Wrocław
University
of Science
and Technology

Struktury danych

Wykład 3

Listy

dr inż. Jarosław Rudy

HR EXCELLENCE IN RESEARCH

Lista (ADT)

- Lista jest kontenerem przechowującym elementy.
- Elementy mogą być identycznego lub różnego typu.
- Lista zachowuje (określa) kolejność elementów.
- Elementy mogą się powtarzać.
- Jeden element ma jedną wartość (ale tą wartością może być kolejna kolekcja).
- Lista ma zmienną długość i zawartość (elementy można dodawać, usuwać i modyfikować).

Lista – operacje (1)

Na liście typowo rozważa się następujące operacje:

- Stworzenie pustej listy.
- Dostęp (zwrócenie) elementu na pozycji i.
- Dodanie elementu e na pozycji i (tj. przed elementem i-tym). Specjalne przypadki:
  - Dodanie elementu e na początek listy.
  - Dodanie elementu e na koniec listy.

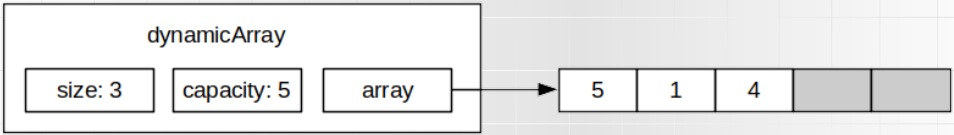
Lista – operacje (2)

- Usunięcie elementu e na pozycji i. Specjalne przypadki:
  - Dodanie elementu e na początku listy.
  - Dodanie elementu e na końcu listy.
- Zwrócenie rozmiaru (liczby elementów) listy.
- Sprawdzenie czy lista jest pusta.
- Wyszukanie elementu e.

Tablica dynamiczna (1)

- Struktura danych będąca implementacja listy (ADT).
- Najczęściej przechowuje:
  - `array` – wskaźnik na tablicę utworzoną dynamicznie (`new`, `malloc()` itp.).
  - `capacity` – rozmiar tablicy `array` (w liczbie elementów).
  - `size` – liczba przechowywanych elementów.
- Zajmowana pamięć: $capacity + 3$, czyli $O(capacity)$.
  - W praktyce $capacity < 2n$, więc zajmowany rozmiar to pomiędzy $n + 3$ a $2n + 3$, czyli $O(n)$.

Tablica dynamiczna (2)



„Pierwsza" implementacja operacji:

- Stworzenie pustej listy:
  - $array \leftarrow null$
  - $size \leftarrow 0$
  - $capacity \leftarrow 0$
  - czas: $O(1)$.

Tablica dynamiczna (3)

- Zwrócenie rozmiaru listy:
  - Zwrócenie *size*.
  - czas: $O(1)$.
- Sprawdzenie czy lista jest pusta:
  - Zwrócenie wartości wyrażenia $size == 0$.
  - czas: $O(1)$.
- Zwrócenie elementu na pozycji i – w czasie $O(1)$ dzięki arytmetyce wskaźników (tablica *array* ma dostęp swobodny przez indeks i).
  - Błąd jeśli $i < 0$ lub $i \geqslant size$.

Tablica dynamiczna (4)

- Wyszukanie elementu e.
  - Sprawdzamy kolejne elementy, porównując je z e do czasu znalezienia pasującego elementu (zwrócenie e, prawdy itp.) lub do wyczerpania elementów (zwrócenie nulla, fałszu itp.).
  - Optymistycznie: sprawdzany jeden element – czas $O(1)$.
  - Średnio: sprawdzamy połowę elementów tj. $\lceil \frac{n}{2} \rceil$ – czas $O(n)$.
  - Pesymistycznie: sprawdzamy wszystkie n elementów – czas $O(n)$.
- Przejrzenie (np. wypisanie wszystkich elementów) również zajmuje czas $O(n)$.

