Struktura danych

Sposób uporządkowania informacji w komputerze. Na strukturach danych operują algorytmy.

Przykładowe struktury danych:

– rekord
– tablica
– lista
– stos
– kolejka
– drzewo i jego odmiany (np. drzewo binarne)
– graf

Zbiory

Zbiór dynamiczny: zbiór, który może się powiększać czy też zmniejszać w czasie.

Słownik: zbiór dynamiczny, z określonymi operacjami wstawiania elementów do zbioru, usuwania elementów ze zbioru oraz sprawdzania, czy dany element należy do zbioru.

Elementy zbioru dynamicznego
Każdy element zbioru jest reprezentowany przez obiekt, którego pola można odczytywać oraz modyfikować, jeżeli dysponujemy **wskaźnikiem** do tego obiektu.
Czasem jedno z pól każdego obiektu należącego do zbioru jest wyróżnione jako jego **klucz** (ang. *key*).

Operacje na zbiorach dynamicznych: zapytania oraz operacje modyfikujące.

`Search(S, k)`
Zapytanie, które dla danego zbioru `S` oraz wartości klucza `k`, daje w wyniku wskaźnik `x` do takiego elementu w zbiorze `S`, że `key[x]=k` lub `NIL` (wskazanie „do niczego”), jeżeli żaden taki element nie należy do `S`.

`Insert(S, x)`
Operacja, która do zbioru `S` dodaje element wskazywany przez `x`.

`Delete(S, x)`
Operacja, która dla danego wskaźnika `x` do elementu w zbiorze `S` usuwa ten element ze zbioru.

`Minimum(S)`
Zwraca w wyniku element zbioru `S` o najmniejszym kluczu. Zbiór `S` powinien być liniowo uporządkowany.

`Maximum(S)`

Zwraca w wyniku element zbioru `S` o największym kluczu. Zbiór `S` powinien być liniowo uporządkowany.

`Successor(S, x)`

Zapytanie, które dla danego elementu `x` o kluczu należącym do uporządkowanego zbioru `S` da w wyniku następnik elementu `x` w `S` (czyli najmniejszy element zbioru `S`, większy od `x`) lub `NIL`, jeżeli `x` jest największym elementem w `S`.

`Predecessor(S, x)`

Zapytanie, które dla danego elementu `x` o kluczu należącym do uporządkowanego zbioru `S` da w wyniku poprzednik elementu `x` w `S` (czyli największy element zbioru `S`, mniejszy od `x`) lub `NIL`, jeżeli `x` jest najmniejszym elementem w `S`.

Realizacja zbiorów dynamicznych za pomocą prostych struktur danych

Stosy i kolejki

Stos – liniowa struktura danych, w której dane dokładane są na koniec zbioru i z końca zbioru są pobierane (czyli pobierany jest element dodany najpóźniej).
Strategia LIFO (ang. Last In, First Out; ostatni na wejściu, pierwszy na wyjściu).

Kolejka – liniowa struktura danych, w której dane pobierane są w kolejności ich dołożenia do zbioru (czyli pobierany jest element dodany najwcześniej).
Strategia FIFO (ang. First In, First Out; pierwszy na wejściu, pierwszy na wyjściu).

