Algorytmy i Struktury Danych

Abstrakcyjne Struktury Danych

(c) Marcin Sydow

Zawartość wykładu:

- Typy operacji na ciągach
- Listy dowiązaniowe (ang. linked lists):
  - lista jednokierunkowa
  - lista dwukierunkowa
- **Abstrakcyjne Struktury Danych**
  - Stos
  - Kolejka
  - przykład rozszerzenia: kolejka dwustronna

Ciągi

Ciągi elementów są najczęściej występującą w algorytmach strukturą danych.

Wśród operacji wykonywanych na ciągach można wyodrębnić m.in. dwa typy:

- dostęp absolutny
  (identyfikowany przez bezwzględny adres (indeks) w ciągu)
- modyfikacja
  (w miejscu identyfikowanym przez konkretny element ciągu)

Tablice jako reprezentacja ciągów

- Najprostszą konkretną strukturą danych do reprezentowania ciągów są **tablice**.
- Zapewniają one bardzo szybki dostęp absolutny (po indeksie) - w czasie stałym.
- Jednak operacje modyfikacji (np. wstawianie elementów w środku tablicy) wymagają liniowej złożoności czasowej (gdyż trzeba najpierw "zrobić miejsce", przesuwając inne elementy)
- Wtedy przydają się inne struktury danych np.: **listy dowiązaniowe**.

Motywacja dla użycia list dowiązaniowych

Alternatywną konkretną strukturą danych do reprezentowania ciągów są listy dowiązaniowe.

Potrafią one zapewnić szybkie operacje typu modyfikującego, w tym wewnątrz ciągu, np:

- dodaj/usuń pierwszy lub ostatni element
- dodaj/usuń element za/przed wskazanym (np. przez wskaźnik) konkretnym elementem
- wstaw/usuń cały podciąg elementów pomiędzy wskazanymi elementami
- połącz kilka ciągów w jeden
- odwróć ciąg

Listy dowiązaniowe

Lista dowiązaniowa jest konkretną strukturą danych typu złożonego, zrealizowaną za pomocą:

- tzw. **węzłów** (ang. nodes), przechowujących elementy ciągu
- połączonych za pomocą tzw. **dowiązań** (ang. links), zrealizowanych jako wskaźniki

Jest kilka wariantów, np:

- listy jednokierunkowe (ang. singly linked lists)
- listy dwukierunkowe (ang. doubly linked lists)
- listy cykliczne (jedno i dwukierunkowe)

Implementacja listy jednokierunkowej

Najprostsza implementacja klasy węzła listy jednokierunkowej (gdzie Type jest dowolnym typem przechowywanych elementów ciągu) i klasy listy jednokierunkowej:

```
Class SLNode<Type>{
 Type element
 SLNode<Type> next //wskazuje kolejny węzeł w liście
}

Class SList<Type>{
 SLNode<Type> head //wskazuje początek listy (jedyny dostęp do listy)
}
```

przykład (4 węzły):
`head-> (2)-> (3)-> (5)-> (8)-> null`
Ostatni wskaźnik zawsze pokazuje na NULL. W pustej liście wskaźnik `head` (czoło) wskazuje na NULL

Przykład prostego algorytmu na liście jednokierunkowej

Przykład: wydrukowanie ciągu elementów przechowywanych w liście będącej argumentem:

```
print(SList l){
 node = l.head
 while(node not null)
  print node.element
  node = node.next
}
```

Bardzo charakterystyczna dla algorytmów sekwencyjnie pracujących na listach dowiązaniowych jest instrukcja `while(node not null)` (przeglądanie listy do końca - zauważmy, że dzięki takiemu warunkowi algorytm działa również na pustej liście) oraz ostatnia linia `node=node.next`, oznaczająca przejście do kolejnego węzła.

Listy jednokierunkowe wystarczają dokładnie w tych zadaniach, gdzie wystarcza jednokierunkowa nawigacja (np. wyszukiwanie w ciągu nieuporządkowanym).

Czasami jednak niezbędne są np. listy dwukierunkowe.

Lista dwukierunkowa

Przykład najprostszej implementacji listy dwukierunkowej (klasy: węzeł i lista):

```
Class DLNode<Type>{
 Type element
 DLNode<Type> next // wskazuje na następny węzeł
 DLNode<Type> prev // wskazuje na poprzedni węzeł
}

Class DLList<Type>{
 DLNode<Type> head // wskazuje na pierwszy element listy
}
```

(czasami użyteczny jest też dodatkowy wskaźnik do ostatniego elementu takiej listy)

Lista dwukierunkowa zużywa na dowiązania 2 razy więcej pamięci niż jednokierunkowa, ale zapewnia szybką dwustronną nawigację, co jest niezbędne w niektórych algorytmach.

