Podstawy Programowania

Wykład IX

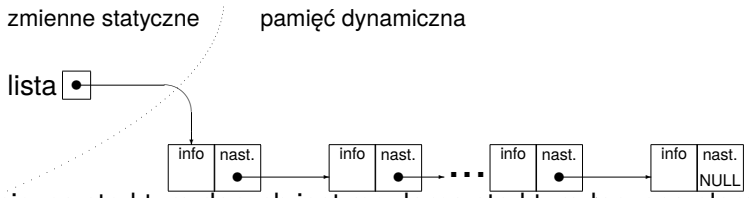
Listy i stosy

Robert Muszyński
ZPCiR IIAiR PWr

Zagadnienia: listy: tworzenie, wyszukiwanie, przeglądanie, usuwanie, problemy, listy z głową, z wartownikiem, dwustronnie połączone, kołowe, abstrakcyjne typy danych: stosy, kolejki chronologiczne, bufory cykliczne.

Lista – regularna struktura dynamiczna

```
typedef struct elem {
  t_info info;
  struct elem *nastepny;
} t_elem;              /* i gdzies dalej  t_elem *lista; */
```



- dynamiczna struktura danych jest regularną strukturą łączącą elementy treści z elementami konstrukcyjnymi (wskaźnikami)
- niezbędna jest co najmniej jedna zmienna statyczna zapewniająca dostęp do struktury dynamicznej
- specjalna wartość wskaźnikowa `NULL` jest przydatna do oznakowania pustych zakończeń struktury dynamicznej

Listy – tworzenie

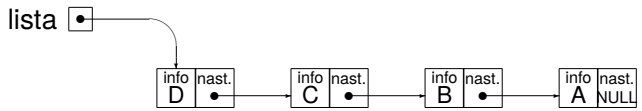
```
  t_elem *lista, *pom;

  lista = NULL;                        /* zainicjuj pusta liste */

  while (!koniec_danych) {                    /* dopoki sa dane */
    pom = (t_elem *)malloc(sizeof(t_elem));  /* utworz element */
    wczytaj_element(str_danych, pom); /* wczytaj do niego dane */
    pom->nastepny = NULL              /* to na wszelki wypadek */
    dodaj_element(lista, pom);       /* wlacz element do listy */
}
```

Trzy etapy w procesie konstrukcji listy:

- stworzenie nowej zmiennej dynamicznej
- wpisanie do niej właściwej treści
- utworzenie właściwych wskaźników konstrukcyjnych dla włączenia nowego elementu do reszty listy
  - należy pamiętać o wartości wskaźnikowej `NULL`

Listy – tworzenie cd.

```
void dodaj_element(t_elem **wsk_lista, t_elem *elem) {
                                          /* na poczatek listy */
  elem->nastepny = *wsk_lista;
  *wsk_lista = elem;
} /* dodaj_element */
  lista = NULL;                  /* i wtedy w jakiejs funkcji */
  pom = malloc(sizeof(t_elem)); /*pomijamy kwestie rzutowania*/
  pom->info = 'A'; dodaj_element(&lista,pom);
   pom = malloc(sizeof(t_elem));
   pom->info = 'B'; dodaj_element(&lista,pom);
   pom = malloc(sizeof(t_elem));
   pom->info = 'C'; dodaj_element(&lista,pom);
   pom = malloc(sizeof(t_elem));
   pom->info = 'D'; dodaj_element(&lista,pom);
```



Listy – tworzenie cd.

```
void dodaj_element(t_elem **wsk_lista, t_elem *elem) {
                          /* na koniec listy, rekurencyjnie */
  if (*wsk_lista == NULL)
    *wsk_lista = elem;
  else
    dodaj_element(&(wsk_lista->nastepny), elem);
  elem->nastepny = NULL;   /* moze niekonieczne, ale poprawne */
} /* dodaj_element */
```

- Zauważmy, że sprawdzenie czy `'*wsk_lista == NULL'` dotyczy nie tyle przypadku pustej listy (początkowo), ale ogólnie przypadku końca listy; a więc każdy nowy element będzie wpisywany w miejsce tej pustej wartości NULL,
- zauważmy również, że ta niewinnie wyglądająca funkcja wykonuje przeszukiwanie całej istniejącej listy, w dodatku szeregiem tylu wywołań rekurencyjnych, ile elementów ma lista,
- gdybyśmy tylko mieli wskaźnik ostatniego elementu :-(

