Abstrakcja zbiorów w informatyce

Struktury danych potrzebne są w programach komputerowych dla przechowywania danych. Są one odpowiednikiem matematycznej koncepcji zbiorów, ale są istotne różnice pomiędzy zbiorami matematycznymi a zbiorami danych w komputerze. Te ostatnie zawsze są budowane z poszczególnych elementów, nigdy nie są nieskończone, i zawsze posiadają jakąś kolejność przechowywanych w nich elementów, podczas gdy w zbiorach matematycznych taka kolejność nie istnieje.

Zbiory danych w informatyce różnią się między sobą — między innymi — zestawem operacji jakie chcemy na nich wykonywać. Do podstawowych operacji należą: **dodawanie** oraz **usuwanie** elementu do/ze zbioru, oraz wyszukiwanie elementu, na przykład według jakiegoś **klucza**, który jest cechą lub składnikiem elementu.

Ale w konkretnym zastosowaniu zbiór danych może wymagać użycia bardziej wyspecjalizowanych operacji, czego przykładem może być znajdowanie i/lub usuwanie elementu maksymalnego, ze strukturą danych dostosowaną do łatwego wykonywania tych operacji w postaci kopców. Zatem dobra, efektywna implementacja struktury danych do danego zastosowania może zależeć od operacji jakie chcemy na tych danych wykonywać.

Tablice

Tablice (*arrays*) są strukturą danych w większości języków programowania zapisaną w ciągłym obszarze pamięci, gdzie poszczególne elementy tablicy identyfikowane są za pomocą indeksu. Tablice mogą być indeksowane począwszy od 0, albo od 1, albo za pomocą jeszcze inaczej wybranego zakresu indeksów.

Tablice mogą być jedno- lub wielowymiarowe, przy czym na przykład język programowania ANSI C traktuje tablice wielowymiarowe jako tablice tablic. W podręczniku CLRS tablice nominalnie są zawsze jednowymiarowe, a dla przypadków wielowymiarowych stosuje się określenie **macierzy** (*matrix/matrices*). Tu nie będziemy stosować tego rozróżnienia, dopuszczając tablice zarówno jedno- jaki wielowymiarowe.

Podstawowym założeniem i własnością tablic jako struktur danych, w większości implementacji, jest, że ich elementy są jednakowego rozmiaru. Wynika z tego możliwość obliczania adresów wszystkich elementów tablicy w jednolity sposób, a więc dostęp do każdego elementu nawet bardzo dużej tablicy zajmuje czas stały, niezależny od jej rozmiaru.

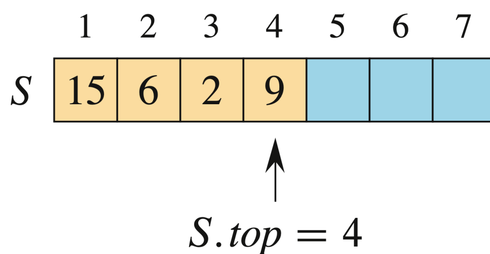
Istnieje szereg wariantów implementacji tablic wielowymiarowych, ale w większości zachowują one tę zasadę stałego czasu dostępu do każdego elementu (przy założeniu, że liczba wymiarów tablicy jest ograniczona, a nie np. rośnie wraz z liczbą elementów).

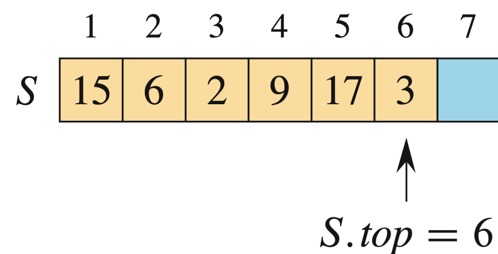
Stosy

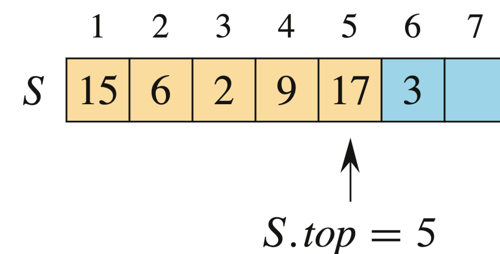
Stos jest uporządkowaną listą elementów, w której dostęp jest możliwy tylko do ostatnio wprowadzonego elementu. Nazwa stosu ma nawiązywać do stosu talerzy, do którego ma sens jedynie dodawać nowe elementy na wierzch, bez ruszania już zawartych na stosie, albo zdejmować wyłącznie z wierzchu. Operację dodawania elementu do stosu tradycyjnie nazywa się $\textsc{Push}$ a usuwania ostatniego elementu $\textsc{Pop}$.