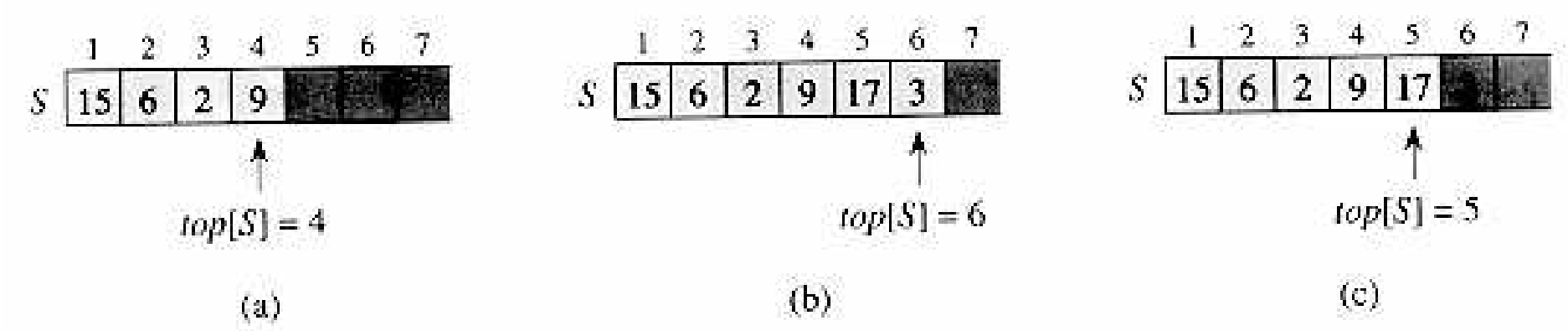
Operacje na stosie

Oprerację `Insert` wstawiającą element do zbioru zwyczajowo nazywa się `Push`, a operację `Delete`, usuwającą element ze zbioru – `Pop`. Operacja `Pop` jest bezargumentowa (zawsze wiadomo, który element usunąć – ten na szczycie stosu, dodany najpóźniej).

Stos zawierający nie więcej niż `n` elementów można zaimplementować w tablicy `S[1...n]`. Z taką tablicą związany jest dodatkowy atrybut `top[S]`, którego wartość jest numerem ostatnio wstawionego elementu. Stos składa się z elementów `S[1... top[S]]`, gdzie `S[1]` jest elementem na dnie stosu, a `top[S]` na jego wierzchołku.

Jeżeli `top[S]=0`, to stos jest pusty. Do sprawdzenia, czy stos jest pusty, używamy operacji `Stack-Empty`. Próba zdjęcia elementu ze stosu pustego powinna być sygnalizowana jako błąd niedomiaru, a jeżeli `top[S]` jest większe niż `n`, to mamy do czynienia z przepełnieniem.

Wynik działania operacji `Push` i `Pop` na stosie implementowanym za pomocą tablicy:



a) Stos ma cztery elementy, na jego wierzchołku znajduje się liczba `9`.

b) Stos po wykonaniu operacji `Push(S,17)` i `Push(S,3)`.

c) Stos po wykonaniu operacji `Pop`. W jej wyniku otrzymujemy liczbę `3` jako ostatnio dodaną. Liczba `3` znajduje się dalej fizycznie w tablicy, ale nie należy do stosu (bo element ostatni to teraz `17`).

Implementacja operacji na stosie

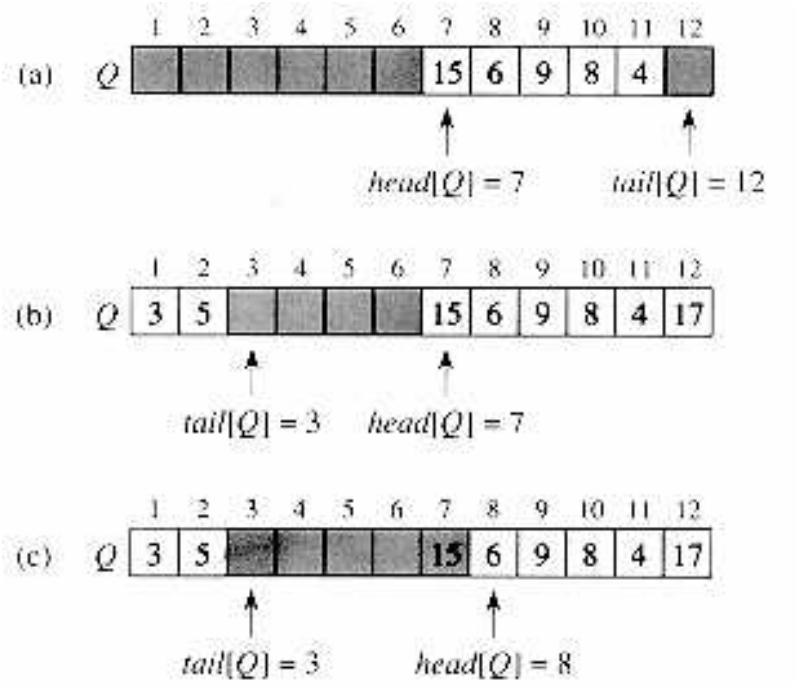
```
Stack-Empty(S):
  if top[S]=0
     return TRUE
  else
     return FALSE

Push(S,x):
  top[S] := top[S]+1
  S[top[S]] := x

Pop(S):
  if Stack-Empty(S)
     error "niedomiar"
  else
     top[S] := top[S]-1
     return S[top[S]+1]
```