Listy – wyszukiwanie elementu

```
char porownaj_element(t_elem e1, t_elem e2) {...}
                /*niech zwraca jeden ze znakow: '<', '=', '>' */
     /* wersja 1 - zla; moze nie zatrzymac sie na koncu */
  pom = lista;
  while (porownaj_element(*pom, elem_wzorcowy) != '=')
    pom = pom->nastepny;
     /* wersja 2 - dobra; ale dlaczego nie zawsze? */
  pom = lista;
  while (pom != NULL &&
     porownaj_element(*pom, elem_wzorcowy) != '=')
    pom = pom->nastepny;
     /* wersja 3 - tez dobra i to zawsze */
  pom = lista;  znalazl = 0;
  while (pom != NULL && !znalazl) {
    if (porownaj_element(*pom, elem_wzorcowy) == '=')
      znalazl = 1;
    else pom = pom->nastepny;
```

Listy – usuwanie elementu

- Musimy odróżnić czynność wyłączania elementu ze struktury danych od trwałego unicestwiania elementu i odzyskiwania zajmowanej pamięci (przy użyciu funkcji `free`).
- Najprostszy jest przypadek wyłączenia z listy pierwszego elementu, w pewnym sensie analogiczny do wstawiania elementu na początek listy:

```
void usun_element(t_elem **lista, t_elem *elem) {
  /* wylacza pierwszy element z listy, i zwraca go */
                                    /* w parametrze elem */
    element = *lista;
    *lista = (*lista)->nastepny;
} /* usun_element */
```

⇒ zauważmy, że procedura nie sprawdza, czy lista nie jest przypadkiem pusta (co spowodowałoby błąd wykonania).

- Wyłączanie z listy elementu, gdy mamy wskaźnik elementu poprzedniego

```
poprzedni->nastepny = poprzedni->nastepny->nastepny;
```


Listy – usuwanie elementu cd.

- Usunięcie z listy konkretnego elementu, do którego mamy wskaźnik, wymaga przeszukiwania.
- Nie zawsze mamy jawny wskaźnik elementu, który chcemy usunąć — często wiemy tylko **jaki** element chcemy usunąć; możemy wtedy porównywać kolejno wszystkie elementy z elementem „wzorcowym” — poszukać go.

⇒ Jeśli przy przeszukiwaniu zapamiętamy położenie elementu szukanego w celu jego usunięcia będziemy musieli szukać ponownie.

- Schemat ten możemy powtarzać, aby usunąć z listy kolejne (wszystkie) elementy pasujące do elementu wzorcowego.
- Ostateczne kasowanie elementu ze zwalnianiem pamięci dynamicznej:

```
free(elem);
elem = NULL;
```

⇒ Przypisanie wskaźnikowi wartości `NULL` ułatwia wykrywanie późniejszych błędnych odwołań do skasowanego elementu pamięci, ale im nie zapobiega!

Problemy ze wskaźnikami

- Typowym błędem zdarzającym się w programach ze wskaźnikami jest „wypadnięcie” przez koniec listy, co jest trochę podobne do przekroczenia zakresu indeksów tablicy.
- Wskaźniki umożliwiają jednak popełnianie znacznie więcej rodzajów błędów, np. używanie niezainicjowanych wskaźników, bądź wskaźników do skasowanych zmiennych dynamicznych.

  $\Rightarrow$ Takie błędy możemy łatwiej wykrywać systematycznie ustawiając nieużywane wskaźniki na `NULL`.
- Innym rodzajem błędów jest gubienie wskaźników do zmiennych dynamicznych (tworzenie nieużytków pamięci), bądź niezwalnianie niepotrzebnej pamięci dynamicznej, nawet jeśli nadal mamy do niej wskaźnik.

  $\Rightarrow$ Aby temu zaradzić dobrze jest systematycznie zwalniać każdy element pamięci dynamicznej w chwili, gdy stał się niepotrzebny.
- Jeszcze innym rodzajem błędów, jest przypadek wyczerpania pamięci dynamicznej.

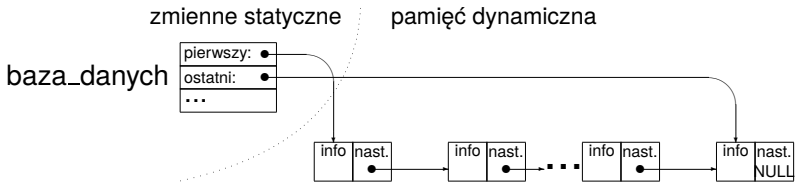
Listy – przeglądanie

```
void przegladaj_element(t_elem);
void przegladaj_liste(t_elem *lista) {   /* wersja iteracyjna */

  while (lista != NULL) {
    przegladaj_element(*lista);
    lista = lista->nastepny;
  }
} /* przegladaj_liste */
```

```
void przegladaj_liste(t_elem *lista) { /* wersja rekurencyjna */

  if (lista != NULL) {
    przegladaj_element(*lista);
    przegladaj_liste(lista->nastepny);
  }
} /* przegladaj_liste */
```

