**Практическая работа BST деревья часть 2**

**Теоретическая часть**

**Введение**

Одной из разновидностей двоичных деревьев являются бинарным поисковым деревьям.

Мы рассмотрим теоретические алгоритмы для выполнения различных операций: вставка и удаление узла, поиск максимума/минимума и (примеры практических реализации данных алгоритмов на языке СИ) Вам нужно их переписать на языке Джава

**Определение. Бинарные поисковые деревья**

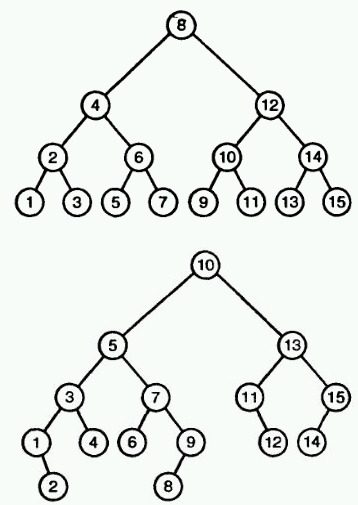
***Бинарные поисковые деревья*** (*англ.* ***binary search tree*** *или сокращенно BST)* — это разновидность двоичного дерева, обладающая следующими отличительными свойствами:

1. для каждого узла **X** (если **X** не равен **NULL**) все узлы в его левом поддереве содержат значения, которые ВСЕГДА меньше значения узла **X**.
2. все узлы в правом поддереве узла **Х** содержат значения, которые соответственно ВСЕГДА больше значения узла **X**.
3. каждый узел в таком дереве является родителем для своих поддеревьев.

Преимущество двоичного поискового дерева в том, что все значения располагаются в нем в отсортированном порядке. В общем случае узлы необязательно должны подвергаться сортировке перед вставкой в бинарное поисковое дерево. Узлы могут вставляться в такое дерево в произвольном порядке, а сама сортировка может происходить в процессе вставки.

Результат вставки узлов в бинарное поисковое дерево может оказаться непредсказуемым. Если вставлять в такое дерево узлы, которые уже были предварительно отсортированы, то это приведет к вырождению дерева. На рисунке 1 показаны две одинаковые по содержанию реализации поискового дерева, но с различными корневыми узлами.

**Рисунок 1. Два варианта представления бинарного поискового дерева**



В листинге 1 приведено объявление структур для узла бинарного дерева и самого дерева на языке Си.

**Листинг 1. Определение структур для представления дерева и его узлов**

struct node

{

int data;

struct node \* left;

struct node \* right;

};

struct tree

{

struct node \* root; // указатель на корень дерева

int count; // количество узлов в дереве

};

Структура, использующаяся для описания узла дерева, полностью совпадает со структурой, использующейся для описания связного списка. Используя эти две структуры, можно подготовить функцию на языке Си для создания дерева (см. листинг 2).

**Листинг 2. Функция для создания дерева**

struct tree \* tree\_create(void)

{

struct tree \* new\_tree = malloc(sizeof \* new\_tree);

if (new\_tree == NULL) return NULL;

new\_tree->root = NULL;

new\_tree->count = 0;

return new\_tree;

}

**Вставка узла в бинарное поисковое дерево**

Перед вставкой нового элемента в бинарное дерево обязательно нужно проверить, нет ли в нем уже такого элемента. Для этого необходимо начать обход дерева с корневого узла и проверить, не превосходит ли значение корневого узла добавляемого значения. Если корневой узел больше добавляемого элемента, то необходимо переместиться в левое дочернее дерево. В противном случае - в правое. Функция поиска определенного узла в дереве, приведенная в листинге 3, вернет **1** в случае, если искомое значение будет найдено, или **0**, если указанное значение в дереве отсутствует.

**Листинг 3. Функция для поиска узла в дереве**

int bin\_search(const struct tree \* search\_tree, int item)

{

const struct node \* search\_node;

search\_node = search\_tree->root;

for(;;)

{

if (search\_node == NULL) return 0;

else if (item == search\_node->data) return 1;

else if (item > search\_node->data) search\_node = search\_node->right;

else search\_node = search\_node->left;

}

}

После выполнения данной проверки можно переходить к добавлению узла в бинарное дерево. Функция вставки аналогична функции поиска: необходимо пройти по дереву и вставить узел в нужное место. Для этого следует выбрать левое или правое поддерево корневого узла, а затем рекурсивно перемещаться по выбранному поддереву до тех пор, пока не будет найдена позиция для вставки узла, как показано в листинге 4.