Kolejki

Operacja wstawiania elementu do kolejki: `Enqueue`, operacja usuwania elementu: `Dequeue`.
Kolejka ma początek (*głowę*) oraz koniec (*ogon*). Po wstawieniu elementu do kolejki, zostaje on umieszczony na jej końcu (w *ogonie*). Element może zostać z niej usunięty tylko wtedy, gdy znajduje się na jej początku (w *głowie*).

Kolejkę o co najwyżej `n-1` elementach można zaimplementować za pomocą tablicy `Q[1..n]`.
Atrybut `head[Q]` wskazuje na *głowę* kolejki. Atrybut `tail[Q]` wyznacza następną wolną pozycję, na którą można wstawić do kolejki nowy element.
Elementy kolejki znajdują się na pozycjach:
`head[Q]`, `head[Q]+1`, ... `tail[Q]-1`.

Umawiamy się, że kolejka jest cykliczna, tzn. pozycja o numerze `1` jest bezpośrednim następnikiem pozycji o numerze `n`. Zatem, jeżeli `head[Q]=tail[Q]` to kolejka jest pusta. Początkowo `head[Q]=tail[Q]=1`. Jeżeli kolejka jest pusta, to operacja `Dequeue` powinna zwrócić błąd niedomiaru. Jeżeli `head[Q]=tail[Q]+1`, to kolejka jest pełna. Operacja `Enqueue` powinna w takim wypadku zwrócić błąd przepełnienia.



a) kolejka zawiera `5` elementów na pozycjach od `7` do `11`.
b) kolejka po operacjach: `Enqueue(Q,17), Enqueue(Q,3)` oraz `Enqueue(Q,5)`
c) operacja `Dequeue(Q)` daje w wyniku element `15` (element „z *głowy*”). Po jej wykonaniu w *głowie* znajduje się liczba `6`.

Implementacja operacji na kolejkach

```
Enqueue(Q,x):
  Q[tail[Q]] := x
  if tail[Q] = length[Q]
     tail[Q] := 1
  else
     tail[Q] := tail[Q]+1

Dequeue(Q):
  x := Q[head[Q]]
  if head[Q] = length[Q]
     head[Q] := 1
  else
     head[Q] := head [Q]+1
     return x
```

Powyższe procedury nie obejmują kontroli błędów przepełnienia i niedomiaru.

Listy

Lista tablicowa – struktura danych, w której elementy są ułożone w liniowym porządku. Lista zaimplementowana w ten sposób opiera się na tablicy obiektów (lub rekordów) danego typu.

