Algorytmy i struktury danych

Wykład 3

Krzysztof M. Ocetkiewicz
Krzysztof.Ocetkiewicz@eti.pg.edu.pl

Katedra Algorytmów i Modelowania Systemów, WETI, PG

Listy dynamiczne

- tablica zajmuje stały rozmiar pamięci niezależnie od tego, ile trzymamy w niej elementów
- lista dynamiczna zajmuje tylko tyle pamięci, ile trzeba
- wstawienie/usunięcie elementu do tablicy zachowujące kolejność zajmuje $O(n)$ czasu
- wstawienie/usunięcie elementu do listy zachowujące kolejność zajmuje $O(1)$ czasu (jeżeli wiemy, gdzie wstawiać)
- dużo łatwiej jest "ciąć" i łączyć listy dynamiczne niż tablice

Lista jednokierunkowa

- każdy element listy (węzeł) przechowuje pojedynczą daną (w sensie obiektu — może się składać z wielu zmiennych) oraz wskaźnik na następny element
- np.

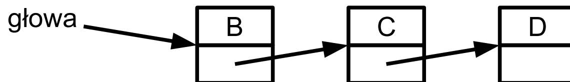
```
struct Wezel {
     int dana;
     struct Wezel *next;
     };
```

Lista jednokierunkowa

- oprócz tego, potrzebujemy jeszcze wskaźnik na pierwszy element listy (głowę)
  - musimy wiedzieć, gdzie zaczyna się nasza lista
  - jeśli znamy adres pierwszego węzła, to znajdziemy wszystkie
  - głowa jest dobrym kandydatem na właściciela całej listy
- zakładamy, że gdy lista jest pusta, głowa jest równa *nullptr*
- wskaźnik na głowę w pewnym sensie identyfikuje listę — jeżeli dwie głowy są równe (pokazują na ten sam element), to jest to ta sama lista

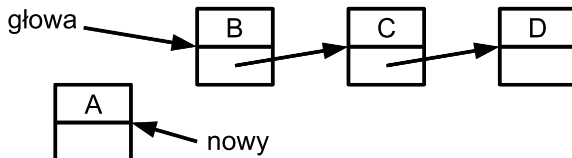
Wstawianie na początek

1: $nowy$ = **new** `Wezel`
2: $nowy.dana = dana$
3: $nowy.next = glowa$
4: $glowa = nowy$



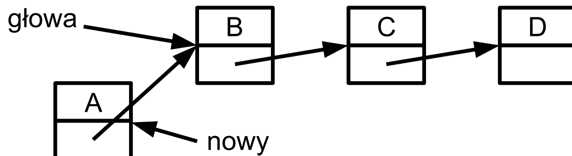
Wstawianie na początek

1: $nowy = \textbf{new}\ \texttt{Wezel}$
2: $nowy.dana = dana$
3: $nowy.next = glowa$
4: $glowa = nowy$



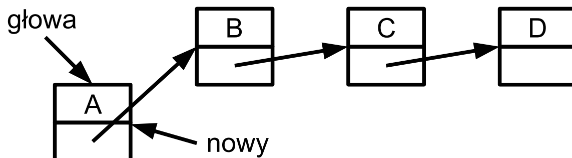
Wstawianie na początek

1: $nowy = $ **new** `Wezel`
2: $nowy.dana = dana$
3: $nowy.next = glowa$
4: $glowa = nowy$



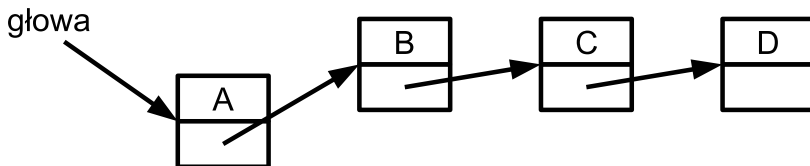
Wstawianie na początek

1: $nowy = \textbf{new}\ \texttt{Wezel}$
2: $nowy.dana = dana$
3: $nowy.next = glowa$
4: $glowa = nowy$



