Drzewa wyszukiwań binarnych (BST)

Krzysztof Grządziel

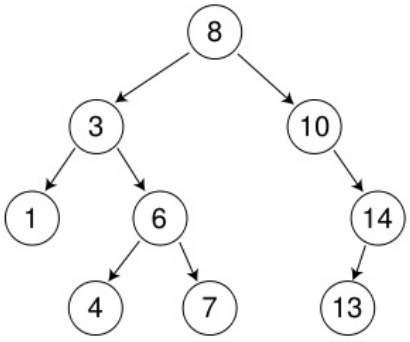
12 czerwca 2007 roku

1 Drzewa Binarne

Drzewa wyszukiwań binarnych, w skrócie BST (od ang. *binary search trees*), to szczególny przypadek drzew binarnych. Są to struktury, na których można wykonywać różne operacje właściwe dla zbiorów dynamicznych, takie jak **SEARCH**, **MINIMUM**, **MAXIMUM**, **PREDECESSOR**, **SUCCESSOR**, **INSERT**, **DELETE**. Drzewo wyszukiwań może więc być użyte zarówno jako słownik, jak i jako kolejka priorytetowa.

Podstawowe operacje na drzewach wyszukiwań binarnych wymagają czasu proporcjonalnego do wysokości drzewa. W pełnym drzewie binarnym o n węzłach takie operacje działają w przypadku pesymistyzcznym w czasie $\theta(lg\ n)$. Jeśli jednak drzewo składa się z jednej gałezi o długości n, to te same operacje wymagają w przypadku pesymistycznym czasu $\theta(n)$.

1.1 Co to jest drzewo wyszukiwań binarnych?



Rysunek 1: *Drzewo poszukiwań binarnych o wadze równej 9, a wysokości równej 3; wierzchołek '8' jest tu korzeniem, a wierzchołki '1', '4', '7' i '13', to liście.*

Drzewo wyszukiwań binarnych, ma strukturę drzewa binarnego. Takie drzewa można zrealizować za pomocą struktury danych z dowiązaniami, w której każdy węzeł jest obiektem. Oprócz pola *key* (ew. danych dodatkowych), każdy węzeł zawiera pola $left$, $right$, oraz p (parent), które wskazują, odpowiednio, na jego lewego syna, prawego syna, oraz rodzica. Jeśli węzeł nie ma następnika albo poprzednika, to odpowiednie pole ma wartość

NIL. Węzeł w korzeniu drzewa jest jedynym węzłem, którego pole wskazujące na ojca ma wartość NIL.

Klucze są przechowywane w drzewie BST w taki sposób, ale spełniona była **własność drzewa BST**:

> Niech x będzie węzłem drzewa BST. Jeśli y jest węzłem znajdującym się w lewym poddrzewie węzła x, to $key[y] \leqslant key[x]$. Jeśli y jest węzłem znajdującym się w prawym poddrzewie węzła x, to $key[y] \geqslant key[x]$.

1.2 Wybrane złożoności

Wypisanie drzewa	$\theta(n)$
Wyszukanie	$O(h)$
Znajdowanie minimum	$O(h)$
Znajdowanie maksimum	$O(h)$
Znajdowanie poprzednika	$O(h)$
Znajdowanie następnika	$O(h)$
Wstawianie	$O(h)$
Usuwanie	$O(h)$

gdzie n - ilość węzłów w drzewie, h - wysokość drzewa

2 Funkcje

2.1 echo

void echo(BST *root)
Wypisuje na standardowe wyjście *key* danego węzła.

Parametry:
root - wskaznik to korzenia

2.2 inorder

void inorder(BST *root)
Przechodzenie drzewa BST metoda *inorder*. Klucz drzewa zostaje wypisany między wartościami jego lewego poddrzewa, a wartościami z jego prawego poddrzewa.

Parametry:
root - wskaznik to korzenia

2.3 preorder

void preorder(BST *root)
Przechodzenie drzewa BST metoda *preorder*. Wypisuje klucz korzenia przed wypisaniem wartości znajdących się w obu poddrzewach.

