**Анализ двоичного дерева поиска**

void straight(tree\* tr) //прямой

{

if (NULL == tr) return; //условие если дерево пустое, выходим

{

cout << tr->inf << " ";//заносим дерево в переменную

straight(tr->left);//кладем личинку в левую ветку

straight(tr->right);//кладем в правую

}

}

void back(tree\* tr)//обратный

{

if (NULL == tr) return;//если дерево пустое - выходим

{

back(tr->left);//кладем в левый узел

back(tr->right);//кладем в правый узел

cout << tr->inf << " ";//выводим дерево

}

}

void inorder(tree\* tr)//симметричный обход

{

if (tr)

{

inorder(tr->left);

cout << tr->inf << " ";

inorder(tr->right);

}

}

Рассмотрим сначала всевозможные обходы. Данные алгоритмы выполняют обход за время O(n), поскольку функции вызываются ровно два раза для каждого узла дерева.

void insert(tree\*& tr, int x) //вставка

{

tree\* n = node(x);

if (!tr) tr = n; //если дерево пустое - корень

else {

tree\* y = tr;

while (y) //ищем куда вставлять

{

if (n->inf >= y->inf) //правая ветка

if (y->right)

y = y->right;

else

{

n->parent = y; //узел становится правым ребенком

y->right = n;

break;

}

else if (n->inf < y->inf)//левая ветка

if (y->left)

y = y->left;

else

{

n->parent = y;//узел становится левым ребенком

y->left = n;

break;

}

}

}

}

Вставка будет осуществляться за О(h), где h-высота дерева, т.к в худшем случае потребуется пройти h вершин, двигаясь вниз. Сама операция создания узла выполняется за константное время.

void Delete(tree\*& tr, tree\* v)//удаление узла

{

tree\* p = v->parent;

if (tr->left && tr->right && !tr->parent) { // Если есть левый и правый, но нет родителя / (Корень)

tree\* L = tr->left; // Создаем дубликат левой ветки

tree\* R = tr->right; // И правой

tr->inf = R->inf; // Копируем информацию правой ветки как основную

tr->left = R->left;

tr->right = R->right;

supdel(L, tr); // И на нее цепляем значение

}

else if (!tr->left && tr->right && !tr->parent) { // Если нет левого и родителя, но есть правый / (Корень)

tree\* R = tr->right;

tr->inf = R->inf;

tr->left = NULL;

tr->right = R->right;

}

else if (tr->left && !tr->right && !tr->parent) { // Если нет правого и родителя, но есть левый / (Корень)

tree\* L = tr->left;

tr->inf = L->inf;

tr->left = L->left;

tr->right = NULL;

}

else if (!v->left && !v->right) //если нет детей

{

if (p->left == v) //указатель у родителя меняем на NULL

p->left = NULL;

if (p->right == v)

p->right = NULL;

delete v;

}

else if (!v->left || !v->right)//если только один ребенок

{

if (!p)//если удаляем корень, у которого 1 ребенок

{

if (!v->left)//если есть правый ребенок

{

tr = v->right; //он становится корнем

v->parent = NULL;

}

else //аналогично для левого

{

tr = v->left;

v->parent = NULL;

}

}

else

{

if (!v->left) //если есть правый ребенок

{

if (p->left == v) //если удаляемый узел явл. левым ребенком

p->left = v->right; //ребенок удаляемого узла становится левым ребенком

else

p->right = v->right; ////ребенок удаляемого узла становится правым

v->right->parent = p; //родителем ребенка становится его "дед"

}

else //аналогично для левого ребенка

{

if (p->left == v)

p->left = v->left;

else

p->right = v->left;

v->left->parent = p;

}

delete v;

}

}

else //есть оба ребенка

{

tree\* succ = Next(tr, v->inf);//следующий за удаляемым узлом

v->inf = succ->inf; //присваиваем значение

if (succ->parent->left == succ)//если succ левый ребенок

{

succ->parent->left = succ->right; //его правый ребенок становится левым

if (succ->right) //если этот ребенок существует

succ->right->parent = succ->parent; //его родителем становится "дед"

}

else //аналогично если succ - правsq ребенок

{

succ->parent->right = succ->right;

if (succ->right)

succ->right->parent = succ->parent;

}

delete succ;

}

}

Рассмотрим операцию удаления. Операции присваивания ссылок, проверка условий нас интересовать не будут, т.к они будут выполняться за константное время. В худшем случае, если элемент находится в самом низу дерева, то потребуется пройти h вершин, где h- высота дерева. Значит время удаления из бинарного дерева будет составлять O(h).

tree\* find(tree\* tr, int x)//поиск

{

if (!tr || x == tr->inf)//нашли или дошли до конца ветки

return tr;

if (x < tr->inf)

return find(tr->left, x);//ищем по левой ветке

else

return find(tr->right, x);//ищем по правой ветке

}

Поиск элемента осуществляется на основе сравнения с другими элементами. Операция поска происходит от вершины левой ветки и правой за O(n) время.