Pytanie kontrolne: jaki typ listy powinien być użyty do dowiązaniowej wersji algorytmu InsertionSort?

Listy Cykliczne

W niektórych zastosowaniach najwygodniejszą konkretną strukturą danych jest *lista cykliczna*.
W liście takiej nie ma ostatniego lub pierwszego węzła (równoważnie: ostatni jest dowiązany do pierwszego, tworząc zamknięty cykl) Może wystąpić zarówno wariant jednokierunkowy jak i dwukierunkowy (w zależności od potrzeb)

W dwukierunkowej liście cyklicznej, dla każdego węzła zachodzi następujący "niezminnik" (w innym sensie niż dla pętli):
(next.prev) == (prev.next) == this

W niektórych przypadkach stosuje się podobnie tzw. "cykliczne tablice", gdzie arytmetyka indeksów jest "modulo n", gdzie n to długość tablicy. Stosuje się je np. w bardziej wyrafinowanych szybkich kolejkach priorytetowych lub w implementacji kolejki na tablicy.

Operacje na ciągach

Przykłady operacji na ciągach:

- isEmpty (daje odpowiedź logiczną "prawda" tylko dla pustego ciągu)
- first (zwraca pierwszy element ciągu)
- last (zwraca ostatni element ciągu)
- insertAfter/insertBefore (wstaw element za/przed inny element)
- moveAfter/moveBefore (przenieś element za/przed inny element)
- removeAfter/removeBefore
- pushBack/pushFront (dołącz element na końcu/początku ciągu)
- popBack/popFront (zdejmij element z końca/początku ciągu)
- concat (połącz dwa ciągi w jeden)
- splice (techniczna operacja pomocnicza, opisana później)
- size (zwróć liczbę elementów w ciągu)
- findNext (zwróć następny element do podanego), etc.

Powyższe operacje na ciągach mogą być różnie zaimplementowanie w zależności np. od tego przez jaką konkretną strukturę danych reprezentowany jest ciąg (np. tablicę, listę jednokierunkową, dwukierunkową, etc.).

Operacja pomocnicza Splice

Większość modyfikujących operacji na listach jest specjalnym przypadkiem ogólnej technicznej operacji `splice`:

INPUT: a,b,t - a,b: wskaźniki do węzłów w jednej liście, t: wskaźnik do węzła w innej liście (obie listy mogą być tożsame, ale wtedy t nie może być pomiędzy a i b)

OUTPUT: wycina podlistę (a,...,b) i wstawia tuż za t

Przykład implementacji operacji splice na liście dwukierunkowej:
(zauważmy, że operacja ma **stałą złożoność czasową**, niezależnie od długości list!)

```
splice(a,b,t){
 // cut out (a,...,b):
 a' = a.prev; b' = b.next; a'.next = b'; b'.prev = a'
 // insert (a,...,b) after t:
 t'= t.next;  b.next = t'; a.prev = t; t.next = a; t'.prev = b
}
```

Proszę sprawdzić, że większość operacji modyfikujących listy jest specjalnym przypadkiem splice, np:

`moveAfter(a,b){ splice(a,a,b)}`

Podsumowanie: listy dowiązaniowe a tablice

Listy dowiązaniowe, jako reprezentacja ciągów, mają zarówno zalety jak i wady w porównaniu z tablicami: Linked Lists (compared with arrays):

- pozytywne: szybkie (czas stały) w porównaniu z tablicami (czas liniowy) operacje modyfikujące ciąg
- pozytywne: dynamiczny rozmiar listy w porównaniu z tablicami, który mają sztywno ustalony rozmiar
- negatywne: dodatkowy narzut pamięci na dowiązania (zwykle tyle co na wskaźnik, czyli max. 8 bajtów na każde dowiązanie dla 64-bitowej architektury)
- negatywne: wolny (liniowy czas) dostęp absolutny do elementów (w porównaniu ze stałym dla tablic)

Abstrakcyjne Struktury Danych

Abstrakcyjne Struktury Danych

Bardzo ważne pojęcie w projektowaniu algorytmów i programowaniu.

Abstrakcyjna Struktura Danych zdefiniowana jest przez **zestaw operacji**, które można wykonać na danych, nie wnikając w ich sposób implementacji.

Opakowuje więc ona niejako implementację zrealizowaną m.in. za pomocą konkretnych struktur danych (np. list dowiązaniowych, tablic, drzew, etc.) ukrywając tę implementację a eksponując jedynie **interfejs** do danych.

Przykłady abstrakcyjnych struktur danych:
stos, kolejka

Stos

Jeden z najbardziej podstawowych przykładów abstrakcyjnej struktury danych.