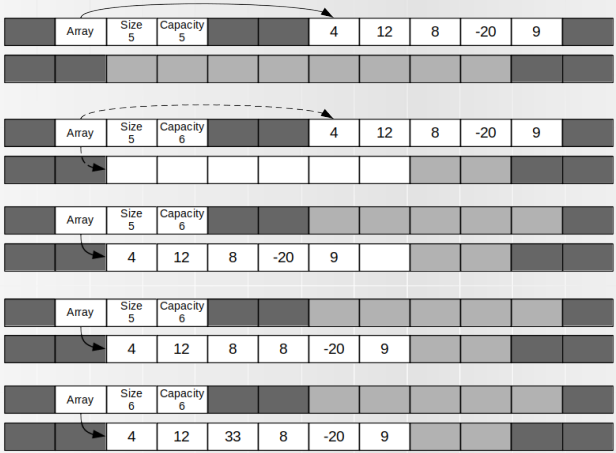
Tablica dynamiczna (5)

W najprostszej implementacji dodanie elementu e na pozycji i składa się z kilku etapów:

- Zwiększenie rozmiaru tablicy *array* o 1.
  - Funkcja `realloc()` zmienia rozmiar tablicy i może wymagać przeniesienia tablicy do nowej lokalizacji (`memcpy()`).
  - $capacity \leftarrow capacity + 1$
  - Czas $O(1)$ lub $O(n)$, zależnie czy kopiowano tablicę.
- Przeniesienie (np. w pętli) elementów od pozycji $n - 1$ do i o jeden w prawo.
  - Czas wynosi $O(n - i)$, pesymistycznie $O(n)$.
- Wstawienie e na pozycję i w czasie $O(1)$:
  - $array[i] \leftarrow e$
  - $size \leftarrow size + 1$
- Czas całej operacji pesymistycznie i średnio $O(n)$, optymistycznie $O(1)$.

Tablica dynamiczna (6)

Przykładowe wstawienie ($e = 33$, $i = 2$).



Tablica dynamiczna (7)

- Dodanie elementu e na początku ($i = 0$) jest takie samo, ale czas jest zawsze $O(n)$, nawet jeśli nie trzeba przenosić tablicy podczas `realloc()`.
- Dodanie elementu e na końcu ($i = n - 1$) jest takie samo, ale czas wynosi $O(1)$ z wyjątkiem sytuacji, gdy `realloc()` przeniesię tablicę.
- Powyższa podstawowa implementacja operacji dodawania zakłada, że zaczynamy od pustej tablicy ($capacity = 0$) i zwiększamy rozmiar zawsze o 1 (tzn. zawsze $size = capacity$).

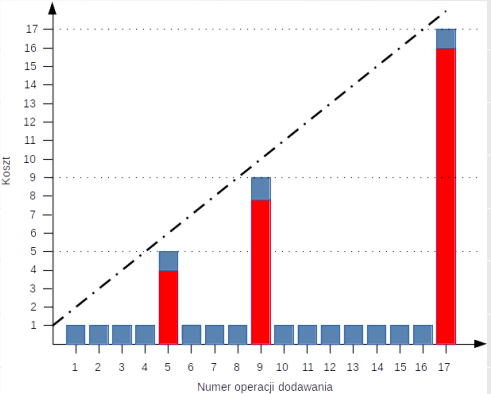
Tablica dynamiczna (8)

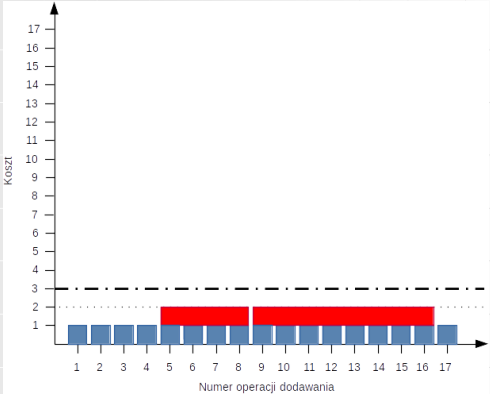
Implementacja ulepszona.

- Zaczynamy od zaalokowanej, ale pustej tablicy (np. $size = 0$, $capacity = 4$).
- Przez pierwsze 4 operacje dodawania, nie trzeba będzie kopiować pamięci!
- Za każdym razem kiedy brakuje miejsca, zwiększamy rozmiar tablicy dwukrotnie ($capacity = 2size$).
- Podobnie jak poprzednio, przy każdym braku miejsca w tablicy może dojść do konieczności kopiowania pamięci... ale brak miejsca zdarza się rzadziej.
- Ściślej, jeśli odbywa się kopiowanie zajmujące $O(n)$, to wiemy, że n poprzednich operacji dodawania nie wymagało kopiowania!
  - Pesymistycznie jest dalej $O(n)$, ale w koszcie zamortyzowanym jest $O(1)$!

