą być przyjmowane przez określone dane.
- System typów zawiera **typy proste**, np. liczby całkowite, oraz zbiór **zasad formowania typów**, dzięki którym możemy konstruować coraz bardziej skomplikowane typy na bazie typów już znanych.
- **Typy podstawowe:**
  - znaki (char, signed char, unsigned char),
  - liczby całkowite (int, short, long int, unsigned),
  - liczby zmiennoprzecinkowe (float, double, long double),
  - Wyliczenia (enum).
- Liczby całkowite i zmiennoprzecinkowe traktowane są jako typy arytmetyczne.

Modele danych języka C

- Reguły formowania typów wymagają istnienia pewnych typów które mogą być albo typami podstawowymi; albo typami wcześniej skonstruowanymi za pomocą takich reguł.
- **Typy tablicowe:**
  - Możemy stworzyć tablice, której elementy są typu T: **T A[n];**
  - Powyższa instrukcja deklaruje tablice **n** elementów, każdy typu **T**.
  - W języku **C** indeksy tablic rozpoczynają się od **0**, zatem pierwszym elementem jest **A[0]**, ostatnim **A[n-1]**.
  - Tablice mogą być skonstruowane ze znaków, typów arytmetycznych, wskaźników, struktur, unii lub innych tablic.

Modele danych języka C

- **Struktury:**
  - Struktura jest grupowaniem zmiennych zwanych **składnikami** (ang. members) lub **polami** (ang. fields). Różne składniki struktur mogą być różnych typów, jednak każdy musi zawierać elementy jednego określonego typu.
  - Jeśli $T_1, T_2, \ldots, T_n$ są typami oraz $M_1, M_2, \ldots, M_n$ są nazwami składników, to deklaracja:

```
struct S {
        T1 M1;
        ....
        Tn Mn;
}
```

    definiuje strukturę której wyróżnik (nazwa jej typu) to S, zaś **n** to liczba jej składników, **i**-ty składnik nosi nazwę M_i i jest typu T_i.
  - Wyróżnik struktury S jest opcjonalny, udostępnia jednak wygodny skrót podczas odwoływania się do tego typu w późniejszych deklaracjach.

Modele danych języka C

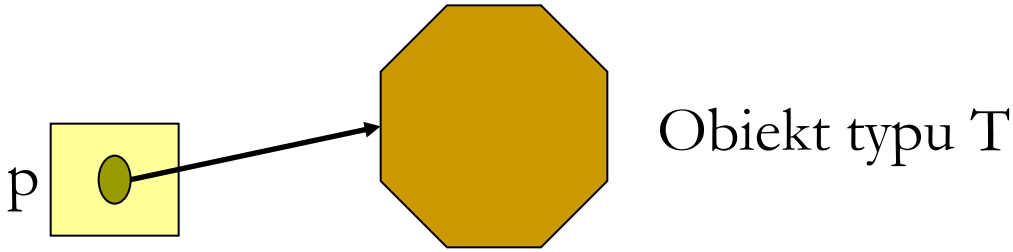
- **Unie:**
  - Unia pozwala na przechowywanie zmiennych przyjmujących wartości różnych typów w różnych momentach wykonywania programu.
  - Deklaracja:

```
union {
        T1 M1;
        T2 M2;
        …
        Tn Mn;
}
x;
```

definiuje zmienną **x**, która może przechowywać wartość dowolnego typu z grupy $T_1, T_2, \ldots, T_n$. Nazwy składników $M_1, M_2, \ldots, M_n$ pomagają wyróżnić typ aktualnej wartości zmiennej. Oznacza to ze $x.M_i$ wskazuje na wartość zmiennej **x** traktowanej jako wartość typu T_i.

Modele danych języka C

- **Wskaźniki:**
  - Język **C** wyróżnia się znaczeniem jaki mają w nim wskaźniki. Zmienna typu wskaźnikowego zawiera adres obszaru pamięci. Za pomocą wskaźnika możemy uzyskać dostęp do wartości innej zmiennej.
  - Deklaracja: **T *p**;
    definiuje zmienną **p**, jako wskaźnik do zmiennej typu **T**.
  - Zmienna **p** nazywa więc pudełko typu wskaźnikowego do **T**, wartością w pudełku **p** jest wskaźnik. Tym co „naprawdę" znajduje się w pudełku jest adres (lokacja), pod którym obiekt typu **T** jest przechowywany w komputerze.



Modele danych języka C

- **Typedef:**
  - Język **C** udostępnia instrukcje **typedef**, która umożliwia tworzenie synonimów dla nazw typów.
  - Deklaracja: **typedef int Odległość**; pozwala na późniejsze używanie nazwy **Odległość** zamiast typu **int**.
- **Funkcje:**
  - Funkcje także posiadają związane ze sobą typy, mimo że nie łączymy z nimi pudełek ani wartości.
  - Dla dowolnej listy typów $T_1, T_2, \ldots, T_n$ możemy zdefiniować funkcje pobierającą odpowiednio **n** parametrów tych typów. Typ wartości zwracanych przez funkcje nazywamy typem funkcji. Jeżeli funkcja nie zwraca żadnej wartości wykorzystujemy typ **void**.
- W ogólności możemy budować typy dowolnie, stosując reguły ich konstrukcji, istnieje jednak kilka ograniczeń. Przykładowo nie możemy konstruować tablicy funkcji mimo że możemy zbudować tablice wskaźników do funkcji.