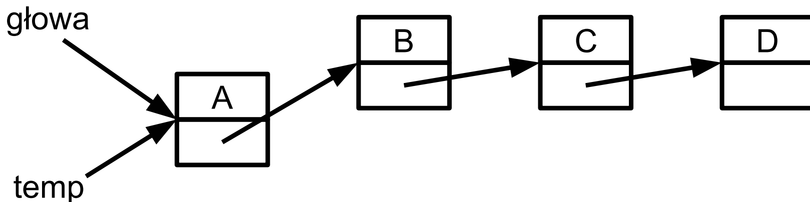
Usuwanie głowy

1: **if** $glowa = nullptr$ **then**
2:     **return** "pusta lista"
3: **end if**
4: $temp = glowa$
5: $glowa = glowa.next$
6: zwolnij pamięć zajmowaną przez *temp*



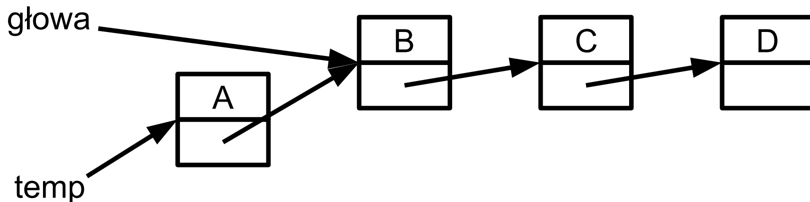
Usuwanie głowy

1: **if** $glowa = nullptr$ **then**
2: **return** "pusta lista"
3: **end if**
4: $temp = glowa$
5: $glowa = glowa.next$
6: zwolnij pamięć zajmowaną przez *temp*



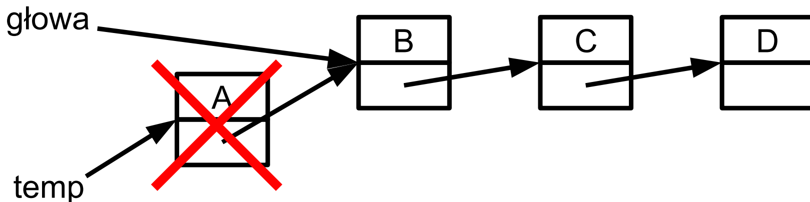
Usuwanie głowy

1: **if** $glowa = nullptr$ **then**
2: **return** "pusta lista"
3: **end if**
4: $temp = glowa$
5: $glowa = glowa.next$
6: zwolnij pamięć zajmowaną przez $temp$



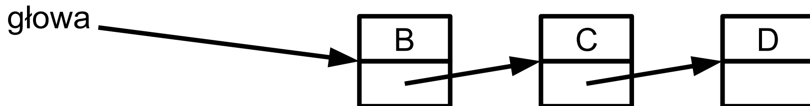
Usuwanie głowy

1: **if** *glowa* = *nullptr* **then**
2:     **return** “pusta lista”
3: **end if**
4: *temp* = *glowa*
5: *glowa* = *glowa.next*
6: zwolnij pamięć zajmowaną przez *temp*



Usuwanie głowy

1: **if** *glowa* = *nullptr* **then**
2:     **return** "pusta lista"
3: **end if**
4: *temp* = *glowa*
5: *glowa* = *glowa.next*
6: zwolnij pamięć zajmowaną przez *temp*



Wstawianie za elementem

- jeżeli mamy wskaźnik na pewien element (*elem*), jesteśmy w stanie wstawić za nim nowy

1: $nowy = \textbf{new}$ `Wezel`
2: $nowy.dana = dana$
3: $nowy.next = elem.next$
4: $elem.next = nowy$

Usuwanie następnika

- analogicznie, jeżeli mamy wskaźnik na pewien element (*elem*), możemy także usunąć jego następnika

1: $temp = elem.next$
2: $elem.next = elem.next.next$
3: zwolnij pamięć zajmowaną przez *temp*

Przeglądanie listy

1: $elem = glowa$
2: **while** $elem \neq nullptr$ **do**
3: zrób co trzeba z $elem$
4: $elem = elem.next$
5: **end while**

Wyszukiwanie elementu

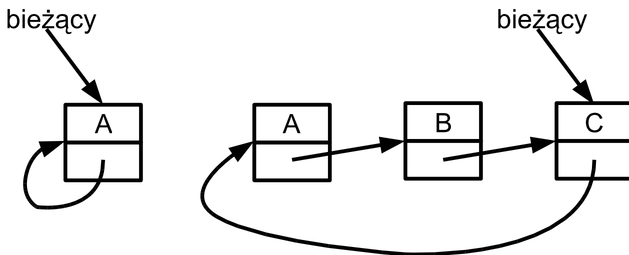
1: $p = glowa$
2: **while** $p \neq nullptr$ **do**
3:     **if** $w = p.dana$ **then**
4:         **return** p
5:     **end if**
6:     $p = p.next$
7: **end while**
8: **return** "nie znaleziono"

Wyszukiwanie elementu

- czy wyszukiwanie binarne nadaje się do list uporządkowanych?
- jaka będzie jego złożoność?

