Все три варианта вертикального обхода элементарно реализуются рекурсивными функциями.

```
void recPreOrder(){
    treatment();
    if (left!=null) left.recPreOrder();
    if (right!=null) right.recPreOrder();
}
void recInOrder(){
    if (left!=null) left.recInOrder();
    treatment();
    if (right!=null) right.recInOrder();
}
void recPostOrder(){
    if (left!=null) left.recPostOrder();
    if (right!=null) right.recPostOrder();
    treatment();
}
```

Рекурсия крайне удобна не только при обходе, но также при построении дерева, поиска в дереве, а также балансировки. Однако рекурсией нельзя осуществить горизонтальный обход дерева. В этом случае, а так же при обеспокоенности перегрузкой программного стека, следует применять итерационный подход.

В случае использования итераций необходимо хранить сведенья о посещенных, но не обработанных узлах. Используются контейнеры типа стек (для вертикального обхода) и очередь (для горизонтального обхода).

Горизонтальный обход: обрабатываем первый в очереди узел, при наличии дочерних узлов заносим их в конец очереди. Переходим к следующей итерации.

```
static void contLevelOrder(Node top){
    Queue<Node> queue=new LinkedList<> ();
    do{
        top.treatment();
        if (top.left!=null) queue.add(top.left);
        if (top.right!=null) queue.add(top.right);
        if (!queue.isEmpty()) top=queue.poll();
    }while (!queue.isEmpty());
}
```

Вертикальный прямой обход:
обрабатываем текущий узел, при наличии правого поддерева добавляем его в стек для последующей обработки. Переходим к узлу левого поддерева. Если левого узла нет, переходим к верхнему узлу из стека.

```
static void contPreOrder(Node top){
     Stack<Node> stack = new Stack<> ();
     while (top!=null || !stack.empty()){
         if (!stack.empty()){
             top=stack.pop();
```

```
        }
        while (top!=null){
            top.treatment();
            if (top.right!=null) stack.push(top.right);
            top=top.left;
        }
    }
}
```

Вертикальный обратный обход: из текущего узла «спускаемся» до самого нижнего левого узла, добавляя в стек все посещенные узлы. Обрабатываем верхний узел из стека. Если в текущем узле имеется правое поддерево, начинаем следующую итерацию с правого узла. Если правого узла нет, пропускаем шаг со спуском и переходим к обработке следующего узла из стека.

```
static void contInOrder(Node top){
    Stack<Node> stack = new Stack<> ();
    while (top!=null || !stack.empty()){
        if (!stack.empty()){
            top=stack.pop();
            top.treatment();
            if (top.right!=null) top=top.right;
            else top=null;
        }
        while (top!=null){
            stack.push(top);
            top=top.left;
        }
    }
}
```

Вертикальный концевой обход:
Здесь ситуация усложняется – в отличие от обратного обхода, помимо порядка спуска нужно знать обработано ли уже правое поддерево. Одним из вариантов решения является внесение в каждый экземпляр узла флага, который бы хранил соответствующую информацию (не рассматривается). Другим подходом является «кодирование» непосредственно в очередности стека — при спуске, если у очередного узла позже нужно будет обработать еще правое поддерево, в стек вносится последовательность «родитель, правый узел, родитель». Таким образом, при обработке узлов из стека мы сможем определить, нужно ли нам обрабатывать правое поддерево.

```
static void contPostOrder(Node top){
    Stack<Node> stack = new Stack<> ();
    while (top!=null || !stack.empty()){
        if (!stack.empty()){
            top=stack.pop();
            if (!stack.empty() && top.right==stack.lastElement()){
                top=stack.pop();
            }else{
```

```
                top.treatment();
                top=null;
            }
        }
        while (top!=null){
            stack.push(top);
            if (top.right!=null){
                stack.push(top.right);
                stack.push(top);
            }
            top=top.left;
        }
    }
}
```

Об указателе на родителя

Наличие в экземпляре класса указателя на родителя приносит определенные хлопоты при построении и балансировки деревьев. Однако, возможность из произвольного узла дерева «дойти» до любого из его узлов может придтись весьма кстати. Все, за чем нужно следить при «подъеме» на верхний уровень – пришли ли от правого потомка или от левого.