Operacje w modelu danych języka C

- Przewidywane operacje na danych w modelu języka **C** możemy podzielić na trzy kategorie:
  - operacje tworzące i usuwające obiekt danych,
  - operacje dostępu i modyfikacji części obiektu danych,
  - operacje łączące części obiektu danych w celu sformowania nowej wartości obiektu danych.

Operacje w modelu danych języka C

- **Tworzenie i usuwanie obiektu danych:**
  - Język **C** udostępnia wiele elementarnych mechanizmów przeznaczonych do tworzenia danych.
  - W momencie wywołania funkcji tworzone są pudełka dla wszystkich jej lokalnych argumentów (parametrów). Pozwala to na przechowywanie wartości tych parametrów.
  - Innym mechanizmem jest procedura biblioteczna **malloc(n)**, która zwraca wskaźnik do **n** kolejnych pozycji znaków w niewykorzystanej pamięci. Obiekty danych mogą być wówczas utworzone właśnie w tych obszarach pamięci.
  - Metody usuwania obiektów danych są analogiczne. Procedura biblioteczna **free** zwalnia pamięć zarezerwowaną przez **malloc**.

Operacje w modelu danych języka C

- **Dostęp do danych i ich modyfikacja:**
  - Język **C** zawiera mechanizm umożliwiający dostęp do komponentów składających się na obiekty. Wykorzystujemy:
    - zapis **a[i]** do uzyskania dostępu do **i-tego** elementu tablicy **a**,
    - zapis **x.m** do uzyskania dostępu do składnika **m** struktury o nazwie **x**
    - zapis ***p** do uzyskania dostępu do obiektu wskazanego przez wskaźnik **p**.
  - Modyfikowanie (przypisywanie) wartości w języku **C** realizujemy za pomocą operatorów przypisania, które umożliwiają zmianę wartości obiektu. Np.:
    - **a[0].(*pole[3]) = 99**.

Łączenie danych

- Język **C** zawiera bogaty zbiór operatorów umożliwiających manipulowanie danymi i łączenie ich wartości. Oto podstawowe operatory:
  - **Operatory arytmetyczne**:
    dwuargumentowe **+**, **–**, **•**, **/**, **%**,
    - 3 + 2 = 5;
    - 3 – 2 = 1;
    - 3 • 2 = 6;
    - 3 / 2 = 1.5;
    - 3 % 2 = 1 (modulo – reszta z dzielenia);

    jednoargumentowe **+**, **–**, **++**, **– –**,
    - n = – k (– jako zmiana znaku liczby);
    - n++, ++n (inkrementacja, zwiekszanie wartości o 1);
    - n– –, – –n (dekrementacja, zmniejszenie wartości o 1);

Łączenie danych

- **Operatory logiczne**:
  Język **C** nie zawiera typu **Boolean**, wykorzystuje **0** do reprezentowania wartości logicznej fałszu oraz liczby różnej od zera do reprezentowania prawdy.
  Język **C** udostępnia:
  - koniunkcje **&&** (dwuargumentowy)
  - alternatywę **||** (dwuargumentowy)
  - negacje **!** (jednoargumentowy)
  - operator warunkowy **warunek ? y : z** (trzyargumentowy), znaczący:
    if (**warunek**)
    then return **y**;
    else return **x**;

Łączenie danych

- **Operatory porównania:** (==, !=, <, >, ≤, ≥)
  - Dla liczb całkowitych i zmiennoprzecinkowych. Wynikiem jest prawda lub fałsz.
- **Operatory działań na poziomie bitowym**
- **Operatory przypisania**
- **Operatory koercji (konwersja, rzutowanie):**
  - przekształcenie wartości jednego typu na odpowiadającą jej wartość innego typu.

- Często uzupełnia się podstawowe typy przez identyfikatory zdefiniowane w pliku nagłówkowym stdio.h: **NULL**, **TRUE**, **FALSE**, **BOOLEAN**, **EOF**.