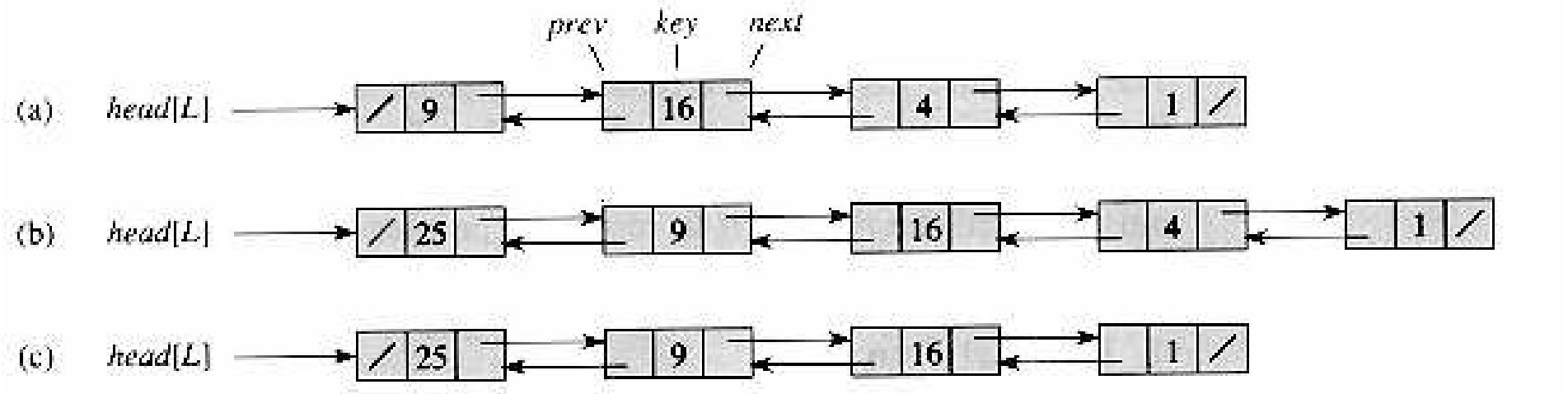
Dopisanie elementu do listy to wstawienie elementu do tablicy.
Jeśli ma ono nastąpić na końcu listy, będzie to kolejny element w tablicy.
Jeśli nowy element ma znaleźć się między innymi elementami, należy przesunąć o jedno pole w prawo wszystkie elementy o indeksie wyższym niż pole, na które będzie wstawiany obiekt; następnie w powstałą lukę wpisuje się nowy element.

Usunięcie elementu znajdującego się pod danym indeksem tablicy to przesunięcie o jedno pole w lewo wszystkich elementów o indeksie wyższym.

Lista z dowiązaniami (lista wskaźnikowa) – struktura danych, w której elementy są ułożone w liniowym porządku. O ile jednak w tablicy porządek jest wyznaczony przez indeksy tablicy, to porządek na liście z dowiązaniami określony jest przez wskaźniki związane z każdym elementem listy.

Lista dwukierunkowa – odmiana listy wskaźnikowej, w której w każdym elemencie listy jest przechowywane odniesienie zarówno do następnika jak i poprzednika elementu w liście. Taka reprezentacja umożliwia swobodne przemieszczanie się po liście w obie strony.
Każdy element takiej listy musi się więc składać co najmniej z trzech pól: `key` (to pole zawiera klucz elementu), `next` oraz `prev`.
Dla danego elementu `x` na liście, atrybut `next[x]` wskazuje na jego następnik na liście, natomiast `prev[x]` na jego poprzednik.
Jeśli `prev[x]=NIL`, to oznacza, że element `x` nie ma poprzednika – jest *głową* listy (pierwszym elementem), jeśli `next[x]=NIL`, to element nie ma następnika – jest ostatnim elementem listy, czyli jej *ogonem*.

Atrybut `head[L]` wskazuje na pierwszy element listy, jeżeli `head[L]=NIL`, to lista jest pusta.



a) Lista dwukierunkowa `L`. Wskaźniki do następnego i poprzedniego elementu zobrazowane są za pomocą strzałek. Ukośnik symbolizuje wartość `NIL`.
b) Lista po wykonaniu operacji `List-Insert(L,x)`, gdzie `key[x]=25`.
c) Lista po wykonaniu operacji `List-Delete(L,x)`, gdzie `x` wskazuje na element listy o kluczu `4`.

