ASD—7
Abstrakcyjne struktury danych - słowniki
DRZEWA POSZUKIWAŃ BINARNYCH

Do najczęściej wykonywanych operacji na zbiorze elementów należą wstawianie elementu do zbioru, usuwanie elementu ze zbioru i wyszukiwanie elementów o zadanych własnościach (np. najmniejszego elementu lub elementu o podanej wartości klucza). Struktura danych umożliwiająca wykonywanie takich operacji nazywa się *słownikiem*. Ważnymi przykładami zastosowań słownika są bazy danych, tablice identyfikatorów w kompilatorach i komputerowe słowniki języków naturalnych.

Przez problem słownika rozumie się następujące zadanie: podać strukturę danych umożliwiającą wykonywanie następujących operacji na skończonym zbiorze S:

1. construct(S):: $S = \emptyset$;
2. search(v,S):: sprawdź, czy element v należy do S; jeśli tak, wyznacz miejsce, gdzie znajduje się v;
3. insert(v,S):: $S = S \cup \{v\}$;
4. delete(v,S):: $S = S - \{v\}$;

Uwaga: Specyfikacja operacji search odbiega od pozostałych specyfikacji, gdyż odwołuje się do konkretnej implementacji zbioru S, w której ma sens mówienie o miejscu zapisania elementu zbioru.

W zastosowaniach zwykle element zbioru jest rekordem informacji posiadającym swoją wewnętrzną strukturę. Wówczas wyszukiwanie odbywa się względem pewnego atrybutu rekordu nazywanego kluczem. Gdy może być więcej niż jeden rekord o tym samym kluczu, rekordy takie trzymane są zwykle na jednej liście. W naszym sformułowaniu problemu wyszukiwania nie ma tej trudności - zakładamy, że S jest zbiorem.

1 Implementacja listowa nieuporządkowana

Najprostsza implementacja listowa polega na zapisie elementów zbioru $S = \{a_1, a_2, ..., a_n\}$ w dowolnej kolejności na liście $L = [a_1, a_2, \ldots, a_n]$. Z kolei lista może być przechowywana, albo w tablicy albo w strukturze dowiązań. Dla każdej operacji (2)–(4) otrzymujemy $W(n) = A(n) = O(n)$ (zakładając sekwencyjne szukanie elementu na liście). Zaletą jest prostota struktury danych i niewielka ilość dodatkowej pamięci; wadą jest natomiast długi czas działania operacji.

Warto zwrócić uwagę na jedną odmianę bezpośredniej implementacji listowej, w której element będący argumentem operacji search lub insert jest ustawiany na początku listy. Gdy rozkład częstości dostępu do elementu nie jest równomierny, ułożenie elementów na liście dostosowuje się do rozkładu częstości; elementy będące często argumentami operacji przejawiają tendencje do pozostawania blisko początku listy i w związku z tym dostęp do nich jest szybki.

2 Implementacja listowa uporządkowana

Elementy na liście mogą być też trzymane w kolejności uporządkowanej. Wygodnie jest wówczas używać tablicy. Elementy zbioru S zapisujemy w tablicy $a[1..n]$ tak, aby $a[0] < a[1] < ... < a[n-1]$.

Operację search można zrealizować używając metody "dziel-i-zwyciężaj". Aby sprawdzić czy v znajduje się w przedziale $a[l] < a[l+1] < ... < a[r]$ wiedząc, że $a[l] \leq v \leq a[r]$, wybieramy indeks k, $l \leq k < r$, porównujemy v z $a[k]$ i w zależności od wyniku porównania, albo szukamy v w $a[l] < ... < a[k]$, albo w $a[k+1] < ... < a[r]$. Wówczas $W(n) = \log n + O(1)$, $A(n) = \log n + O(1)$.

Pozostałe operacje słownika insert i delete w implementacji za pomocą tablic uporządkowanych są pracochłonne: $W(n) = A(n) = O(n)$.

3 Drzewa poszukiwań binarnych

Drzewo uporządkowane jest drzewem ukorzenionym, w którym zbiór następników każdego węzła jest uporządkowany. *Drzewo binarne* jest drzewem uporządkowanym spełniającym warunki:

1. wśród następników każdego wierzchołka rozróżniamy *lewy* i *prawy* następnik, oraz

2. każdy wierzchołak ma co najwyżej jeden prawy i jeden lewy następnik.

Poddrzewo T_l (jeżeli istnieje), którego korzeń jest lewym następnikiem wierzchołka v, nazywamy *lewym poddrzewem* v. Podobnie definiujemy prawe poddrzewo T_r.

Uogólnieniem wyszukiwania w tablicy uporządkowanej są *drzewa poszukiwań binarnych*, czyli drzewa BST.

Zakładamy, że każdy węzęl x w drzewie ma trzy atrybuty:

x: left(x) key(x) right(x)

W drzewach BST mamy do czynienia z *porządkiem symetrycznym.* W porządku tym dla każdego węzła x jest spełniony następujący warunek:
jeśli węzeł y leży w lewym poddrzewie x, to key(y) < key(x),

jeśli y leży w prawym poddrzewie x, to key(x) < key(y).