Mówi się o „dyscyplinie stosowej" (LIFO: *Last-In-First-Out*) w odniesieniu do sytuacji, kiedy kolejność przetwarzania elementów jest odwrotna do kolejności ich pojawienia się.

Stos można zaimplementować za pomocą tablicy, która posiada atrybuty $S.top$ zawierający indeks ostatniego elementu stosu, lub 0 gdy stos jest pusty, oraz $S.size$ określający rozmiar tablicy, czyli pojemność stosu. Następujący rysunek ilustruje przykładowy początkowy stan stosu (a), stan stosu po dodaniu kolejno elementów 17 i 3 (b), i po usunięciu elementu 17 (c):







Interfejs tak implementowanego stosu odzwierciedla następujący pseudokod:

```
Stack-Empty(S)
1  if S.top == 0
2      return TRUE
3  else return FALSE

Push(S, x)
1  if S.top == S.size
2      error "overflow"
3  else S.top = S.top + 1
4      S[S.top] = x

Pop(S)
1  if Stack-Empty(S)
2      error "underflow"
3  else S.top = S.top − 1
4      return S[S.top + 1]
```

Inicjalizacja $S.top = 0$

Wskaźnikowa implementacja stosu

```
STACK-EMPTY(S)
1  if S.top == NIL
2        return TRUE
3  else return FALSE

PUSH(S, x)
1  x.next = S.top
2  S.top = x

POP(S)
1  if STACK-EMPTY(S)
2        error "underflow"
3  else S.top = NIL
```

Zakładamy tu, że tworzeniem elementów, tzn. alokacją ich pamięci, zajmuje się program wywołujący.

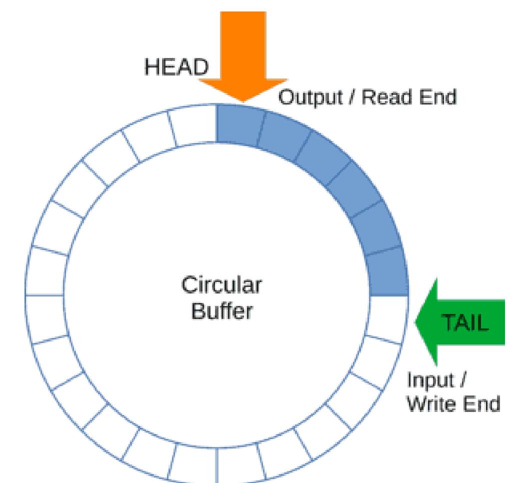
Kolejki

W **kolejce** dostęp istnieje do elementu pierwszego, zwanego również **głową** (*head*) kolejki, oraz ostatniego, zwanego **ogonem** (*tail*). Elementy dodawane do kolejki ustawiane są zawsze za ogonem, i stają się nowym ogonem kolejki, natomiast elementy usuwane brane są „z głowy”, tym samym odsłaniając nową głowę kolejki.

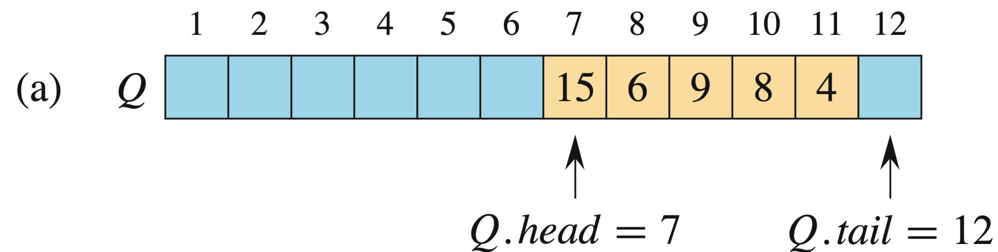
Analogicznie do dyscypliny stosowej LIFO, w przypadku kolejek mówimy o dyscyplinie FIFO (*First-In-First-Out*).