Bazy danych

- W wielu zastosowaniach komputerów same struktury danych nie wystarczają. Nie zawsze jest to bowiem tylko kwestia rozważenia zadania algorytmicznego i zdefiniowania dobrych i użytecznych do jego rozwiązania struktur danych. Czasem potrzeba bardzo obszernych zasobów danych, stanowiących dla wielu algorytmów potencjalne dane wejściowe, a wiec mające ustalona strukturę i nadające się do odszukiwania i manipulowania nimi.
- Przykładami takich danych mogą być finansowe i osobowe dane przedsiębiorstwa, rezerwacje miejsc i informacje o lotach towarzystwa lotniczego, dane katalogowe biblioteki, itd.…

Bazy danych

- Bazy danych są zazwyczaj bardzo obszerne i zawierają wiele różnych rodzajów danych, począwszy od nazwisk i adresów po specjalne kody i symbole, a czasami nawet zwykły tekst.
- Zgromadzone dane są zazwyczaj przedmiotem licznych rodzajów operacji wstawiania, usuwania i wyszukiwania, wykorzystywanych w różnych celach przez różnych ludzi.
- O ile dodanie nowej informacji do bazy danych lub usuniecie już istniejącej są zadaniami stosunkowo łatwymi, o tyle **zapytanie** bazy danych z zamiarem wydobycia z niej informacji zazwyczaj jest dużo bardziej skomplikowane.

Bazy danych

- Ogromne znaczenie ma dobra organizacja bazy danych.
- Tak jak w przypadku struktur danych, dobry projekt bazy danych – to projekt przejrzysty, łatwy do zapisania, najważniejsza zaleta to duża sprawność działania i wykonalność opartego na tym projekcie systemu zarządzania bazą danych który potrafi odpowiedzieć na zapytania w krótkim czasie.
- Stosuje się różne modele organizacji baz danych.
- Modele, zaprojektowane do obsługi dużych ilości danych, jednocześnie wiernie i sprawnie wychwytują związki zachodzące miedzy obiektami danych.
- Istnieje wiele metod i języków manipulacji danymi i zapytań baz danych.

Bazy danych

- Jeden z najpopularniejszych, **model relacyjny,** zaspokaja potrzeby związane z układami danych w postaci ogromnych tabel, przypominających tablicowe struktury danych.
- Inny model, **model hierarchiczny**, wymaga pewnych rodzajów układów drzewiastych albo sieciowych. Ten model organizuje dane w drzewiastej formie o wielu poziomach.
- Na niektóre rodzaje danych lepiej patrzeć jak na fragmenty **wiedzy** niż tylko jako na liczby, nazwiska czy kody.
- Oprócz dużej bazy danych opisującej inwentarz przedsiębiorstwa produkcyjnego moglibyśmy chcieć mieć dużą bazę informacji dotyczących prowadzenia tego przedsiębiorstwa. Tego rodzaju fragmenty wiedzy wymagają organizacji bardziej złożonej niż obiekty danych o mniej więcej ustalonym formacie, zwłaszcza wówczas gdy zależy nam na sprawnym wyszukiwaniu.

Bazy wiedzy

- **Bazy wiedzy** stają się następnym naturalnym stopniem po **bazach danych**, są bogatym źródłem ciekawych pytań związanych z reprezentowaniem, organizacją i wyszukiwaniem algorytmicznym.
- Problem **reprezentacji wiedzy** jest faktycznie jednym z podstawowych zagadnień **sztucznej inteligencji**.
- Trudność wynika z tego, że wiedza składa się nie tylko z wielkiego zbioru faktów, ale także wielu zawiłych związków miedzy nimi. Te związki implikują inne, wyższego poziomu związki z innymi elementami wiedzy.

Bazy wiedzy

- Zaproponowano wiele **modeli wiedzy** które można by wykorzystać w inteligentnych programach. Niektóre opierają się na pojęciach czysto informatycznych, takich jak relacyjne czy hierarchiczne bazy danych. Inne na logicznych formalizmach takich jak rachunek predykatów czy logika modalna.
- Pewne języki programowania, jak **Lisp** czy **Prolog**, łatwiej nadają się do manipulowania wiedzą niż inne.
- Np. **Prolog** wydaje się trafnie dobrany jeżeli chodzi o fragmenty wiedzy dotyczące prostych relacji.
- Wymagane związki stają się coraz bardziej zagmatwane gdy wyjdziemy poza małą dobrze określoną dziedzinę dyskusji. Sięganie do wiedzy wiążącej się z pewna decyzją, którą program musi podjąć, staje się ogromnym zadaniem.

Bazy wiedzy

- „Efektywny” model algorytmicznej reprezentacji wiedzy wciąż czeka na odkrycie…

LISTY

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

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.
- **Długość listy**
  - Jest to liczba wystąpień elementów należących do listy. Jeżeli liczba tych elementów wynosi **0** to mówimy że lista jest pusta.

Podstawowa terminologia

- **Części listy**
  - Jeżeli lista nie jest pusta to składa się z pierwszego elementu zwanego **nagłówkiem (ang. head)** oraz reszty listy, zwanej **stopką (ang. tail).** Istotne jest że nagłówek listy jest elementem, natomiast stopka listy jest listą .
- **Podlista**
  - Jeśli **L = $(a_1, a_2, \ldots, a_n)$** jest listą, to dla dowolnych **i** oraz **j**, takich że **$1 \leq i \leq j \leq n$**, lista **$(a_i, a_{i+1}, \ldots, a_j)$** jest podlistą (ang. sublist) listy **L**. Oznacza to, że podlista jest tworzona od pewnej pozycji **i**, oraz że zawiera wszystkie elementy aż do pozycji **j**.
  - Lista pusta jest podlistą dowolnej listy.
- **Przedrostek** (ang. prefix)
  - Przedrostkiem listy jest dowolna podlista rozpoczynająca się na początku tej listy (czyli **i=1**).
- **Przyrostek** (ang. suffix)
  - Przyrostek jest dowolna podlista kończąca się wraz z końcem listy (czyli **j=n**).
  - Lista pusta jest zarówno przedrostkiem jak i przyrostkiem.