**Листинг 4. Функция для вставки узла в бинарное поисковое дерево**

int insert(struct tree \* search\_tree, int item)

{

struct node \* search\_node, \*\*new;

new = &search\_tree->root;

search\_node = search\_tree->root;

for(;;)

{

if(search\_node == NULL)

{

search\_node = \*new = malloc(sizeof \* search\_node);

if(search\_node != NULL)

{

search\_node->data = item;

search\_node->left = search\_node->right=NULL;

search\_tree->count++;

return 1;

}

else return 0;

}

else if(item == search\_node->data) return 2;

else if(item > search\_node->data)

{

new = &search\_node->right;

search\_node = search\_node->right;

}

else

{

new = &search\_node->left;

search\_node = search\_node->left;

}

}

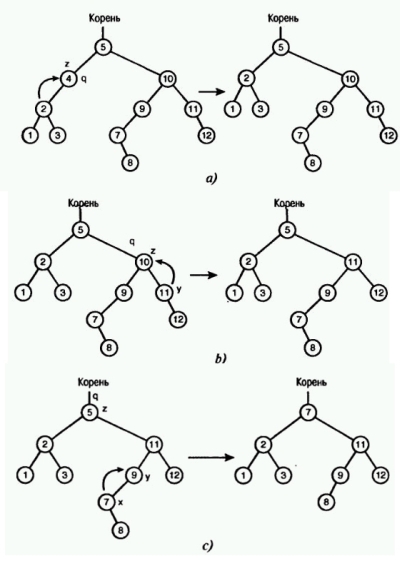
}

Элемент, добавленный первым, автоматически становится корнем дерева. Функция может вернуть одно из 3-х значений: **0** (при вставке произошла ошибка), **1** (узел успешно вставлен), **2** (такой элемент уже есть в дереве).

**Удаление узла из бинарного поискового дерева**

Функция удаления узла из бинарного дерева также будет возвращать **0**, если возникла ошибка, или **1** в случае удачного удаления элемента. В данной функции используется уже знакомый алгоритм поиска, который будет искать элемент, помеченный для удаления. После того, как этот элемент будет обнаружен, возможны 3 варианта действий, которые представлены на рисунке 2.

**Рисунок 2. Три варианта удаления элемента из бинарного поискового дерева**



В первом случае (вариант **a**) узел **z** имеет только левый дочерний узел, и при удалении он и будет заменен им. Во втором варианте (рисунок **b**) узел **z** заменяется узлом **y**.

В третьем варианте (рисунок **с**) корневой узел **z** заменяется узлом **x**, что приводит к дополнительным перестроениям в правом поддереве. Узел **x** становится преемником узла **z**, так как в левом поддереве узла **z** все узлы меньше 5, и преемника нужно искать в правом поддереве узла **z**.

**Листинг 5. Функция для удаления узла из бинарного поискового дерева**

int delete(struct tree \* search\_tree, int \*\* item)

{

struct node \*\* q,\*z;

q=&search\_tree->root;

z=search\_tree->root;

//поиск удаляемого элемента

for(;;)

{

if(z == NULL) return 0;

else if(item == (int \*\*)z->data) break;

else if(item > (int \*\*)z->data)

{

q = &z->right;

z = z->right;

}

else

{

q = &z->left;

z = z->left;

}

}

// непосредственное удаление элемента

if(z->right == NULL) \*q = z->left;

else

{

struct node \* y = z->right;

if(y->left == NULL)

{

y->left = z->left;

\*q-y;

}

else

{

struct node \* x=y->left;

while(x->left != NULL)

{

y = x;

x = y->left;

}

y->left = x->right;

x->left = z->left;

x->right = z->right;

\*q=x;

}

}

search\_tree->count --;

free(z);

return 1;

}

**Проход по дереву**

Для прохода по дереву можно использовать рекурсию, так как любое двоичное дерево - это рекурсивная структура. Функции, приведенные в листинге 6, могут распечатать дерево, начиная с любого узла: сначала печатается левое поддерево, потом правое.