Parametry:
root - wskaznik to korzenia

2.4 postorder

void postorder(BST *root)

Przechodzenie drzewa BST metoda *postorder*. Wypisuje klucz korzenia po wypisaniu wartości znajdujących się w obu poddrzewach.

Parametry:
root - wskaznik to korzenia

2.5 postorderinverse

void postorderinverse(BST *root)

Przechodzenie drzewa BST metoda odwrotną do *preorder*. Wypisuje klucze korzenia w kolejności odwrotnej do *preorder*.

Parametry:
root - wskaznik to korzenia

2.6 search

BST *search(BST *root, int val)

Wyszukiwanie węzła (rekurencyjnie), który zawiera klucz *val*.

Parametry:
root - wskaznik to korzenia val - szukany klucz

2.7 isearch

BST *isearch(BST *root,int val)

Iteracyjne wyszukiwanie węzła, który zawiera klucz *val*.
Uwaga!
Efektywniejsze od *serach(BST*,int)*;

Parametry:
root - wskaznik to korzenia val - szukany klucz

2.8 min

BST *min(BST *root)

Znajdowanie najmniejszej wartości w drzewie.

Parametry:
root - wskaznik to korzenia

2.9 max

BST *max(BST *root)
Znajdowanie największej wartości w drzewie.

Parametry:
root - wskaznik to korzenia

2.10 trace

void trace(BST *root, int v1, int v2)
Znajduje droge pomiedzy dwoma kluczami.

Parametry:
root - wskaznik to korzenia v1 - klucz początkowy v2 - klucz końcowy

2.11 draw

void draw(BST* root)
Rysuje drzewo.

Parametry:
root - wskaznik to korzenia

2.12 nastepnik

BST *nastepnik(BST *root)
Znajduje następnik danego węzła.

Parametry:
root - wskaznik to węzła

2.13 poprzednik

BST *poprzednik(BST *root)
Znajduje poprzednika danego węzła.

Parametry:
root - wskaznik to węzła

2.14 del

BST *del(BST *root, int val)
Usuwa węzeł o danym kluczu *val*.

Parametry:
root - wskaznik to korzenia val - wartość klucza

2.15 add

BST *add(BST *root, int val)
Dodaje węzeł o kluczu *val* do drzewa wyszukiwań binarnych.

Parametry:
root - wskaznik to korzenia val - wartość klucza

2.16 waga

int waga(BST *root)
Zwraca ilość węzłów w drzewie.

Parametry:
root - wskaznik to korzenia

3 Implementacja

3.1 Plik nagłówkowy

Listing 1: bst.h

```
#ifndef INC_BST_H
#define INC_BST_H

typedef struct drzewo BST;
struct drzewo {
        int key;
        BST *left;
        BST *right;
        BST *p; // parent
};

extern void echo(BST *root);
extern void inorder(BST *root);
extern void preorder(BST *root);
extern void postorder(BST *root);
extern void postorderinverse(BST *root);
extern BST *search(BST *root, int val);
extern BST *isearch(BST *root,int val);
extern BST *min(BST *root);
extern BST *max(BST *root);
extern void trace(BST *root, int v1, int v2);
extern void draw(BST* root);
extern BST *nastepnik(BST *root);
extern BST *poprzednik(BST *root);
extern BST *del(BST *root, int val);
extern BST *add(BST *root, int val);
extern int waga(BST *root);

#endif
```

