Лекция 15 Двоичные деревья поиска

15.1. Проблема: организовать хранилище данных, которое позволяет хранить большие объемы данных и предоставляет возможность быстро находить и модифицировать данные.

15.1.1. Хранилище данных обеспечивает пользователю *интерфейс*, в котором определены *словарные операции*: *search* (найти), *insert* (вставить) и *delete* (удалить). Одним из способов эффективной реализации описанного интерфейса является хеширование. Еще одним способом является использование деревьев поиска.

15.1.2. Помимо перечисленных операций деревья поиска позволяют эффективно реализовать и такие операции, как *min* (минимум), *max* (максимум), *pred* (предыдущий) и *succ* (следующий).

15.1.3. Время выполнения всех перечисленных операций пропорционально высоте дерева. Если дерево, содержащее n узлов, сбалансировано, его высота пропорциональна $\log n$, если дерево вытянуто в линейную цепочку, его высота пропорциональна n.

15.2. Двоичные деревья поиска

15.2.1. *Двоичное дерево* проще всего определить следующим образом: двоичное дерево это набор узлов, который либо пуст (пустое дерево), либо разбит на три непересекающиеся части: узел, называемый *корнем*, двоичное дерево, называемое *левым поддеревом*, и двоичное дерево, называемое *правым поддеревом*.

15.2.2. Узел двоичного дерева представляется следующей структурой:

```
struct BT_node {
   int key;
   struct BT_node *left;
   struct BT_node *right;
   struct BT_node *parent;
}
```

15.2.3. Ключи в двоичном дереве поиска хранятся с соблюдением свойства упорядоченности: Пусть x – произвольный узел двоичного дерева поиска. Если узел y принадлежит левому поддереву, то $key[y] \le key[x]$, если y находится в правом поддереве узла x, то $key[y] \ge key[x]$.

15.2.4. ***Пример***.

Дерево с ключами 2, 3, 5, 5, 7, 8; (а) корень с ключом 5; (б) корень с ключом 2. (В случае (а) высота дерева – 2, в случае (б) – 4).

15.3. ***Реализация операций*** из 15.1.1 и 15.1.2.

15.3.1. ***Поиск***. **На входе**: искомый ключ k и указатель *root* на корень поддерева, в котором производится поиск. **На выходе**: указатель на узел с ключом *key* (если такой узел есть), либо пустой указатель NULL.

```
struct BT_node *BTsearch(struct BT_node *root, int k) {
   if(root == NULL || k == root -> key) return root;
   if(k < root -> key) return BTsearch(root -> left, k);
   else return BTsearch(root -> right, k);
}
```

Итеративная версия поиска.

```
struct BT_node *BTsearch(struct BT_node *root, int k) {
   struct BT_node *p;
   p = root;
   while(p != NULL && k != p -> key)
      if(k < p -> key) p = p -> left;
      else p = p -> right;
   return p;
}
```

Среднее время поиска $O(h)$, где h – высота дерева.

15.3.2. ***Минимум и максимум***.

(1) **На входе**: указатель *root* на корень поддерева. **На выходе**: указатель на узел с минимальным ключом *k*.

```
struct BT_node *BTmin(struct BT_node *root) {
   struct BT_node *p;
   p = root;
   while(p -> left != NULL)
      p = p -> left;
   return p;
}
```

(2) **На входе**: указатель *root* на корень поддерева. **На выходе**: указатель на узел с максимальным ключом *k*.

(3) Среднее время выполнения функций **BTmin** и **BTmax** есть $O(h)$, где h – высота дерева *root*.

15.3.3. ***Следующий и предыдущий***.

(1) **На входе**: указатель *node* на узел дерева. **На выходе**: указатель на следующий за *node* узел дерева.

```
struct BT_node *BTsucc(struct BT_node *node) {
   struct BT_node *p, *q;
   p = node;
   /* I случай: правое поддерево узла не пусто */
   if(p -> right != NULL) return BTmin(p -> right);
   /* II случай: правое поддерево узла пусто */
   q = p -> parent;
   while(q != NULL && p == q -> right) {
      p = q;
      q = q -> parent;
   }
   return q;
}
```

Во II случае идем по родителям до тех пор, пока не найдем родителя, для которого наше поддерево левое; этот родитель и будет следующим за *node* узом дерева.

(2) **На входе**: указатель *node* на узел дерева. **На выходе**: указатель на предшествующий *node* узел дерева.

```
struct BT_node *BTpred (struct BT_node *node) {
```

(3) Среднее время выполнения функций **`BTsucc`** и **`BTpred`** есть $O(h)$, где h – высота дерева *root*.

15.3.4. ***Добавление узла***.

На входе: указатель *root* на корень дерева и указатель *node* на новый узел, у которого есть значение ключа, а все поля с указателями имеют значение *NULL*. Функция **`BTinsert`** вставляет узел с указателем *node* в дерево с корнем *root*, сохраняя упорядоченность ключей.

```
void BTinsert(struct BT_node *root, struct BT_node *node) {
   struct BT_node *p, *q, *s;
   p = root;
   q = NULL;
   while(p != NULL) {
      q = p; /*чтобы не испортить последний элемент*/
      if(node -> key < p -> key) p = p -> left;
      else p = p -> right;
   }
   /*подвешиваем node к q */
   node -> parent = q;
   /*подвешиваем node к корню, либо к q слева или справа*/
   if(q == NULL) root = node;
      else if(node -> key < q -> key) q -> left = node;
      else q -> right = node;
```

Среднее время выполнения функции **`BTinsert`** есть $O(h)$, где h – высота дерева *root*.

15.3.5. ***Удаление узла***.

На входе: указатель *root* на корень дерева и указатель *node* на один из его узлов. Функция **`BTdelete`** удаляет узел с указателем *node* из дерева с корнем *root*, сохраняя упорядоченность ключей.

Необходимо рассмотреть три случая: (1) у удаляемого узла нет детей; (2) у удаляемого узла только один ребенок; (3) у удаляемого узла два ребенка.

```
void BTdelete (struct BT_node *root, struct BT_node *node) {
```