Tablica dynamiczna (9)

Klasyczny analiza przypadku pesymistycznego vs koszt zamortyzowany





Tablica dynamiczna (10)

Klasyczna analiza:

Operacja	Optymistycznie	Średnio	Pesymisycznie
Dodanie na dowolnej pozycji	$O(1)$	$O(n)$	$O(n)$
Dodanie na początku	$O(n)$	$O(n)$	$O(n)$
Dodanie na końcu	$O(1)$	$O(n)$	$O(n)$

Koszt zamortyzowany:

Operacja	Optymistycznie	Średnio	Pesymisycznie
Dodanie na dowolnej pozycji	$O(1)$	$O(n)$	$O(n)$
Dodanie na początku	$O(n)$	$O(n)$	$O(n)$
Dodanie na końcu	$O(1)$	$O(1)$	$O(1)$

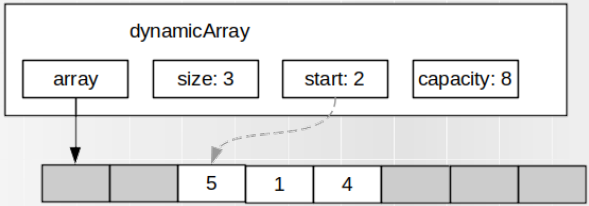
Tablica dynamiczna (11)

Usuwanie elementu z pozycji i zasadniczo działa podobnie do dodawania, lecz kolejność jest inna:

- Skopiowanie elementów od $i+1$ do $n-1$ o jedną pozycję w lewo oraz zmniejszenie *size* o 1.
- Zmniejszenie rozmiaru tablicy (oraz *capacity*). Może następować o 1 lub co jakiś czas (analogicznie do dodawania).
- Jeśli usuwany element ma zostać zwrócony, należy go zapamiętać przed pierwszym krokiem!
- Zmniejszanie tablicy dalej może wymagać kopiowania pamięci (zależnie od implementacji `realloc()`).
- Można pominąć zmniejszanie rozmiaru tablicy, ale rozmiar struktury nie będzie wtedy $O(n)$.

Tablica dynamiczna (12)

- Czy da się zredukować złożoność $O(n)$ dla dodawania i usuwania na początku?
- Zaalokowanie więcej miejsca i przesunięcie indeksu fizycznego względem logicznego – zostaje z przodu miejsce na indeksy „ujemne".
- Konieczne dodanie i aktualizacja pozycji początkowej.
- Alokacja i dealokacja raz na jakiś czas analogicznie jak poprzednio.



Tablica dynamiczna (13)

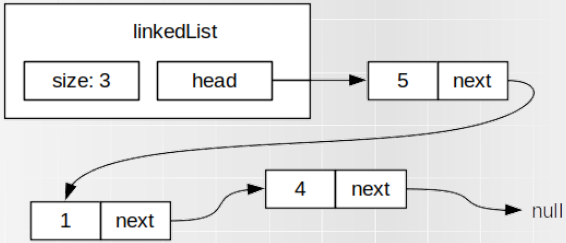
Po dodaniu wspomnianego usprawienia:

Operacja	Optymistycznie	Średnio	Pesymisycznie
addFront(e)	$O(1)$	$O(1)$ (amort.)	$O(1)$ (amort.)
removeFront()	$O(1)$	$O(1)$ (amort.)	$O(1)$ (amort.)
addBack(e)	$O(1)$	$O(1)$ (amort.)	$O(1)$ (amort.)
removeBack()	$O(1)$	$O(1)$ (amort.)	$O(1)$ (amort.)
add(e,i)	$O(1)$	$O(n)$	$O(n)$
remove(i)	$O(1)$	$O(n)$	$O(n)$

Lista wiązana (jednokierunkowa)

- Struktura danych będąca implementacja listy (ADT).
- Każdy element (węzeł) zawiera:
  - Właściwą wartość (*value*).
  - Wskaźnik na kolejny element listy (*next*).
- Elementy mogą mieć różną lokalizację w pamięci!
- Lista kończy się, gdy następny wskaźnik ma wartość null.
- Struktura przechowuje wskaźnik *head* na pierwszy element. Oprócz tego przechowywany jest *size*.
- Zajmowana pamięć: $2n + 2$, czyli $O(n)$.