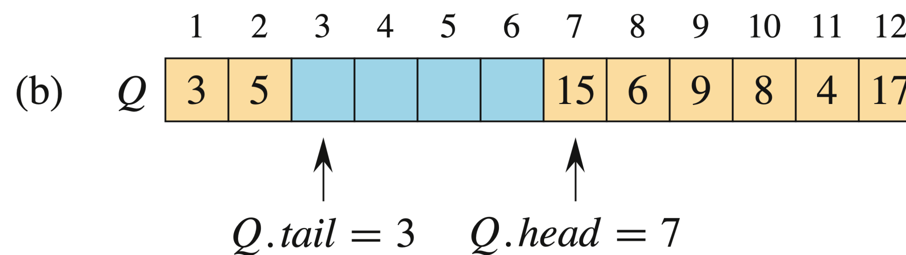
Aby zaimplementować kolejkę za pomocą tablicy, trzeba zwrócić uwagę, że w odróżnieniu od stosu, próba zapełnienia początkowego zakresu tablicy elementami kolejki prowadzi do bardzo nieefektywnej implementacji, kiedy co prawda dodanie elementu na koniec kolejki jest łatwe, ale usunięcie pierwszego elementu będzie wymagało przesunięcia (przepisania) wszystkich elementów o jedną pozycję.

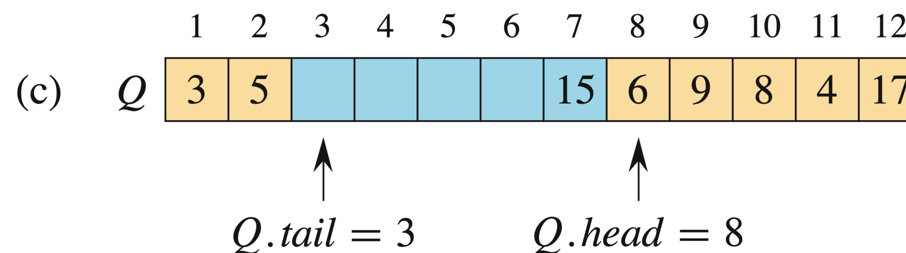
Zamiast tego, poprawna i efektywna implementacja kolejki w tablicy wykorzystuje strukturę **bufora kołowego** (*circular buffer/ring buffer*). Kolejne elementy są dodawane na koniec kolejki, a początkowe elementy usuwane. Początek kolejki nie przypada zawsze w pierwszym elemencie tablicy, tylko „kroczy”. Po zapełnieniu tablicy elementy są wypełniane od początku.