Lista cykliczna

- "zapętlamy" listę jednokierunkową: następnikiem ostatniego elementu staje się pierwszy
- zamiast głowy, mamy wskaźnik na bieżący element
- większość operacji wykonuje się tak samo jak w przypadku listy jednokierunkowej



Wstawianie i usuwanie elementu

- wstawienie elementu za bieżącym elementem wygląda dokładnie tak samo jak w przypadku listy jednokierunkowej
- jeżeli chcielibyśmy wstawić element przed bieżącym elementem, musimy "obiec" listę dookoła aby znaleźć poprzednika bieżącego elementu
- tak samo w przypadku usuwania:
  - następnika elementu bieżącego usuwamy jak w liście jednokierunkowej
  - aby usunąć bieżący element, musimy wykonać rundę dookoła listy, aż do poprzednika bieżącego elementu

Usuwanie bieżącego elementu

1: $elem = biezacy$
2: **while** $elem.next \neq biezacy$ **do**
3: $\quad elem = elem.next$
4: **end while**
5: $elem.next = elem.next.next$
6: **if** $elem = biezacy$ **then**
7: $\quad elem = nullptr$
8: **end if**
9: zwolnij pamięć zajmowaną przez *biezacy*
10: $biezacy = elem$

Przeglądanie listy

- przeglądamy podobnie jak listę jednokierunkową
- przeglądamy nie do *nullptr* ale do momentu, gdy wrócimy do elementu, z którego zaczęliśmy

1: $elem = biezacy$
2: **repeat**
3: zrób co trzeba z $elem$
4: $elem = elem.next$
5: **until** $elem \neq biezacy$

Przykład zastosowania

- mamy zbudować listę jednokierunkową z elementów $e_1, e_2, \ldots e_n$ w czasie $O(n)$
- zwykłe dodawanie na początek da nam listę $[e_n, e_{n-1}, \ldots, e_1]$
- tworzymy listę cykliczną
- wstawiamy elementy za elementem bieżącym i po każdym wstawieniu ustawiamy nowy element jako bieżący
- po dodaniu wszystkiego robimy krok w przód jednocześnie rozłączając listę

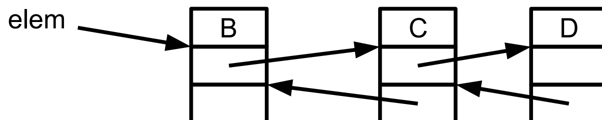
Lista dwukierunkowa

- rozbudowujemy listę jednokierunkową o wskaźnik na poprzedni element
- jak poprzednio, ostatni element ma następnika *nullptr*
- pierwszy element ma poprzednika *nullptr*
- np.

```
struct Wezel {
     int dana;
     struct Wezel *next;
     struct Wezel *prev;
     };
```

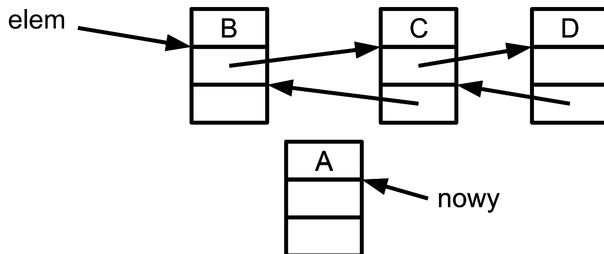
Wstawianie za elementem

1: $nowy = $ **new** `Wezel`
2: $nowy.dana = dana$
3: $nowy.next = elem.next$
4: $nowy.prev = elem$
5: **if** $nowy.next \neq nullptr$ **then** $nowy.next.prev = nowy$
6: **if** $nowy.prev \neq nullptr$ **then** $nowy.prev.next = nowy$



