

Listy

# Zbiór dynamiczny

- Zbiór dynamiczny to zbiór wartości pochodzących z pewnego określonego uniwersum, którego zawartość zmienia się w trakcie działania programu.
- Elementy zbioru dynamicznego musimy co najmniej umieć porównać pod względem identyczności (czy dwa elementy są równe albo różne).
- W multizbiorze elementy mogą się powtarzać.

# Dynamiczne struktury danych

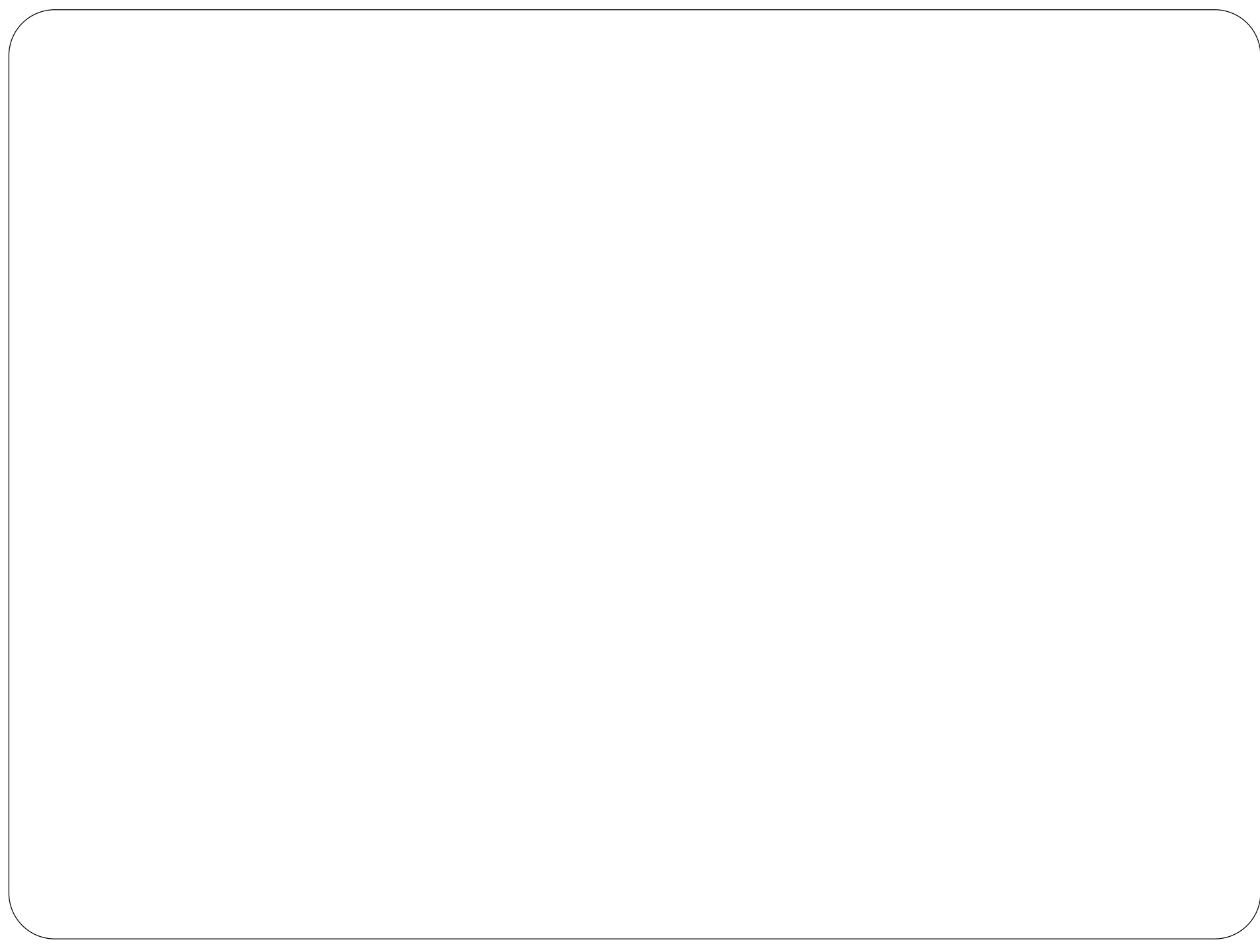
- Dynamiczna struktura danych, to struktura danych pozwalająca na przechowywanie zbioru dynamicznego; rozmiar tej struktury dostosowuje się do rozmiaru danych.
- W zbiorze dynamicznym musimy umieć realizować operację dodania nowego elementu do zbioru i usunięcia ze zbioru wskazanego elementu.
- Tablica dynamiczna jest dynamiczną strukturą danych.
- Zastosowanie: przechowywanie pewnego zbioru danych, którego zawartość będzie się zmieniać w trakcie pracy programu i nie jesteśmy w stanie przewidzieć jak dużo tych danych będzie gromadzonych.

# Abstrakcyjne struktury danych

- **Abstrakcyjne struktury danych** (ang. ADT) to opis funkcjonalności danej struktury.
- Struktura danych jest **homogeniczna**, jeśli składa się z elementów tego samego typu.

# Słownik

- **Słownik** (ang. dictionary) to struktura danych pozwalająca efektywnie realizować następujące operacje:
  - `insert(x)` – dodanie nowego elementu  $x$  do zbioru dynamicznego,
  - `delete(x) / remove(x)` – usunięcie elementu o wartości  $x$  ze zbioru dynamicznego,
  - `search(x)` – sprawdzenie czy w zbiorze dynamicznym znajduje się element o wartości  $x$ .