**Листинг 6. Функции для вывода содержимого дерева**

//функция для распечатки фрагмента дерева - с любого узла

static void walk(const struct node \* search\_node)

{

if(node == NDLL) return;

walk(node->left);

printf("%d ", search\_node->data);

walk(node->right);

}

//функция для распечатки дерева целиком - с корня

void walk2(const struct tree \* my\_tree)

{

walk(my\_tree->root);

}

Для прохода по дереву можно обойтись и одной функцией, если ввести массив указателей на узлы, который будет заполняться во время прохода. В функции будет выполняться вложенный цикл для прохода дерева от текущего узла влево до конца. А дополнительный массив поможет обойти все дерево за один проход, так как в нем будет храниться указатель на текущий корневой узел.

**Листинг 8. Нерекурсивная реализация обхода дерева**

void traverse(const struct tree \* search\_tree)

{

struct node \* stack[32];

int count;

struct node \* search\_node;

count = 0;

search\_node = search\_tree->root;

for(;;)

{

while(search\_node != NOLL)

{

stack[count++] = search\_node;

search\_node = search\_node->left;

}

if(count == 0) return ;

search\_node = stack[-count];

printf("%d",search\_node->data);

search\_node = search\_node->right;

}

}

В листинге 8 присутствует намеренное упрощение, так как в нем используется упрощенный статический вариант стека, хотя в реальной ситуации память под стек обязательно должна выделяться динамически.

**Удаление дерева**

В этом разделе приведены функции для полного удаления дерева, начиная с листьев и заканчивая корневым узлом.

**Листинг 9. Функции для удаления дерева**

//функция для удаления отдельного узла дерева и его потомков

static void destroy(struct node \* search\_node)

{

if(search\_node == NOLL) return;

destroy(search\_node->left);

destroy(search\_node->right);

free(search\_node);

}

//функция для полного удаления дерева

void destroy(struct tree \* search\_tree)

{

destroy(search\_tree->root);

free(search\_tree);

}

**Нахождение минимума и максимума в бинарном поисковом дереве**

В листинге 10 приведены две функции для поиска в дереве узлов с минимальным и максимальным значением. Минимальное значение нужно искать в левом поддереве корневого элемента, а максимальное - в правом.

**Листинг 10. Функции для поиска минимального и максимальных значений**

struct node \* tree\_minimum(struct tree \* search\_tree)

{

struct node \* search\_node;

search\_node = search\_tree->root;

while (search\_node->left != NULL)

search\_node = search\_node->left;

return search\_node;

}

struct node \* tree\_maximum(struct tree \* search\_tree)

{

struct node \* search\_node;

search\_node = search\_tree->root;

while (search\_node->right != NULL)

search\_node = search\_node->right;

return search\_node;

}

**Определение типа двоичного дерева**

Чтобы определить относится ли данное двоичное дерево к поисковым бинарным деревьям, можно использовать функцию из листинга 11.

**Листинг 11. Определение типа двоичного дерева**

int isBST(struct node\* node)

{

return(isBSTUtil(node, INT\_MIN, INT\_MAX));

}

int isBSTUtil(struct node\* node, int min, int max)

{

if (node==NULL) return(true);

if (node->data<min || node->data>max) return(false);

return (

isBSTUtil(node->left, min, node->data) &&

isBSTUtil(node->right, node->data+1, max)

);

}

**Практическая реализация алгоритмов**

1. Сначала создать бинарное поисковое действие из 10 узлов, затем содержимое дерева распечатать
2. из него удалить один узел.
3. После этого выполняется еще одна распечатка, чтобы проверить, как прошло удаление
4. дерево удаляется уже полностью с освобождением памяти.

**Заключение**

* Мы рассмотрели основные алгоритмы, использующиеся для работы с одной из самых популярных структур данных – бинарным поисковым деревом.
* Кроме теоретических описаний были представлены и практические реализации данных алгоритмов на языке программирования Си.
* Для лучшего усвоения материала необходимо самостоятельно разработать реализацию этих алгоритмов на языке Java и решить ответить на следующие вопросы:

1. если вставить в бинарное поисковое дерево числа в диапазоне от 1 до 64 в порядке возрастания таким образом, что сначала вставляются нечетные числа, потом четные, то какова будет высота этого дерева
2. если сначала вставить 3 числа: 32, 16, 48, а потом все остальные в прежнем порядке, то какова будет высота этого дерева?