Teoretyczne podstawy informatyki

Wykład 5b:
Model danych oparty na listach

http://hibiscus.if.uj.edu.pl/~erichter/Dydaktyka2009/TPI-2009

Słowem wstępu

- 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

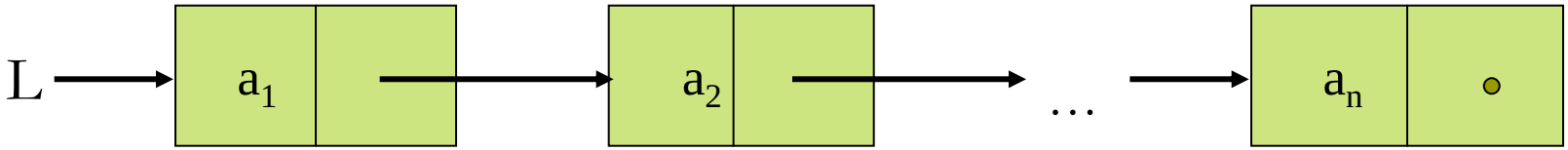
- **Podciąg**
  - Jeśli **L = (a_1, a_2, …, a_n)** jest listą, to lista utworzona przez wyciągnięcie zera lub większej liczby elementów z listy **L** jest podciągiem listy **L**.
  - Pozostałe elementy, które także tworzą podciąg, muszą występować w tej samej kolejności, w której występowały na oryginalnej 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:**

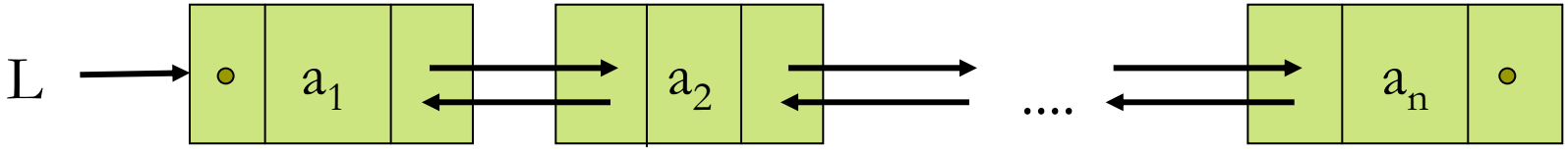
Lista	Wstawianie	Usuwanie	Przeszukiwanie
Brak Duplikatów	$n/2 \rightarrow n$	$n/2 \rightarrow n$	$n/2 \rightarrow n$
Duplikaty	0	m	$m/2 \rightarrow m$
Lista posortowana	n/2	n/2	n/2

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).**

Struktura danych → lista dwukierunkowa

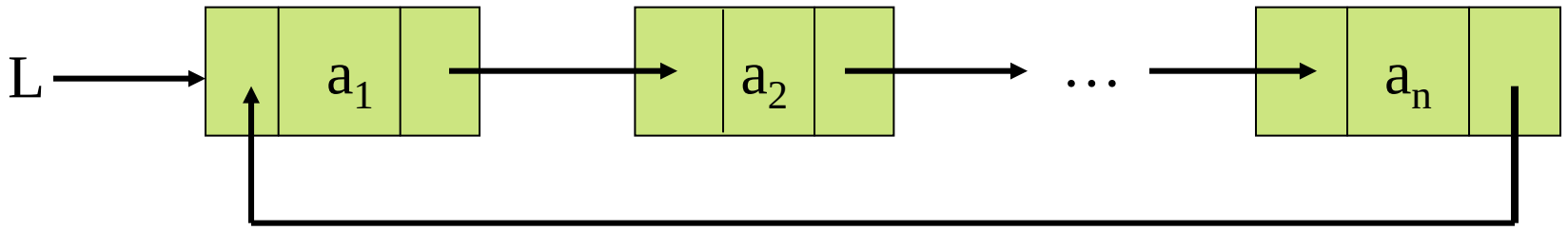
- W przypadku listy jednokierunkowej nie ma mechanizmu na przejście od dowolnej komórki do początku listy.
- Rozwiązaniem tego problemu jest lista dwukierunkowa – struktura danych umożliwiająca łatwe przemieszczanie się w obu kierunkach. Komórki listy dwukierunkowej zawierające liczby całkowite składają 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**).