Stos zdefiniowany jest poprzez następujący zestaw operacji (interfejs):

Interfejs stosu (elementów typu Type):

- push(Type e) (dodaj do stosu element e)
- Type pop() (zdejmij i zwróć ze stosu element ostatnio dodany - operacja modyfikująca)
- Type top() (zwróć bez zdejmowania element ostatnio dodany - operacja niemodyfikująca)

Ze względu na powyższe zdefiniowanie wzajemnych relacji pomiędzy push a pop i top stos nazywany jest też strukturą "LIFO" (od ang. last in - first out), gdyż efekt stosu jest taki, że elementy wyjmowane są ze stosu w odwrotnej kolejności do tej w jakiej były do niego dodawane.

Kolejka

Zdefiniowana za pomocą następującego zestawu operacji (interfejsu): Kolejka (elementów typu Type):

- inject(Type e) (wstaw nowy element do kolejki)
- Type out() (wyjmij i zwróć element najdawniej dodany do kolejki - operacja modyfikująca)
- Type front() (zwróć bez wyjmowania element najdawniej dodany do kolejki - operacja niemodyfikująca)

uwaga: nazwy operacji kolejki nie są tak tradycyjnie ustalone jak w przypadku stosu i mogą być różne (np. in zamiast inject, etc.), ale zawsze mają powyższe znaczenie.

Kolejka ma więc tę własność, że elementy wyjmowane są z niej w tej samej kolejności w jakiej były do niej dodane. Stąd czasami używana nazwa "FIFO" (ang. first in - first out).

Kolejki mają rozliczne zastosowania jako wszelkiego rodzaju bufory operacji wejścia/wyjścia (sieciowe, strumieniowe, plikowe, etc.), oraz np. do przechowywania kolejności zadań do wykonywania.

Kolejka dwustronna

Kolejka dwustronna (elementów typu Type) jest abstrakcyjną strukturą danych, która odwołuje się do pojęcia "dwóch końców" ciągu (albo "początku" i "końca")

(ang. Dequeue od **D**ouble **E**nded **Queue** (wymawiane tak jak "deck"))

Jej interfejs jest następujący:

- Type first() (pokaż element z "pierwszego końca")
- Type last() (pokaż element z "drugiego końca")
- pushFront(Type) (dodaj element do pierwszego końca)
- pushBack(Type) (dodaj element do drugiego końca)
- Type popFront() (zwróć i zdejmij element z pierwszego końca)
- Type popBack() (zwróć i zdejmij element z drugiego końca)

Jest to relatywnie nieco rzadziej używana abstrakcyjna struktura danych, będąca skrzyżowaniem (uogólnieniem) stosu z kolejką (tzn. można na niej naturalnie symulować zachowanie zarówno stosu jak i kolejki)

Implementacja abstrakcyjnych struktur danych

Zauważmy, że definicja abstrakcyjnej struktury danych wogóle nie precyzuje jak jest ona zaimplementowana.

W istocie, jest to pozostawione do wyboru programiście i możliwe są różne implementacje np. stosu, kolejki, etc. jednak zawsze spodziewany jest taki sam interfejs i jego zachowanie. Należy też zauważyć, że świadomy wybór takiej czy innej implementacji może istotnie wpływać na złożoność czasową czy pamięciową określonych operacji danej abstrakcyjnej struktury danych. Np. abstrakcyjną strukturę danych: stos czy kolejkę można zaimplementować używając jako "konkretnej" struktury danych zarówno listy dowiązaniowej jak i tablicy.
Bardzo dobrym ćwiczeniem jest implementacja wszystkich możliwych kombinacji wg poniższej tabelki:

abstrakcyjna str. danych	implementacja:
Stos	lista jednokierunkowa, tablica (dlaczego?)
Kolejka	lista jednokierunkowa, tablica cykliczna (dlaczego?)
Kolejka dwustronna	lista dwukierunkowa (dlaczego?), tablica cykliczna

We wszystkich powyższych implementacjach można zapewnić stałą złożoność czasową (czyli niezależną od liczby przechowywanych elementów)

Przykładowe zadania:

- opisz listę dowiązaniową jedno- i dwukierunkową, oraz cykliczną
- podaj pseudokod klas listy jedno- i dwukierunkowej
- napisz pseudokod prostych funkcji operujących na powyższych (np. dodaj element na końcu listy, zwróć liczbę elementów)
- podaj przykłady algorytmów, gdzie lista jednokierunkowa wystarcza i takie, gdzie dwukierunkowa jest niezbędna, dokonaj analizy złożnoności czasowej
- porównaj zalety i wady list dowiązaniowych i tablic
- zaimplementuj kolejkę i stos zarówno na liście dowiązaniowej, jak i na tablicy, dokonaj porównawczej analizy tych implementacji (złożoność, inne wady/zalety)
- zaprojektuj i zaimplementuj abstrakcyjną strukturę danych będącą kolejką z dodatkową szybką operacją odwracania kolejności.

Dziękuję za uwagę.