Lista wiązana (2)



Tworzenie pustej listy (czas $O(1)$):

- $size \leftarrow 0$
- $head \leftarrow null$

Lista wiązana (3)

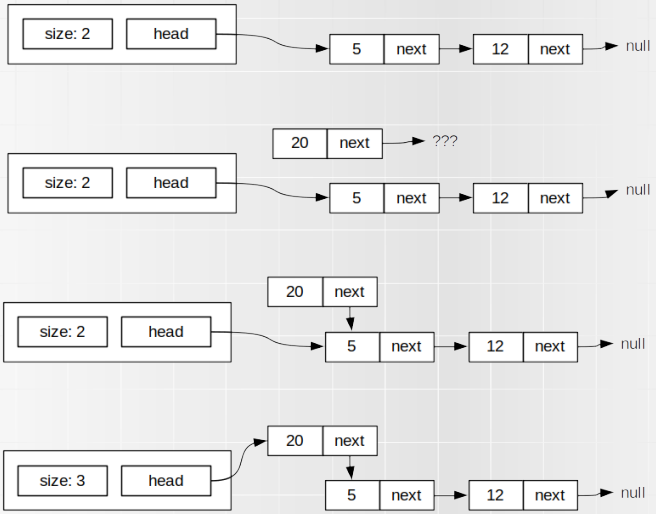
- Zwrócenie rozmiaru i sprawdzenie czy lista jest pusta działa identycznie jak dla tablicy dynamicznej.
- Wyszukanie elementu działa podobnie, lecz by przejść do następnego elementu, należy skorzystać ze wskaźnika *next* poprzedniego (i sprawdzić czy nie jest on null).
- Zwrócenie elementu na pozycji i wymaga przejścia przez elementy od 0 do i. Czas operacji jest więc $O(i)$:
  - Optymistycznie ($i = 0$) mamy $O(1)$.
  - Średnio ($i = \lceil \frac{n}{n} \rceil$) mamy $O(n)$.
  - Pesymistycznie ($i = n - 1$) mamy $O(n)$.

Lista wiązana (4)

Dodanie elementu e na początek ($O(1)$)

- Tworzymy (dynamicznie, `new`, `malloc()` itp.) nowy węzeł i zapamiętujemy jego wskaźnik (*node*).
- $node.value \leftarrow e$
- $node.next \leftarrow head$
- $head \leftarrow node$
- $size \leftarrow size + 1$

Lista wiązana (5)



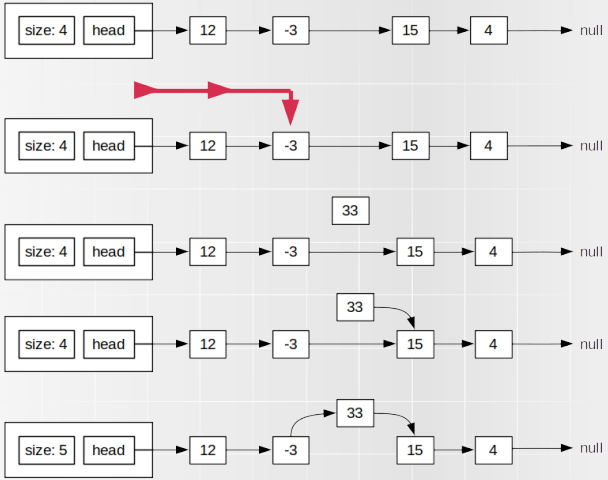
Lista wiązana (6)

Dodanie elementu e na pozycji i (innej niż $i = 0$ oraz $i = n - 1$).

- Najpierw musimy dotrzeć do węzła $i - 1$ (czas $O(i)$), nazwijmy go *old*.
- Tworzymy nowy węzeł *node* i przypisujemy mu wartość e.
- $node.next \leftarrow old.next$
- $old.next \leftarrow node$
- $size \leftarrow size + 1$
- Optymistycznie $O(1)$, średnio i pesymistycznie $O(n)$.