3.2 Funkcje drzewa BST

Listing 2: Implementacja operacji na drzewie BST (bst.c)

```
#include<stdio.h>
#include<malloc.h>
#include"bst.h"

// wyswietla key z podanego wskaznika
void echo(BST *root){
```

```
7           if(root != NULL) printf("%d\n",root->key);
8           else printf("brak\n");
9           return ;
10  }
11
12  // wypisuje drzewo w porzadku inoder (posortowane rosnaco)
13  void inorder(BST *root){
14          if(root != NULL){
15                  inorder(root->left);
16                  printf("%d ",root->key);
17                  inorder(root->right);
18          }
19          return ;
20  }
21
22  // wypisywanie drzewa w porzadku preorder
23  void preorder(BST *root){
24          if(root != NULL){
25                  printf("%d ",root->key);
26                  preorder(root->left);
27                  preorder(root->right);
28          }
29          return ;
30  }
31
32  // wypisywanie drzewa w porzadku postorder
33  void postorder(BST *root){
34          if(root != NULL){
35                  postorder(root->left);
36                  postorder(root->right);
37                  printf("%d ",root->key);
38          }
39          return ;
40  }
41
42  // wypisywanie drzewa w porzadku odwrotnym do postorder
43  void postorderinverse(BST *root){
44          if(root != NULL){
45                  printf("%d ",root->key);
46                  postorderinverse(root->right);
47                  postorderinverse(root->left);
48          }
49          return ;
50  }
51
52  // szukanie rekurencyjne
53  BST *search(BST *root, int val){
54          if((root == NULL) || (val == root->key)) return root;
55          if(val < root->key){
56                  return search(root->left,val);
57          }
58          else {
59                  return search(root->right,val);
60          }
61          return root;
62  }
63
64  // przeszukiwanie iteracyjne (efektywniejsze)
65  BST *isearch(BST *root,int val){
66          while((root != NULL) && (val != root->key)){
67                  if(val < root->key) root = root->left;
68                  else root = root->right;
69          }
70          return root;
71  }
72
73  // znajdowanie minimum
74  BST *min(BST *root){
75          while(root->left != NULL){
76                  root = root->left;
77          }
78          return root;
79  }
```

```
80
81  // znajdowanie maksimum
82  BST *max(BST *root){
83          while(root->right != NULL){
84                  root = root->right;
85          }
86          return root;
87  }
88
89  //wskazywanie drogi pomiedzy dwoma elementami
90  void trace(BST *root, int v1, int v2){
91          BST* x = isearch(root,v1);
92          BST* y = isearch(root,v2);
93
94          if(v1 > v2){
95                  int tmp=v2;
96                  v2=v1;
97                  v1=tmp;
98          }
99          //printf("\tkey:%d\n",root->key);
100         if(root==NULL || x==NULL || y==NULL){
101                 printf("brak drogi");
102         }
103         else{
104                 if(v1<root->key && v2 < root->key){
105                         trace(root->left,v1,v2);
106                 }
107                 if(v1>root->key && v2 > root->key){
108                         trace(root->right,v1,v2);
109                 }
110                 if(v1<=root->key && v2 >= root->key){
111                         while(x != root){
112                                 printf("%d ",x->key);
113                                 x=x->p;
114                         }
115                         printf("%d ",x->key);
116                         while(root != y){
117                                 if(y->key < root->key){
118                                         root = root->left;
119                                         printf("%d ",root->key);
120                                 }
121                                 if(y->key > root->key){
122                                         root = root->right;
123                                         printf("%d ",root->key);
124                                 }
125                         }
126                 }
127         }
128 }
129
130 // rysuje drzewo
131 void draw(BST* root){
132         if(root != NULL){
133                 printf("%-3d", root->key);
134                 if(root->left == NULL && root->right == NULL);
135                 else{
136                         printf("(");
137                         if(root->left != NULL) printf("%-3d,", root->left->key); else printf(".
    ,");
138                         if(root->right != NULL) printf("%3d", root->right->key); else printf("
    .");
139                         printf(")");
140                 }
141                 printf("\n");
142         }
143         if(root->left !=NULL) draw(root->left);
144         if(root->right !=NULL) draw(root->right);
145 }
146
147 //successor
148 BST *nastepnik(BST *root){
149         BST* y = root;
150
```

```
151         //exception
152         if(root==NULL) return y;
153
154         if(root->right != NULL){
155                 return min(root->right);
156         }
157         y = root->p;
158         while(y != NULL && root == y->right){
159                 root = y;
160                 y = y->p;
161         }
162         return y;
163 }
164
165 //predecessor
166 BST *poprzednik(BST *root){
167         BST* y = root;
168
169         //exception
170         if(root==NULL) return y;
171
172         if(root->left != NULL){
173                 return max(root->left);
174         }
175         y = root->p;
176         while(y!=NULL && root == y->left){
177                 root = y;
178                 y = y->p;
179         }
180         return y;
181 }
182
183 //usuwanie
184 BST *del(BST *root, int val){
185         // wskazniki pomocnicze
186         BST* x = root;
187         BST* y = (BST*)malloc(sizeof(BST));
188         // to co usuwamy
189         BST* del = search(root,val);
190
191         if(del == NULL) return y;
192
193         if(del->left==NULL || del->right==NULL) y=del;
194         else y=nastepnik(del);
195
196         if(y->left!=NULL) x=y->left;
197         else x=y->right;
198
199         if(x!=NULL) x->p = y->p;
200
201         if(y->p == NULL) root = x;
202         else if(y == y->p->left) y->p->left = x;
203         else y->p->right = x;
204
205         if(y!=del){
206                 del->key = y->key;
207         }
208         return y;
209 }
210
211 // dodawanie
212 BST *add(BST *root, int val){
213         BST *x = root;
214
215         // nowy obiekt, ktory wpinany jest do drzewa
216         BST *nowe = (BST *)malloc(sizeof(BST));
217         nowe->key = val;
218         nowe->p = nowe->left = nowe->right = NULL;
219
220         BST *y = NULL;
221         while(x != NULL){
222                 y = x;
223                 if(val < x->key) x = x->left;
```

```
224                     else x = x->right;
225             }
226             nowe->p = y;
227             if(y == NULL) root = nowe;
228             else {
229                     if(nowe->key < y->key) y->left = nowe;
230                     else y->right = nowe;
231             }
232             return root;
233     }
234
235     // oblicza ilosc wezlow w drzewie
236     int waga(BST *root){
237             if(root == NULL) return 0;
238             else return waga(root->left) + waga(root->right) + 1;
239     }
```