Listy z głową

```
typedef struct elem {
  t_info info;
  struct elem *nastepny;
} t_elem, *t_lista;
typedef struct glowa {
  t_lista pierwszy;
  /* inne informacje o liscie, np. ostatni, dlugosc */
} t_listaZG;
                        /* i wtedy gdzies w jakiejs funkcji */
  t_listaZG baza_danych;          /* powinna byc statyczna */

  baza_danych.pierwszy = NULL;          /* inicjacja listy */
  baza_danych.dlugosc = 0;
```



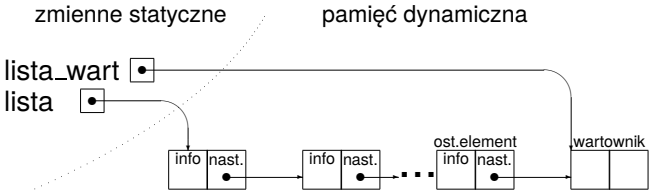
Listy z głową cd

- Zastosowania głowy:
  - ⋆ przechowywanie dwóch wskaźników listy, pierwszego i ostatniego elementu, w celu np.:
    - ∗ szybkiego dodawania węzłów na koniec listy,
    - ∗ szybkiego włączania całej listy w środek innej listy,
  - ⋆ pojemnik na informacje: liczba elementów, liczba referencji, itp.,
  - ⋆ umożliwienie usuwania pierwszego elementu listy, gdy wskaźnik listy jest skopiowany w kilku miejscach, np. w strukturach danych,
  - ⋆ znacznik na listach kołowych.
- Użycie głowy komplikuje procedury rekurencyjne.

Listy z wartownikiem

- Posługiwanie się listą ma pewne niespójności ponieważ operacje na wskaźniku listy czasem muszą być różne dla listy pustej i niepustej.
- Możemy to zmienić stosując zwykły schemat listy z dodatkiem **wartownika**, czyli wyróżnionego, dodatkowego elementu, którego treść jest nieistotna i nieużywana i który jest niewidoczny dla programisty wykorzystującego listę.

  $\Rightarrow$ np. lista z wartownikiem na początku lub na końcu jak poniżej



Listy dwustronnie połączone

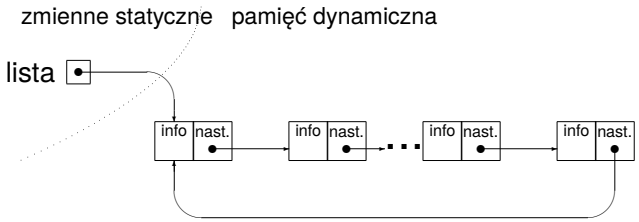
```
typedef struct elem {
  t_info info;
  struct elem *nastepny;
  struct elem *poprzedni;
} t_elem, *t_lista;

void dodaj_element(t_lista *lista, t_elem *elem) {
  elem->nastepny = *lista;
  elem->poprzedni = NULL;
  if ((*lista) != NULL) {
    elem->poprzedni = (*lista)->poprzedni;
    if ((*lista)->poprzedni != NULL)
      (*lista)->poprzedni->nastepny = elem;
    (*lista)->poprzedni = elem;
  } else
    *lista = elem;
} /* dodaj_element */
```

Listy dwustronnie połączone cd

- Listy dwustronnie połączone mają wiele cech odmiennych od „zwykłych” list, np. można się na nich „cofnąć”, łatwo włączać i usuwać elementy, dostęp do całej listy zapewnia wskaźnik dowolnego elementu.
- Jednak zasadnicze cechy list pozostają te same: sekwencyjny dostęp do elementów, brak istotnych korzyści z porządkowania elementów.
- Posługiwanie się listami dwustronnie połączonymi jest bardziej złożone niż zwykłymi listami, np. trzeba uważać na koniec listy po obu stronach.

```
void usun_element(t_lista *lista, t_elem *elem) {
                                  /*wylaczanie dowolnego elementu z listy*/
  if (elem->nastepny != NULL)                  /*w el.nastepnym, zaktualizuj*/
    elem->nastepny->poprzedni = elem->poprzedni;                 /*poprzedni*/
  else if (elem->poprzedni != NULL)                  /*jesli istn.poprzedni,*/
    elem->poprzedni->nastepny = NULL;                /*wpisz,ze nie ma nast.*/
  if (elem->poprzedni != NULL)                 /*w el.poprzednim,zaktualizuj*/
    elem->poprzedni->nastepny = elem->nastepny;                   /*nastepny*/
  else IF (elem->nastepny != NULL)                   /*jesli istn.nastepny, */
    elem->nastepny->poprzedni = NULL                 /*wpisz,ze nie ma poprz*/
  else *lista = NULL;                            /*w tym przyp.lista jest pusta*/
  if (*lista == elem)  /*...*/;                  /*ten przyp.tez wymaga korekty*/
} /*usun_element*/
```


Listy kołowe



- Listy kołowe nie mają fizycznego końca (ani początku), nie mają pustych wskaźników NULL i są całkowicie symetryczne.
- Nadają się np. do realizacji tzw. buforów kołowych (cyklicznych), służących np. do przechowywania pewnej ilości danych historycznych, automatycznie kasowanych w miarę zapisywania nowych danych czy też synchronizacji pracy funkcji programu (procesów).