Lista wiązana (7)

Dodanie $e = 33$ na pozycję 2:

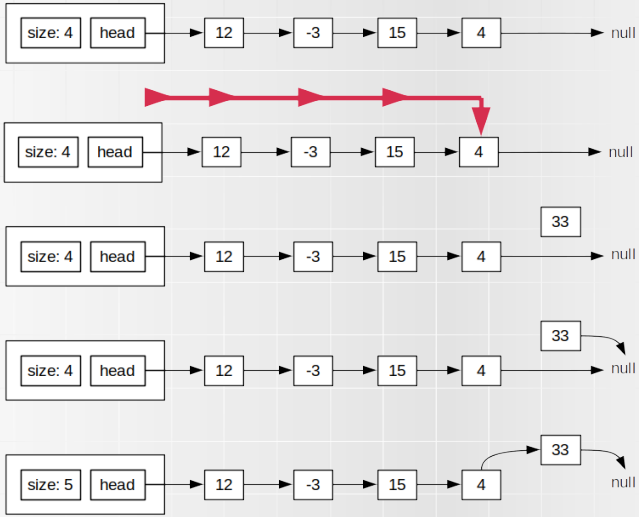


Lista wiązana (6)

Dodanie elementu e na ostatniej pozycji ($i = n - 1$).

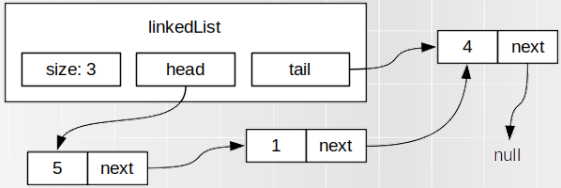
- Identycznie jak dodawanie na dowolną pozycję, z tym że *node.next* należy ustawić na null.
- Trzeba dotrzeć do końca listy, więc czas wynosi $O(n)$.

Lista wiązana (7)

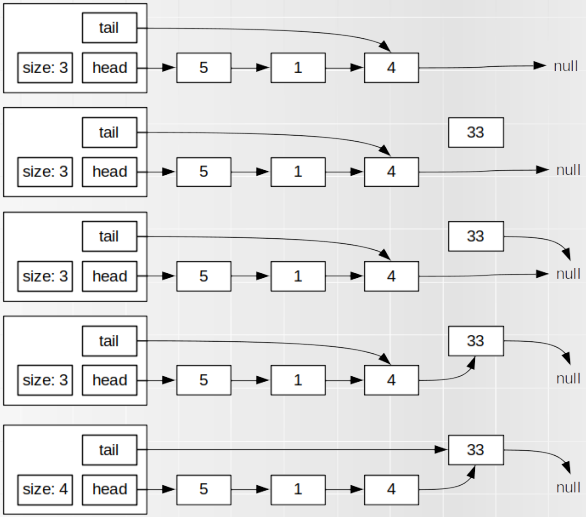


Lista wiązana (8)

- Prostym ulepszeniem jest dodanie wskaźnika *tail* wskazującego na koniec listy (ostatni element lub null dla listy pustej).
- Należy go ustawić na null na początku i pamiętać o ustawieniu gdy zmienia się ostatni element.
- Umożliwia dodawanie na koniec listy w czasie $O(1)$.



Lista wiązana (9)

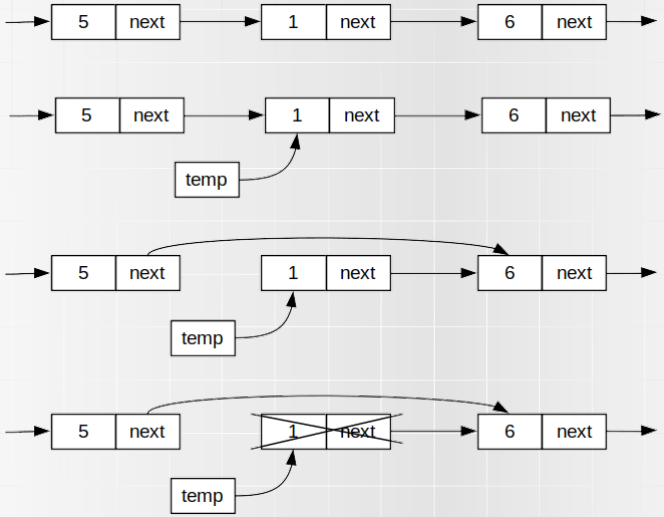


Lista wiązana (10)

Usuwanie elementu na pozycji i

- Dotarcie do węzła $i - 1$ (nazwijmy go *old*).
- $temp \leftarrow old.next$
- $old.next \leftarrow old.next.next$
- Usunięcie węzła *temp* (`delete`, `free()`).
- Specjalne przypadki:
  - Usunięcie pierwszego węzła (konieczność modyfikacji *head*).
  - Usunięcie ostatniego węzła (konieczność modyfikacji *tail*, jeśli jest).