3.3 Przykład użycia

Listing 3: Przykład użycia

```
1   #include<stdio.h>
2   #include<malloc.h>
3   #include"bst.h"
4
5   int main(){
6           BST *tree = NULL;
7
8           tree = add(tree,15);
9           tree = add(tree,6);
10          tree = add(tree,3);
11          tree = add(tree,7);
12          tree = add(tree,2);
13          tree = add(tree,4);
14          tree = add(tree,13);
15          tree = add(tree,9);
16          tree = add(tree,18);
17          tree = add(tree,17);
18          tree = add(tree,20);
19
20          printf("inorder(): ");
21          inorder(tree);
22          printf("\npostorder(): ");
23          postorder(tree);
24          printf("\npostorderinverse(): ");
25          postorderinverse(tree);
26          printf("\npreorder(): ");
27          preorder(tree);
28          printf("\nsearch(2): ");
29          search(tree,2);
30          printf("\nsearch(7): ");
31          search(tree,7);
32          printf("\nmin(7): ");
33          echo(min(tree));
34          printf("max(7): ");
35          echo(max(tree));
36          int nst=15;
37          printf("Nastepnik %d: ",nst);
38          echo(nastepnik(search(tree,nst)));
39          printf("Poprzednik %d: ",nst);
40          echo(poprzednik(search(tree,nst)));
41          del(tree,nst);
42          printf("Waga drzewa: %d\n",waga(tree));
43          printf("Droga(17,20): ");
44          trace(tree,17,20);
45          printf("\nDroga(20,17): ");
46          trace(tree,20,17);
47          printf("\nDroga(20,26): ");
48          trace(tree,20,26);
49          printf("\n");
50          return 0;
51  }
```

Literatura

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Wprowadzenie do algorytmów*, WNT, Warszawa 2005.