Lista jednokierunkowa – lista, w której pomijamy wskaźnik `prev`.

Lista cykliczna – lista, w której pole `prev` elementu w *głowie* wskazuje na *ogon*, a pole `next` w *ogonie* wskazuje na *głowę*. Elementy takiej listy tworzą pierścień.

Lista jest **posortowana**, jeżeli kolejność elementów na liście jest zgodna z porządkiem na ich kluczach; element o najmniejszym kluczu znajduje się w *głowie* listy, a ten o kluczu największym – w *ogonie*. Na liście **nieposortowanej** kolejność elementów jest dowolna.

Wyszukiwanie na listach z dowiązaniami.

Procedura `List-Search(L,k)` wyznacza pierwszy element o kluczu `k` na liście `L`, za pomocą prostego liniowego przeglądania (zakładamy, że lista jest dwukierunkowa i nieposortowana). Wynikiem działania procedury jest wskaźnik do tego elementu; jeżeli na liście nie ma elementu o kluczu `k`, zwracana jest wartość `NIL`.

```
List-Search(L,k):
  x := head[L]
  while x ≠ NIL and key[x] ≠ k
      x := next[x]
  return x
```

Pesymistyczny czas działania procedury dla listy o `n`-elementach wynosi $\Theta(n)$.

Wstawanie do listy z dowiązaniami

Procedura `List-Insert` przyłącza element `x` na początek listy.

```
List-Insert(L,x):
  next[x] := head[L]
  if head[L] ≠ NIL
      prev[head[L]] :=  x
  head[L] := x
  prev[x] := NIL
```

Procedura działa w czasie stałym `O(1)`.

Usuwanie z list z dowiązaniami

Wywołanie procedury `List-Delete` powoduje usunięcie elementu `x` z listy `L`. Element zostaje wycięty poprzez modyfikację odpowiednich wskaźników.

```
List-Delete(L,x):
  if prev[x] ≠ NIL
     next[prev[x]] := next[x]
  else
     head[L] := next[x]
  if next[x] ≠ NIL
     prev[next[x]] := prev[x]
```

Procedura działa w czasie $O(1)$. W celu usunięcia elementu o zadanej wartości klucza, należy najpierw wywołać procedurę `List-Search` w celu wyznaczenia wskaźnika do takiego elementu. W takim przypadku procedura usuwania usunięcia elementu wymaga czasu $\Theta(n)$.