Podstawowa terminologia

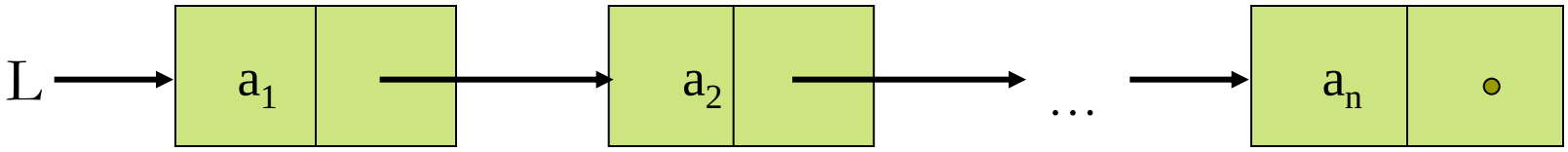
- **Podciąg**
  - Jeśli **L = (a_1, a_2, …, a_n)** jest listą, to lista utworzona przez wyciągnięcie zera lub większej liczby elementów z listy **L** jest podciągiem listy **L**.
  - Pozostałe elementy, które także tworzą podciąg, muszą występować w tej samej kolejności, w której występowały na oryginalnej liście **L**.
- **Pozycja elementu na liście**
  - Każdy element na liście jest związany z określoną pozycja.
    Jeśli **(a_1, a_2, …, a_n)** jest listą oraz **n≥1**, to o elemencie **a_1** mówimy, że jest **pierwszym** elementem, o **a_2** że jest **drugim** elementem, itd. aż dochodzimy do elementu **a_n** o którym mówimy że jest **ostatnim** elementem listy.

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.

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***)*.

Słownik

- Często stosowaną w programach komputerowych strukturą danych jest zbiór, na którym chcemy wykonywać operacje:
  - wstawianie nowych elementów do zbioru (ang. **insert**)
  - usuwanie elementów ze zbioru (ang. **delete**)
  - wyszukiwanie jakiegoś elementu w celu sprawdzenia, czy znajduje się w danym zbiorze (ang. **find**)
- Taki zbiór będziemy nazywać **słownikiem** (niezależnie od tego jakie elementy zawiera).

Abstrakcyjny typ danych = słownik

Struktura danych → lista jednokierunkowa

Abstrakcyjny typ danych = słownik
Abstrakcyjna implementacja = lista
Implementująca struktura danych = lista jednokierunkowa

- Słownik zawiera zbiór elementów $\{a_1, a_2, \ldots, a_n\}$.
- Uporządkowanie elementów w zbiorze nie ma znaczenia.
- **Operacje:**
  - **wstawianie** **x** do słownika **D**,
  - **usuwanie** elementu **x** ze słownika **D**,
  - **wyszukiwanie**, czyli sprawdzanie czy element **x** znajduje się w słowniku **D**.

Struktura danych → lista jednokierunkowa

- **Wyszukiwanie:**
  - Aby zrealizować tą operację musimy przeanalizować każdą komórkę listy reprezentująca słownik **D**, by przekonać się czy zawiera on szukany element **x**.
    - Jeśli tak odpowiedź jest „prawda".
    - Jeśli dojdziemy do końca listy i nie znajdziemy elementu odpowiedź jest „fałsz".
  - Wyszukiwanie może być zaimplementowane rekurencyjnie.

Dla listy **L** o długości **n** operacja wyszukiwania **$T(n) = O(n)$.**
Podstawa: **$T(0) = O(1)$**, ponieważ jeśli lista **L=NULL**, nie wykonujemy żadnego wywoływania rekurencyjnego.
Indukcja: **$T(n)=T(n-1)+O(1)$**.

  - Średni czas wykonywania operacji wyszukiwania jest O(n/2).
- **Usuwanie:**
  - Aby zrealizować tą operację musimy przeanalizować każdą komórkę listy reprezentującą słownik **D**, by przekonać się czy zawiera on szukany element **x**. Jeśli tak, następuje usunięcie elementu **x** z listy.
  - Ta operacja może być zaimplementowana rekurencyjnie, czas wykonania jest ***O*(n)**, średni czas wykonania jest ***O*(n/2)**.