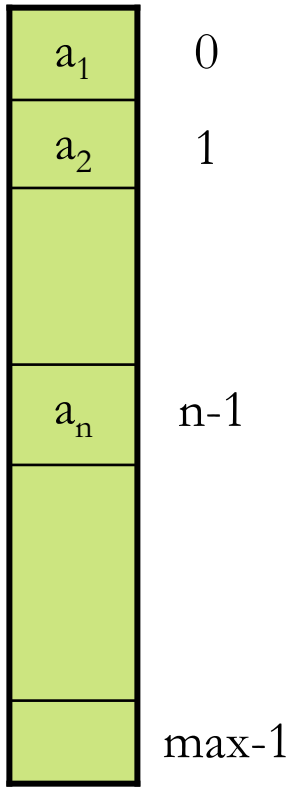
Struktura danych → lista cykliczna

- 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**.



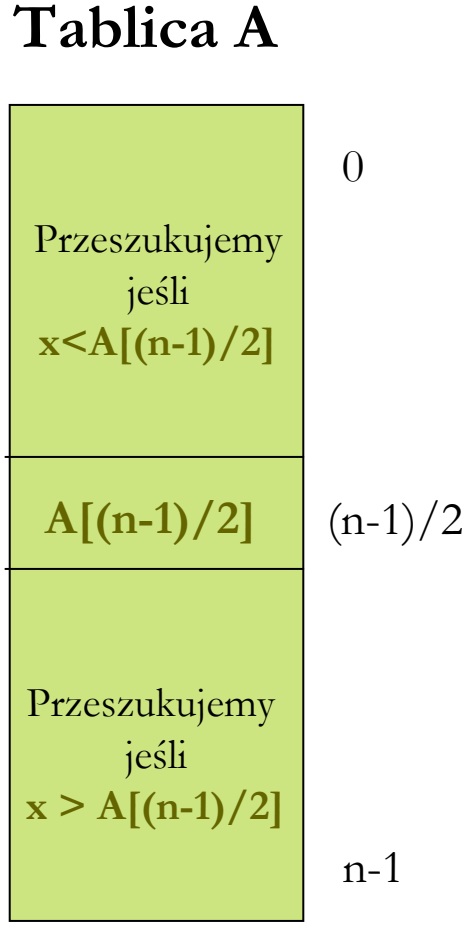
Struktura danych → lista oparta na tablicy

- 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 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

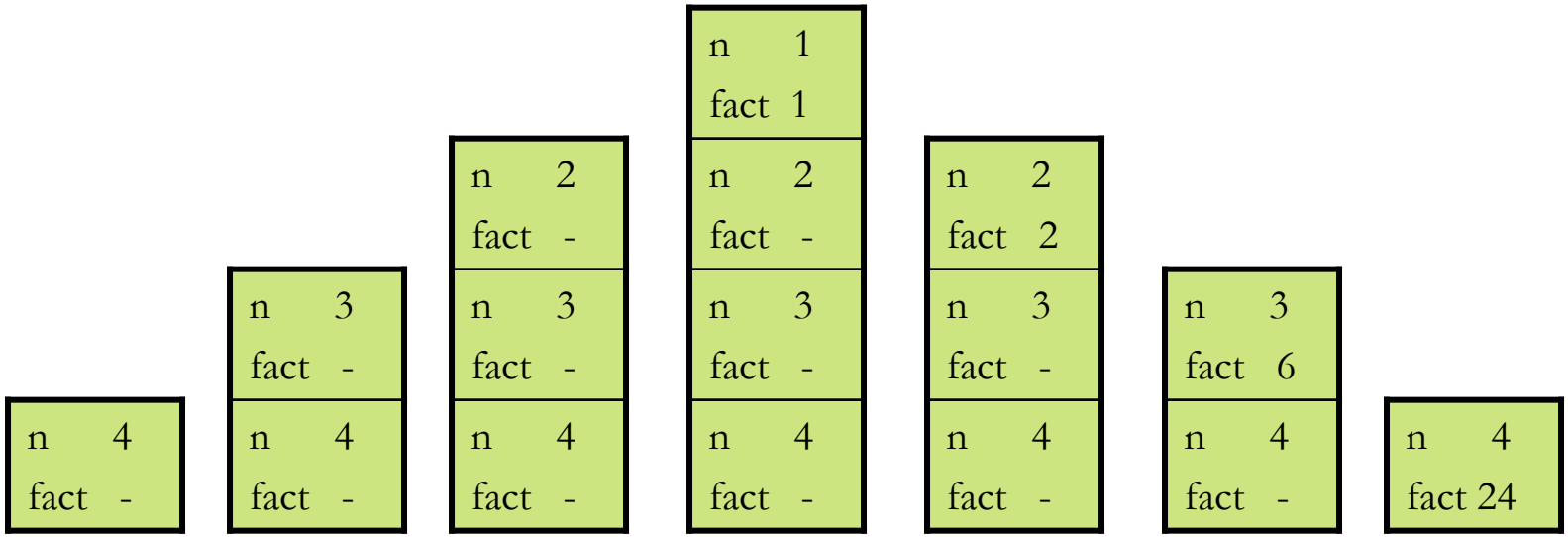
Abstrakcyjny typ danych
= stos
Implementująca struktura danych
= lista jednokierunkowa, lista oparta na tablicach