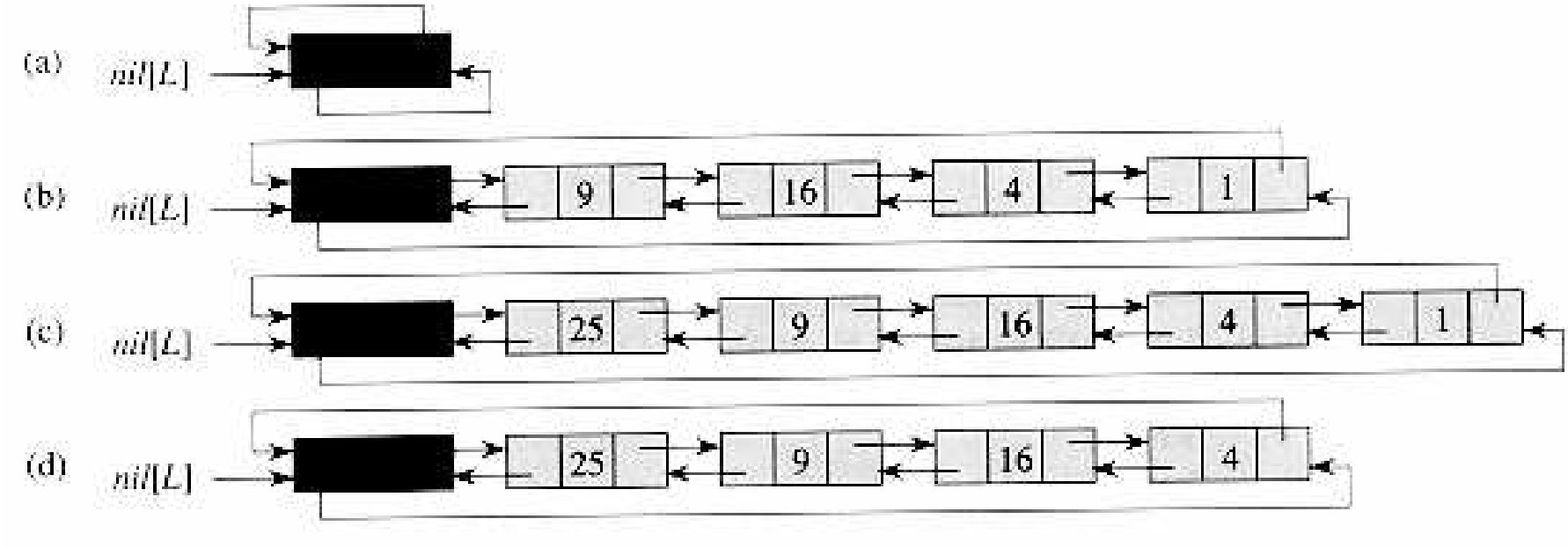
Wartownicy

Gdyby można było pominąć warunki brzegowe dotyczące *ogona* i *głowy* listy, procedura `List-Delete` by się znacznie uprościła:

```
List-Delete'(L,x):
     next[prev[x]] := next[x]
     prev[next[x]] := prev[x]
```

Wartownik jest sztucznym elementem `nil[L]`, który pozwala uprościć warunki brzegowe.

Lista `L` z dowiązaniami oraz wartownikiem `nil[L]` to cykliczna dwukierunkowa, w której element `nil[L]` zawsze znajduje się między głową a ogonem. Atrybut `head[L]` staje się zbędny, ponieważ na głowę zawsze wskazuje `next[nil[L]]`.



a) Lista pusta.
b) Lista z kluczem `9` w *głowie* oraz `1` w *ogonie*.
c) Lista po wykonaniu `List-Insert'[L,x]`, gdzie `key[x]=25`. Nowy element został umieszczony w *głowie*.
d) Lista po usunięciu elementu o kluczu `1`.

Zmodyfikowane procedury wyszukiwania i wstawiania:

```
List-Search'(L,k):
  x := next[nil[L]]
  while x ≠ nil[L] and key[x] ≠ k
      x := next[x]
  return x


List-Insert'(L,x):
  next[x] := next[nil[L]]
  prev[next[nil[L]]] := x
  next[nil[L]] := x
  prev[x] := nil[L]
```

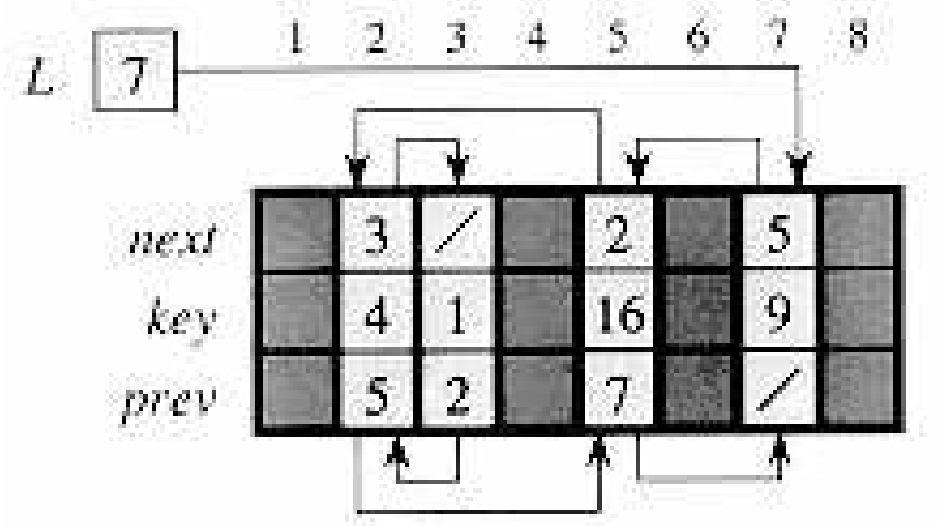
Użycie wartowników prowadzi do zmniejszenia stałych współczynników, ale nie do poprawy asymptotycznej złożoności operacji. Pozwala również na uproszczenie kodu procedur.