Так, с использованием родительских указателей будет выглядеть код вертикального концевого обхода.

```
static void parentPostOrder(Node top){
    boolean fromright=false;
    Node shuttle=top, holder;
    while(true){
        while (fromright){
            shuttle.treatment();
            if (shuttle==top) return;
            holder=shuttle;
            shuttle=shuttle.parent;
            fromright=shuttle.right==holder;
            if (!fromright && shuttle.right!=null) shuttle=shuttle.right;
            else fromright=true;
        }
        while (shuttle.left!=null) shuttle=shuttle.left;
        if (shuttle.right!=null) shuttle=shuttle.right;
        else fromright=true;
    }
}
```

Другой класс задач, которые позволяет решить родительский указатель, как уже было упомянуто — перемещение внутри дерева.

Так, что бы перейти на n-ый по счету узел от текущего узла, без «ориентации в дереве» пришлось бы обходить дерево с самого начала, до известного узла, а потом еще n-узлов. С использованием же родительского указателя при обратном обходе дерева

перемещение на steps узлов от текущего узла (start) будет иметь следующий вид.

```
  public static Node walkTheTree(Node start, int steps){
     boolean fromright=true;
     Node shuttle=start, holder;
     if (shuttle.right!=null){
        shuttle=shuttle.right;
        while (shuttle.left!=null) shuttle=shuttle.left;
        fromright=false;
     }
     int counter=0;
     do{
        while (true){
           if (!fromright && ++counter==steps) return shuttle;
           if (!fromright && shuttle.right!=null){
                 shuttle=shuttle.right;
                 break;
              }
           holder=shuttle;
           shuttle=shuttle.parent;
           fromright=(holder==shuttle.right);
        }
        while (shuttle.left!=null) shuttle=shuttle.left;
     }while (true);
  }
```

Поиск в глубину - это тип обхода, который максимально углубляется в каждого дочернего элемента, прежде чем исследовать следующего родственного.

Существует несколько способов выполнить углубленный поиск: по порядку, перед заказом и после заказа.

Обход по порядку состоит из посещения сначала левого поддерева, затем корневого узла и, наконец, правого поддерева:

```
public void traverseInOrder(Node node) {
   if (node != null) {
      traverseInOrder(node.left);
      System.out.print(" " + node.value);
      traverseInOrder(node.right);
   }
}
```

Если мы вызовем этот метод, в выводе консоли будет показан обход по порядку:

3 4 5 6 7 8 9

Обход по предварительному заказу посещает сначала корневой узел, затем левое поддерево и, наконец, правое поддерево:

```
public void traversePreOrder(Node node) {
   if (node != null) {
      System.out.print(" " + node.value);
```

```
        traversePreOrder(node.left);
        traversePreOrder(node.right);
    }
}
```

Давайте проверим обход предварительного заказа в выводе консоли:

6 4 3 5 8 7 9

При обходе после заказа посещаются левое поддерево, правое поддерево и корневой узел в конце:

```
public void traversePostOrder(Node node) {
    if (node != null) {
        traversePostOrder(node.left);
        traversePostOrder(node.right);
        System.out.print(" " + node.value);
    }
}
```

Вот узлы в последующем порядке:

3 5 4 7 9 8 6

Поиск по ширине
Это еще один распространенный тип обхода, который **посещает все узлы уровня, прежде чем перейти к следующему уровню**.

Этот вид обхода также называется level-order и посещает все уровни дерева, начиная с корневого и слева направо.

Для реализации мы будем использовать *очередь* для упорядоченного хранения узлов каждого уровня. Мы извлекем каждый узел из списка, выведем его значения, затем добавим его дочерние элементы в очередь:

```
public void traverseLevelOrder() {
    if (root == null) {
        return;
    }

    Queue<Node> nodes = new LinkedList<>();
    nodes.add(root);

    while (!nodes.isEmpty()) {

        Node node = nodes.remove();

        System.out.print(" " + node.value);

        if (node.left != null) {
            nodes.add(node.left);
        }
```

```
        if (node.right != null) {
            nodes.add(node.right);
        }
    }
}
```