Struktura danych → lista jednokierunkowa

- **Wstawianie:**
  - Aby wstawić **x** musimy sprawdzić, czy takiego elementu nie ma już na liście (jeśli jest nie wykonujemy żadnej operacji).
  - Jeśli lista nie zawiera elementu **x** dodajemy go do listy. Miejsce w którym go dodajemy nie ma znaczenia, np. dodajemy go na końcu listy po dojściu do wskaźnika **NULL**.
  - Podobnie jak w przypadku operacji wyszukiwania i usuwania, jeśli nie znajdziemy elementu **x** na liście dochodzimy do jej końca co wymaga czasu $O(n)$.
  - Średni czas wykonywania operacji wstawiania jest $O(n)$.

Słownik jako abstrakcyjny typ danych nie dopuszcza duplikatów (z definicji) ale struktura danych która go implementuje (lista jednokierunkowa) **może te duplikaty dopuszczać.**

Struktura danych → lista jednokierunkowa z duplikatami

- **Wstawianie:**
  - tworzymy tylko nową komórkę: **T(n) = *O*(1)**.
- **Wyszukiwanie:**
  - wygląda tak samo, możemy tylko musieć przeszukać dłuższą listę: **T(n) = *O*(n)**, średni czas jest **$O(n/2)$**.
- **Usuwanie:**
  - Wygląda tak samo ale zawsze musimy przejrzeć całą listę: **T(n) = *O*(n)**, średni czas również **$O(n)$**.

Struktura danych → lista jednokierunkowa

- Słownik jako abstrakcyjny typ danych nie wymaga uporządkowania ale struktura danych która go implementuje (*lista jednokierunkowa*) to uporządkowanie wprowadza.

- **Wstawianie, wyszukiwanie, usuwanie:**
  Musimy znaleźć właściwe miejsce na wstawienie, dla wszystkich T(n)= O(n/2).

Struktura danych → lista jednokierunkowa

- **Wstawianie, wyszukiwanie, usuwanie:**

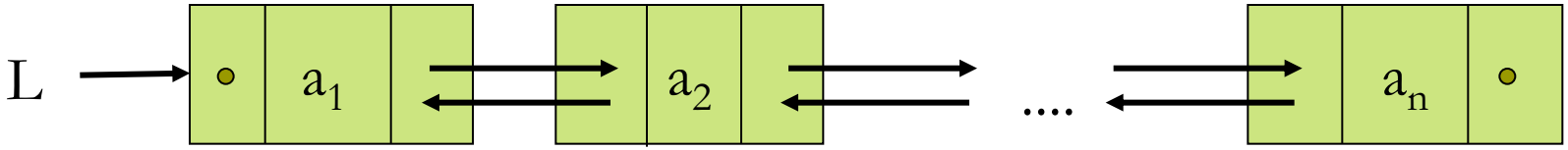
Lista	Wstawianie	Usuwanie	Przeszukiwanie
Brak Duplikatów	n/2 → n	n/2 → n	n/2 → n
Duplikaty	0	m	m/2 → m
Lista posortowana	n/2	n/2	n/2

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).**

Struktura danych → lista dwukierunkowa

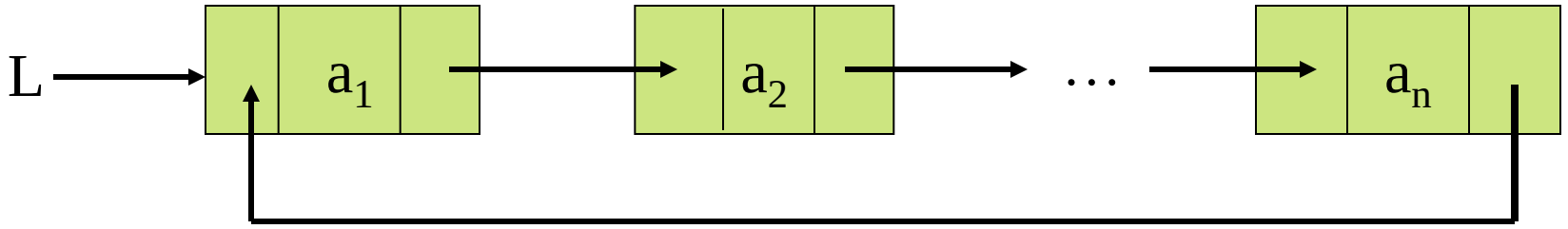
- W przypadku listy jednokierunkowej nie ma mechanizmu na przejście od dowolnej komórki do początku listy.
- Rozwiązaniem tego problemu jest lista dwukierunkowa – struktura danych umożliwiająca łatwe przemieszczanie się w obu kierunkach. Komórki listy dwukierunkowej zawierające liczby całkowite składają się z trzech pól. Dodatkowe pole zawiera wskaźnik do poprzedniej komórki na liście.
- Dla każdej komórki znamy wskaźnik poprzedniej i następnej (ang. **previous** i **next**).



- Zaleta:
  - operacja usuwania jest ***O*(1)** ponieważ mając wskaźnik do elementu który chcemy usunąć nie musimy przeglądać listy aby znaleźć komórkę poprzedzającą tą którą usuwamy (za pomocą pól **previous** i **next**).