Drzewem poszukiwań binarnych (drzewem BST) nazywamy dowolne drzewo binarne, w którym elementy zbioru są wpisane do wierzchołków zgodnie z porządkiem symetrycznym.

Oto moduł implementujący drzewa poszukiwań binarnych.

```
/* bstdef.h */
#ifndef BSTDEF
#define BSTDEF
#include <stdio.h>

typedef long Tklucz;
typedef int Tinfo;

#define rowne(s1,s2)  ((s1) == (s2))
#define mniejsze(s1,s2) ((s1) < (s2))
#define skopiujklucz(s1, s2)   ((s1) = (s2))
#define skopiujinfo(i1, i2)    ((i1) = (i2))

#endif
```

```
/* bst.h */
#ifndef BST
#define LISTA
#include <stdio.h>
#include "bstdef.h"

typedef struct Wezel{
  Tklucz klucz;
  Tinfo info;
  struct Wezel *lewy, *prawy;
} *PtrWezel;

PtrWezel construct(void);
/* Zbuduj puste drzewo */
PtrWezel search(PtrWezel korzen,
                Tklucz kl, PtrWezel *poprz);
/* znajdz wezel z kluczem kl w drzewie
   o korzeniu ``korzen'';
   *poprz --- wskaznik do poprzednika
   znalezionego wezla;
   jesli kl nie ma w drzewie,
   to zwracane jest NULL, *poprz
   jest poprzednikiem ``tego wlasnie'' NULL */
void insert(PtrWezel *korzen,
```

```
            Tklucz kl, Tinfo nowainfo);
/* wstaw nowa informacje
   ‘‘nowainfo’’ z kluczem ‘‘kl’’ */
void delete(PtrWezel *korzen, Tklucz kl);
/* usun inormacje z kluczem kl */
#endif
```

```
/* bst.c */
#include "bst.h"

/*
typedef struct Wezel{
  Tklucz klucz;
  Tinfo info;
  struct Wezel *lewy, *prawy;
} *PtrWezel;
*/

PtrWezel construct(void)
{
  return NULL;
}
```

```
/* wyszukiwanie rekurencyjne
PtrWezel search(PtrWezel korzen,
                Tklucz kl, PtrWezel *poprz)
{
  if (korzen == NULL)
    return NULL;
  else if (korzen->klucz == kl)
    return korzen;
  else{
    *poprz = korzen;
    if (mniejsze(kl,korzen->klucz))
      return search(korzen->lewy,kl,poprz);
    else
      return search(korzen->prawy,kl,poprz);
  }
}
*/

/*wyszukiwanie nierekurencyjne*/

PtrWezel search(PtrWezel korzen,
                Tklucz kl, PtrWezel *poprz)
{
  *poprz = NULL;
```

```
  while ((korzen != NULL)
         &&
         (!rowne(korzen->klucz,kl))){
    *poprz = korzen;
    if (mniejsze(kl,korzen->klucz))
      korzen = korzen->lewy;
    else
      korzen = korzen->prawy;
  }
  return korzen;
}

void insert(PtrWezel *korzen,
            Tklucz kl, Tinfo nowainfo)
{
  PtrWezel p, q;

  if (search(*korzen,kl,&p) == NULL){
    q = malloc(sizeof(struct Wezel));
    skopiujklucz(q->klucz,kl);
    skopiujinfo(q->info,nowainfo);
    q->lewy = NULL;
    q->prawy = NULL;
    if (*korzen == NULL)
```

```
      *korzen = q;
    else if (mniejsze(kl,p->klucz))
      p->lewy = q;
    else
      p->prawy = q;
  }
}

void delete(PtrWezel *korzen, Tklucz kl)
{
  PtrWezel t, x, y, z;
  int b;

  x = search(*korzen,kl,&y);
  if (x != NULL)
  {
    if ((x->lewy == NULL)  || (x->prawy == NULL))
    {
      if ((x->lewy == NULL) && (x->prawy == NULL))
        z == NULL;
      else if (x->lewy == NULL)
        z = x->prawy;
      else
        z = x->lewy;
```

```
    if (y == NULL)
      *korzen = z;
    else if ( x == y->lewy )
      y->lewy = z;
    else
      y->prawy = z;
  }
  else{
    b = rand()%2;
    /* jesli b=0, to w miejsce  x
       wstawiamy  element
       z kluczem bezposrednio poprzedzajacy kl;
       jesli b=1, to w miejsce  x
       wstawiamy  element
       bezposrednio z kluczem nastepujacy po kl */
    if (b == 0){
      z = x->lewy;
      if (z->prawy == NULL)
        x->lewy = z->lewy;
      else{
        do{
          t = z;
          z = z->prawy;
        }while (z != NULL);
```

```
        t->prawy = z->lewy;
      }
    }
    else{
      z = x->prawy;
      if (z->lewy == NULL)
        x->prawy = z->prawy;
      else{
        do{
          t = z;
          z = z->lewy;
        }while (z != NULL);
        t->lewy = z->prawy;
      }
    }
    skopiujklucz(x->klucz,z->klucz);
    x = z;
  }
  free(x);
 }
}
```