Lista wiązana (11)



Lista wiązana (12)

Lista wiązana jednokierunkowa (tylko *head*)

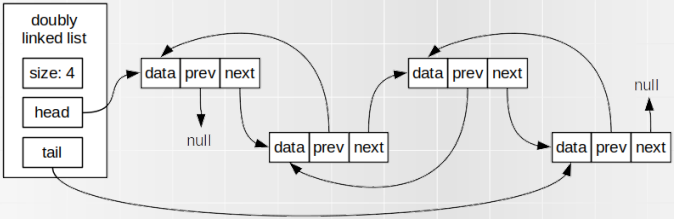
Operacja	Optymistycznie	Średnio	Pesymisycznie
Dodanie na dowolnej pozycji	$O(1)$	$O(n)$	$O(n)$
Dodanie na początku	$O(1)$	$O(1)$	$O(1)$
Dodanie na końcu	$O(n)$	$O(n)$	$O(n)$

Lista wiązana jednokierunkowa (*head* i *tail*)

Operacja	Optymistycznie	Średnio	Pesymisycznie
Dodanie na dowolnej pozycji	$O(1)$	$O(n)$	$O(n)$
Dodanie na początku	$O(1)$	$O(1)$	$O(1)$
Dodanie na końcu	$O(1)$	$O(1)$	$O(1)$

Lista dwukierunkowa (1)

- Lista wiązana, gdzie każdy element, ma zarówno wskaźnik na następny element (*next*), jak i na poprzedni (*prev*).
- Dla ostatniego elementu $next = null$.
- Dla pierwszego elementu $prev = null$.
- Struktura przechowuje *head* i *tail*.

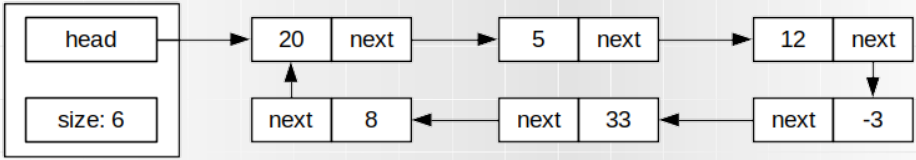


Lista dwukierunkowa (2)

- Zajętość pamięci: $3n + 3$ (wciąż $O(n)$, ale narzut jest znaczny).
- Łatwiejsze przemieszczanie się po liście.
- Czas dotarcia do węzła i dwukrotnie mniejszy (choć wciąż $O(i)$).
  - Prawie dwukrotny mniejszy czas dodawania/usuwania elementów na dowolnej pozycji, szukania oraz przeglądania.
- Więcej wskaźników do ustawienia podczas operacji.
  - Nieco wolniejsze dodawanie/usuwanie na końcach.

Lista cykliczna (1)

- Ostatni element wskazuje na pierwszy, zamiast na null.
  - Dla listy dwukierunkowej, pierwszy element wskazuje też na ostatni.
- Wciąż istnieje *head* (inaczej nie można dostać się do listy), ale dowolny element jest początkiem/końcem listy.
- Koniec listy rozpoznajemy po dotarciu drugi raz do elementu początkowego



Lista cykliczna (2)

- Poszukiwanie można zacząć od dowolnego elementu.
- Przydatna w implementacji niektórych kolejek, buforów cyklicznych czy kopca Fibonacciego
- Przydatna, gdy przechodzimy po liście wielokrotnie.
- Bardziej złożona, trudniejsza w kontroli (znalezienie końca, możliwość nieskończonych pętli itp.).

Rozszerzenia listy wiązanej

Lista z wartownikiem (sentinel):

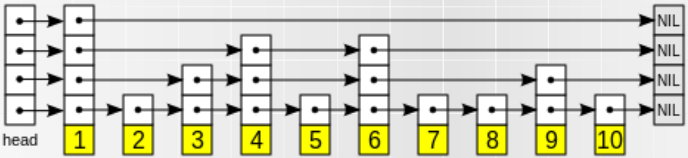
- Dodatkowy węzeł (wartownik) przed początkiem/po końcu.
- Ułatwia obsługę listy.
  - Każdy wskaźnik można wyłuskać (brak nulla).
  - Zawsze istnieje jakiś element, nawet jak lista jest pusta.

Lista wielokrotnie wiązana:

- Węzeł posiada więcej wskaźników.
- Możliwość posiadania różnych kolejności na tych samych danych.