Przykładowe operacje na kolejce: stan kolejki z elementami 15,6,9,8,4 na pozycjach tablicy od 7 do 11 (a), stan kolejki po dodaniu kolejno elementów: 17, 3, i 5 (b), i stan po usunięciu (pierwszego) elementu (c):







```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.size
3        Q.tail = 1
4  else Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.size
3        Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

Uwaga: inicjalizacja $Q.head = Q.tail = 1$

Powyższy pseudokod ignoruje wyjątki, czyli próbę usunięcia elementu z pustej kolejki, oraz próbę dodania elementu do pełnej.

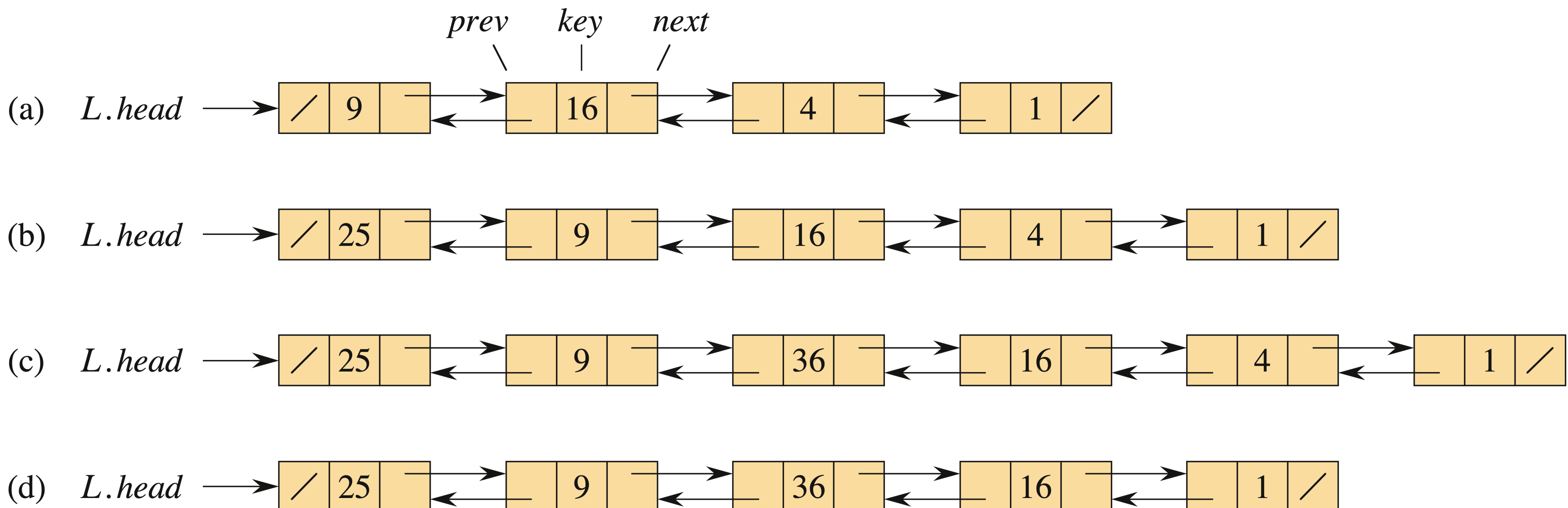
Ćwiczenie: uzupełnij powyższy pseudokod o obsługę tych wyjątków, podobnie jak w przypadku stosów.

Wskaźnikowa implementacja kolejki

Listy wskaźnikowe

Listy wskaźnikowe zawierają ciąg elementów połączony wskaźnikami. Nie rozpatrujemy tablicowej implementacji listy, ponieważ nie różni się ona niczym od tablicy jako takiej.

Przykład **listy dwukierunkowej** (a) z elementami 9, 16, i 1, (b) po wykonaniu operacji $\text{LIST-PREPEND}(L, x)$, gdzie $x.key = 25$, (c) po wykonaniu operacji $\text{LIST-INSERT}(x, y)$, gdzie $x.key = 36$, a y jest wskaźnikiem do elementu listy zawierającego klucz 9, (d) po wywołaniu $\text{LIST-DELETE}(L, x)$, gdzie jest wskaźnikiem do elementu listy zawierającego klucz 4:



```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3        x = x.next
4  return x
```

```
LIST-PREPEND(L, x)
1  x.next = L.head
2  x.prev = NIL
3  if L.head ≠ NIL
4        L.head.prev = x
5  L.head = x
```

```
LIST-INSERT(x, y)
1  x.next = y.next
2  x.prev = y
3  if y.next ≠ NIL
4        y.next.prev = x
5  y.next = x
```

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2        x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5        x.next.prev = x.prev
```

Warianty list, listy z wartownikiem

Powyższy przykład listy dwukierunkowej jest jednym z wielu możliwych wariantów list. Listy mogą być połączone jedno- lub dwukierunkowo, mogą być uporządkowane lub nie.

Jako specjalny przypadek rozważymy **listy z wartownikiem**. Zauważmy najpierw, że kod procedury LIST-DELETE mógłby być znacznie prostszy, gdyby nie konieczność każdorazowego sprawdzania warunków brzegowych:

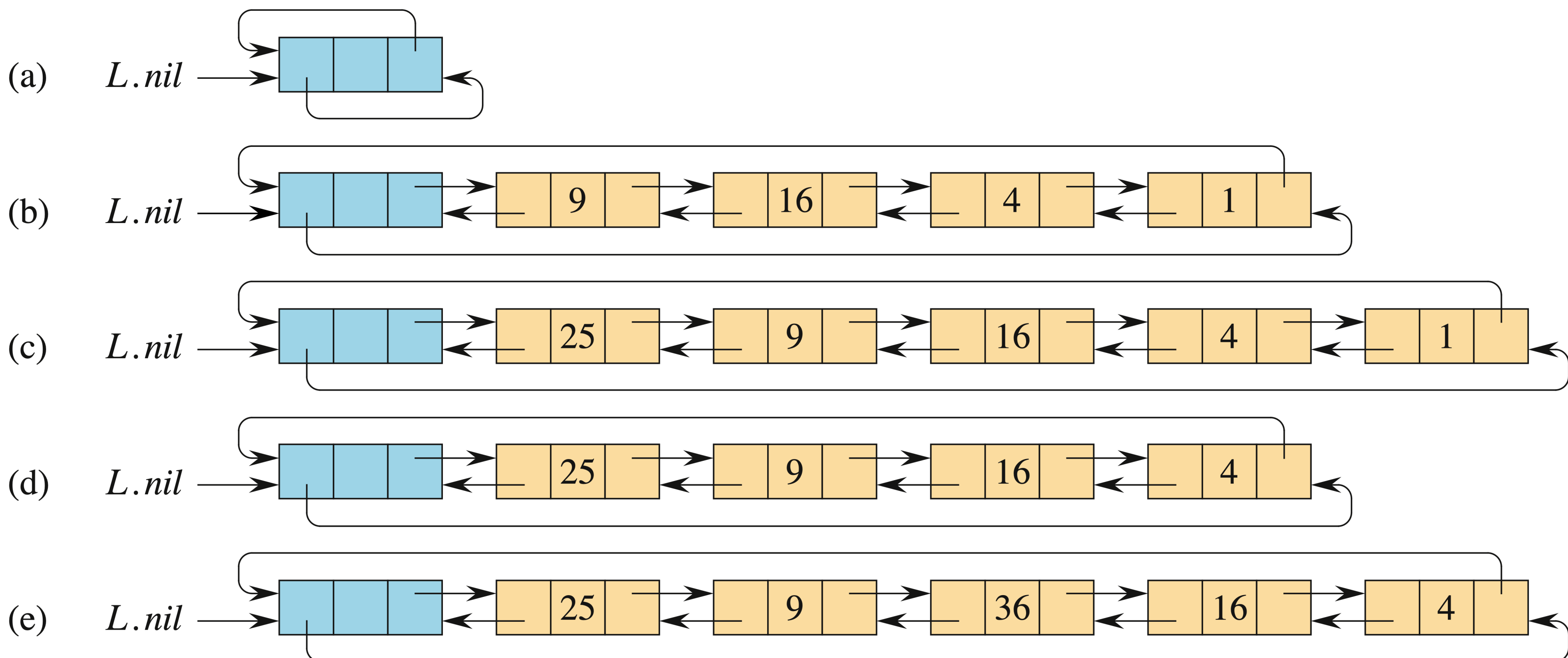
```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2        x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5        x.next.prev = x.prev
```

```
LIST-DELETE'(L, x)
1  x.prev.next = x.next
2  x.next.prev = x.prev
```

Taka implementacja byłaby możliwa, gdybyśmy mieli pewność, że lista nigdy nie będzie pusta. Jest możliwa prosta modyfikacja listy dwukierunkowej w **dwukierunkową listę cykliczną z wartownikiem**. Taka lista zawsze zawiera jeden nadmiarowy, całkowicie sztuczny element, zwany **wartownikiem**, który zastępuje wartość NIL.

Lista z wartownikiem — przykład

Przykładowe operacje na dwukierunkowej liście cyklicznej z wartownikiem: (a) pusta lista, (b) lista podobna jak w poprzednich przykładach, z elementami: 9, 16, 4, i 1, (c) lista po wywołaniu LIST-INSERT'$(x, L.nil)$ gdzie $x.key = 25$; ponieważ nowy element wstawiany jest bezpośrednio za wartownikiem, to staje się pierwszym elementem listy, (d) lista po usunięciu elementu z kluczem 1, (e) lista po wywołaniu LIST-INSERT'(x, y) gdzie $x.key = 36$ i y jest wskaźnikiem do elementu z kluczem 9:



Listy z wartownikiem — dodawanie elementu

Zastosowanie listy z wartownikiem upraszcza również nieco procedurę LIST-INSERT:

LIST-INSERT(x, y)

1 $x.next = y.next$
2 $x.prev = y$
3 **if** $y.next \neq NIL$
4 $\quad y.next.prev = x$
5 $y.next = x$

LIST-INSERT'(x, y)

1 $x.next = y.next$
2 $x.prev = y$
3 $y.next.prev = x$
4 $y.next = x$

Listy z wartownikiem — przeszukiwanie

Przeszukiwanie listy z wartownikiem jest podobne do zwykłej listy nieuporządkowanej, jednak można wykonać w nim osiągnąć skrócenie czasu wykonania. Oryginalna procedura LIST-SEARCH wykonuje w wierszu 2 dwa porównania: pierwsze sprawdza koniec listy, i drugie, czy został znaleziony poszukiwany element. W przypadku listy z wartownikiem pierwsze sprawdzenie można pominąć, ale wtedy, w przypadku braku na liście elementu z poszukiwanym kluczem, procedura „zawinie się" na końcu listy i będzie kręciła się w kółko. Aby tego uniknąć, można wpisać poszukiwany klucz do wartownika zapewniając, że na pewno zostanie znaleziony, co zatrzyma przeszukiwanie:

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3      x = x.next
4  return x

