Задача C. Простое двоичное дерево поиска

```
struct node {
   int value;
   node* l;
   node* r;
}root;
```

Сначала я создам root для хранения NODE двоичного дерева поиска

```
node* insert(node *root, int data){
   if(root == NULL) {
      root = new node();
      root->value = data;
      root->l = root->r = NULL;
   }
   else if(data <= root->value)
      root->l = insert(root->l, data);
   else
      root->r = insert(root->r, data);
   return root;
```

1. Если root равно NULL, мы добавляем data в root

2. Если data <= root->value, то мы будем использовать рекурсию, чтобы добавить левое поддерево

3. Если data > root->value, то мы будем использовать рекурсию, чтобы добавить правильное поддерево

```
bool search(node* root, int data){
   if(root == NULL) {
      return false;
   }
   else if(root->value == data) {
      return true;
   }
   else if(data <root->value) {
      return search(root->l, data);
   }
   else {
      return search(root->r, data);
   }
```

1. Если root равно NULL, можно значит, что в двоичном дереве нет элементов для поиска.

2. Если root->value == data можно значит, что есть data в двоичном дереве

3. Если root->value < data, то мы будем использовать рекурсию, чтобы поиск в левом поддереве

4. Если root->value < data, то мы будем использовать рекурсию, чтобы поиск в правильном поддереве

```
struct node* delete_node(struct node *root, int data){
    if(root == NULL) return root;
    else if(data < root->value) root->l = delete_node(root->l, data);
    else if (data > root->value) root->r = delete_node(root->r, data);
    else {
       if(root->l == NULL && root->r == NULL) {
          delete root;
          root = NULL;
       }
       else if(root->l == NULL) {
          struct node *temp = root;
          root = root->r;
          delete temp;
       }
       else if(root->r == NULL) {
          struct node *tem = root;
          root = root->l;
          delete tem;
       }
       else {
          struct node *temp = find_min(root->r);
          root->value = temp->value;
          root->r = delete_node(root->r, temp->value);
       }
    }
    return root;
}
```

1. Если root равно NULL -> ничего не делать

2. Если data < root->value, то мы будем использовать рекурсию, чтобы удалить в левом поддереве

3. Если data > root->value, то мы будем использовать рекурсию, чтобы удалить в правильном поддереве

4. Если data = root->value:

4.1 Если удаляемая позиция не имеет левого и правого поддерева, то мы удалим

4.2 Если удаляемая позиция имеет правого поддерева или левого поддерева, то мы свяжем parents of root с поддеревом и удалим root

4.3 Если удаляемая позиция имеет левого и правого поддерева, то мы найдем наименьшее node из правого поддерева root Затем мы присваиваем к root и мы используем рекурсию для удаления наименьшее node из правого поддерева root

```
void find_Prede_Succeccor(node* root, node*& pre, node*& suc, int value){
   if (root == NULL)
      return;
   if (root->value == value){
      if (root->l != NULL){
         node* tmp = root->l;
         while (tmp->r)
            tmp = tmp->r;
         pre = tmp ;
      }

      if (root->r != NULL){
         node* tmp = root->r ;
         while (tmp->l)
            tmp = tmp->l ;
         suc = tmp ;
      }
      return;
   }

   if (root->value > value){
      suc = root ;
      find_Prede_Succeccor(root->l, pre, suc, value) ;
   }
   else{
      pre = root ;
      find_Prede_Succeccor(root->r, pre, suc, value) ;
   }
}
```