Lista z przeskokiem

- Probabilistyczna struktura danych.
- Lista wielokrotnie wiązana, wiązania mogą pomijać elementy (wg prawdopodobieństwa).
- Średni czas operacji dodawania/usuwania/wyszukiwania $O(\log n)$.
- Pesymistyczna zajętość pamięci $O(n \log n)$.

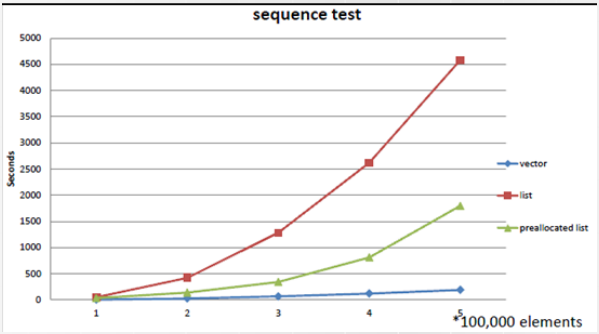


Tablica dynamiczna vs lista wiązana (1)

- Tablica dynamiczna ma szybszy czas dostępu do węzła oraz dodatkową pamięć niż lista wiązana – $O(1)$ vs $O(n)$.
- Lista wiązana ma szybsze dodawanie na początku ($O(1)$ vs $O(n)$) chyba, że zastosujemy usprawnienie z „ujemnymi" indeksami.
- Dodawanie na końcu jest w obu przypadkach $O(n)$, ale można je ulepszyć.
  - Tablica dynamiczna – koszt zamortyzowany plus odpowiednio rzadka alokacja pamięci.
  - Lista wiązana – dodanie wskaźnika na ogon.

Tablica dynamiczna vs lista wiązana (2)

- Czy na pewno? Bjarne Stroustrup (twórca C++) przedstawił eksperyment, w którym lista okazała się wolniejsza.
- Winowajcą jest „niespójna" reprezentacja listy wiązanej w pamięci w połączeniu ze sposobem działania współczesnych pamięci podręcznych (cache).



[1]

Samoorganizujące się listy (1)

- Średni czas wyszukiwania na liście (ADT) wynosi $\lceil \frac{n}{2} \rceil = O(n)$.
- Jest to znacznie gorzej niż czas wyszukiwania dla drzew poszukiwań binarnych czy posortowanych list/tablic.
- Możliwym rozwiązaniem są listy samoorganizujące, które zmieniają kolejność w wyniku kolejnych operacji dostępu/wyszukiwania.
- Najlepiej sprawdzają się dla sytuacji, gdzie żądania dostępu (lub ich rozkład) znane są z góry.
  - Zasada 80-20 (20% elementów jest celem 80% wyszukiwań).

Samoorganizujące się listy (2)

Metoda move-to-front:

- Element, do którego był dostęp przesuwany jest na początek listy.
  - Proste dla listy wiązanej (dodatkowy czas $O(1)$).
  - Trudniejsze dla tablicy dynamicznej (dodatkowy czas $O(n)$).
- Względnie prosta implementacja.
- Podatny na przeszacowanie (przenoszenie na sam już po pierwszym dostępie).

Samoorganizujące się listy (3)

Metoda transpose (swap):

- Element, do którego był dostęp przesuwany jest na pozycję o 1 wcześniej.
- Proste zarówno dla tablicy dynamicznej i listy wiązanej.
  - Jeśli lista nie jest dwukierunkowa, to należy pamiętać element poprzedni.
- Przenoszenie elementów do przodu jest stopniowe.
- Dobrze dostosowuje się do sytuacji, gdzie wzorzec (rozkład prawdopodobieństwa) żądanych elementów zmienia się w czasie.

Samoorganizujące się listy (4)

Metoda count:

- Każdy węzeł ma licznik odwołań (ile razy był dostęp).
  - Wymaga dodatkowo $O(n)$ pamięci.
- Węzły układane są w kolejności malejącego licznika.
  - Po odwołaniu wykonuje się tyle swapów ile potrzeba, by uzyskać poprawne sortowanie.
  - Wyszukiwanie może wydłużyć się o dodatkowe $O(n)$, ale średnio będzie szybsze.
  - Podobnie zwykły dostęp dla listy wiązanej. Dla tablicy dynamicznej dostęp się pogorszy!

Bibliografia

 https://bulldozer00.blog/2012/02/09/vectors-and-lists/