# Lista

- **Lista** (ang. list) to homogeniczna struktura danych służąca do reprezentowania zbioru dynamicznego, w której elementy ułożone w ciąg (struktura sekwencyjna).
- Element listy nazywa się **węzłem** (ang. node); każdy węzeł zawiera pole `value` służące do przechowywania jednej wartości z pewnego określonego uniwersum oraz pole `next` ze wskaźnikiem na następny element listy (ostatni element listy ma w polu `next` wpisany wskaźnik pusty).

# Lista

- Pierwszy węzeł listy jest nazywany **głową** (ang. head) albo początkiem listy; reszta listy za pierwszym węzłem jest nazywana **ogonem** listy (ang. tail).
- Dostęp do elementów listy jest **sekwencyjny** – więc dojście do elementu k-tego wymaga przejścia przez kolejne elementy listy od pierwszego do docelowego.
- Zastosowanie: lista najlepiej nadaje się do danych, które będą przetwarzane sekwencyjnie.



# Lista

- **Lista jednokierunkowa** (ang. single linked list) to lista, po której można się poruszać tylko od głowy do ogona – w każdym węźle jest tylko wskaźnik do następnika.
- Lista dwukierunkowa (ang. double linked list) to lista, po której można się poruszać w obu kierunkach: w stronę głowy i w stronę ogona – w każdym węźle są dwa wskaźniki `next` do następnika i `prev` do poprzednika.
- Gdy dane pochodzą z uniwersum z porządkiem liniowym, to dane w liście można przechowywać w sposób uporządkowany – mamy wtedy do czynienia z **listą uporządkowaną**.

# Lista

- Lista cykliczna to lista, w której ostatni węzeł posiada wskaźnik do pierwszego węzła.
- Lista dwukierunkowa może być cykliczna.
- Lista z wartownikiem to lista, w której na końcu umieszczony jest węzeł zwany wartownikiem – wartownik nie przechowuje danych, pełni rolę pomocniczą w nawigacji po liście.
- Lista z wartownikiem może być cykliczna lub dwukierunkowa.

# Trwałe struktury danych

- **Trwałe struktury danych** to takie struktury, w których elementy tej struktury nie mogą być modyfikowane a jedynie zainicjalizowane w momencie ich tworzenia.
- Modyfikacja trwałych struktur danych polega na całkowitej wymianie modyfikowanych elementów na nowe (funkcyjne przetwarzanie danych w zbiorach dynamicznych).

# Lista

Wyszukiwanie wartości w liście jednokierunkowej – wersja iteracyjna

```
search(node *n, x) -> boolean
{
    while (n.value != x) {
        if (n.next != null) n := n.next;
        else return false;
    }
    return true;
}
```

Czas:  $O(n)$  gdzie  $n$  to ilość elementów na liście

Pamięć:  $O(1)$

# Lista

Wyszukiwanie wartości w liście jednokierunkowej – wersja rekurencyjna

```
search(node *n, x) -> boolean
{
    if (n == null) return false;
    if (n.value == x) return true;
    return search(n.next, x);
}
```

Czas:  $O(n)$  gdzie  $n$  to ilość elementów na liście

Pamięć:  $O(n)$  zależy od liczby wywołań rekurencyjnych

# Lista

Wyszukiwanie wartości w liście jednokierunkowej z wartownikiem  
– wersja iteracyjna

```
search(node *n, x, node *sentinel) -> boolean
{
    sentinel.value := x;
    while (n.value != x) n := n.next;
    return n != sentinel;
}
```

Czas:  $O(n)$  gdzie  $n$  to ilość elementów na liście

Pamięć:  $O(1)$

# Lista

Wyszukiwanie wartości w liście posortowanej – wersja rekurencyjna

```
search(node *n, x) -> boolean
{
    if (n == null) return false;
    if (n.value == x) return true;
    if (n.value > x) return false;
    return search(n.next, x);
}
```

Czas:  $O(n)$  gdzie  $n$  to ilość elementów na liście

Pamięć:  $O(n)$  zależy od liczby wywołań rekurencyjnych

# Lista

- Wstawianie elementu do listy nieuporządkowanej:
  - na początek listy,
  - na koniec listy,
  - na zadaną pozycję.
- Wstawianie elementu do listy uporządkowanej:
  - wstawiamy zachowując uporządkowanie
- Usuwanie elementu z listy:
  - usuwanie elementu z zadanej pozycji,
  - usuwanie elementu o zadanej wartości.



# Listy

- Technika zwracania wskaźnika do struktury po zmodyfikowaniu (semitrwałe struktury danych) – w technice tej aktualizujemy za każdym razem wskaźnik do podstruktury, na której rekurencyjnie wywołujemy procedurę modyfikującą.
- Przykład: wstawienie elementu na zadaną pozycję:

```
insert(node *n, x, pos) -> node*
{
    if (pos < 0) error;
    if (n == null and pos > 0) error;
    if (pos > 0) {
        n.next := insert(n.next, x, pos-1);
        return n;
    }
    else return new node(x, n);
}
```

wywołanie:

```
head := insert(head, x);
```

# Listy

- Operacje słownikowe na liście n-elementowej wymagają:
  - czasu  $O(n)$ ,
  - pamięci  $O(1)$  gdy używamy iteracji albo  $O(n)$  gdy korzystamy z rekurencji.