Połączenia

Często przydatne jest łączenie różnych modyfikacji podstawowego schematu listy, np.:

- lista z wartownikiem i wskaźnikiem ostatniego elementu,
- lista kołowa dwustronnie połączona,
- nie ma sensu dodawanie wartownika ani wskaźnika ostatniego elementu do listy kołowej,
- głowa przydaje się i można ją dodać do każdego rodzaju listy.

Abstrakcyjne typy danych (wstęp)

- Przez ATD rozumiemy określenie typu danych poprzez zdefiniowanie zestawu operacji, jakie mają być dlań dostępne.
- ATD wnoszą wyższy poziom abstrakcji niż typy danych definiowane w programach i pozwalają rozdzielić proces tworzenia programu na implementację ATD, oraz pisanie właściwego programu przy ich użyciu.
- Przykłady ATD:
  - ⋆ stosy (*LIFO*)
    - ∗ dodaj na stos (*push*)
    - ∗ zdejmij ze stosu (*pop*)
    - ∗ sprawdź szczyt stosu (*top*)
    - ◇ sensowna realizacja przez zwykłe listy, a także tablice statyczne
  - ⋆ kolejki chronologiczne (*FIFO*)
    - ∗ dodaj na koniec
    - ∗ usuń z początku
    - ∗ podaj długość kolejki (?)
    - ◇ sensowna realizacja przez listy ze wskaźnikiem ostatniego elementu, a także tablice statyczne (z „zawijaniem”)
  - ⋆ bufory cykliczne

Podsumowanie

- Zagadnienia podstawowe
  1. Co oznacza pojęcie zgubienia wskaźnika do zmiennej dynamicznej?
  2. Czy można zdefiniować listę tylko i wyłącznie przy użyciu dynamicznych struktur danych?
  3. W jaki sposób dodaje się i usuwa elementy z listy?
  4. Jakie czynności należy wykonać by z listy usunąć element na który mamy wskaźnik?
  5. Czy można zbudować listę, która nie będzie miała końca? Jeśli tak, ile elementów musi zawierać najkrótsza taka lista?
  6. Jakie są różnice między listą a tablicą?
  7. Wskaż wady i zalety list dwustronnie połączonych w porównaniu ze standardowymi listami jednokierunkowymi.
  8. Czym różni się abstrakcyjna struktura danych od rzeczywistej struktury danych?
  9. W jaki sposób definiuje się stos? Jak może być on zaimplementowany? Który rodzaj listy najlepiej nadaje się do jego implementacji?
  10. Co to jest kolejka FIFO?
- Zagadnienia rozszerzające
  1. Znajdź informacje na temat zastosowań abstrakcyjnych struktur danych typu stos i kolejka chronologiczna. Jakie inne abstrakcyjne struktury danych można wyróżnić?

  2. Programy w trakcie swojego działania wykorzystują stos m.in. do przechowywania danych związanych z uruchamianymi funkcjami. Używając debugera `gdb` do uruchomienia dowolnego programu rekurencyjnego przeanalizuj zmiany zawartości stosu podczas działania programu.
  3. Czym jest bufor pośredniczący przy wymianie danych pomiędzy procesami w systemie? Znajdź przykładowe implementacje.
- Zadania
  1. Napisz dwa moduły: pierwszy implementujący listę jednokierunkową za pomocą zmiennych dynamicznych i drugi realizujący tę samą funkcję bazując na tablicy statycznej. Następnie przygotuj program pozwalający na przetestowanie poprawności napisanych modułów (driver). Program testujący i struktura całości powinny być na tyle uniwersalne, by zmiana modułu poddawanego testom nie wymagała zmian w kodzie całości, a jedynie zamianę dolinkowywanego modułu.
  2. Napisz funkcję umożliwiającą usunięcie z listy wszystkich elementów zgodnych z podanym wzorcem.
  3. Zapisz strukturę danych do przechowywania elementu listy dwukierunkowej zawierającej liczby zespolone.
  4. Zaproponuj strukturę kolejki priorytetowej oraz sposób jej implementacji za pomocą struktur dynamicznych.
  5. Zaproponuj implementację bufora cyklicznego. W jaki sposób definiuje się funkcje dodawania, usuwania, wyszukiwania elementów?