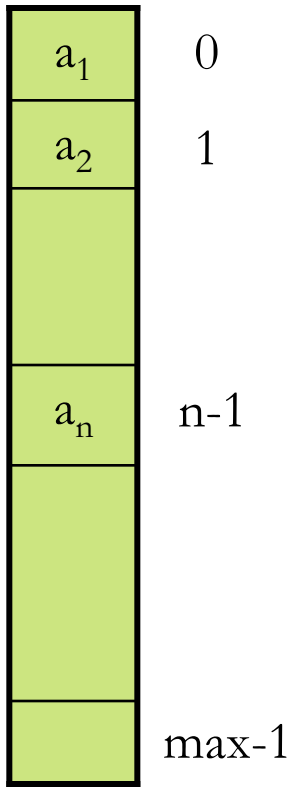
Struktura danych → lista cykliczna

- Czasem nie potrzebujemy przechodzić listy wstecz, a jedynie mieć dostęp z każdego z elementów do wszystkich innych.
- Prostym rozwiązaniem jest użycie listy cyklicznej. Na takiej liście, dla ostatniego elementu **next ≠ NULL**, zamiast tego **next** wskazuje na początek listy.
- Nie ma więc pierwszego i ostatniego elementu, ponieważ elementy tworzą cykl. Potrzebny jest co najmniej jeden zewnętrzny wskaźnik do jakiegoś elementu listy.
- Jeśli lista zawiera tylko jeden element, to musi on wskazywać sam na siebie. Jeżeli lista jest pusta to wartością każdego zewnętrznego wskaźnika do niej jest **NULL**.



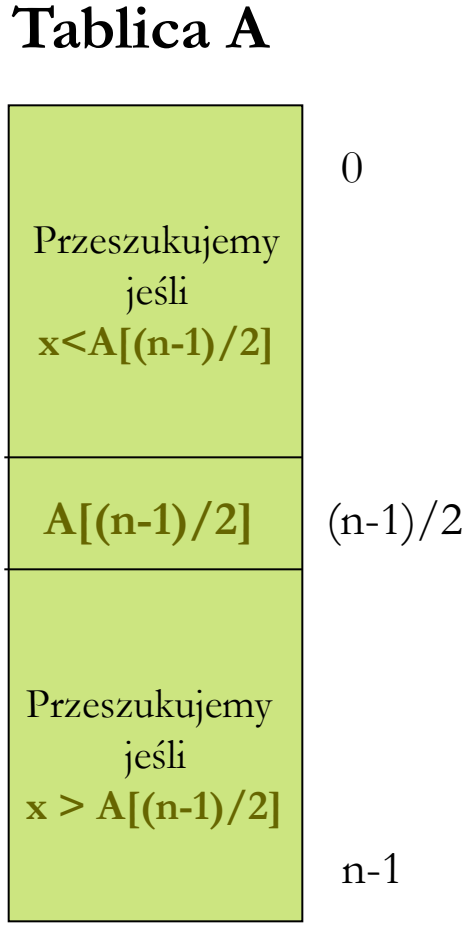
Struktura danych → lista oparta na tablicy

- 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.



Struktura danych → lista oparta na tablicy

- **Operacja przeszukiwania liniowego:**
  - Przeglądamy wszystkie elementy występujące w liście **L** (a więc w implementującej ją macierzy),
  - Operacja jest **T(n) = O(n)**.
- **Operacja przeszukiwania binarnego:** (Możliwa jeśli lista była posortowana)
  - Znajdujemy indeks środkowego elementu, czyli **m=(n-1)/2**.
  - Porównujemy element **x** z elementem **A[m]**.
  - Jeśli **x=A[m]** kończymy, jeśli **x<A[m]** przeszukujemy podlistę **A[0, m-1]**, jeśli **x>A[m]** przeszukujemy podlistę **A[m+1, n-1]**.
  - Powtarzamy operację rekurencyjnie.
  - Operacja jest **T(n) = *O*(log n)**.