LIST-SEARCH'(L, k)
1  L.nil.key = k            // store the key in the sentinel to guarantee it is in list
2  x = L.nil.next           // start at the head of the list
3  while x.key ≠ k
4      x = x.next
5  if x == L.nil            // found k in the sentinel
6      return NIL           // k was not really in the list
7  else return x            // found k in element x
```

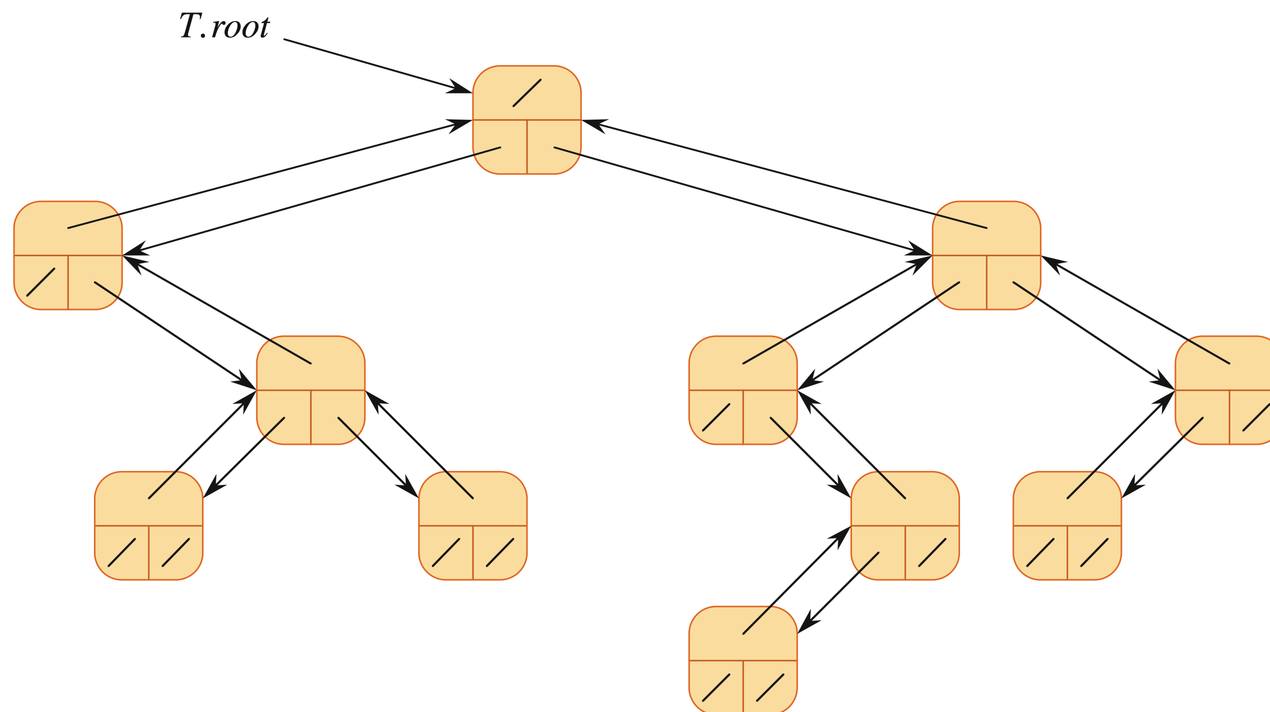
Listy z wartownikiem — podsumowanie

Listy z wartownikiem są przykładem struktury danych, która kosztem minimalnej rozbudowy struktury pozwala uprościć kod stosowanych funkcji, a także przyspieszyć podstawową operację przeszukiwania całej listy (ale jest to przyspieszenie tylko o stały współczynnik, bez poprawy złożoności asymptotycznej).

Jednak o ile w przypadku intensywnie wykorzystywanych list zastosowanie takiej struktury jak najbardziej ma sens, to nie zawsze jest uzasadnione. Na przykład, gdy program często tworzy niewielkie listy, i potem je kasuje, to narzut na dodatkową pamięć i utworzenie listy z wartownikiem może niwelować ewentualne zyski.

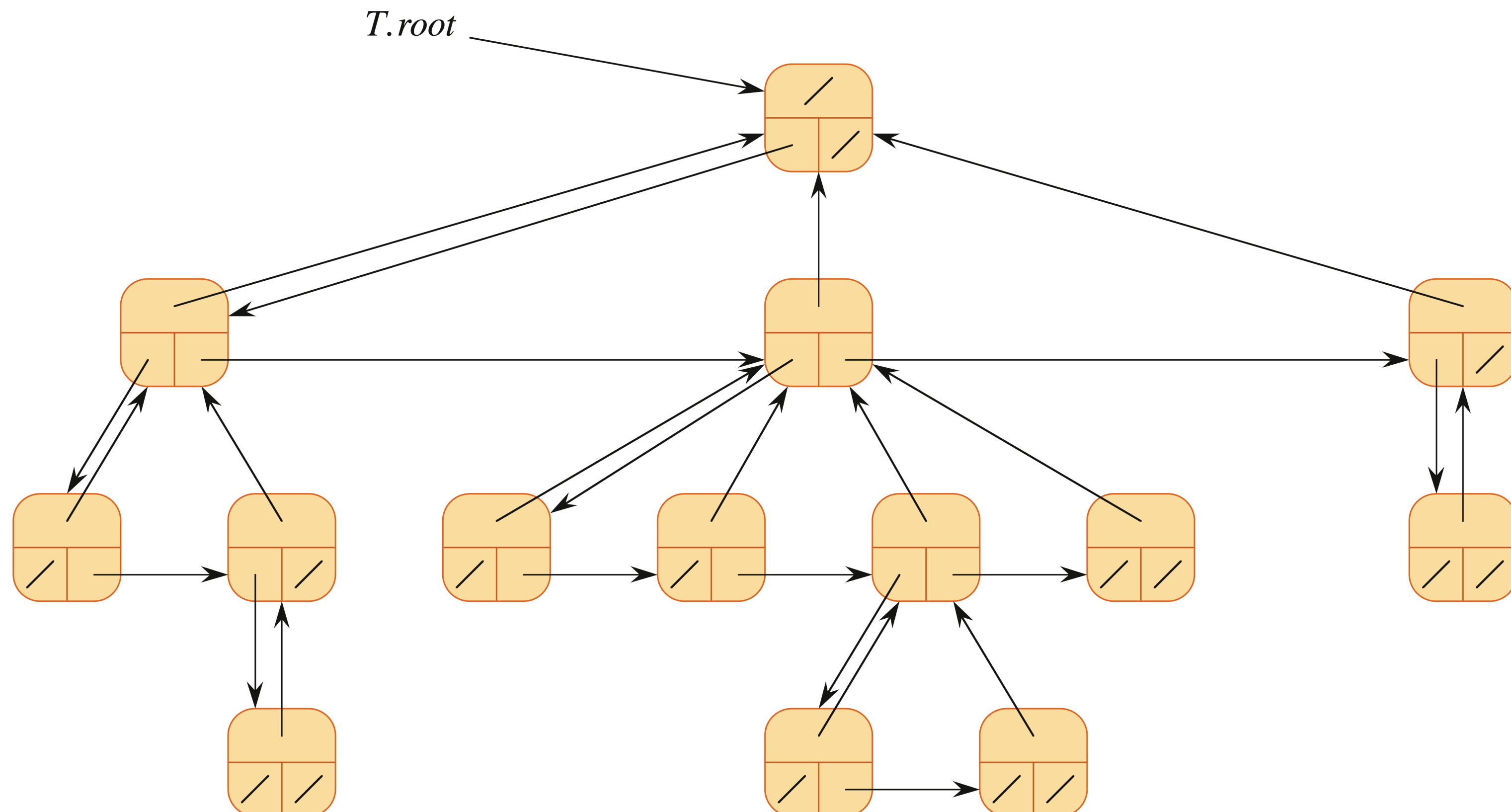
Drzewa binarne

Listy są dobrą reprezentacją kolekcji elementów, które naturalnie układają się w sekwencję. Jednak nie zawsze jedyną relacją jaką chcemy odzwierciedlić w strukturze danych jest sekwencja. Poniższy rysunek przedstawia **drzewo binarne** zbudowane z jednakowych obiektów zawierających wskaźniki do lewego i prawego poddrzewa, oraz wskaźnik do rodzica. Umieszczenie wszystkich tych wskaźników w każdym elemencie pozwala na łatwe **przeglądanie** drzewa w dowolnym porządku. Podobnie jak ze wskaźnikami do poddrzew, pusty wskaźnik do rodzica wskazuje, że dany węzeł nie ma rodzica, czyli jest korzeniem drzewa. Rysunek nie pokazuje pól na klucz i pozostałe dane przechowywane w każdym elemencie drzewa:



Drzewa wskaźnikowe z dowolną liczbą podgałęzi

Można zastosować podobne podejście jak do drzew binarnych aby budować drzewa o dowolnej **arności**. Jednak wtedy określona liczba poddrzew danego węzła determinuje ile wskaźników do poddrzew musi być pamiętanych w każdym węźle. Nie może być ich więcej, a jeśli będzie mniej, to niektóre będą musiały być puste. Alternatywnie, poniższy rysunek ilustruje ciekawy schemat reprezentacji drzew niebinarnych, zwany **lewy-potomek, prawy-brat**. Ten schemat dopuszcza dowolną liczbę potomków każdego węzła wykorzystując pamięć $O(n)$ dla drzewa o n węzłach:



Krótkie podsumowanie — pytania sprawdzające

1. Wyjaśnij jak zaimplementować dwa stosy w jednej tablicy $A[1:n]$ aby do przepełnienia któregoś z nich dochodziło dopiero wtedy, gdy łączna liczba elementów w obu stosach osiąga n. Procedury $\textsc{Push}$ i $\textsc{Pop}$ powinny działać w czasie $O(1)$.

2. Uzupełnij procedury $\textsc{Enqueue}$ i $\textsc{Dequeue}$ o wykrywanie błędów niedomiaru i przepełnienia.

3. Zaimplementuj stos (napisz jego pseudokod) za pomocą listy jednokierunkowej (w której węzły nie zawierają wskaźnika $prev$. Operacje $\textsc{Push}$ i $\textsc{Pop}$ powinny działać w czasie $O(1)$. Czy konieczne jest dodanie w tym celu jakichś atrybutów do listy?

4. Napisz rekurencyjną procedurę (pseudokod) działającą w czasie $O(n)$, która dla danego drzewa binarnego o n węzłach wypisuje wszystkie jego klucze.

5. Napisz rekurencyjną procedurę (pseudokod) działającą w czasie $O(n)$, która wypisuje wszystkie klucze danego drzewa o n węzłach, pamiętanego w reprezentacji **lewy-potomek, prawy-brat**.

Literatura i materiały pomocnicze

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L Rivest, Clifford Stein: Wprowadzenie do algorytmów, PWN, 2024, rozdział 10.