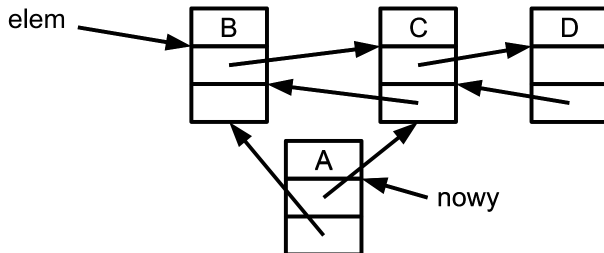
Wstawianie za elementem

1: $nowy = \textbf{new}$ `Wezel`
2: $nowy.dana = dana$
3: $nowy.next = elem.next$
4: $nowy.prev = elem$
5: **if** $nowy.next \neq nullptr$ **then** $nowy.next.prev = nowy$
6: **if** $nowy.prev \neq nullptr$ **then** $nowy.prev.next = nowy$



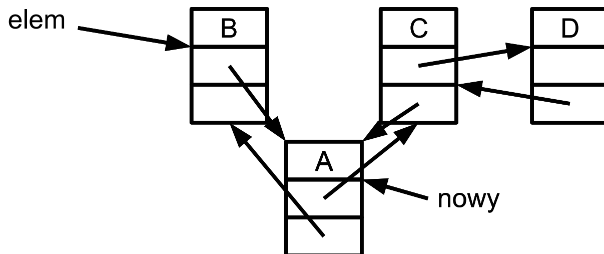
Wstawianie za elementem

1: $nowy = \textbf{new}\ \texttt{Wezel}$
2: $nowy.dana = dana$
3: $nowy.next = elem.next$
4: $nowy.prev = elem$
5: **if** $nowy.next \neq nullptr$ **then** $nowy.next.prev = nowy$
6: **if** $nowy.prev \neq nullptr$ **then** $nowy.prev.next = nowy$



Wstawianie za elementem

1: $nowy = \textbf{new}\ \texttt{Wezel}$
2: $nowy.dana = dana$
3: $nowy.next = elem.next$
4: $nowy.prev = elem$
5: **if** $nowy.next \neq nullptr$ **then** $nowy.next.prev = nowy$
6: **if** $nowy.prev \neq nullptr$ **then** $nowy.prev.next = nowy$



Wstawianie przed elementem

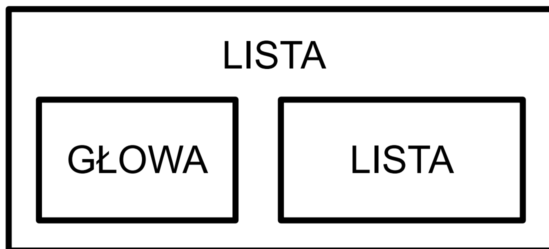
- wstawianie przed elementem — analogicznie (zamieniamy *prev* na *next* i *next* na *prev*)

Usuwanie węzła

```
1:  if elem.next ≠ nullptr then
2:          elem.next.prev = elem.prev
3:  end if
4:  if elem.prev ≠ nullptr then
5:          elem.prev.next = elem.next
6:  end if
7:  if biezacy = elem then
8:          biezacy = elem.prev
9:          lub biezacy = elem.next
10:         lub biezacy = nullptr
11: end if
12: zwolnij pamięć zajmowaną przez elem
```

Lista jako struktura rekurencyjna

- każda lista (jednokierunkowa) składa się z głowy (danej, zawartości węzła) i ogona (listy)



Lista jako struktura rekurencyjna

- każda lista (jednokierunkowa) składa się z głowy (danej, zawartości węzła) i ogona (listy)

```
struct Lista {
    int glowa;
    Lista *ogon;
    };
```

Lista jako struktura rekurencyjna

- np. lista $4 \to 3 \to 2 \to 1 \to$ *nullptr* składa się z:
  - głowy: 4
  - ogona: listy $3 \to 2 \to 1 \to$ *nullptr*, który składa się z:
    - głowy: 3
    - ogona: listy $2 \to 1 \to$ *nullptr*, który składa się z: ...
- lista $1 \to$ *nullptr* składa się z głowy 1 i pustej listy

