Обход двоичного дерева поиска принято осуществлять в глубину, т.к. это позволяет посещать вершины в определённом порядке, зависящим от правила построения дерева. Проблема в том, что задание требует итеративный обход, который заметно сложнее рекурсивного в плане реализации. Любой итеративный обход дерева будет требовать использования дополнительных данных. Таких, как список пройденных вершин или двоичную карту обхода.

Моя идея состоит в том, чтобы обходить дерево в ширину и при этом строить упорядоченный список посещенных вершин. Вершины в список будут записываться по требуемым правилам, т.е. обход списка после его построения, будет равносилен обходу дерева по требуемому правилу. Итак, суть: выгрузка дерева в список при обходе в ширину (используя итерации).

Список состоит из элементов типа element\_t. Каждый элемент содержит указатель на узел дерева, к которому он относится и глубину этого узла в дереве. Глубина служит для идентификации тех узлов списка, которые должны посещаться на очередной итерации цикла (каждая итерация внешнего цикла соответствует обходу одного уровня списка).

element\_t = **record**

pNode : nodePtr; {указатель на узел, которому принадлежит}

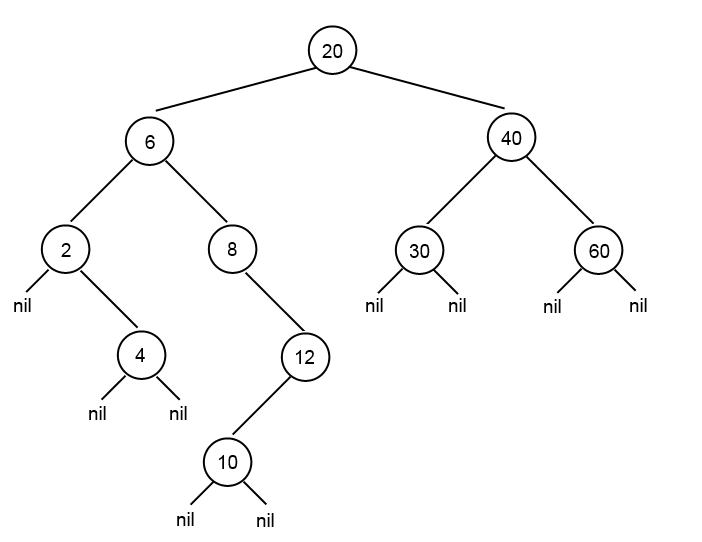
depth : integer; {глубина этого узла в дереве}

pPrev, pNext : elPtr; {список двусвязный}

**end**;  
  
Алгоритм (р3):  
в переменной currDepth содержится тот уровень дерева, который должен быть проверен на очередной итерации.

* На каждой итерации внешнего цикла:
  + вычисляем currDepth (просто наращивая значение на 1).
  + проходим по списку посещенных узлов и ищем те узлы, которые находятся на текущем уровне (currDepth). Для каждого такого узла:
    - добавляем всех его потомков (если они есть) в список обхода на соответствующие им места (с указанием глубины currDepth + 1, что гарантирует их посещение на следующем проходе основного цикла).

Важная деталь заключается в том, как проходит вставка элементов в список. Дело в том, что задание подразумевает обход по правилу левое-корень-правое. Т.е. элементы в список должны входить в том же прядке. Левый потомок встает в список слева от родителя, а правый – справа. Это гарантирует упорядоченность получаемого списка.

Для приведенных входных данных (20, 6, 8, 2, 4, 12, 10, 40, 30, 60) дерево будет выглядеть следующим образом. 

Функция getPassList осуществляет обход дерева по указанному алгоритму. В ходе этого строится обходной список, указатель на который как раз и возвращает эта функция.

Обходной список строится следующим образом:

1)создаются 2 пустых элемента списка (головной и хвостовой). Это сделано для того, чтобы при вставке элементов в список не приходилось отдельно обрабатывать случаи, когда вставляемый элемент находится на границе списка. Список сразу после создания:

|  |  |
| --- | --- |
| Глубина 0 | Глубина 0 |
| Указатель на - | Указатель на - |

2)в список заносится первый элемент (корень)

|  |  |  |
| --- | --- | --- |
| Глубина 0 | Глубина 0 | Глубина 0 |
| Указатель на - | Указатель на 20 | Указатель на - |

3) **=== начинается основной цикл ===**

3.1) на каждой итерации цикла проходим по списку и ищем элементы с глубиной, соответствующей глубине текущего прохода (для первой итерации это 0).  
Для каждого из таких элементов мы добавляем их левого и правого потомка в список рядом с элементом (слева и справа, соответственно).

После 1 итерации:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Глубина 0 | Глубина 1 | Глубина 0 | Глубина 1 | Глубина 0 |
| Указатель на - | Указатель на 6 | Указатель на 20 | Указатель на 40 | Указатель на - |

После 2 итерации:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Глубина 0 | Глубина 2 | Глубина 1 | Глубина 2 | Глубина 0 | Глубина 2 | Глубина 1 | Глубина 2 | Глубина 0 |
| Указатель на - | Указатель на 2 | Указатель на 6 | Указатель на 8 | Указатель на 20 | Указатель на 30 | Указатель на 40 | Указатель на 60 | Указатель на - |

Итак: Общее число итераций соответствует глубине дерева. На каждой итерации мы рассматриваем сразу все элементы текущего уровня (выбирая их из списка) и добавляем в список их потомки, чтобы рассмотреть их на следующей итерации.

Условие выхода из внешнего цикла:  
Условием выхода из цикла служит следующее выражение: desCount <> 0; Переменная disCount служит для подсчёта числа вершин на след. уровне дерева. Значение её увеличивается при добавлении очередного элемента в список. Очевидно, что если на следующем уровне нет ни одного узла, то обход дерева окончен (мы дошли до самого нижнего уровня).

Использование полученного списка для обхода элементов дерева (p2):  
Полученный список представляет последовательность указателей на элементы дерева, стоящих по возрастанию. Т.е. фактически мы выгрузили дерево в список. Теперь для того, чтобы вывести все элементы дерева по возрастанию достаточно просто пройти в цикле по списку от начала и до конца, выводя каждый элемент.

Анализ данных при помощи списка (p3):  
Задача – вывести число людей, моложе указанного возраста. Очевидно, что если список содержит все элементы дерева, то достаточно пройти по этому списку, сравнивая каждое значение с ключом и получая результат. Именно так я и поступил. Прошел по списку, сравнивая значения и суммируя их для получения общего числа.  
Тут стоит отметить, что список отсортирован по возрастанию, а это значит, что можно проходить не весь список, а только его часть. Ведь когда очередной встреченный во время прохода элемент стал >= ключ (заданный возраст), сразу можно сделать вывод о том, что это же будет выполняться и для всех оставшихся элементов списка (а значит, они уже не подходят по условию).

Маленький бонус: расширяемость алгоритма.  
Очевидно, что данный алгоритм можно без усилий подстроить под любое требуемое правило обхода. Будь то (корень-левое-правое), или (правое-левое-корень). Для подобных изменения достаточно лишь подправить место вставки новых элементов в список. Так для правила (корень-левое-правое) мы будем вставлять оба потомка элемента за ним, а для (правое-левое-корень) оба потомка встанут перед текущим элементом. Других изменения не потребуется!