Stos

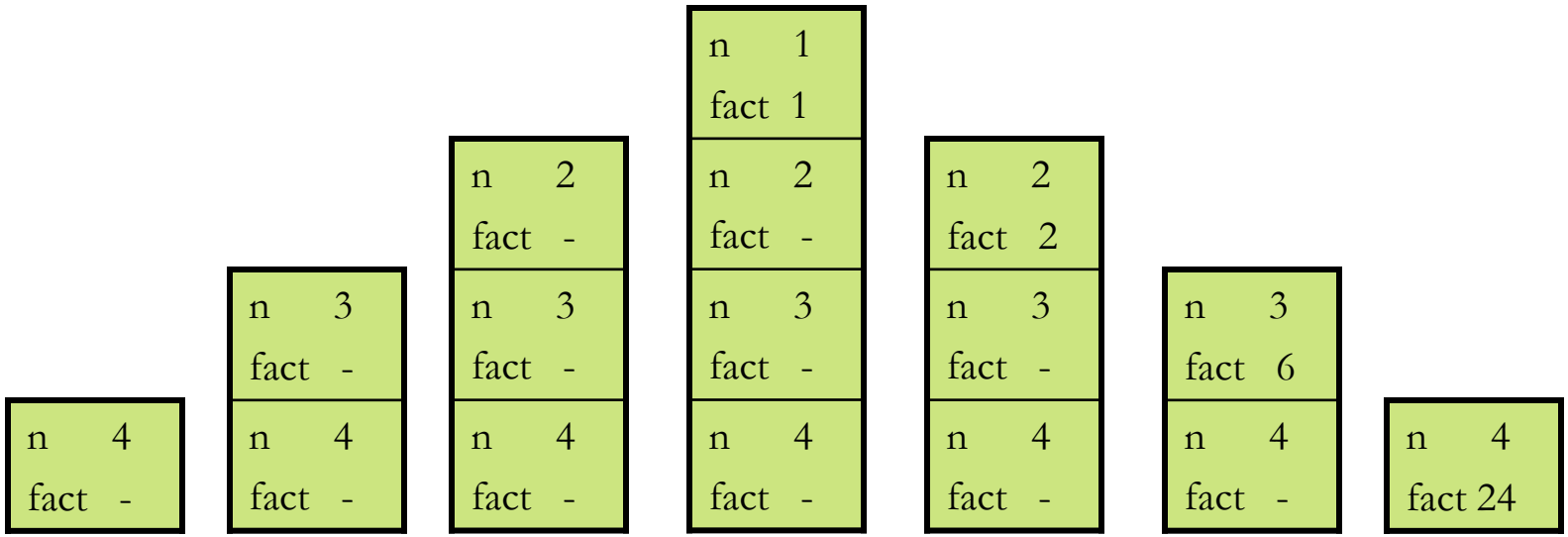
Abstrakcyjny typ danych
= stos
Implementująca struktura danych
= lista jednokierunkowa, lista oparta na tablicach

- **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 $a_1, a_2, \ldots, a_n$ 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 $\mathbf{T(n) = \mathit{O}(1)}$.

Więcej abstrakcyjnych typów danych...

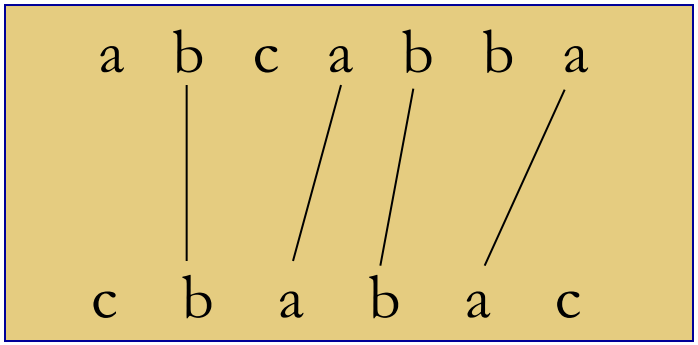
Abstrakcyjny typ danych	Abstrakcyjna implementacja	Struktura danych
Słownik	Drzewa przeszukiwania binarnego	Struktura lewe dziecko – prawe dziecko
Kolejka priorytetowa	Zrównoważone drzewo częściowo uporządkowane	Kopiec
Słownik	Lista	1. Lista jednokierunkowa 2. Tablica mieszająca
Stos	Lista	1. Lista jednokierunkowa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

Najdłuższy wspólny podciąg

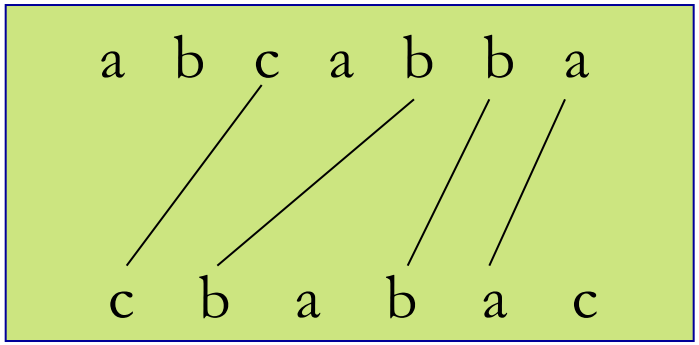
- Mamy dwie listy i chcemy je za sobą porównać, tzn. dowiedzieć się co je różni. Problem ten może mieć wiele różnych zastosowań.
- Traktujemy oba pliki jako sekwencje symboli:
  $\mathbf{x = a_1, a_2, \ldots, a_m}$ $\mathbf{y = b_1, b_2, \ldots, b_m}$
  gdzie $\mathbf{a_i}$ reprezentuje **i**-ty wiersz pierwszego pliku, natomiast $\mathbf{b_j}$ reprezentuje **j**-ty wiersz drugiego pliku.
- Abstrakcyjny symbol $\mathbf{a_i}$ może być w rzeczywistości dużym obiektem, np. całym zdaniem.
- Aby znaleźć długość najdłuższego wspólnego podciągu list **x** i **y**, musimy znaleźć długości najdłuższych wspólnych podciągów wszystkich par przedrostków, gdzie jeden pochodzi z listy **x**, drugi z listy **y**. (Przedrostek to początkowa podlista listy.)
- Jeśli **i=0** lub **j=0** to oczywiście wspólny przedrostek ma długość **0**.
- Jeśli **i≠0** oraz **j≠0** to wygodną metodą poszukiwania najdłuższego wspólnego podciągu jest dopasowywanie kolejnych pozycji dwóch badanych ciągów.