Lista jako struktura rekurencyjna

- chcemy napisać funkcję która wykona pewną operację na liście, zaczynając od pierwszego elementu
- piszemy funkcję `WYKONAJ` która pobiera listę (wskaźnik na głowę) jako argument
- funkcja `WYKONAJ`:
  - jeżeli lista jest pusta (wskaźnik jest równy *nullptr*), kończy się od razu
  - przetwarza daną (głowę)
  - trzeba jeszcze przetworzyć ogon (który jest listą)
  - ale przecież do przetwarzania listy służy funkcja `WYKONAJ`
  - wywołuje `WYKONAJ`(*ogon*)

Wypisywanie listy

- np. funkcja `WYPISZ` wypisująca wszystkie elementy listy

```
void Wypisz(Lista *lista) {
     if(lista == nullptr) return;
     cout << lista->glowa << ' ';
     Wypisz(lista->ogon);
     };
```

Wypisywanie listy od końca

- np. funkcja `WypiszOdw` wypisująca wszystkie elementy listy od końca

```
void WypiszOdw(Lista *lista) {
    if(lista == nullptr) return;
    WypiszOdw(lista->ogon);
    cout << lista->glowa << ' ';
    };
```

Ostatni element listy

- funkcja `Ostatni` zwracająca ostatni element listy
- jaka jest jej złożoność?

```
Lista *Ostatni(Lista *lista) {
     if(lista == nullptr) return nullptr;
     if(lista->ogon == nullptr) return lista;
     return Ostatni(lista->ogon);
     };
```

Wstawienie na koniec

- np. funkcja `WstawKoniec` wstawiająca element *v* na koniec listy
- będziemy zwracać nową listę, która jest rezultatem wstawienia

```
Lista *WstawKoniec(Lista *lista, int v) {
     if(lista == nullptr) {
          Lista *t = new Lista;
          t->glowa = v;
          t->ogon = nullptr;
          return t;
          };
     lista->ogon = WstawKoniec(lista->ogon, v);
     return lista;
     };
```

Odwracanie listy

- np. funkcja `Odwroc` odwracająca kolejność elementów na liście

```
Lista *Odwroc(Lista *lista) {
     if(lista == nullptr) return nullptr;
     Lista *t = Odwroc(lista->ogon);
     t = WstawKoniec(t, lista->glowa);
     delete lista;
     return t;
     };
```

Odwracanie listy

- albo tak:

```
Lista *Odwroc2(Lista *lista) {
     if(lista == nullptr) return lista;
     if(lista->ogon == nullptr) return lista;
     Lista *t = Odwroc2(lista->ogon);
     lista->ogon->ogon = lista;
     lista->ogon = nullptr;
     return t;
     };
```

Lista dynamiczna bez wskaźników

- jeżeli z jakiegoś powodu nie jesteśmy w stanie skorzystać ze wskaźników, możemy ich działanie zasymulować, używając tablicy i indeksów w niej
- nasze "wskaźniki" będą mogły jednak wskazywać tylko na elementy wewnątrz "umówionej" tablicy
- jako *nullptr* możemy użyć pewnej wyróżnionej wartości, najlepiej niepoprawnego indeksu, np. -1

Lista dynamiczna bez wskaźników

```
struct Wezel {
     int dana;
     int next;
     };
Wezel tab[100];
int pierwszy = 10;
tab[pierwszy].dana = 15; // dereferencja
tab[pierwszy].next = 7; // pobranie adresu
tab[tab[pierwszy].nast].dana = 15; // deref.*2
tab[pierwszy].next = -1; // ‘‘nullptr’’
```

Lista dynamiczna bez wskaźników

- np.

```
Wezel tablica[...];
int lista = ...;
...
void Wypisz(int glowa) {
     if(glowa == -1)
          { cout << '\n'; return; };
     cout << tablica[glowa].dana << ' ';
     Wypisz(tablica[glowa].next);
     };
```

Listy

	jednokierunkowa	dwukierunkowa
złożoność pamięciowa	$O(n)$	$O(n)$
wstawienie (początek)	$O(1)$	$O(1)$
wstawienie (koniec)	$O(n)$	$O(1)$
wstawienie przed	$O(n)$	$O(1)$
wstawienie za	$O(1)$	$O(1)$
usunięcie wskazanego	$O(n)$	$O(1)$
wyszukanie	$O(n)$	$O(n)$
modyfikacja wskaz.	$O(1)$	$O(1)$
trwałość wskaźników	tak	tak