Reprezentowanie struktur wskaźnikowych za pomocą tablic

Niektóre języki programowania nie znają pojęcia wskaźnika. W takim przypadku wskaźniki można symulować za pomocą tablic.

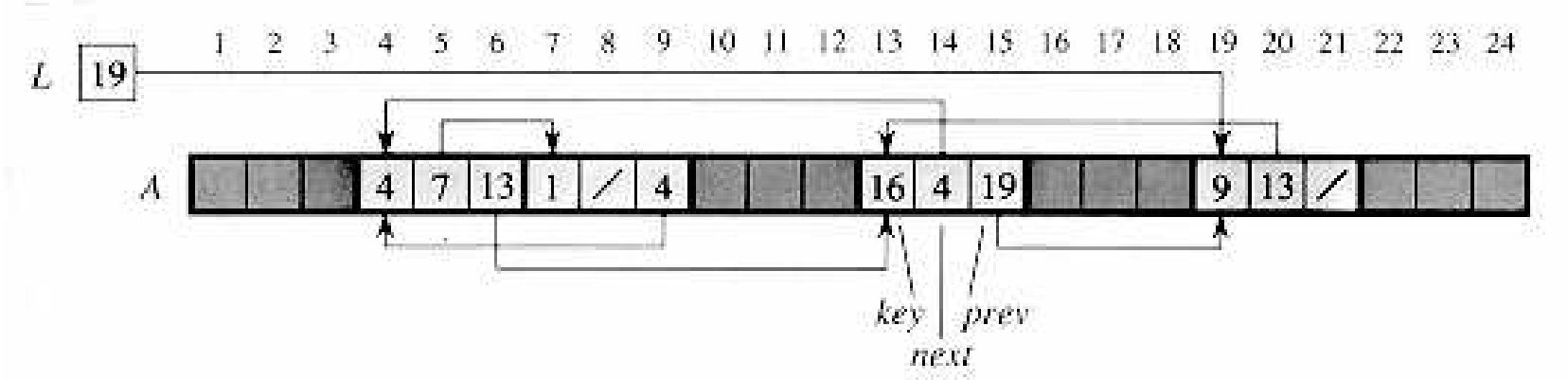
Reprezentacja wielotablicowa

Rolę wskaźników odgrywają indeksy w trzech tablicach zawierających elementy `key`, `next` oraz `prev`.



W zmiennej `L` jest pamiętana pozycja głowy listy.

Reprezentacja jednotablicowa



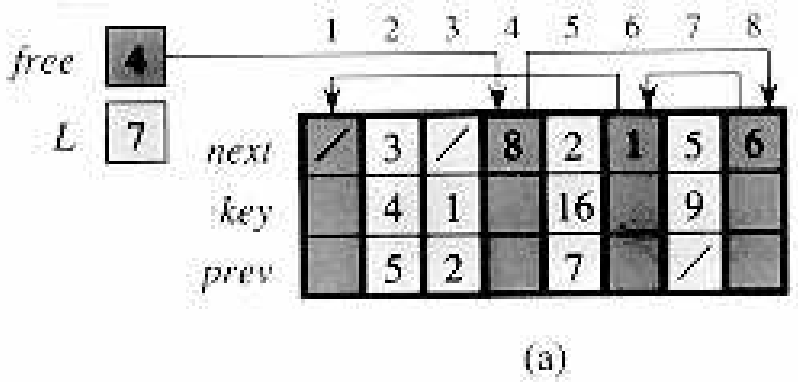
Każdy element listy zajmuje spójny fragment tablicy `A[j...k]`.
Aby np. odczytać wartość `prev[i]` mając dane `i`, należy odczytać wartość pod indeksem `i+2`. Poszukiwana wartość znajduje się zatem pod indeksem `A[i+2]`.
W zmiennej `L` jest pamiętana pozycja głowy listy.