Najdłuższy wspólny podciąg

Przyporządkowanie dla **baba**



Przyporządkowanie dla **cbba**



- Dopasowane pozycje musza zawierać takie same symbole a łączące je linie nie mogą się przecinać.

Najdłuższy wspólny podciąg

- Rekurencyjna definicja dla **L(i, j)**, czyli długości najdłuższego wspólnego podciągu listy $(a_1, a_2, \ldots, a_i)$ oraz $(b_1, b_2, \ldots, b_j)$.

- **Podstawa:**
  - Jeśli **i+j = 0**, to zarówno **„ i”** jak i **„j”** są równe **0**, zatem najdłuższym wspólnym podciągiem jest **L(0, 0) = 0**.
- **Indukcja:**
  - Rozważmy **„i”** oraz **„j”**, przypuśćmy, że mamy już wyznaczone **L(g, h)** dla dowolnych **g** i **h** spełniających nierówność **g+h < i+j**.
  - Musimy rozważyć następujące przypadki:
    - Jeśli **i** lub **j** są równe **0**, to **L(i, j) =0**.
    - Jeśli **i>0** oraz **j>0** oraz $a_i \neq b_j$, to **L(i, j) = max (L(i, j-1), L(i-1, j))**.
    - Jeśli **i>0** oraz **j>0** oraz $a_i = b_j$, to **L(i, j) = 1 + L(i-1, j-1)**.

Najdłuższy wspólny podciąg

- Ostatecznie naszym celem jest wyznaczenie **L(m, n)**.
- Jeżeli na podstawie podanej poprzednio definicji napiszemy program rekurencyjny to będzie on działał w czasie wykładniczym, zależnym od mniejszej wartości z pary **m, n**.
- Możemy znacznie zwiększyć wydajność rozwiązania jeżeli zbudujemy dwuwymiarową tabelę w której będziemy przechowywali wartości **L(i, j)**.
- Wyznaczenie pojedynczego elementu tabeli wymaga jedynie czasu **O(1)**, zatem skonstruowanie całej tabeli dla najdłuższego podciągu zajmie ***O*(m•n)** czasu.
- Aby tak się działo, elementy należy wypełnić w odpowiedniej kolejności. (np. wypełniać wierszami, a wewnątrz każdego wiersza kolumnami)

- **Zastosowanie techniki wypełniania tabeli to element tzw. programowania dynamicznego.**

Pseudokod programu, który wypełnia tabelę

```
for (i = 0; i <= m; i++)        L[i][0] = 0          // ustawianie wartości 0 dla
for (j = 0; j <= n; j++)        L[0][j] = 0          // przypadku 1 z naszej listy

for (i=1; i <=m ; i++){                              // dla każdego wiersza
      for (j=1; j <=n ; j++) {                       // dla każdej kolumny
         if( a[i] != b[j] ) {                        // czyli przypadek 2 z naszej listy
            L[i][j] = max (L[i-1][j], L[i][j-1])
         }else {                                     // czyli przypadek 3 z naszej listy
            L[i][j] = 1 + L[i-1][j-1];
         }
      }
}
```

Czas działania kodu dla list o długości **m** i **n** wynosi $O(m \cdot n)$.

Przykład dla list: abcabba i cbabac

Numer w tablicy		0	1	2	3	4	5	6	7
			a	b	c	a	b	b	a
0		0	0	0	0	0	0	0	0
1	c	0	0	0	1	1	1	1	1
2	b	0	0	1	1	1	2	2	2
3	a	0	1	1	1	2	2	2	3
4	b	0	1	2	2	2	3	3	3
5	a	0	1	2	2	3	3	3	4
6	c	0	1	2	3	3	3	3	4

Ścieżka wskazująca na jeden z najdłuższych (długości 4) wspólnych podciągów: caba

Podsumowanie

- Ważnym modelem danych reprezentującym sekwencje elementów są listy.
- Do implementowania list możemy wykorzystać dwie struktury danych – listy jednokierunkowe i tablice.
- Listy umożliwiają prostą implementacją słowników, jednak efektywność takiego rozwiązania jest znacznie gorsza niż efektywność implementacji opartej na drzewach przeszukiwania binarnego.
  (Jest też gorsza od implementacji przy użyciu tablic mieszających, patrz następny wykład).
- Ważnymi specyficznymi odmianami list są stosy i kolejki.
- Stos jest wykorzystywany w tle do implementowania funkcji rekurencyjnych.
- Znajdowanie najdłuższego wspólnego podciągu za pomocą techniki znanej jako „programowanie dynamiczne” pozwala efektywnie rozwiązać ten problem.