- **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...

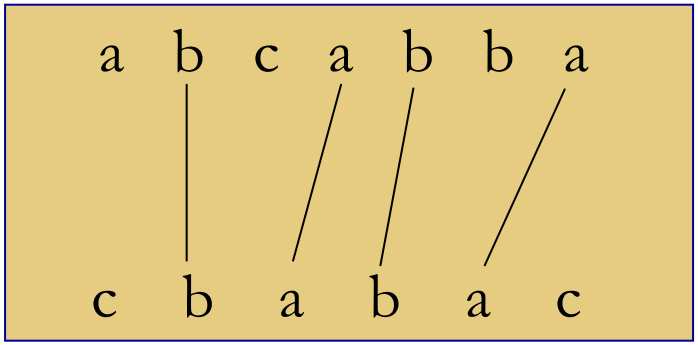
Abstrakcyjny typ danych	Abstrakcyjna implementacja	Struktura danych
Słownik	Drzewa przeszukiwania binarnego	Struktura lewe dziecko – prawe dziecko
Kolejka priorytetowa	Zrównoważone drzewo częściowo uporządkowane	Kopiec
Słownik	Lista	1. Lista jednokierunkowa 2. Tablica mieszająca
Stos	Lista	1. Lista jednokierunkowa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

Najdłuższy wspólny podciąg

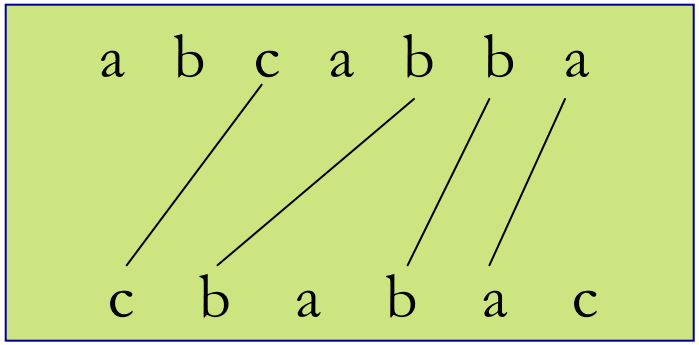
- Mamy dwie listy i chcemy je za sobą porównać, tzn. dowiedzieć się co je różni. Problem ten może mieć wiele różnych zastosowań.
- Traktujemy oba pliki jako sekwencje symboli:
  $\mathbf{x = a_1, a_2, \ldots, a_m}$ $\mathbf{y = b_1, b_2, \ldots, b_m}$
  gdzie $\mathbf{a_i}$ reprezentuje **i**-ty wiersz 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•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.