Przydzielanie i zwalnianie pamięci.

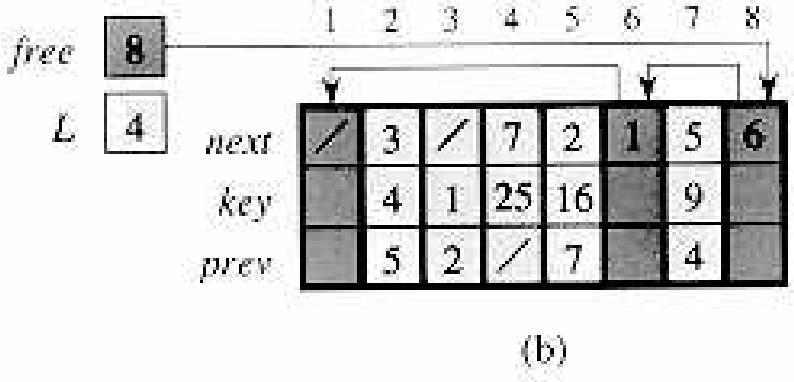
Przyjmijmy, że tablice w wielotablicowej reprezentacji listy dwukierunkowej mają długość `m`, oraz że w pewnej chwili na liście znajduje się $n \leq m$ elementów. Zatem `n` rekordów reprezentuje elementy należące do listy, a pozostałe `m-n` rekordów jest wolnych.

Wolne rekordy będą przechowywane na specjalnej liście jednokierunkowej – liście wolnych pozycji. Do jej reprezentacji wykorzystamy „puste” miejsca w tablicy `next`. Wskaźnik do głowy tej listy będzie przechowywany w zmiennej `free`.

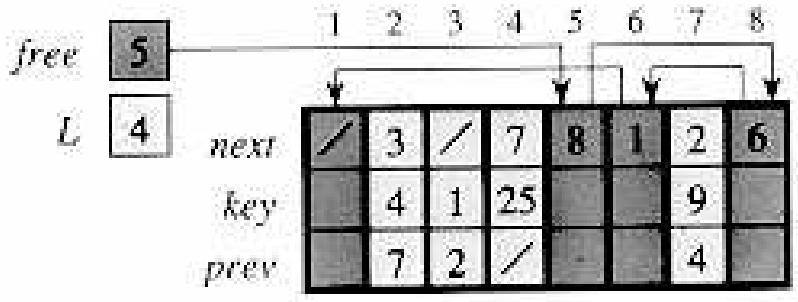
Formalnie, lista wolnych pozycji jest stosem. Przydzielana jest zawsze ostatnio zwolniona pozycja.



(a)



(b)



(c)

a) Lista reprezentowana za pomocą trzech tablic. W wolnych miejscach tablicy `next` przechowywane są kolejne wskaźniki wolnych rekordów. Zmienna `free` przechowuje indeks głowy listy.
b) Ilustracja procedury przydzielania pamięci (`Allocate-Object`).
c) Ilustracja procedury zwalniania pamięci (`Free-Object`).

Kod procedur przydzielających i zwalniających pamięć

```
Allocate-Object():
  if free = NIL
     error "brak pamięci"
  else
     x := free
     free := next[x]
     return x


Free-Object(x):
  next[x] := free
  free := x
```

Obie procedury działają w czasie `O(1)`.